Scalable Ranked Publish/Subscribe

Ashwin Machanavajjhala* , Erik Vee† , Minos Garofalakis† , and Jayavel Shanmugasundaram†
*Cornell University and †Yahoo! Research

ABSTRACT

Publish/subscribe (pub/sub) systems are designed to efficiently match incoming events (e.g., stock quotes) against a set of subscriptions (e.g., trader profiles specifying quotes of interest). However, current pub/sub systems only support a simple binary notion of matching: an event either matches a subscription or it does not; for instance, a stock quote will either match or not match a trader profile. In this paper, we argue that this simple notion of matching is inadequate for many applications where only the "best" matching subscriptions are of interest. For instance, in targeted Web advertising, an incoming user ("event") may match several different advertiser-specified user profiles ("subscriptions"), but given the limited advertising real-estate, we want to quickly discover the best (e.g., most relevant) ads to display.

To address this need, we initiate a study of *ranked* pub/sub systems. We focus on the case where subscriptions correspond to interval ranges (e.g, age in [25,35] and salary $> \$50,000$), and events are points that match all the intervals that they stab (e.g., age=28, salary = \$65,000). In addition, each interval has a score and our goal is to quickly recover the top-scoring matching subscriptions. Unfortunately, adapting existing index structures to solve this problem results in either an unacceptable space overhead or a significant performance degradation. We thus propose two novel index structures that are both compact and efficient. Our experimental evaluation shows that the proposed structures provide a scalable basis for designing ranked pub/sub systems.

1. INTRODUCTION

The exploding volume of information available on the Internet has fueled the development of middleware systems that are based on the publish/subscribe (or pub/sub) paradigm. Such systems rely on efficiently matching streams of published events to a large number of subscriptions that correspond to subscriber interests in specific classes of events. A canonical example of pub/sub systems involves stock trading: publishers such as the New York Stock Exchange publish stock quotes and stock traders register their interest in specific stock events, e.g., notify me when the stock price of Apple exceeds \$200.

While there has been a large body of work on building scalable pub/sub systems (e.g., [3, 6, 13, 15, 25]), all of them rely on a simple binary notion of matching that assumes that each event either matches a subscription or it does not, and *all* matching subscriptions are returned. However, many emerging applications require a more sophisticated notion of matching, where only the "best" matching subscriptions are of interest. This gives rise to a new class of pub/sub systems that we call *ranked* pub/sub systems. We motivate the need for such systems using three application scenarios.(Note that we are using the term "pub/sub" in a somewhat unconventional manner, to capture scenarios where a dynamic stream of events must be quickly matched against a large collection of standing subscriptions. These subscription matchings are not necessarily tied to an underlying event-notification or data-dissemination service.)

Targeted Web Advertising: An emerging trend in online advertising is enabling advertisers to target users based on information such as user demographics, profile information and online activity [1, 11, 28]. For instance, a mortgage vendor may wish to target online users between 20 and 35 years of age, who have a credit score between 400 and 500, and who have visited a real estate Web site at least 3 times in the past month, and show an ad tailored to such users when they visit an online website. This can be modeled as a pub/sub problem, where the stream of incoming users corresponds to events (e.g., a user with age = 25, credit score = 441, and real estate count = 6), and the advertiser specifications are subscriptions (e.g., $20 \leq$ age ≤ 35 and $400 \leq$ credit score ≤ 500 and real estate count ≥ 3). However, unlike traditional pub/sub systems, we do not wish to retrieve all the subscriptions (ads) that correspond to a given event (user) because we can only show a small number of ads in a Web page. Rather, we only wish to retrieve the "best" subscriptions based on some criteria such as the most targeted ads (tightest enclosing rectangles), the most profitable ads, the most underserved ads, etc.

Online Job Sites: Several online job sites (e.g., `HotJobs.com`, `Monster.com`) allow job seekers to register profiles, and also allow job posters to specify job seeker profiles that they are interested in. For instance, a job seeker may register a profile for nursing jobs that pay \$50 per hour and have a 25 hour work week, while a job poster may express an interest in nurses who are willing to work between 20 and 30 hours per week for \$45-60 per hour. Then, when a job seeker visits the site, she can be presented with jobs that match her profile. This can again be modeled as a pub/sub problem, where the events are job seekers (e.g., job type = nursing, hourly rate = \$50 and hours per week = 25) and the subscriptions are job poster interests (e.g., job type = nursing, $45 \leq$ hourly rate ≤ 60, and $20 \leq$ hours per week ≤ 30). However, as in the targeted advertising case, we cannot show all the jobs that match a user profile because of the limited real estate on the Web page. Thus, we want to retrieve only the best jobs for a given user based on criteria such as the monetary value to the job poster, fairness of exposure across job postings, etc.

Application-level Routers: In information dissemination applications, application-level routers are commonly used to route docu-

Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright for components of this work owned by others than VLDB Endowment must be honored.
Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept., ACM, Inc. Fax +1 (212) 869-0481 or permissions@acm.org.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

ments based on their content [3, 6, 25]. For instance, in a financial news feed application, a news document can have fields such as the date posted and average analyst ratings, and subscribers can request documents within a specified date and analyst rating range. This corresponds to a typical pub/sub application, where the events are news documents (e.g., date posted = 16 Nov 2007 and analyst rating = 3) and the subscriptions are subscriber interests (e.g., date posted $>$ 1 Nov 2007 and analyst rating $<$ 4). However, in high-volume applications, an application-level router may be required to shed some load (i.e., avoid delivering some events to certain subscribers) due to CPU and/or network bandwidth limitations, and this load shedding needs to be guided by factors such as subscriber priority or service level agreements. This can again be modeled as a ranked pub/sub problem, where the score of a subscription is the subscriber priority or the deviation from the service level agreement.

In this paper, we initiate a study of scalable and efficient techniques for the ranked pub/sub problem. We focus on the problem where each event is represented as a point $(v_1, \ldots, v_d)$ over a d-dimensional space, and each subscription is represented as a set of intervals $(I_1, \ldots, I_d)$ over that space. (It is easy to see how the above motivating examples can be mapped to this model.) Further, we consider two notions of matching: exact matching and relaxed matching.

Exact Matching: A subscription $(I_1, \ldots, I_d)$ matches an event $(v_1, \ldots, v_d)$ if and only if $\forall i \in [1, d](v_i \in I_i)$, i.e., the event is fully contained in the subscription's hyper-rectangle. Further, every subscription has an associated score, and the goal is to return the top few subscriptions ordered by the score.[1] As an illustration of this semantics, consider an application-level router where we only wish to route messages that satisfy *all* the subscription constraints, and the score of a subscription represents the priority of the subscriber.

Relaxed Matching: A subscription $(I_1, \ldots, I_d)$ matches an event $(v_1, \ldots, v_d)$ if and only if $\exists i \in [1, d](v_i \in I_i)$, i.e., at least one dimension of the event is contained in the corresponding interval of the subscription. Further, a weight w_d is associated with *each dimension* of a subscription, and the score of a subscription is the sum[2] of the weights of the matching dimensions, i.e., $\Sigma_{i \in \{j | j \in [1,d] \wedge v_j \in I_j\}} w_i$. For instance, in online advertising and job sites, it is preferable but not necessary to satisfy all the subscription constraints (e.g., we might show an ad even if it does not fully satisfy the user profile). Further, different dimensions of an ad may be weighted differently (e.g., a credit score match may be more important to an advertiser than an age match).

Given the above problem statement, a natural question arises: How do we implement the these ranked pub/sub systems in order to achieve scalability and efficiency? One naive approach is to use a traditional (unranked) pub/sub system to retrieve *all* the subscriptions that match an event, and then perform some post-processing to retrieve the top few matches. Such a naive solution, however, is clearly inefficient since it produces all the matching subscriptions even though only the top few matches are desired. This issue is particularly problematic in applications like online advertising and job sites, where the number of matches (i.e., ads, jobs) far exceeds the number that can be shown on a single Web page, and in application-level routers, where producing all the matches consumes already scarce resources.

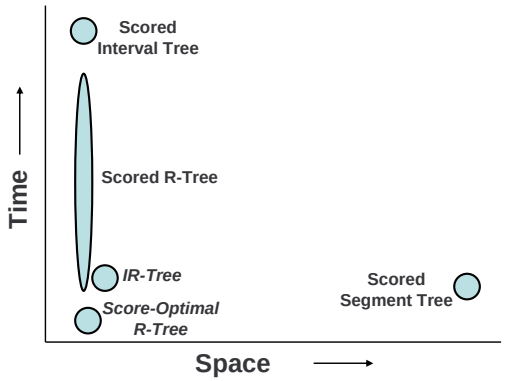

Figure 1: Space-Time Tradeoffs for Scored Interval Indices

Hence, we propose an alternative solution that works as follows. For each dimension i, we build a *Scored Interval Index* over the subscription intervals in that dimension. A Scored Interval Index is designed to take in a event value v_i and provide an efficient `getnext()` iterator that returns the intervals containing v_d *in the order of their score*. (For exact matching, the score of a subscription interval is simply the score of the subscription; for flexible matching, the score is the weight of the subscription interval in the given dimension.) Given these indices, an incoming event $(v_1, \ldots, v_n)$ is processed as follows. First, the indices are probed with the event values to produce a set of iterators. Then, in the case of exact matching, the intervals produced by the iterators are intersected to produce the subscriptions in score order. In the case of flexible matching, the Threshold Algorithm [17], which is an instance-optimal algorithm for merging multiple ranked lists with different rank (weight) orders, is used to efficiently find the subscriptions with the highest scores.

Our decision to use many one-dimensional Scored Interval Indices instead of using a single multi-dimension index warrants some discussion. The primary reason for this choice is our requirement to support relaxed matching, which requires the ability to drop certain subscription dimensions from consideration at event processing time. While we are not aware of any multi-dimensional index with this capability, the Threshold Algorithm easily handles this case by ignoring the weights for the dropped dimensions when computing the overall score. The other reason for our choice is more pragmatic: to the best of our knowledge, there are no interval index structures (one-dimensional or multi-dimensional) in the literature that are optimized for ranked retrieval of scored matches. So, the focus of this paper is on the easier (albeit still challenging!) problem of developing scored one-dimensional indices for processing multi-dimensional events.

Given the above system architecture, the main technical challenge is devising efficient Scored Interval Indices. Existing interval index structures such as interval trees [23], segment trees [12] and (1-dimensional) R-trees [18] are not directly applicable to this problem because they do not produce results in score order. Thus, we propose some simple adaptations to these structures that can produce results in score order. Unfortunately, our analytical and experimental results show that these adaptations are either time-inefficient (i.e., slow response time) or space-inefficient. Figure 1 pictorially depicts this qualitative tradeoff (not drawn to scale).

Based on the above observations, one of our main contributions is the development of two new index structures – the *Interval R-tree* (*IR*-tree) and the *Score-Optimal R-Tree* (*SOPT-R*-tree) – that

[1] In our work, we assume that subscription scores are given and are *independent of the matching event/point* — extending our techniques to handle event-dependent subscription scores is a challenging area for future work.

[2] More generally, the score of a subscription can described by *any monotonic function* of the weights.

are both time- and space- efficient (Figure 1). The *SOPT-R*-tree, which relies on intelligent pre-processing of the underlying interval set before indexing it using an *R*-tree, is the most efficient in terms of both time and space. In fact, we can prove that we can retrieve the top-k results in $O(k \times \log n)$ time, where n is the number of subscriptions. However, *SOPT-R*-trees cannot handle incremental updates easily. On the other hand, the *IR*-tree, which is a hybrid between an Interval Tree and an *R*-tree, is marginally slower than the *SOPT-R*-tree, but is incrementally updateable.

In summary, the main contributions of this paper are:

- We propose and formalize the novel problem of *ranked publish/subscribe*.
- While exploring simple adaptations of existing structures to support scoring, we identify an interesting space-time tradeoff (Section 2) for ranked subscription retrieval.
- We devise new, score-aware index structures that are both space and time efficient (Section 3).
- We give an experimental evaluation of the proposed indexing structures that convincingly demonstrates the benefits of our approach (Section 4)

2. RANKED RETRIEVAL USING EXISTING INTERVAL INDEX STRUCTURES

In this section we describe three existing interval index structures, namely the Interval Tree, the Segment Tree and the *R*-tree. These index structures are designed to support *interval stabbing queries*, i.e., queries that return the set of all intervals that are stabbed by a given query point. We, however, are interested in *top-k interval stabbing queries*, i.e., queries that return the top-k scoring intervals that are stabbed by a query point. In the latter part of this section, we describe scored adaptations of these index structures that support top-k interval stabbing queries.

2.1 Standard Interval Index Structures

Interval- and segment-tree indexes are the "standard" known solutions for efficiently processing simple interval-stabbing queries over the real line. We now briefly describe the key ideas behind each index. In either case, the input comprises of a collection of n intervals $\mathcal{I}$, where each interval $I_i \in \mathcal{I}$ is a pair of left/right endpoints ($I_i = [x_i^l, x_i^r], i = 1, \ldots, n$).

2.1.1 Interval Trees

An *interval tree* [23] over $\mathcal{I}$ is constructed in a recursive manner as follows. We pick the median endpoint x_{med} of the interval collection, and let $\mathcal{I}(x_{med}) \subset \mathcal{I}$ denote the subset of intervals in our collection that are stabbed by x_{med}. Also, let $\mathcal{I}^l(x_{med})$ ($\mathcal{I}^r(x_{med})$) be the subset of intervals completely to the left (resp., right) of the median x_{med}. Create an interval-tree node v containing two sorted lists of the intervals in $\mathcal{I}(x_{med})$: one sorted by interval left-endpoints and one sorted by interval right-endpoints. Then, for the left and right child subtree of v, recurse the above construction on $\mathcal{I}^l(x_{med})$ and $\mathcal{I}^r(x_{med})$, respectively. Note that, assuming a roughly even split of intervals across the x_{med} (i.e., $|\mathcal{I}^l(x_{med})| \approx |\mathcal{I}^r(x_{med})|$), the height of the interval tree is $O(\log n)$. To process a stabbing query for point q over the interval tree, start from the root node, and, for each visited tree node v: If $q = x_{med}$ at v, then simply return all the intervals in the v node; else, if $q < x_{med}$ at v, then traverse the left-endpoint-sorted list at v to return all intervals that begin before q, and recurse the search on the left child of v; otherwise, traverse the right-endpoint sorted list at v to return all intervals that end after q, and recurse the search on the right child of v.

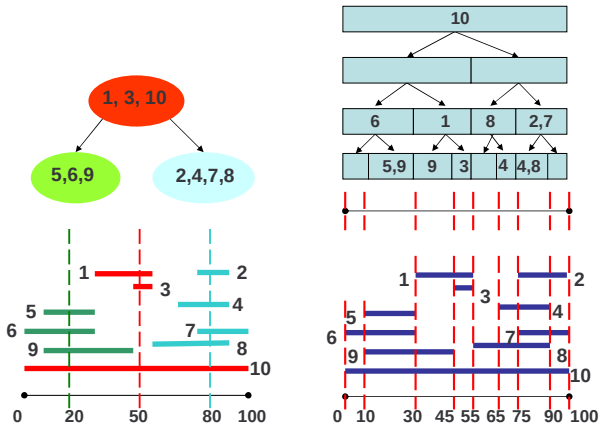

Figure 2: Interval and Segment Trees

It is not difficult to see that the space requirements of the interval tree over $\mathcal{I}$ are $O(n)$ (as each interval is stored only in a single tree n node and replicated in two lists). The construction time for the interval tree is $O(n \log n)$, and, letting $m(q)$ denote the size of the answer set for a stabbing query q, the time to answer q is $O(m(q) + \log n)$.

EXAMPLE 1. *Figure 2 (on the left) shows a set of ten intervals (labeled* 1 *through* 10*) on a line segment between* 0 *and* 100*.* 50 *is the median end point. This partitions the intervals into the set of intervals stabbed by* 50*,* $\{1, 3, 10\}$*, the set of intervals completely to the left of* 50*,* $\{5, 6, 9\}$*, and the set of intervals to the right of* 50*,* $\{2, 4, 7, 8\}$*. The recursive construction stops at the next level after finding medians* 20 *and* 80 *that stab all the intervals in* $\{5, 6, 9\}$ *and* $\{2, 4, 7, 8\}$*, respectively. The left end-point and the right end-point sorted lists maintained in the root node (not shown in the figure), for instance, would be* $\{10, 1, 3\}$ *and* $\{1, 3, 10\}$*, respectively.*

2.1.2 Segment Trees

In contrast to interval trees, a *segment tree* [12] over $\mathcal{I}$ relies on partitioning the intervals in $\mathcal{I}$ into a collection of *disjoint, atomic segments*, and then indexing these segments using a binary-tree structure. The atomic segments are simply defined by sorting the collection of all $2n$ endpoints in $\mathcal{I}$ and taking the segments defined by consecutive endpoints in the list (including $-\infty$ and $+\infty$ as the leftmost and rightmost points, respectively); note that this results in at most $2n+1$ atomic segments over $\mathcal{I}$. The segment tree over $\mathcal{I}$ is a balanced binary tree over the above sequence of atomic segments. Note that each node v of the tree can be described by a single *extent interval* $\texttt{interval}(v)$ that is equal to the union of all atomic segments under v's subtree. A node v stores the interval ids for all intervals $I \in \mathcal{I}$ such that $\texttt{interval}(v) \subseteq I$ but $\texttt{interval}(u) \not\subseteq I$, where u is the parent node of v (in other words, $\texttt{interval}(v)$ is a maximal node-extent interval in the tree that is completely contained in I). Processing a stabbing query using a segment tree is greatly simplified by the fact that the atomic segments partition the underlying domain, which, in turn, implies that, at each level of the segment tree, the query point q stabs *exactly one* of the node-extent intervals. Thus, starting from the root, we only need to follow a path of stabbed nodes to a leaf and return all the interval ids stored at each stabbed node.

The construction time and query time requirements for a segment tree are similar to those for an interval tree: $O(n \log n)$ and $O(m(q) + \log n)$, respectively (where $m(q)$ is again the size of the answer set for q). A key difference lies in the space requirements of the two structures: By partitioning each interval in $\mathcal{I}$ across node-extent intervals, in the worst case, each interval id can be replicated across at most two distinct (non-sibling) nodes *at each level* of a segment tree. Thus, the worst-case space requirements of the segment tree are $O(n \log n)$.

EXAMPLE 2. *Figure 2 (on the right) shows the same set of intervals* $\{1, \ldots, 10\}$ *when indexed by a segment tree. Let* $x_1, \ldots, x_9$ *denote the* 9 *distinct end points. Each segment* $[x_i, x_{i+1}]$, $i = , 1 \ldots, 8$, *is an atomic segment. Hence, the segment tree has four levels, with the lowest level containing one node* $v_{x_i, x_{i+1}}$ *for each atomic segment. The interval* 9*, for instance, is stored in the nodes* $v_{10,30}$ *and* $v_{30,45}$*, since both* $[10, 30]$ *and* $[30, 45]$ *are contained in interval* 9*, but none of their parent nodes are contained in* 9*. To illustrate query processing, let* $75 < q < 90$*. An interval stabbed by* q *contains the atomic segment* $[75, 90]$ *and hence should appear in the node* $v_{75,90}$ *or one of its parents. Therefore, the intervals* $\{10, 2, 7, 4, 8\}$ *intersect* q*.*

2.1.3 R-Trees for Stabbing Queries

Conventionally, R-trees [18] have been used for indexing hyper-rectangles in order to efficiently search for all rectangles that overlap with a query rectangle. In a single dimension, intervals "overlap" a query point q if and only if they are stabbed by q. Hence, we can use R-trees to solve our problem. The R-tree groups intervals into partitions of size at most b, where b is the *branching factor*. Various heuristics can be used for grouping intervals, including minimizing the size of the bounding interval for a group, minimizing bounding interval overlap between groups, or grouping intervals by their start or end points.

Each group of intervals is stored in a *leaf node* of the R-tree. The leaf node is associated with an *extent interval* which is the *minimum bounding interval* of the intervals in leaf node. Suppose $[\ell_i^g, r_i^g]$, $i = 1, \ldots, b$, are the intervals in a leaf node g. Then $I_g = [\ell^g, r^g]$, where $\ell^g = \min_i \ell_i^g$ and $r^g = \max_i r_i^g$ is the minimum bounding interval. The R-tree is constructed recursively on these minimum bounding intervals. Finally, we add a child pointer from the entry corresponding to interval I_g to the leaf node g. In order to answer a stabbing query q, we start from the root and keep chasing child pointers as long as q is in the extent of each intermediate node. Once, we reach a leaf node, we return the set of intervals that contain q. Note that intervals in multiple leaf nodes might contain the point q, and hence we might need to go down multiple root-to-leaf paths. It can be shown that an R-tree only requires $O\left(n \times \left(1 + \frac{2}{b-1}\right)\right)$ space, but in the worst case might take $O(n)$ time to return all the stabbed intervals.

EXAMPLE 3. *Figure 3 shows a set of intervals indexed by an* R*-tree with branching factor* 4*. The intervals are grouped so as to try and minimize the size of the bounding intervals. The leaf nodes partition the intervals into groups of at most* 4 *and each entry in the root node is a minimum bounding interval of the leaf nodes. The* R*-tree constructed in Figure 3 is especially bad since, for instance, every node in this* R*-tree needs to be visited to answer a query* $q = 35$*.*

2.2 Scored Interval Index Structures

We now describe *Scored Interval* trees, *Scored Segment* trees, and *Scored R*-trees, simple adaptations of the three standard interval indexes. Our analysis demonstrates that these scored variants

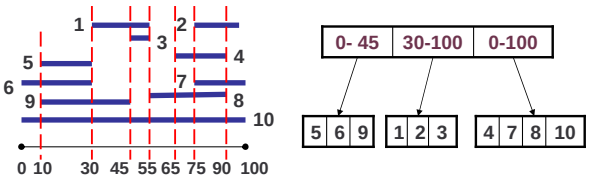

Figure 3: R-Tree

of the standard index structures give rise to an interesting space/-time tradeoff: While being very space efficient, interval trees and R-trees require significantly more time to process top-k stabbing queries (time linear in the number of intervals, in the worst case); in contrast, segment trees allow for extremely efficient ranking but, of course, can also incur a $O(\log n)$ factor blowup in space. Note that, with subscription numbers n in the millions, such $O(\log n)$ can be very significant — in the worst case, they can render main-memory indexing infeasible.

We first introduce some basic notation that will be useful for our analysis of the scored interval- and segment-trees. Given an index node v (in either an interval tree or a segment tree) and a query point q, we use n_v to denote the number of intervals in $\mathcal{I}$ that are stored in node v, and $m_v(q)$ to denote the number of those intervals that are stabbed by q. We also define $m_v(\bar{q}) = n_v - m_v(q)$ (i.e., the number of v's intervals that do *not* contain q). Finally, we let $\texttt{path}(q)$ denote the set of nodes on a root-to-leaf tree path that are traversed when processing query point q.

2.2.1 Scored Segment Trees

A segment-tree index can be easily adapted to return the top-k scoring intervals of $\mathcal{I}$ stabbed by a query point q. Recall that, for each I stored in v, $I \supseteq \texttt{interval}(v)$. Hence, the key observation here is that if the extent interval of v is stabbed by q (and, thus, accessed during the basic retrieval algorithm), then *all interval ids* stored in node v are also guaranteed to be stabbed by q.

For score-based retrieval, the intervals in each segment-tree node are stored sorted in the order of their scores. (Note that this does not increase the asymptotic segment-tree construction cost, which remains $O(n \log n)$.) Then, since a query point only stabs $O(\log n)$ tree nodes along a root-to-leaf path, we can retrieve the stabbed intervals in rank order by simply maintaining a max-heap of size $O(\log n)$ across these stabbed nodes. A $\texttt{getnext}()$ operation simply extracts the maximum element from the heap (belonging to, say, stabbed node v), and then replenishes the heap by inserting the next-best interval from node v — the total cost of both operations is only $O(\log \log n)$. (In fact, since this next-best interval has a lower score, we can use the more efficient $\texttt{decreaseKey}()$ operation on the heap [10].) The overall cost for retrieving the top-k scoring stabbed intervals for q (including the cost to build the initial max-heap) is $O(\log n \log \log n + k \log \log n)$.

2.2.2 Scored Interval Trees

In contrast to segment trees, the intervals stored in an interval-tree node that is explored during a (conventional) stabbing query, are *not* all guaranteed to be stabbed by the query point. Thus, to support ranked stabbing queries, the retrieval algorithm needs to query both *interval end points and scores* at each interval tree node. Hence, the interval tree can be adapted to support ranking naturally in two ways.

Conventional (Endpoint-Sorted) Interval Tree. One approach is to simply employ the basic interval-tree index structure. Like in the basic stabbing query algorithm, at each visited tree node v, we re-

trieve the intervals stabbed by the query point q. We also keep track of the top-k scoring intervals in v using a per-node max-heap structure of size k. (Of course, the heap is only needed if $m_v(q) > k$.) Then, to extract the global top-k stabbed intervals in the tree, we maintain a global max-heap of size $O(\log n)$ to keep track of each of the nodes on $\texttt{path}(q)$. Each call to $\texttt{getnext}()$ extracts the best interval (say, from node v) from the global max-heap, which then replenishes itself by inserting the next-best interval from v (similar to the heap described for the segment-tree scheme). The overall (worst-case) time complexity, which includes the time to build the per-node max-heaps as well as the time to build and probe the traversed-path max-heap is $O(m(q)\log k + k(\log k + \log\log n))$.

Score-Sorted Interval Tree. Rather than sorting intervals in a node by their endpoints, an obvious alternative is to sort intervals by their scores (thus, essentially, favoring score ranking instead of stabbing-based selection). Each node in this *score-sorted* interval tree index also maintains the minimum and maximum endpoints across all intervals stored in the node — this allows us to quickly determine whether the intervals in a node are potentially stabbed by an input query point.

In order to retrieve the top-k stabbed intervals, we maintain a global max-heap of size $O(\log n)$ across the nodes on the root-to-leaf path stabbed by the query point q (i.e. $\texttt{path}(q)$) that, at each point, contains the next-best stabbed interval from each node. The problem here is that, since intervals in each node v are sorted by score, getting the next-best interval from v that is actually stabbed by q might require an expensive linear scan over the score-sorted list; in the worst case, we may need to examine (and discard) $O(m_v(\bar{q}))$ intervals from each node v on the query path. Thus, we expect this indexing scheme to perform well only if most of the intervals in the stabbed nodes on $\texttt{path}(q)$ are actually stabbed by q (i.e., the $m_v(\bar{q})$'s are small). The overall worst-case time complexity for top-k retrieval using the score-sorted interval tree is $O(\sum_{v\in\texttt{path}(q)} m_v(\bar{q}) + (\log n + k)\log\log n)$.

Our experiments have shown that the score-sorted interval tree is typically much more efficient than the conventional interval tree for top-k stabbing queries, since the running time only depends on the number of intervals not stabbed along a path, rather the total number of intervals stabbed by the query. Hence, in the rest of the paper, we define the scored interval tree to be a score-sorted interval tree.

2.2.3 Scored R-Trees

Recall that in an R-tree, we have the flexibility to group intervals together based on different criteria. In order to answer top-k stabbing queries, it is natural to group intervals by their scores so that the top scored intervals are grouped together, the next lower scored intervals are grouped together, and so on. In other words, we order intervals in decreasing order of their scores and pick consecutive blocks of size b to form the leaf node groups. Recursively, if $(g_1, \ldots, g_k)$ are the set of internal nodes at any level of the R-tree (in that order), then every interval in the subtree of g_1 has a score at least as large as that of every interval in the subtree of g_2. This property ensures that the following simple *[left-first] depth-first* traversal implements a $\texttt{getnext}()$: Starting with the root node of the R-tree, at each internal node, scan each entry from left to right and recurse on its child node only if its extent interval contains the query point q. At a leaf node, scan the intervals from left to right and record an interval if it is stabbed by q. Return from the recursive call either if all entries in the node have been processed or if k intervals have been recorded.

LEMMA 2.1. *Given an R-tree on a set of intervals $\mathcal{I}$ arranged in descending order of score and a query q, performing a [left-first] depth-first search till k intervals stabbed by q correctly returns the top-k scoring intervals stabbed by q.*

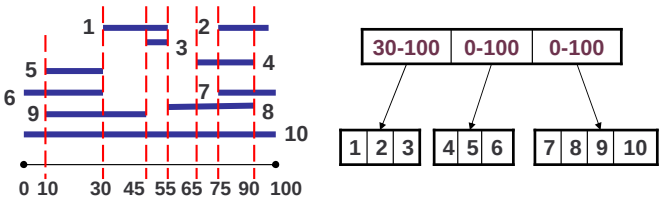

Figure 4: Scored R-Tree (Interval Ids Sorted by Score)

The problem with the above simple solution is that, in many cases, this [left-first] depth-first search traversal ends up visiting leaf nodes where the query point does not stab *any* of the intervals. The reason should be intuitively obvious: Recall that a scored *R*-tree groups intervals based solely on their score ranking and with no regard to their spatial extents. Unfortunately, this implies that the coverage of the "bounding" extent intervals for internal nodes in the resulting *R*-tree often contains a *large number of "holes"* — in other words, there will often be a large number of sub-ranges in a node's coverage that do not intersect *any actual interval* in the underlying collection. As a simple example, consider the collection of intervals depicted in Figure 4, where the ordering of intervals on the y-axis corresponds to their scores. (Thus, interval 1 is the highest-scoring interval, interval 2 is the second highest, and so on.) Figure 4 also shows an example scored R-tree for that interval collection. Note that the [left-first] depth-first traversal for, say, $q = 90$ would visit *all the nodes* in the tree incurring $O(n)$ time for a single $\texttt{getnext}()$.

In summary, our analysis quantifies parts of the qualitative space/-time tradeoff illustrated in Figure 1. Scored segment trees are very efficient in answering top-k stabbing queries; however, they are space inefficient and might not allow an in-memory implementation. On the other hand, scored interval trees and R-trees are very space efficient. However, modulo the score-sorted variant, scored interval tree adaptations are only good at indexing the intervals but not their score. Similarly, R-trees can only index either scores or the intervals, but not both. Hence, these index structures are lacking in terms of time efficiency. Nevertheless, we carry over the insights from these simple adaptations to design two novel index structures – the *IR*-tree (Section 3.1) and the *SOPT-R*-tree (Section 3.2) – that are as space-efficient as interval and R-trees, and, at the same time, can answer top-k stabbing queries as quickly as segment trees.

3. TIME AND SPACE EFFICIENT TOP-K INTERVAL INDEXES

While requiring significantly smaller space than the segment-tree solution (essentially, avoiding the $O(\log n)$ replication blowup), the interval-tree schemes described above can also be significantly more expensive in terms of computation time. Likewise, the scored R-tree, although compact, has unpredictable performance times. We now describe two novel scored interval indexing structures, *Interval R-trees* (*IR*-trees) and *Score-Optimal R-trees* (*SOPT-R*-trees), that are provably efficient in terms of both query time and memory requirements.

3.1 The IR-Tree Index Structure

We saw earlier that interval trees and their variants store lists of intervals at their nodes; in the worst case, answering a query

may require traversing the entire list. The key idea in *IR*-trees is to employ a more time-efficient data structure than a list — more specifically, we index the set of intervals at each interval tree node by an R-tree. For example, in Figure 2(a), intervals 5, 6, 9 are indexed with an R-tree, and similarly 2, 4, 7, 8 and 1, 3, 10. As we mentioned earlier, R-trees may still have linear search times in the worst case. However, we are saved by a crucial technical observation: By the construction of interval-tree nodes, every interval stored at a node is stabbed by a common point (namely, the median point corresponding to the node). For instance, intervals 5, 6, 9 are stabbed by a common point in Figure 2(a). This observation allows us to guarantee efficient query times.

More formally, we can prove the following lemma, which relies on the fact that the extent of every internal node in this R-tree index has *no "holes" in the node's coverage*; in other words, if the R-tree node is stabbed by a query point q, then at least one interval stored in its subtree is guaranteed to contain q. This fact ensures efficient ranked retrieval.

LEMMA 3.1. *Given an R-tree constructed over a set of n intervals in which every pair of intervals overlap and a query q, retrieving the set of top-k scoring intervals stabbed by q takes $O(bk \log_b \frac{n}{k})$ steps.*

PROOF. The extent interval I_g of an internal node g is the minimum bounding interval of all the intervals in its subtree. Since every pair of intervals intersect, if q stabs I_g, there should exist an interval I in g's subtree such that q stabs I. If we ever go down a child pointer, we are guaranteed that the extent interval in one of the entries in the child node contains q. Hence, one root to leaf traversal is enough to find the top scoring interval stabbed by q. Thereafter, finding the next best interval involves at most traversing back up to the root and an additional root-to-leaf traversal. Hence, this takes at most $O(bk \log_b(n/k))$ steps. □

We construct an *IR*-tree over $\mathcal{I}$ as follows. First, we build a score-sorted interval tree on $\mathcal{I}$. Then, at each node v of the tree, we index the sorted list of n_v intervals at v (in order of decreasing score) by building a (scored) R-tree index on top of the list.

The top-k interval retrieval algorithm over an *IR*-tree is similar to the basic interval-tree search algorithm, but also employs the embedded R-tree structure at each traversed node v to efficiently find the top-scoring interval stored in v. More specifically, starting with $v =$ root of the *IR*-tree, we can find the top-scoring interval in v by performing the [left-first] depth-first traversal of the R-tree at node v until we find an interval that is stabbed by the query q. That is, we call `getnext()` on the R-tree at node v. A trivial extension of Lemma 3.1 shows that each of these calls to `getnext()` takes time only $O(b \log_b n_v)$.

Once the above step for searching node v is complete, we check the location of q compared to the median endpoint of v, and recurse on the left or right child of v in the *IR*-tree as in traditional interval-tree search. Finally, we return the best interval found from amongst the $O(\log n)$ nodes we traversed.

To discover the next-best interval (for a `getnext()` operation on the *IR*-tree) during top-k processing, we maintain an $O(\log n)$-size max-heap for the best intervals along the traversed query path (as earlier). If we returned the best interval from node v, then we must replenish the heap with the next-best interval from node v. Hence, we call `getnext()` on the R-tree associated with node v. If the call returns an interval, we place it into our max-heap. Otherwise, we have exhausted node v's stabbed intervals. As shown in Lemma 3.1, fetching k items from node v has total time complexity of just $O(bk \log_b(n_v/k))$, and, in general, is quite fast.

The time to set up the initial max-heap (which requires a traversal of the *I*-tree from the root to a leaf node) takes time $O((b \log_b n + \log \log n) \log n)$. Each subsequent call to `getnext()` for the *IR*-tree, along with a heap update, takes time $O(b \log_b n + \log \log n)$. Hence, the worst-case running time for a top-k retrieval is bounded by $O((k + \log n)\, b \log_b n)$.

In terms of space complexity, the *IR*-tree clearly requires only $O(n)$ space (in fact, its size is at most $1.5\times$ the size of a conventional interval tree, even with an R-tree of branching factor $b = 2$). Using an R-tree with branching factor $b > 2$ decreases the space blow-up over score-sorted interval trees (which take even less space than conventional interval trees) to be just $1 + \frac{2}{b-1}$. However, it also increases the worst-case running time asymptotics by a factor $O(b/\log b)$. Due to caching affects and other overhead, performance can actually improve for modest values of b, while simultaneously decreasing memory requirements. We summarize these performance guarantees below.

THEOREM 3.1. *An* IR*-tree indexing n intervals has space complexity $O(n)$, and in general takes at most a factor $(1 + \frac{2}{b-1})$ more space than a score-sorted interval tree, where b is the branching factor of the R-trees at the nodes of the* IR*-tree. The time to process a top-k query is bounded by $O((k + \log n)\, b \log_b n)$.*

3.2 The SOPT-R-Tree Index Structure

We now explore the *SOPT-R*-tree data structure which has the memory requirements of an R-tree, but also guarantees fast query times — its worst-case running time is just $O(kb \log_b n)$ to produce a top-k list over n items.

The *SOPT-R*-tree is, in fact, a scored R-tree, in which we carefully sort the intervals in such a way that we hit very few "holes". Recall that, in the score-sorted R-tree discussed earlier, intervals are sorted by their score, and the R-tree is built on top of these intervals. For certain distributions, this approach works well. However, for many distributions, this will produce many "holes," leading to poor performance. By a clever rearrangement of the intervals, our *SOPT-R*-tree index can avoid most of these holes. In fact, we prove that, *for any top-k query*, we explore at most $2k$ leaf nodes of the tree, corresponding to hitting at most k holes.

The main optimization idea stems from the following realization. Suppose that I_1 and I_2 are intervals that we wish to index. Further, suppose that the score of I_1 is greater than the score of I_2, and that no interval has a score between the score of I_2 and the score of I_1. If I_1 and I_2 intersect, then any R-tree indexing them must place I_1 before I_2. (To see this, suppose that $q \in I_1 \cap I_2$; then, a query q must return I_1 before I_2.) However, if I_1 and I_2 do not intersect, we are free to place them in either order, since no query point can stab both intervals — their relative ordering is immaterial. In the next section, we show how to leverage this simple property to produce a provably good interval arrangement.

3.2.1 *Generating a* SOPT-R*-tree*

Before describing the underlying arrangement of intervals in a *SOPT-R*-tree, we first define a *constraint graph* for the intervals. In essence, this constraint graph captures the allowable arrangements of intervals.

Consider the set $\mathcal{I}$ of n input intervals, each with a score, and define $\widetilde{G}(\mathcal{I})$ to be the directed graph $(V, \widetilde{E})$, where V and $\widetilde{E}$ are as follows: The set V consists of n nodes, one node for each interval $I \in \mathcal{I}$. We refer to the node associated with I by $\texttt{node}(I)$. We include an edge in $\widetilde{E}$ from $\texttt{node}(I_1)$ to $\texttt{node}(I_2)$ if and only if $I_1 \cap I_2 \neq \emptyset$ and $\texttt{score}(I_1) > \texttt{score}(I_2)$.

However, it will be useful for us to use a more efficient representation of this simple intersection graph that, intuitively, tries to

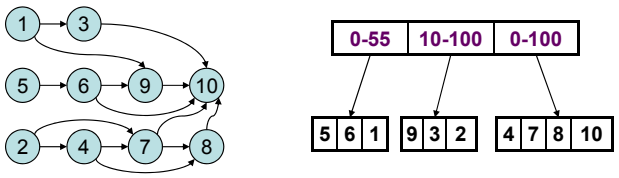

Figure 5: *SOPT-R*-Tree

avoid some extraneous "transitive" edges. Formally, define graph $G = (V, E)$ to have the same vertex set as $\widetilde{G}$, and E defined as follows. Suppose $I_1, I_2 \in \mathcal{I}$ with $\texttt{score}(I_1) > \texttt{score}(I_2)$. Then, E contains an edge from $\texttt{node}(I_1)$ to $\texttt{node}(I_2)$ if and only if (a) $I_1 \cap I_2 \neq \emptyset$; and, (b) there exists a point $q \in I_1 \cap I_2$ such that, for all $I \in \mathcal{I}$ with $\texttt{score}(I_1) > \texttt{score}(I) > \texttt{score}(I_2)$, the point $q \notin I$. Clearly G contains only a subset of the edges in $\widetilde{G}$; furthermore, it is not difficult to see that, if there is an edge from $\texttt{node}(I_1)$ to $\texttt{node}(I_2)$ in $\widetilde{E}$, then there is a *path* from $\texttt{node}(I_1)$ to $\texttt{node}(I_2)$ in E. Note that G includes some unnecessary "transitive" edges; however, we will be able to compute G extremely efficiently.

EXAMPLE 4. *Figure 5 shows the constraint graph G for our running example of intervals (see, e.g., Figure 4 — recall that the y-axis and interval ids are sorted by score). Interval* 1 *intersects intervals* 3 *and* 9*, and* $\texttt{score}(1) > \texttt{score}(3)$, $\texttt{score}(1) > \texttt{score}(9)$*; furthermore,* $1 \cap 3$ *and* $1 \cap 9$ *do not intersect any other intervals of intermediate scores. Hence, edges* $(1, 3)$ *and* $(1, 9)$ *appear in G. Note that, even though interval* 1 *also intersects interval* 10 *and* $\texttt{score}(1) > \texttt{score}(10)$*, there is no edge* $(1, 10)$ *in G; however, this edge is "covered" by the* $(1, 3, 10)$ *path in G.*

We say that an arrangement of the intervals in $\mathcal{I}$ *respects* $G(\mathcal{I})$ if for all intervals $I_1, I_2 \in \mathcal{I}$ such that there is an edge from $\texttt{node}(I_1)$ to $\texttt{node}(I_2)$, the interval I_1 comes before I_2 in the arrangement. Note that, by the fact that edges in $\widetilde{G}(\mathcal{I})$ always map to paths in $G(\mathcal{I})$, an arrangement respects $G(\mathcal{I})$ *if and only if* it respects $\widetilde{G}(\mathcal{I})$. Also, note that the arrangement described for scored R-trees, in which the highest scored intervals come first, clearly respects $G(\mathcal{I})$. The following lemma shows that any R-tree that groups intervals based on *any arrangement respecting* $G(\mathcal{I})$ will produce a ranked top-k list in the expected way.

LEMMA 3.2. *Let $\mathcal{I}$ be a set of scored intervals, and suppose arrangement $\mathcal{A}$ respects $G(\mathcal{I})$. Let T be the R-tree built on arrangement $\mathcal{A}$. Then, for any query q, performing [left-first] depth-first search on T to find the first k intervals stabbed by q will produce the top k scored intervals stabbed by q.*

PROOF. An R-tree built on arrangement $\mathcal{A}$, when performing depth-first search for query q, will simply return intervals stabbed by q in the order they appear in $\mathcal{A}$. So we only need to argue that for all k and q, the first k intervals appearing in $\mathcal{A}$ that are stabbed by q are in fact the top k scoring intervals stabbed by q.

Suppose not. Then we can find an interval I that is among the first k intervals according to $\mathcal{A}$ stabbed by q, and an interval J that is in the true top k list, but such that $\texttt{score}(I) < \texttt{score}(J)$. Thus, there is an edge from $\texttt{node}(J)$ to $\texttt{node}(I)$ in $\widetilde{G}$. That is, $\mathcal{A}$ does not respect $\widetilde{G}(\mathcal{I})$, hence does not respect $G(\mathcal{I})$. This is a contradiction. □

We are now ready to describe the algorithm that builds the interval arrangement for *SOPT-R*-trees. In a nutshell, the idea is to exploit the freedom allowed by the partial-ordering constraints specified in the constraint graph G, to ensure intervals are grouped together in terms of their *spatial proximity* (as long as that does not violate G). More specifically, let $\texttt{left}(I)$ denote the left endpoint for interval I. The first interval in the arrangement is the interval I with the smallest $\texttt{left}(I)$ value, taken over all I who have $\texttt{node}(I)$ with indegree 0. We remove the $\texttt{node}(I)$ from $G(\mathcal{I})$ and repeat this step recursively, until all intervals have been added. We restate this algorithm below. For convenience, we define $\texttt{indeg}(I)$ to be the indegree of $\texttt{node}(I)$. (We set this to -1 if $\texttt{node}(I)$ is not in $G(\mathcal{I})$.)

Algorithm 1 Arrangement for *SOPT-R*-trees

Require: Interval set $\mathcal{I}$ and constraint graph $G(\mathcal{I})$.
1: **while** $G(\mathcal{I})$ is not empty **do**
2: Let I be the interval with the smallest $\texttt{left}(I)$ value, taken over all I with $\texttt{indeg}(I) = 0$.
3: Add I to the arrangement, and remove $\texttt{node}(I)$ from $G(\mathcal{I})$.
4: **end while**
5: Output the arrangement.

For any set of intervals $\mathcal{I}$ with scores, the b-way *SOPT-R*-tree for $\mathcal{I}$ is defined to be the b-way R-tree created using the arrangement produced using Algorithm 1. As an example, Figure 5 shows the *SOPT-R*-tree created for our running-example interval collection from its corresponding constraint graph.

Since the algorithm respects the constraint graph $G(\mathcal{I})$, we know that it produces correct results. But the key property of *SOPT-R*-trees is in the lemma below. It guarantees that while processing a top-k query for point q using the *SOPT-R*-tree, we explore at most k leaf nodes of the tree that do not contain an interval stabbed by q. (That is, we hit at most k "holes.") This translates directly into an upper bound on the running time of any top-k query.

THEOREM 3.2. *Let $\mathcal{I}$ be a set of n scored intervals, and let T be the b-way* SOPT-R*-tree generated for $\mathcal{I}$. For any query q, the time to return a top-k list is at most $O(bk \log_b n)$. The total space taken by T is the same as an R-tree on $\mathcal{I}$, and can be implemented in $(1 + \frac{2}{b-1})$ times the space of the original interval data.*

PROOF. Fix a level of the tree T, and label the nodes on that level from left to right by $\nu_1, \nu_2, ..., \nu_m$, where m is the number of nodes on that level. Fix any query point q. We will first show that if there are two nodes, ν_i and ν_j with $i < j$, whose extent intervals both contain q, and there are no intervening nodes whose extent interval contains q, then either ν_i or ν_j (or both) index an interval that contains q.

Suppose not. That is, suppose that neither ν_i nor ν_j index an interval containing q. Let $\mathcal{A}$ be the arrangement produced by Algorithm 1. Since $i < j$, all of the intervals indexed by ν_i appear in $\mathcal{A}$ before the intervals indexed by ν_j. Let I be the lowest-scoring interval indexed by ν_i that is entirely to the right of q. (Since the extent interval of ν_i contains q, while no interval indexed by ν_i contains q, we know such an interval exists.) Let $\mathcal{S}$ be the set of all intervals that appear *after* the intervals indexed by ν_i in arrangement $\mathcal{A}$, and let J be the highest-scoring interval in $\mathcal{S}$ that lies entirely to the left of q.

By the ordering specified in Algorithm 1, J would appear before I (since its $\texttt{left}()$ value is smaller) unless there were a path from $\texttt{node}(I)$ to $\texttt{node}(J)$ in $G(\mathcal{I})$. This path consists of intervals that intersect each other, and they stretch from the left of q to the right of q. Hence, one of those intervals, say K, must cross q. Furthermore, since $\texttt{node}(K)$ lies on the path from $\texttt{node}(I)$ to $\texttt{node}(J)$, the arrangement $\mathcal{A}$ must order K between I and J. Hence, an intervening node must contain K (which is stabbed by q), a contradiction.

So, for every two nodes the algorithm explores, at least one will contain a stabbed interval. Hence, it will explore at most $2k$ nodes per level. Each node takes $O(b)$ time to explore, and there are $O(\lg_b n)$ levels. The running time follows. The space claims follow directly from the fact that the *SOPT-R*-tree is an R-tree. □

Note that, in the worst case, this is essentially the best query time we can hope to prove for any R-tree structure. There are interval sets such that for any arrangement and any k, there will be a query q whose stabbed intervals are contained in k different leaf nodes of the R-tree. Hence, the top-k search will explore at least k leaf nodes.

Although we are primarily concerned with the query time, the pre-processing time must be kept subquadratic, since we are frequently dealing with millions of intervals. Note that Algorithm 1 runs in time $O(n \log n + |E(\mathcal{I})|)$, where $|E(\mathcal{I})|$ is the size of the edge-set for $G(\mathcal{I})$: The find operation in step 2 can be performed using a heap (of size at most n); it is executed n times, taking $O(n \log n)$ time. Every time we remove a $\texttt{node}(\mathcal{I})$ from $G(\mathcal{I})$, we touch every out-edge of $\texttt{node}(\mathcal{I})$ and update the heap by adding any nodes that now have indegree 0. This takes an additional $O(|E(\mathcal{I})|)$ time overall. In the next section, we show that the graph $G(\mathcal{I})$ can be generated efficiently, and we also bound $|E(\mathcal{I})|$ by $3n$. Using these ideas, we prove that the pre-processing time to produce a *SOPT-R*-tree is $O(n \log n)$.

Handling Updates. While the *SOPT-R*-tree clearly offers the best time/space tradeoff among the different scored interval index structures explored in this work, it also raises some issues with respect to the update-ability of the data structure (when dealing with dynamic interval/subscription collections). The problem, of course, is that our constraint-graph optimizations are highly sensitive to the underlying interval collection: a single update could drastically alter the constraint graph structure, rendering the *SOPT-R*-tree index obsolete. Devising techniques for efficiently updateable *SOPT-R*-tree indexes is definitely an interesting area for future work; in the meantime, *IR*-tree indexes seem to offer the best tradeoff with respect to time/space requirements and the ability to incrementally update the index structure.

3.2.2 Efficient Construction of the Constraint Graph

There is an obvious $O(n^2)$ algorithm to produce the constraint graph $G(\mathcal{I})$. However, since n may well be in the millions, this is an unacceptably long pre-processing time. We show in this section that $G(\mathcal{I})$ can be constructed in time $O(n \log n)$, and that it has at most $3n$ edges. Since Algorithm 1 runs in time $O(n \log n + |E(\mathcal{I})|)$, this immediately shows the following result.

THEOREM 3.3. *Given any set $\mathcal{I}$ of n scored intervals, a* SOPT-R*-tree for $\mathcal{I}$ can be constructed in time $O(n \log n)$.*

In order to describe the efficient process for constructing $G(\mathcal{I})$, we will need an additional concept. For a subset $\mathcal{J} \subseteq \mathcal{I}$ of scored intervals, we say an endpoint p is *visible with respect to* $\mathcal{J}$ if there is some interval $I \in \mathcal{J}$ for which p is an endpoint, and further, there is no other interval $J \in \mathcal{J}$ with $\texttt{score}(I) > \texttt{score}(J)$ and $p \in J$. It may be helpful to consider the example drawn in Figure 3. (Again, recall that intervals are numbered by decreasing score.) Imagine looking upward from below the intervals. If $\mathcal{J}$ consists of the intervals 1 through 10, then the point $p = 30$ is *not* a visible endpoint with respect to $\mathcal{J}$ — intuitively, we may think of interval 10 as obscuring it. However, if $\mathcal{J}$ consists of the intervals 1 through 8, then $p = 30$ is a visible endpoint with respect to $\mathcal{J}$ — 30 is an endpoint of interval 6, and no lower-scoring interval contains (i.e., "obscures") 30.

The set of endpoints that are visible with respect to $\mathcal{J}$ break the real line into intervals, which we refer to as *visible blocks* to avoid confusion. We denote this set of visible blocks by $\texttt{visBlks}(\mathcal{J})$; the set $\texttt{visBlks}(\emptyset)$ contains only the interval $(-\infty, \infty)$. For every block $B \in \texttt{visBlks}(\mathcal{J})$, we say interval $I \in \mathcal{J}$ is *associated with* B if I is the lowest scoring interval in $\mathcal{J}$ such that $B \subseteq I$.

EXAMPLE 5. *In Figure 3,* $\texttt{visBlks}(\{1, 2, ..., 7\})$ *consists of the blocks* $(-\infty, 0]$,$[0, 30]$, $[30, 45]$, $[45, 55]$, $[55, 65]$,$[65, 75]$, $[75, 100]$, $[100, \infty)$. *Interval 6 is associated with block* $[0, 30]$. *Interval 1 is associated with block* $[30, 45]$, *interval 3 with* $[45, 55]$, *interval 4 with* $[65, 75]$, *and interval 7 with* $[75, 100]$. *The blocks* $(-\infty, 30]$, $[55, 65]$, *and* $[100, \infty)$ *have no associated intervals. Notice that each block has at most one interval associated with it.*

Our algorithm uses the following key property. In essence, it says that when considering the ith interval I_i, we only need to find the set of visible blocks that I_i intersects in order to find all edges pointing to $\texttt{node}(I_i)$ in $G(\mathcal{I})$.

LEMMA 3.3. *Let $\mathcal{I} = \{I_1, I_2, ..., I_n\}$ be a set of scored intervals, labeled so that $\texttt{score}(I_i) > \texttt{score}(I_j)$ for all $i < j$. Further, let $\mathcal{J}_\ell = \{I_i | i \leq \ell\}$ for $\ell \geq 1$. For any interval $I \in \mathcal{J}_{i-1}$, there is an edge from $\texttt{node}(I)$ to $\texttt{node}(I_i)$ in $G(\mathcal{I})$ if and only if I is associated with a visible block $B \in \texttt{visBlks}(\mathcal{J}_{i-1})$ such that $B \cap I_i$ is nonempty.*

PROOF. First suppose that I is associated with $B \in \texttt{visBlks}(\mathcal{J}_{i-1})$ with $B \cap I_i$ nonempty. Let $x \in B \cap I_i$. Then $x \in I \cap I_i$ as well. Furthermore, if there were an interval $J \in \mathcal{J}_{i-1}$ such that $x \in J$ and $\texttt{score}(I) > \texttt{score}(J) > \texttt{score}(I_i)$, it would violate the "visibility" condition on B. Hence, there is an edge from $\texttt{node}(I)$ to $\texttt{node}(I_i)$.

On the other hand, suppose there is an edge from $\texttt{node}(I)$ to $\texttt{node}(I_i)$. Then there is some $x \in I \cap I_i$ such that no interval J with $\texttt{score}(I) > \texttt{score}(J) > \texttt{score}(I_i)$ is such that $x \in J$. In other words, there is a block B containing x that is visible with respect to $\mathcal{J}_{i-1}$. □

Below, we give the algorithm to produce the constraint graph $G(\mathcal{I})$. For convenience, we assume that $\mathcal{I}$ contains the interval $(-\infty, \infty)$ with score ∞, so that every visible block throughout the algorithm will have an associated interval. The algorithm works through the intervals in $\mathcal{I}$ in decreasing order of their score, say $I_1, I_2, ...$ For each interval I_i, we determine the set of visible blocks from $\texttt{visBlks}(\mathcal{J}_{i-1})$ that intersect I_i, where $\mathcal{J}_{i-1}$ is defined as in Lemma 3.3. We can then add all edges in $G(\mathcal{I})$ pointing to $\texttt{node}(I_i)$. We repeat this until all intervals are processed.

Algorithm 2 Computing the constraint graph

Require: Interval set $\mathcal{I} = \{I_1, I_2, ..., I_n\}$, sorted in descending order by score.
1: Initialize $\mathcal{J} \leftarrow \{I_1\}$, and initialize G to have no edges and vertices $\texttt{node}(I_1), .., \texttt{node}(I_n)$.
2: **for** $i = 2$ to n **do**
3: **foreach** block $B \in \texttt{visBlks}(\mathcal{J})$ such that $B \cap I_i$ **do**
4: Set I to be the interval associated with block B.
5: Add an edge from $\texttt{node}(I)$ to $\texttt{node}(I_i)$.
6: **end foreach**
7: Add interval I_i to set $\mathcal{J}$.
8: **end for**
9: Output graph G.

Note that step 3 of the algorithm can be performed in time $O(\log n + b_i)$, where b_i is the number of visible blocks that intersect I_i: At each step, we maintain the set of visible endpoints with respect to

$\mathcal{J}$, sorted by value; we may do so using a tree. Given interval I_i, let x be its left endpoint and y be its right endpoint. We find consecutive visible endpoints with respect to $\mathcal{J}$, say $z_1 < z_2$, such that $z_1 \leq x < z_2$. (By a natural extension, we allow $z_1 = -\infty$ or $z_2 = \infty$.) This can be done in a tree in $O(\log n)$ time. Since the visible endpoints are maintained in sorted order, we may then find all visible endpoints greater than x and less than y in time $O(b_i)$. Note that to maintain the list of visible endpoints when we add interval I_i, we simply insert x and y, and remove all previously visible endpoints that lie between x and y.

Hence, the total running time of Algorithm 2 is $O(n \log n + \sum_{i=1}^{n} b_i)$. But notice that b_i is precisely the number of edges that point to $\texttt{node}(I_i)$, by Lemma 3.3. Hence, $\sum_{i=1}^{n} b_i = |E(\mathcal{I})|$. We will show in the next lemma that $|E(\mathcal{I})| \leq 3n$, hence the total running time is bounded by $O(n \log n)$.

LEMMA 3.4. *Let $\mathcal{I}$ be a set of n scored intervals. The graph $G(\mathcal{I})$ has at most $3n$ edges. Furthermore, $G(\mathcal{I})$ can be constructed in $O(n \log n)$ time.*

PROOF. We show $G(\mathcal{I})$ has at most $3n$ edges via a charging argument. Consider running Algorithm 2. Initially, we add I_1 to $\mathcal{J}$ so that there are three visible blocks. (The interval I_1 is associated with the middle block; the other blocks have no associated intervals.) We *charge* each of these three blocks to I_1.

In general, if I_ℓ is about to be added to $\mathcal{J}$ in the ℓth iteration, I_ℓ will intersect a number of visible blocks, say B_1,B_2,..., B_k ordered by their left endpoint. When I_ℓ is added to to $\mathcal{J}$, the blocks B_2,B_3,...,B_{k-1} will cease to be visible. Block B_1 is reduced to cover just $B_1 - I_\ell$, and likewise B_k reduces to $B_k - I_\ell$. Call these reduced blocks B_1' and B_k', respectively. So the addition of I_ℓ produces 3 new blocks, B_1', B_k', and the block whose range is I_ℓ. We charge each of these three blocks to I_ℓ.

But note that whenever an edge is added to a graph, we remove a visible block. (In the case outlined above, we may think of removing B_1 and adding block B_1'; likewise with B_k.) Every time we add a block, it is charged to some interval, so the total number of edges is at most the number of charged blocks. But every time we add an interval, exactly 3 blocks are charged to it. So the total number of edges is at most $3n$. The runtime thus follows from our previous discussion. □

COROLLARY 3.1. *An* SOPT-R*-tree for interval set $\mathcal{I}$ can be constructed in $O(n \log n)$ time.*

4. EXPERIMENTS

In this section, we present an experimental evaluation of the data structures we proposed for ranked pub/sub. As part of our experimental setup, we consider subscriptions on a d dimensional numeric domain. Recall that, each subscription $\mathbf{I}$ defines an interval I_d in each dimension (i.e., a hyper-rectangle). We focus our experimental evaluation on the more general relaxed matching problem. Accordingly, a score $f_j(I_j)$ is associated with each dimension k. An event is a point $\mathbf{q} = (q_1, \ldots, q_d)$ in the d dimensional domain. A subscription matches the event if in each dimension q_j stabs I_j. The combined score of a matching subscription is the sum of the scores of the matched intervals

$$f(I) = \sum_{j=1}^{d} \delta_j f_j(I_j) \qquad \text{where,} \qquad \delta_j = \begin{cases} 1 & \text{if } \mathbf{q} \text{ stabs } I_j \\ 0 & \text{else.} \end{cases}$$

We build d single-dimensional indices using each of the following five indexes – the Scored Interval Tree, the Scored Segment Tree, the Scored R-tree, our *IR*-tree and our *SOPT-R*-tree. We also experimented with the conventional interval tree and R-tree structures, but as expected, they were orders of magnitude slower than their scored counterparts, and we thus do no consider them further. Since we only deal with the scored variants of the standard structures, we will henceforth drop the 'scored' prefix when referring to them. All indices are implemented in main-memory in keeping with the response time requirements for online and routing applications.

We implement Fagin's Threshold Algorithm [17] over the d single dimensional indices that aggregates scores based on the above combination function. The algorithm retrieves the next best intervals from each of the indexes in a round-robin fashion. On retrieving an interval, the algorithm finds out the scores of the subscription in all the other dimensions, and computes the total score for this subscription. The algorithm maintains a heap of size k to keep track of the k best subscriptions retrieved at any point of time. Simultaneously, a threshold value, which is the sum of the scores of the most previous best interval retrieved from every dimension, is also updated. When the k^{th} smallest subscription has a higher score than the threshold, the algorithm terminates and returns the top-k subscriptions.

Data Set: We use a synthetically generated subscription workload for our experiments. We actually did have access to a data set of real subscriptions from Yahoo!'s Behavioral Targeting (BT) group, but on inspecting that data set, we found that most interval predicates were on pre-specified non-overlapping intervals. This was not due to a lack of demand for arbitrary intervals, but due to a limitation of the current user interface; specifically, the interface precluded subscribers from specifying arbitrary intervals because the underlying system did not support such intervals. Since our goal is to enable arbitrary intervals, we decided to use a synthetic data set instead because it illustrates the fundamental tradeoffs between the index structures.

We generate a workload of subscriptions and queries based on a d-dimensional Zipfian data generator [7] described in the literature. The data generator, which we shall describe shortly, is used to generate a distribution of points. We then generate the query workload by sampling $n_{queries}$ points from this distribution. The subscriptions are generated as follows. We draw a point x_{mid} from the data generator. We then independently pick a length ℓ_i for each dimension from a Zipfian distribution of skew controlled by $skew_{length}$. The resulting subscription is the hyper-rectangle $\mathbf{I} = (I_1, \ldots, I_d)$, where $I_i = [x_{mid} - \ell_i, x_{mid} + \ell_i]$.

Recall that the R-tree and, hence, the *SOPT-R*-tree indices are very efficient when there is a large amount of overlap amongst the subscriptions. On the contrary, interval trees should perform better when there is lesser interval overlap as they allow for a finer partitioning of the intervals. The Zipfian parameter $skew_{length}$ controls the overlap between intervals and, hence, is a crucial parameter that we can vary to study the performance of our index structures across a variety of scenarios. We believe that subscription scores are typically correlated with their selectivities. So, we set the score of the subscription in each interval as $f_i(I_i) = 1 - \ell_i$.

The Zipfian Data Generator: The Zipfian data generator [7] assumes that each dimension is discrete and finite. We simulate this by dividing the $[0, 1)$ interval of real numbers into c_i equal parts and numbering them 0 through $c_i - 1$. The data generator randomly selects $n_{regions}$ hyper-rectangles. The number of points within each region is bound by the parameters v_{min} and v_{max}. The points that are generated are divided across the different regions according to a Zipfian distribution with a parameter $skew_{across}$. Within each region, the points are distributed again based on a Zipfian distribution with a parameter $skew_{within}$ that makes points farther away

Param	Description	Default
k	Top-k Parameter	20
$n_{intervals}$	# Subscriptions	1 million
$n_{queries}$	# Queries	1000
d	# Dimensions	1
b	Tree Branching Factor	50
$skew_{length}$	Length Zipfian Param	0.75
$n_{warm-up}$	# Warmup Queries	100
Zipfian Data Generator Parameters		
$n_{regions}$	# Regions	10
c_i	# Points in dimension i	100
v_{min}, v_{max}	Volume (min,max) per region	$100^d/20$
$skew_{across}$	Zipfian Parameter to partition points across regions	0
$skew_{within}$	Zipfian Parameter for points distribution within a region	1

Table 1: Parameters for Synthetic Data

from the mid point of the region more unlikely. Table 1 describes the defaults values for the various parameters.

The Zipfian data generator returns a discrete point in d dimensions. We need to convert this into a point in our original domain $[0,1)^d$.Recall that a point $(p_1, \ldots, p_d)$ corresponds to the hyper-rectangle $R = (R_1, \ldots, R_d)$, where $R_i = \left[\frac{p_i}{c_i}, \frac{p_i+1}{c_i}\right)$. We return a point chosen uniformly at random from R.

We are now ready to describe our experimental results. In every experiment, we first generate a subscription workload and build all the indices. The build time for all the indices was very fast — under a minute per dimension even when we had a million subscriptions. We then perform $n_{warm-up}$ queries on the index and measure its performance on then next $n_{queries}$ queries. We implemented these algorithms in Java and performance measurements were made on a dual core machine with 4 GB RAM.

	100,000	200,000	500,000	1,000,000
Interval	0.92M	1.7M	4.2M	8.3M
R	1.2M	2.4M	6M	12M
Segment	40M	89M	214M	429M
IR	1.76M	3.43M	8.35M	16.5M

Table 2: Memory usage of interval indexes (in MB)

Experiment 1 *Space Complexity of Data Structures*: We measure the memory usage of an index structure by measuring the amount of heap space used by the implementation just before and just after constructing the index. Heap space measurements could be misleading since old objects might not have been garbage collected. However, such effects are eliminated by averaging over a large number of observation. Table 2 shows the memory usage of the index structures per dimension as we vary the number of index intervals. We did not plot the *SOPT-R*-tree since it is essentially an R-tree. As shown, the Segment Tree requires more than an order of magnitude more space than the other structures, and quickly becomes impractical when the number of intervals increases (note that these are space requirements *per dimension*). Regarding the other structures, the *IR*-tree takes up more space than the interval and *R*-tree indexes, but it is at most twice as large as the interval tree. This validates our claim that the *IR*-tree index and the *SOPT-R*-tree index have a low space overhead.

Experiment 2 *Varying the number of* `getnext()` *calls*: The

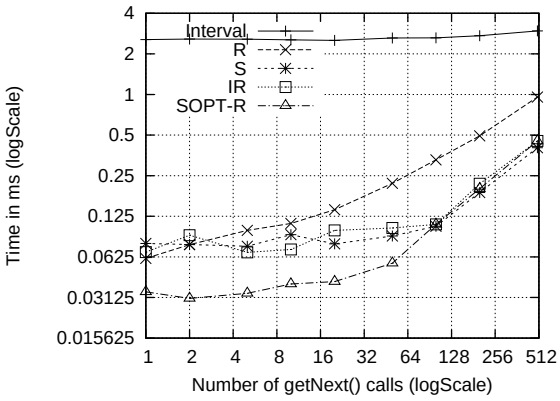

Figure 6: Varying the number of `getnext()` calls

speed of our index structures depends on the time taken for every `getnext()`. Figure 6 shows the total time taken as we vary the number of `getnext()` calls (this is equivalent to varying k in 1 dimension). There are several interesting aspects to note from the graph. First, as expected, the Interval Tree is the least efficient because of the high initial processing cost to find the right interval range (since the lists are in score order). Second, the performance of the Scored R-Tree degrades as the number of `getnext()` calls increase because they encounter many "holes" in their depth-first traversal. Third, the performance of the Segment Tree and the *IR*-tree are about the same, even though the *IR*-tree consumes at least an order of magnitude less space. Finally, and perhaps most interestingly, the *SOPT-R*-tree is always the most efficient and provides up to a factor of 2 speed-up over the other approaches. Note, however, that the difference in performance between the different index structures decreases with the number of `getnext()` calls; this is because almost all the intervals are retrieved when the number of calls is large, and the index structures are roughly similar in this case.

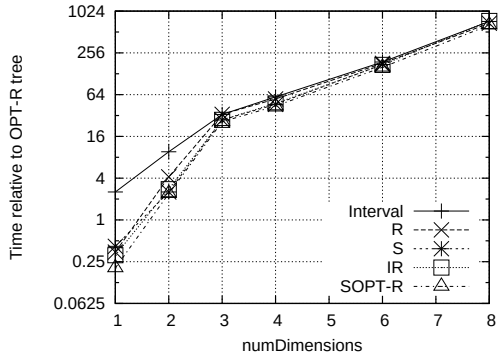

Figure 7: Varying Number of Dimensions

Experiment 3 *Varying the number of dimensions*: Figure 7 shows the performance of a query when varying the number of dimensions, and Figure 8 shows the performance of the different indices relative to *SOPT-R*-treewhen varying the number of dimensions (note the log scale on both the y axes). The first striking aspect is that as the dimensionality increases, the performance of all the indices converges. This is because, at higher dimensions, the number of `getnext()` operations needed to retrieve the top 20 becomes very large and hence almost all the stabbed intervals have to be re-

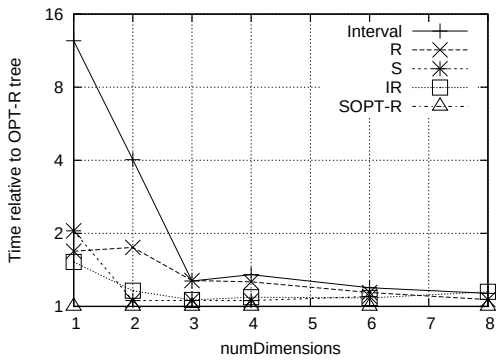

Figure 8: Varying Number of Dimensions

trieved. Since all the index structures are roughly equally efficient in returning the set of all matches, their performance converges (this is one manifestation of the dimensionality curse). However, for a reasonable number of dimensions (1, 2, 3, 4), the *SOPT-R*-tree still offers significant benefits ranging from 16x to 1.5x over other approaches. Further, the *IR*-tree offers roughly similar benefits with the added benefit of supporting incremental subscription insertions and deletions. Finally, while the Segment Tree has a good response time, it ran out of memory at 8 dimensions, showing that it is not well suited for for in-memory implementations.

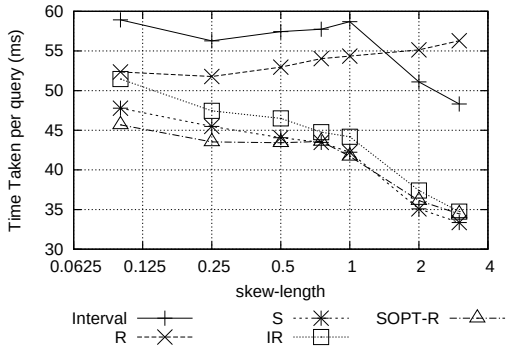

Figure 9: Varying subscription overlap

Experiment 4 *Varying subscription overlap*: Figure 9 shows the effect of varying subscription overlap (a lower value for $skew_{length}$ means larger hyper-rectangles and hence more overlap amongst the subscriptions). We fix the number of dimensions to 4. The most interesting take-away from this graph is that while all the interval indexes become faster as $skew_{length}$ increases, (scored) R-trees degrade in performance. Recall from Section 2.2.3 that Scored R-trees group intervals based on their score rather than their spatial intent. Larger values of $skew_{length}$ cause "holes" in the R-tree structure. Performance degrades as the [left-first] depth-first traversal visits many leaf nodes which do not contain any stabbed interval. The same problem does not affect the *SOPT-R*-tree since non-overlapping intervals are intelligently rearranged to avoid gaps. The rest of the index structures improve in performance because it is easier to partition a space with large gaps.

In summary, the *SOPT-R*-tree has the best performance across the board for a reasonable number of dimensions, while the *IR*-tree comes a close second.

5. RELATED WORK

To the best of our knowledge, this is the first piece of work to address the ranked publish/subscribe problem; still, there are several related problems that have been studied in the research literature. As mentioned earlier, the bulk of conventional pub/sub engines and indexing tools (e.g., [3, 6, 13, 15, 25]) are based on the concept of *strict binary matches* (i.e., find all subscriptions matching an event), and do not incorporate notions of subscription ranking. Of course, our ranked pub/sub problem can be naively solved by first finding the set of *all matching subscriptions* (e.g. using a system like *LeSubscribe* [15] or *Gryphon* [3]), and then ranking these matches in a post-processing step. This can be a very wasteful, time-consuming approach, especially for events that are not very "selective" (e.g., vague job seeker profiles — common in practice), and end up matching a very large number of standing subscriptions. In a sense, the arguments here are very similar to those for optimizing top-k query evaluation in relational systems [20].

More recent work on *symmetric pub/sub* [27] considers a setting where both events and subscriptions are specified as constraints (or, ranges for numeric attributes), and shows how they can be expressed as PostgreSQL queries using its geometric `overlap` and `contains` predicates. Subscription ranking is not considered in their study. Extending our fast interval indexing schemes to the symmetric pub/sub setting is an interesting problem. Liu and Jacobsen [22] propose an interesting extension to traditional pub/sub models by allowing for *fuzzy matchings* of events to subscriptions, using ideas from fuzzy logic and possibility theory. Still, they also do not consider subscription ranking; instead, their schemes rely on using a *user-defined threshold* for the fuzzy-match score and returning all matches above that threshold. Investigating the applicability of these ideas for *ranked* retrieval under such fuzzy-matching score functions opens up a very interesting area for future work.

In traditional *Nearest-Neighbor (NN) search* (e.g., [9, 26]), the goal is to find the top few points that are close to other *points*. In contrast, in ranked pub/sub, we are interested in finding points close to *intervals*. This seemingly simple distinction requires a fundamental rethinking of index structures based on intervals, which is the main focus of this paper. Some variants of NN search do return regions that are close to a query point [24], but the definition of "close" is hard-coded based on geometric distances, and the techniques do not generalize to the ranked pub/sub scenario where each dimension of a subscription can have its own weighted score. *Rank-sensitive B-tree structures* [5] aim to efficiently rank the result points of interval range queries by maintaining ranked lists of data points in the *B*-tree nodes; similarly, in *preference queries* [2, 8], the goal is to find the best data points that match a given user query. In contrast, ranked pub/sub considers the inverse problem, where the goal is to find the best queries (subscriptions) that match a given data point (event). Again, this fundamental distinction of indexing intervals as opposed to points requires the development of new index structures. Work on *bounded continuous queries* [19] considers the problem of selecting the top-k *events* for a given subscription over a specified time period. The ranked pub/sub problem, on the other hand, considers the problem of selecting the top-k *subscriptions* that match a given event.

There are other data structures that are related to our problem. Aggregate R-trees (like Ra* trees [16]), that store aggregates (e.g., the maximum score) at each internal node can be adapted to solve the ranked pub/sub problem if the top-k parameter is known up front. *Partially-persistent data structures* (such as, multi-version B-trees [4]) capture the evolution of a data structure (e.g., a B-tree index) over time, and allow for query points to address any version of the data structure in time. This work obviously has some

strong connections to our ranked pub/sub indexing problem: subscriptions can be viewed as temporal intervals in a multi-version index and stabbing events essentially query a particular version of that index in time. At the same time, compared to, say, the optimal multi-version B-trees of [4], our approach exhibits some key benefits for ranked pub/sub indexing. First, our winning index strategies offer *much stronger space-efficiency guarantees*. For instance, in the space analysis of [4], the constants hidden in the $O()$ factors can actually introduce up to a factor 11.5 blowup in the space requirements of the basic B-tree structure. (This is mainly due to splitting intervals into multiple segments based on their end points (i.e., version changes).) In contrast, the space blowup of our *SOPT-R*-tree is always upper bounded by a factor of $(1 + \frac{2}{b-1})$ (Theorem 3.2), which is typically less that 10% (for realistic values of b). Second, unlike *IR*-trees, multi-version B-trees do not allow for incremental updates to the subscription set, since temporal intervals (i.e., versions) can only be inserted in increasing order of their left endpoints. We should, of course, note that it might be possible to extend/adapt ideas from partially-persistent data structures to provide effective solutions specifically targeted to our ranked pub/sub setting — exploring such adaptations and comparing them to the techniques presented here is an interesting avenue for future work.

6. CONCLUSIONS AND FUTURE WORK

We have introduced the new problem of ranked pub/sub systems, developing indexing solutions for the case where events are points in a n-dimensional space, and subscriptions are intervals in that space. The index structures — *IR*-tree and *SOPT-R*-tree— are compact and efficient, and scale well for reasonable values of n. We believe that this work is only the first step towards building truly flexible and sophisticated ranked pub/sub systems, and that addressing the following open issues will lead us closer to that goal:

More expressive subscriptions: Many applications such as content-based filtering [14, 25] have subscriptions that are specified as paths over hierarchical XML documents. Further, such applications also mix structural and content filters. Supporting such subscriptions would require the development of new scored subscription indexing techniques that go beyond those for indexing intervals.

More expressive events: In applications such as online job sites, the *events* themselves could be intervals, e.g., a job seeker might be interested in jobs that require 20-30 hour work weeks. A related issue occurs in online advertising where some user behavior is inferred and is hence uncertain, e.g., we might be infer an approximate probability of a user being interested in sports, but his estimate may have an error bound. Modeling such events with intervals and uncertainty again requires a rethinking of scored interval indices.

Score updates: Score updates can be very useful in applications such as online advertising, where the priority of a line can depend on how far it is from the delivery goal of, say 10,000 million impressions a day, and can thus change after just a few ads are served in a space of a few minutes. While some of our proposed index structures can support incremental addition/deletion of subscriptions, they do not support score updates, which would require the development of new techniques.

Scaling to high dimensions: With the advent of behavioral targeting in online advertising [1, 11, 28], there can be hundreds of dimensions associated with an user (e.g., propensity for sports, propensity for shopping, etc.). Scaling to such a large number of dimensions requires new techniques that go beyond our current solution of 1-dimensional indices using the Threshold Algorithm. Possible solutions to this problem include the development of scored multi-dimensional indices and dimensionality reduction techniques.

Acknowledgements. We would like to thank Pat O'Neil, Jun Yang, and Pankaj Agarwal for several helpful comments and suggestions.

7. REFERENCES

[1] AOL Audience Targeting. www.aolmedianetworks.com/index.php?id=1936
[2] R. Agrawal, E. Wimmers. A Framework for Expressing and Combining Preferences. SIGMOD 2000.
[3] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. ICDCS 1999.
[4] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer. An Asymptotically Optimal Multiversion B-Tree. The VLDB Journal 5(4), 1996.
[5] W. Bialynicka-Birula, R. Grossi. Rank-Sensitive Data Structures. SPIRE 2005.
[6] A. Carzaniga, A. L. Wolf. Forwarding in a Content-Based Network. SIGCOMM 2003.
[7] K. Chakrabarti, M. Garofalakis, R. Rastogi, K. Shim Approximate Query Processing Using Wavelets VLDB 2000.
[8] J. Chomicki. Querying with Intrinsic Preferences. EDBT 2002.
[9] K. L. Clarkson. A Randomized Algorithm for Closest Point Queries. SIAM Journal of Computing 17(4), 1988.
[10] T.H. Cormen, C.E. Leiserson, R.L. Rivest. Introduction to Algorithms. MIT Press, 1990.
[11] DoubleClick Targeting Filters. www2.doubleclick.com/dk/advertisers/brand/filters.htm
[12] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf. Computational Geometry: Algorithms and Applications. Springer-Verlag, Heidelberg, 2000.
[13] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, P. Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. TODS 28(4), 2003.
[14] Y. Diao, S. Rizvi, M. J. Franklin. Towards an Internet-Scale XML Dissemination Service. VLDB 2004.
[15] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe. SIGMOD 2001.
[16] M. Jurgens, H. J. Lenz. The Ra*-tree: An Improved R-tree with Materialized Data for Supporting Range Queries on OLAP-Data. DEXA Workshop, 1998.
[17] R. Fagin, A. Lotem, M. Naor. Optimal Aggregation Algorithms for Middleware. J. Comput. Syst. Sci. 66(4), 2003.
[18] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. SIGMOD 1984.
[19] D. Kukulenz, A. Ntoulas. Answering Bounded Continuous Search Queries in the World Wide Web. WWW 2007.
[20] C. Li, K. C. Chang, I. Ilyas, S. Song. RankSQL: query algebra and optimization for relational top-k queries. SIGMOD 2005.
[21] Z. Liu, S. Parthasarthy, A. Ranganathan, H. Yang. Scalable Event Matching for Overlapping Subscriptions in Pub/Sub Systems. DEBS 2007.
[22] H. Liu, H.A. Jacobsen. Modeling Uncertainties in Publish/Subscribe Systems. ICDE 2003.
[23] F. P. Preparata, M. I. Shamos. Computational Geometry: An Introduction. Springer-Verlag, 1985.
[24] N. Roussopoulos, S. Kelley, F. Vincent. Nearest Neighbor Queries. SIGMOD 1995.
[25] A. C. Snoeren, K. Conley, D. K. Gifford. Mesh-Based Content Routing using XML. SOSP 2001.
[26] R. L. Sproull. Refinements to Nearest-Neighbor Searching in k-Dimensional Trees. Algorithmica 6, 1987.
[27] A. Tomasic, C. Garrod, K. Popendorf. Symmetric Publish/Subscribe via Constraint Publication. ExpDB 2006.
[28] Yahoo! Advertising Targeting Options. advertising.yahoo.com/central/marketing/targeting.html