Range Aggregation with Set Selection

Yufei Tao, Cheng Sheng, Chin-Wan Chung, Jong-Ryul Lee

Abstract—In the classic *range aggregation* problem, we have a set S of objects such that, given an interval I, a query counts how many objects of S are covered by I. Besides COUNT, the problem can also be defined with other aggregate functions, e.g., SUM, MIN, MAX and AVERAGE.

This paper studies a novel variant of range aggregation, where an object can belong to multiple sets. A query (at runtime) picks any two sets, and aggregates on their intersection. More formally, let $S_1, ..., S_m$ be m sets of objects. Given distinct set ids i, j and an interval I, a query reports how many objects in $S_i \cap S_j$ are covered by I. We call this problem *range aggregation with set selection* (*RASS*). Its hardness lies in that the pair (i, j) can have $\binom{m}{2}$ choices, rendering effective indexing a non-trivial task. The RASS problem can also be defined with other aggregate functions, and generalized so that a query chooses more than 2 sets.

We develop a system called *RASS* to power this type of queries. Our system has excellent efficiency in *both* theory and practice. Theoretically, it consumes linear space, and achieves nearly-optimal query time. Practically, it outperforms existing solutions on real datasets by a factor up to *an order of magnitude*. The paper also features a rigorous theoretical analysis on the hardness of the RASS problem, which reveals invaluable insight into its characteristics.

Index Terms—Range Aggregation, Index, Theory.

1 INTRODUCTION

Range aggregation, such as *"find the number of employees between 30 and 40 years old"*, can be very well supported by modern database systems. Queries like the previous one have two common properties. First, they impose an interval I on a certain attribute. Second, all entities are potential candidates to qualify for the condition of I.

In this paper, we study a variant of range aggregation by relaxing the second property. Let us consider the following query about *Facebook*

> Q : *Find the number of male Facebook users in California that are married, and are aged between 30 and 40.*

Let S_{male}, S_{CA}, and S_{married} be the sets of users that are male, reside in California, and are married, respectively. Then, Q is equivalent to:

> *How many users in $S_{male} \cap S_{CA} \cap S_{married}$ are aged between 30 and 40?*

Compared to traditional range aggregation, Q has a key difference: not all objects (i.e., users) are candidates to qualify for the age condition; *only those in the set intersection* are. Many useful queries follow the same pattern, e.g., how many users in $S_{\text{female}} \cap S_{\text{NJ}} \cap S_{\text{single}}$ are aged between 20 and 30?

Queries like Q report demographic statistics about Facebook, and are important for several reasons. First, such statistics provide valuable information into how successful a social-network site has grown (apparently, similar queries can also be issued on *Foursquare*, *Twitter*, *LinkedIn*, etc.), and indeed have long been the subject of active discussion in the Internet[1]. Second, those statistics play a significant role in marketing, because they allow an advertiser to understand the multitude of the customers targeted. For this purpose, it is paramount to return the answer efficiently *regardless of* the sets whose intersection is concerned. The number of possible sets is gigantic, e.g., the set of users having a particular occupation, or education level, marital status, religion, hobby... Third, the statistics even provide the ground truth for many types of studies on social behavior. An example is association rule mining, which aims at answering questions like: *at Facebook, of the male and married users aged between 30 and 40, what is the most common education level?*

Range aggregation on flexible set intersection is no easy task. To explain, let us provide a quick problem definition (a full-fledged version will appear in the next section). Let $S_1, ..., S_m$ be m sets of objects. Note that these sets are *not* disjoint, that is, an object can appear in multiple sets (e.g., a user can belong to S_{male}, S_{CA}, and S_{married} at the same time). Each object has an *index value* (e.g., age). Given a pair of distinct set ids (i, j) and an interval I, a query reports how many objects of $S_i \cap S_j$ have index values covered by I. We refer to this problem as *range aggregation with set selection* (*RASS*). The value of m can be very large. As mentioned earlier, in the Facebook example, it equals the total number of distinct values in all the categories, (e.g., gender, state of residence, occupation, education level, marital status, etc.). Extending the definition straightforwardly, a query can specify $d \geq 2$ set ids in general.

- *Yufei Tao is with the Chinese University of Hong Kong, Hong Kong. E-mail: taoyf@cse.cuhk.edu.hk.*
- *Cheng Sheng is with Google Switzerland. E-mail: jeru.sheng@gmail.com.*
- *Chin-Wan Chung is with Korea Advanced Institute of Science and Technology, Republic of Korea. E-mail: chungcw@cs.kaist.ac.kr.*
- *Jong-Ryul Lee is with Korea Advanced Institute of Science and Technology, Republic of Korea. E-mail: jrlee@islab.kaist.ac.kr.*

1. Querying a search engine with the keywords "*facebook demographic statistics*" will return a long list of websites discussing a great variety of observations/studies based on these statistics.

The challenge of powering RASS queries is to examine *the fewest objects qualifying a query's predicate*. Put differently, we aim at far better efficiency than enumerating all the qualifying objects. Intuitively, retrieving all such objects is a serious overkill when the query result is merely a single value. Even for the lowest d, however, there are already $\binom{m}{2}$ pairs of sets that can be chosen by the query, rendering it unrealistic to index each pair separately – each object would need to be duplicated a large number of times (as many as $\binom{m}{2}$). Another natural approach to attack the problem is to treat each of the $\binom{m}{2}$ set pairs as a dimension, and then, deploy a multidimensional aggregate index (e.g., the aR-tree [12]) to manage the point set converted from the original data. Unfortunately, even for moderately large m (say, 1000), $\binom{m}{2}$ is already an exceedingly high dimensionality, where multidimensional indexing is known to be notoriously inefficient. It is worth noting that the popular locality sensitive hashing [7] is not suitable here because it is designed for nearest neighbor search, as opposed to range aggregation.

The RASS problem is well defined with many aggregate functions. We have used COUNT as an example, while SUM, AVERAGE, MAX, MIN (and so on) could have been deployed as well. A good solution to the RASS problem should work for all these functions. Furthermore, besides social networks, RASS queries are useful in a great variety of contexts, e.g., "*find the minimum price of 5-star hotels with free parking and a gym whose distances to the beach are at most 10 miles*", "*report the average salary of male, Professor, CA tax payers that are between 30 and 50 years old*", "*return the most profitable action movie produced by an American company during 1990 and 2012*", and so on. In each query, every underlined keyword implies a set of objects.

In spite of its vast importance in reality and fundamental nature in database systems, surprisingly, the RASS problem has not been studied before to our knowledge. Consequently, statistics extraction must currently rely on ill-fitted mechanisms adapted from conventional aggregation methods. Motivated by this, we present the first work to address the RASS problem. Our contributions can be summarized as follows:

- We develop a system called *RASS* to provide powerful support to this new type of aggregate retrieval. The technical core of *RASS* is a novel mechanism combining binary search trees and a multidimensional array. The mechanism is amenable to simple implementation, such that *RASS* can be easily deployed as an extensional plug-in to existing database systems. We believe that the mechanism is of independent interest, and may be used to attack other related problems.
- We prove rigorously that *RASS* has attractive theoretical guarantees. It consumes space linear to the dataset size, and answers any query efficiently even in the *worst case*. This feature is especially important in environments where it is crucial to impose a *hard* bound on the maximum response time. Such a bound is impossible for solutions with poor worst-case performance because their running time must be inevitably long when given a difficult input.
- We also investigate the hardness of the RASS problem. Specifically, we present a sophisticated and insightful analysis that establishes the best query time possible when only linear space consumption is permitted (as usual, a scalable solution needs to use only linear space). The analysis significantly promotes our understanding of the RASS problem, and paves a solid foundation for this topic. Furthermore, our hardness result shows that the proposed *RASS* system has already achieved the *optimal* performance, up to only a very small factor.
- We demonstrate with extensive experiments that, on practical datasets, *RASS* is faster than existing solutions by a factor up to *an order of magnitude* in query time.

The rest of the paper is organized as follows. Section 2 defines the RASS problem formally. Section 3 describes the proposed system, and establishes its theoretical guarantees. Section 4 discusses the hardness of the RASS problem, and reveals the optimality of our solution. Section 5 surveys the previous work related to ours. Section 6 validates the practical efficiency of our techniques with experiments. Finally, Section 7 concludes the paper with a summary of findings.

2 PROBLEM DEFINITION

We now formally define the *range aggregation with set selection* (RASS) problem. We are given m sets of objects: $S_1, ..., S_m$. Each object o is associated with two values in the real domain $\mathbb{R}$:

- an *index value*, denoted as $val(o)$;
- a *weight*, denoted as $weight(o)$.

In a query, index values are examined by a range condition, whereas weights are aggregated according to an *aggregate function* AGG.

*Definition 1: Given distinct set ids i, j and an interval I, a **2-RASS** query reports the result of applying* AGG *on the weights of the objects in $S_i \cap S_j$ whose index values fall in I, or equivalently, the answer is:*

$$\underset{\substack{o \in S_i \cap S_j \\ \text{and } val(o) \in I}}{\text{AGG}} \quad weight(o).$$

We support the entire *distributive* class of aggregate functions. As defined in [9], AGG is distributive if it is *decomposable*. That is, to compute the aggregate value on a set R of weights, we can arbitrarily divide R into disjoint subsets R_1 and R_2, calculate $\text{AGG}(R_1)$ and $\text{AGG}(R_2)$ respectively, and then obtain $\text{AGG}(R)$ from $\text{AGG}(R_1)$ and $\text{AGG}(R_2)$ in constant time. Common functions such as COUNT, SUM, MAX, and MIN are all distributive.

One may wonder at this point about AVERAGE, which is not distributive, despite our claim in Section 1 that we can support it. In fact, the ability of answering COUNT and SUM queries implies that we can deal with AVERAGE at no extra overhead but a single division. In general, aggregate functions such as AVERAGE that can be computed in constant time from distributive functions are classified as *algebraic* in [9]. Our solutions apply to all algebraic functions as well.

Define

$$n = \sum_{i=1}^{m} |S_i|. \quad (1)$$

Trivially, a 2-RASS query can be answered in $O(n)$ time. One may say that an accurate bound ought to be $O(|S_i| + |S_j|)$ for a query that concerns sets S_i and S_j. This is correct, but not very helpful, because $|S_i| + |S_j|$ can go up to $\Omega(n)$ if we are not lucky.

The goal of the RASS problem is to answer every query in time substantially shorter than $O(n)$ – even in the worst case. For practical scalability, a good solution ought to consume only $O(n)$ space, i.e., linear to the dataset size. Besides a nice theoretical guarantee, the solution must perform well in practice too.

We now generalize Definition 1:

*Definition 2: Given d distinct set ids $i_1, ..., i_d$ and an interval I, a **d-RASS** query reports the result of applying* AGG *on the weights of the objects in $S_{i_1} \cap ... \cap S_{i_d}$ whose index values fall in I, or equivalently, the answer is:*

$$\underset{\substack{o \in S_{i_1} \cap ... \cap S_{i_d} \\ \text{and } val(o) \in I}}{\text{AGG}} \quad weight(o).$$

We consider that d is at most a certain constant, that is, we do not allow the query to specify a number of sets that is sensitive to n (e.g., $\log n$).

$d = 1$. Let us first get rid of this uninteresting case. We can easily solve any 1-RASS query in $O(\log n)$ time with a structure of $O(n)$ space. Notice that the sets can be treated independently when a query touches only one of them. Thus, the problem degenerates into traditional range aggregation, whose solution, e.g., the *SB-tree* [19], can be directly applied. Specifically, we can create an SB-tree on each of $S_1, ..., S_m$, respectively. The total space is $O(n)$ because the SB-tree on S_i requires only $O(|S_i|)$ space for each $i \in [1, m]$. Given a query, we perform range aggregation on the SB-tree responsible for the (only) set the query specifies. The query time is $O(\log n)$, thanks to the SB-tree. In the remainder of the paper, we consider $d \geq 2$.

Membership test. Our techniques often need to test whether an object o belongs to a set S_i ($1 \leq i \leq m$). Call this a *membership test*. It is rudimentary knowledge that each membership test can be completed in constant time, by indexing each S_i with a hash table. This hash table occupies $O(|S_i|)$ space and can be constructed in $O(|S_i|)$ expected time. Hence, the hash tables of $S_1, ..., S_m$ altogether occupy $O(n)$ space. Henceforth, we will use it as a fact that each membership test takes constant time.

Remark. In practice (e.g., the Facebook example in Section 1), demographic analysis is typically done on a sample set, as opposed to the entire database (which is not only large in volume but may also be distributed across numerous servers). The memory of today's machines can easily accommodate a sizable sample set, thus enabling fast, I/O-free, statistical exploration. A good system must have the ability of building its access methods from *scratch* quickly, so that when a new sample set is taken (e.g., on a daily basis), all the access methods can be refreshed with minimum cost.

3 THE *RASS* SYSTEM

For simplicity, we will describe the system using COUNT as the aggregate function AGG, because our discussion can be easily extended to any distributive (and hence, algebraic) function. Equivalently, one can regard all object weights as 1, and set AGG to SUM in Definitions 1 and 2. Sections 3.1 through 3.4 will present the access method in the *RASS* system, the query and construction algorithms, and their analysis for $d = 2$. Section 3.5 will discuss general values of d and several extensional issues.

3.1 Structure

Intersection count. Recall that we are given m sets of objects $S_1, ..., S_m$, where each object o has an index value $val(o)$. For each S_i ($1 \leq i \leq m$), we create a complete binary search tree (BST)[2] T_i on the index values of the objects in S_i. In particular, T_i stores all the index values at leaf nodes. Figure 1 shows an example with $m = 3$ sets and 9 distinct objects o_1, ..., o_9. Object o_1, for example, has index value 20 and belongs to S_2 and S_3. The figure also illustrates the BST T_i on each S_i ($i = 1, 2, 3$).

We now introduce a notion *intersection count*. Consider, for example, nodes u_4, u_9 from T_2, T_3, respectively. Denote by $sub(u_4)$ the set of objects in the subtree of u_4. Let $range(u_4)$ be the range of the index values of the objects in $sub(u_4)$, namely, $range(u_4) = [90, 95]$, enclosing the index values of o_8 and o_9. Similarly, $range(u_9) = [60, 95]$. Since one object (i.e., o_9) is in *both* $sub(u_4)$ and $sub(u_9)$, the *intersection count* of (u_4, u_9) is then defined as 1. In general, if u and v are nodes from *different* BSTs, the *intersection count* of (u, v) equals $|sub(u) \cap sub(v)|$.

Materialization of intersection counts is beneficial to query processing. To explain, imagine that we have stored the intersection count 1 of (u_4, u_9). Consider a 2-RASS query that designates S_2, S_3, and an interval I. As long as $range(u_4) \cap range(u_9) = [90, 95]$ is covered by I, we can assert that only a single object in $sub(u_4) \cap sub(u_9)$ contributes to the query answer, *without* traversing these subtrees. Ideally, we would like to store the intersection counts of all pairs of nodes in the BSTs. Unfortunately, there can be $O(n^2)$ such pairs, such that the storage of all intersection counts far exceeds our linear space budget. As we can afford to materialize only $O(n)$ intersection counts, the question is thus to do so for *which* of them.

To answer the question, we need to understand if an intersection count has *not* been pre-computed, how to obtain it *on the fly* during a query. For this purpose, let us re-examine the previous scenario, where the query picks S_2, S_3, and

2. In a complete BST, all but the last level are full, and the nodes at the last level are as far left as possible.

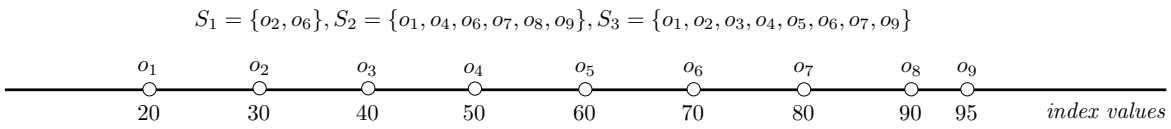


Fig. 1. Example dataset ($m = 3$)

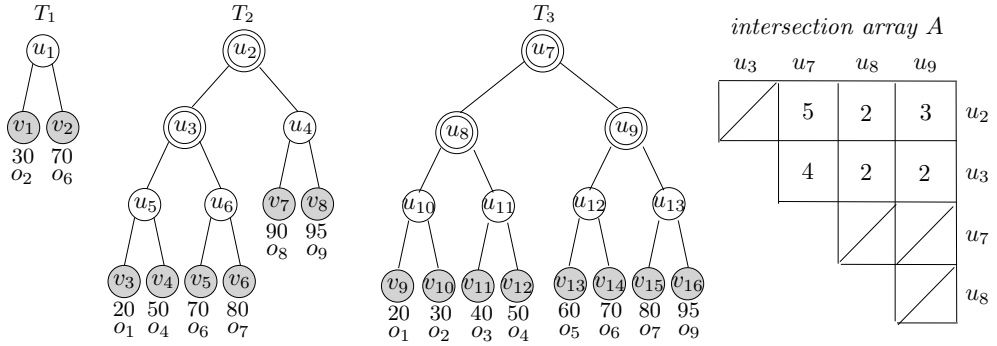


Fig. 2. The RASS-index for the example dataset ($\tau = \sqrt{n} = 4$; double circled nodes are big, while the other nodes are small.)

$range(u_4) \cap range(u_9)$ is covered by I. This time, consider that the intersection count 1 is not available, and needs to be calculated. Since $range(u_9) = [60, 95]$, we can retrieve all the objects in $sub(u_4)$ that fall in $[60, 95]$. The retrieval fetches o_8, o_9 by simply traversing the entire $sub(u_4)$. Then, for each of o_8 and o_9, perform a membership test to see whether it belongs to S_3.[3] After this, we can confirm that only o_9 contributes to the query result, i.e., the intersection count of (u_4, u_9) is 1. The total cost is $O(|sub(u_4)|)$, i.e., the time of traversing $sub(u_4)$ and all the membership tests.

In general, let u and v be nodes from two different BSTs. If an intersection count is *unavailable* for (u, v), we can obtain it in $O(\min\{|sub(u)|, |sub(v)|\})$ time during a query. Therefore, if *either* $sub(u)$ *or* $sub(v)$ has a small size, it is not worthwhile to store an intersection count for (u, v), because on-the-fly computation is cheap. Below, we develop this rationale into a concrete index.

RASS index. As mentioned earlier, there is a BST T_i on each S_i ($1 \leq i \leq m$). As before, if a node u belongs to T_i, $sub(u)$ is the set of objects in the subtree of u. Now, collect the nodes of all BSTs $T_1, ..., T_m$ into a global set U (i.e., U includes the nodes of all different trees). We classify the nodes of U into two categories:

*Definition 3: A node $u \in U$ is **big** if $|sub(u)|$ is at least τ where*

$$\tau = \sqrt{n}. \tag{2}$$

*Otherwise, u is **small**.*

In Figure 2 where $n = 16$ (Equation 1), we have $\tau = 4$. Among all the nodes of T_1, T_2, T_3, only 5 are big: u_2, u_3, u_7, u_8, u_9. They are depicted with double circles.

We create an *intersection array* A as follows. Let (u, v) be a pair of nodes satisfying:

- $u \neq v$, and they are both big.
- u and v are not in the same BST.

Store in $A(u, v)$ the intersection count of (u, v). The intersection array A for our example dataset is given in Figure 2. $A(u_3, u_8)$, for example, equals 2 because objects o_1, o_4 are in the subtrees of both u_3 and u_8. The array has no value for, say, (u_2, u_3) because these two nodes belong to the same BST. Also note that the array is symmetric, which is why only the upper half is stored.

3.2 Query

Preliminary: Canonical partition. Let us first review a basic property of BST. Consider a BST T on a set S of n values. T has height $O(\log n)$, and stores all values of S at the leaves. Given an interval I, denote by $I \cap S$ the set of values in S covered by I. We want to find a small set C of nodes to satisfy all the following:

- for any nodes $u \neq v$ in C, their subtrees $sub(u)$ and $sub(v)$ have no overlap. Equivalently, neither u nor v is an ancestor of the other.
- for each node $u \in C$, the entire $sub(u)$ belongs to $I \cap S$.
- every value in $I \cap S$ is in the subtree of exactly one node in C.

C always exists – naively, simply collect all the leaves corresponding to the values in $I \cap S$, but such a C can be too large for a long I. In fact, for any I, a C of size $O(\log n)$ can always be found in $O(\log n)$ time (see, e.g., [3] for details). As an example, consider T_3 in Figure 2. For $I = [25, 85]$, $C = \{v_{10}, u_{11}, u_{12}, v_{15}\}$ is what we are looking for.

We refer to a C of size $O(\log n)$ as a *canonical node set* of I. The subtrees of the nodes in C form a *canonical partition* of $I \cap S$.

Answering a RASS query. Our attention now goes back to the RASS problem. Given a query which specifies an interval I and set ids i, j ($1 \leq i \neq j \leq m$), we answer it using BSTs T_i, T_j, as well as the intersection array A. First, identify in $O(\log n)$ time the canonical node sets C_i, C_j of I in T_i, T_j,

3. Even though o_8, o_9 must appear in S_2 (since u_2 is a node in T_2), we do not know whether they are in S_3 yet.

respectively. After initializing a temporary result $r = 0$, we obtain the intersection count of each pair of nodes $(u, v) \in C_i \times C_j$:

- if u and v are both big, $A(u, v)$ stores the intersection count directly.
- otherwise, assume $|sub(u)| \leq |sub(v)|$ without loss of generality. We calculate the intersection count by traversing $sub(u)$, as explained in Section 3.1.

Now, increase r by the intersection count of (u, v). The final r at the end of the algorithm is returned as the query answer. No double counting can occur because, in processing (u, v), r changes due to the objects in $sub(u) \cap sub(v)$. The fact that C_i and C_j are canonical, ensures that no object can contribute to the query answer twice.

As an example, suppose that a query designates S_2, S_3, and $I = (-\infty, 92]$ on the structure of Figure 2. The canonical node sets are: $C_2 = \{u_3, v_7\}$ and $C_3 = \{u_8, u_{12}, v_{15}\}$. Hence, the algorithm inspects 6 pairs of nodes in $C_2 \times C_3$. We will discuss only two representative pairs. The first one is (u_3, u_8). Since both nodes are big, r is increased by the intersection count 2 of (u_3, u_8), obtained directly from A. The second one is (u_3, u_{12}). A has no entry for this pair because u_{12} is small. Since $|sub(u_{12})| < |sub(u_3)|$, we traverse $sub(u_{12})$ to find objects o_5, o_6. As only o_6 is a member of S_2 and falls in $range(u_3) = [20, 80]$, the intersection count of (u_3, u_{12}) is 1, which is thus added to r.

Traversing a subtree only once. The above algorithm has an undesired feature: we may need to traverse the subtree of a small node several times. Specifically, let $u \in C_i$ be a small node. It is possible that $sub(u)$ is traversed in processing (u, v) for every $v \in C_j$. As $|C_j| = O(\log n)$, all the traversals of $sub(u)$ would take up $O(|sub(u)| \log n)$ time, instead of $O(|sub(u)|)$. Next, we remedy this drawback, so that $sub(u)$ needs to be traversed only once.

For convenience, list out the nodes in C_j as $v_1, ..., v_t$ for some $t = O(\log n)$. Because C_j is canonical, the subtrees of $v_1, ..., v_t$ are disjoint, and thus, so are their ranges: $range(v_1), ..., range(v_t)$. Hence, these ranges can be sorted in ascending order; without loss of generality, let the order be just $range(v_1), ..., range(v_t)$.

Consider v_1. Recall that, processing (u, v_1) means finding the number of objects in $sub(u) \cap sub(v_1)$, or equivalently, $|range(u) \cap range(v_1)|$. We enumerate the objects in $sub(u)$ in ascending order of their index values, and stop as soon as the current object falls out of $range(v_1)$. So far we have obtained $|range(u) \cap range(v_1)|$. We then turn to process (u, v_2). Since $range(v_2)$ is strictly behind $range(v_1)$, it is unnecessary to re-visit any of the objects already enumerated. Instead, we continue from where we stopped, and repeat the above until we are done processing (u, v_t). In this way, $sub(u)$ has been traversed only once. The total time of processing $(u, v_1), ..., (u, v_t)$ is $O(|sub(u)|)$.

3.3 Construction

Next, we explain how to build a RASS index. The construction of $T_1, ..., T_m$ is straightforward. When this is done, we know which nodes are big, and hence, can initialize the intersection array A with all cells set to 0. What remains is to fill up the cells of A.

For this purpose, we examine every object o again. Let S_i and S_j be two different sets that o belongs to. We identify the leaf node u_i (u_j) in T_i (T_j) that corresponds to o. Let P_i (P_j) be the root-to-leaf path from the root of T_i (T_j) to u_i (u_j). For each pair of nodes $(v_i, v_j) \in P_i \times P_j$, check whether v_i and v_j are *both* big. If not, nothing needs to be done; otherwise, we increase cell $A(v_i, v_j)$ by 1. To illustrate, consider o_1 in Figure 2, which belongs to S_2 and S_3. Hence, $P_2 = \{u_2, u_3, u_5, v_3\}$, and $P_3 = \{u_7, u_8, u_{10}, v_9\}$. P_2 has big nodes u_2, u_3, while P_3 has big nodes u_7, u_8. Hence, we increase each of $A(u_2, u_7)$, $A(u_2, u_8)$, $A(u_3, u_7)$ and $A(u_3, u_8)$ by one.

In general, if o belongs to f sets $S_{i_1}, ..., S_{i_f}$, the extension is straightforward. Following the above description, we are now looking at f root-to-leaf paths. For each combination (u, v) where u, v are big nodes on two different paths, increase $A(u, v)$ by 1.

3.4 Analysis

In this subsection, we will prove the worst-case space and query complexities for the RASS index. In a BST, define the *level* of a node as the number of edges from the root to that node (i.e., root at level 0). In a complete BST, the i-th level has 2^i nodes, except possibly the last level (recall that all BSTs in *RASS* are complete). Let us start with a useful fact:

Lemma 1: There are $O(\sqrt{n})$ big nodes.

Proof: We will need the following rudimentary fact: in a complete BST, if u and v are siblings (i.e., they have the same parent), the subtree of u has at most twice as many leaves as in the subtree of v. In other words, if $sub(u)$ is the set of leaves in the subtree of u, then $|sub(u)| \leq 2|sub(v)|$.

To prove the lemma, we will artificially declare some small nodes to be big, i.e., increasing the number of big nodes. If, after the declaration, the number of big nodes is still $O(\sqrt{n})$, then originally the number must also be $O(\sqrt{n})$. The declaration goes as follows. If, by Definition 3, a node u is big but its sibling v is not, we declare v big as well. Since $|sub(u)|$ is at least $\sqrt{n}$, we know $|sub(v)| \geq \sqrt{n}/2$. For example, in Figure 2, we declare u_4 big because of u_3.

Let Z be the set of big nodes u (after declaration) such that no child of u is big. In Figure 2, $Z = \{u_3, u_4, u_8, u_9\}$. Nodes in Z have disjoint subtrees. As mentioned earlier, $sub(u)$ covers at least $\sqrt{n}/2$ leaves. On the other hand, only n leaves are available in all BSTs. We thus know that Z can have at most $n/\frac{\sqrt{n}}{2} = 2\sqrt{n}$ nodes.

In each BST, the big nodes themselves form a binary tree, whose leaves are in Z. Furthermore, each big node $v \notin Z$ has the property that both child nodes of v are big (due to the way we declared). Hence, the number of big nodes outside Z cannot be more than $|Z|$. It follows that the number of big nodes is at most $4\sqrt{n}$. □

Now we can bound the space and query complexities:

Theorem 1: The RASS index uses $O(n)$ space, and answers a 2-RASS query in $O(\sqrt{n})$ time.

Proof: We focus on the query time because the space complexity is obvious (note that by Lemma 1, the intersection array A occupies $O(n)$ space). Recall that our algorithm obtains two canonical node sets C_i and C_j, each with $O(\log n)$ nodes. Overall, the query time is dominated by the cost of (i) reading $O(\log^2 n)$ cells in A, and (ii) traversing the subtree of each small node in C_1 and C_2. Next, we will show that it takes $O(\sqrt{n})$ time to traverse the subtrees of all the small nodes in C_1. The same holds for C_2 due to symmetry, from which the lemma then follows because $O(\log^2 n + \sqrt{n}) = O(\sqrt{n})$.

Let C_{small} be the set of small nodes in C_1, and x be the *highest* level of the nodes in C_{small}. Since a level-x node of a complete binary tree has $O(n/2^x)$ leaves, we know that $n/2^x = O(\sqrt{n})$ because the subtree of a small node has less than $\sqrt{n}$ leaves.

C_{small} has at most two nodes at level i, for each $i = x, x+1, x+2, ...$ A level-i node has $O(n/2^i)$ leaves in its subtree. Hence, the cost of traversing the subtrees of all the nodes in C_{small} is bounded by

$$O\left(n \cdot \left(\frac{1}{2^x} + \frac{1}{2^{x+1}} + \frac{1}{2^{x+2}} + ...\right)\right) = O(n/2^x).$$

which, as mentioned earlier, is $O(\sqrt{n})$. □

3.5 Extensions

Heuristic. We have shown that the RASS index guarantees $O(\sqrt{n})$ search time. For some queries, it is easy to ensure this bound. To explain, consider a query that picks S_i and S_j. If *one* of S_i and S_j has size at most $\sqrt{n}$, say, $|S_i| \leq \sqrt{n}$, we can simply look at each object $o \in S_i$, and perform a membership test to see if it is in S_j. If so, we increase our current answer by 1 (of course, one still needs to worry about the query interval I, but this is trivial). This algorithm works in $O(|S_i|)$ time.

This observation points to a hybrid solution: leverage the RASS index only if both S_i and S_j have sizes at least $\sqrt{n}$. Otherwise, apply the simple algorithm described above. This heuristic offers no help in improving the time complexity in Theorem 1, but it allows us to avoid spending $\Omega(\sqrt{n})$-time whenever possible.

$d = 3$ and up. Our solution can be straightforwardly extended to higher d, as we will clarify next for $d = 3$. The major change is the value of τ (Equation 2), which becomes $\tau = n^{2/3}$. Furthermore, A is now 3-dimensional. Specifically, let u, v, w be nodes from distinct BSTs. Their intersection count $A(u, v, w)$ equals $|sub(u) \cap sub(v) \cap sub(w)|$, i.e., how many common objects $sub(u), sub(v)$ and $sub(w)$ have.

The query algorithm is also modified slightly. First, instead of two, we obtain three canonical sets C_1, C_2, C_3 of the query interval I, each of which is in the BST of a different set designated by the query. For each node combination $(u, v, w) \in C_1 \times C_2 \times C_3$, if u, v, w are all big, the intersection count of (u, v, w) can be found in $A(u, v, w)$ directly. Otherwise, we compute it by first identifying the node (among u, v, w) with the smallest subtree and traversing its subtree once. The current answer r is then increased with the intersection count.

Following the same argument, Lemma 1 now states that there are $O(n^{1/3})$ big nodes (which ensures that A uses linear space), whereas Theorem 1 becomes: space $O(n)$ and query time $O(n^{2/3})$. Finally, for general d, τ should be set to $n^{1-\frac{1}{d}}$. Accordingly, the RASS index consumes $O(n)$ space, and solves a query in $O(n^{1-\frac{1}{d}})$ time.

4 Hardness of the RASS Problem

This section aims at answering the question: can any structure do better than $O(\sqrt{n})$ time to solve the RASS problem for $d = 2$? The significance of the question is that it indicates to what extent the RASS problem remains unexplored. If $O(\sqrt{n})$ is already the end of theoretical improvement, further research down this line should be devoted to heuristics! We will show, unfortunately, that $O(\sqrt{n})$ is indeed nearly the end:

Theorem 2: If a structure of size $O(n)$ works for all distributive aggregate functions, it must incur $\Omega(\sqrt{n}/\log^3 n)$ time in answering a 2-RASS query in the worst case.

In other words, nobody can design a structure with linear space and query time $O(n^{\frac{1}{2}-\epsilon})$, no matter how small the constant $\epsilon > 0$ is, because $n^{\frac{1}{2}-\epsilon} = o(\sqrt{n}/\log^3 n)$ for any ϵ. It thus follows that the RASS index is already optimal up to only a small factor. The proof of Theorem 2 is rather non-trivial, and constitutes the rest of the section.

Infinite-interval queries. We will limit our attention to a special class of 2-RASS queries, namely, those whose search intervals $I = (-\infty, \infty)$. In other words, what matters is that the set ids i, j chosen by the query. The query answer is the result of applying the aggregate function AGG on $S_i \cap S_j$. Clearly, there are exactly $\binom{m}{2}$ such *infinite-interval* queries. We will show at least one of them demands the execution time claimed in Theorem 2.

Behavior of a query algorithm. Let us recall what is a distributive AGG. Let R be a set of objects, and $\{R_1, R_2\}$ a *partition* of R (i.e., $R_1 \cap R_2 = \emptyset$ but $R_1 \cup R_2 = R$), it must hold that

$$\text{AGG}(R) \quad = \quad \text{AGG}(R_1) \oplus \text{AGG}(R_2)$$

where $\oplus$ is the operator used to combine $\text{AGG}(R_1)$ and $\text{AGG}(R_2)$. For example, for SUM, $\oplus$ is addition, while for MAX, $\oplus$ is simply max.

Even though various query algorithms may differ in many details, they share a common pattern in how to obtain the query answer r. At any time during the algorithm, r equals the result of applying the aggregate function AGG to a certain set R of objects. At the beginning, $R = \emptyset$, whereas at the end, R must be equal to $S_i \cap S_j$ so that r can be correctly returned (S_i and S_j are the sets that the query concerns). Each update of r takes place in the following way:

> Given an aggregated value $r' = \text{AGG}(R')$, the algorithm performs $r \leftarrow r \oplus r'$, whose effect is to union R' into R.

If R' *always* contains only a single object, then clearly the algorithm must carry out the update $|S_i \cap S_j|$ times to obtain the final answer. However, a good index should have stored *pre-aggregates*, each of which is a value r' that may have aggregated a set R' of multiple objects. In the luckiest scenario, there may be an R' that equals exactly $S_i \cap S_j$, in which case fetching the pre-aggregate r' completes the algorithm. In more common scenarios, R' is only a proper subset of $S_i \cap S_j$, so that more updates are necessary.

In any case, in an update, R' cannot contain any object, say o, that does not belong to $S_i \cap S_j$. If it does, then r' has included the contribution of $weight(o)$, in which case the update will force r to contain the contribution of $weight(o)$, too. Such an r is *contaminated* because the aggregate function AGG does not necessarily allow a way to eliminate the contribution of $weight(o)$ from r. As a result, the algorithm can no longer ensure the correctness of the final r. This fact is known as the *faithfulness of a semi-group* [4].

The crux of our proof is to argue that if a pre-aggregate r' is on a large set R' of objects, r' cannot be used to answer many queries. On the other hand, we will force each query to aggregate many objects. This, combined with the earlier statement, helps us to show that at least one query must use quite a number of "small" pre-aggregates, each of which is applied to only a few objects. This query, therefore, necessitates long running time. Of course, for the whole argument to work, we need to craft a dataset to suit our needs, as given below.

A hard dataset. The rest of the proof will utilize a dataset carefully designed to make the RASS problem hard. Our dataset has m sets $S_1, ..., S_m$, in which each object is an integer from 1 to D, where the values of m and D will be determined as needed later. Define

$$L \quad = \quad \log_2(mD) \tag{3}$$

The next lemma generalizes a result in [4]:

Lemma 2: For m and D satisfying $m \leq 2^{cD}$ for some constant c, we can find m subsets $S_1, ..., S_m$ of $\{1, ..., D\}$ to satisfy two conditions:

$C1$: *for any distinct $i, j \in [1, m]$, $|S_i \cap S_j| \geq D/8$.*

$C2$: *For any distinct integers $i_1, ..., i_L \in [1, m]$, $|\bigcap_{j=1}^{L} S_{i_j}| \leq L$.*

Proof: Our proof generalizes an argument in [4] that proves a statement more restrictive than ours. We will deploy the *probabilistic method*. This is a classic technique that utilizes probabilistic arguments to prove the *definite* existence of a certain property (see [11] for a systematic introduction). In the current scenario, we want to prove the existence of sets $S_1, ..., S_m$ satisfying both Conditions $C1$ and $C2$.

Let us construct each S_i ($1 \leq i \leq m$) by adding each integer of $\{1, ..., D\}$ with probability $1/2$. We will show that with a *non-zero* probability, the m sets built this way will fulfill both $C1$ and $C2$. The implication is thus that there *must* exist $S_1, ..., S_m$ as demanded by the lemma (otherwise, how would we get a non-zero probability?). This, therefore, will complete the proof.

So for any $1 \leq i \neq j \leq m$, an integer of $\{1, ..., D\}$ is in both S_i and S_j with probability $1/4$. Let $x = |S_i \cap S_j|$. Clearly, x sums up D Bernoulli random variables, each of which has success probability $1/4$. An application of the Chernoff bound[4] gives:

$$Pr[x < D/8] \quad < \quad (0.962)^D.$$

Let ϵ be a positive real value less than $1/2$. When

$$D \quad \geq \quad \frac{\log \frac{1}{1/2-\epsilon} + 2 \log m}{0.055} \tag{4}$$

it holds that: $(0.962)^D \leq (1/2 - \epsilon)/m^2$. As there are less than m^2 pairs of (i, j), by the union bound, Condition $C1$ is satisfied with probability at least $1/2+\epsilon$. Note that Inequality 4 essentially requires $m \leq 2^{cD}$ for some constant c.

Now consider a set of distinct $i_1, ..., i_L \in [1, m]$. Each integer of $\{1, ..., D\}$ simultaneously belongs to *all* of $S_{i_1}, ..., S_{i_L}$ with probability $1/2^L$. Let $y = |\bigcap_{j=1}^{L} S_{i_j}|$. The expectation of y is $D/2^L$. As $L = \log_2(mD) > \log_2 D$, an application of the Chernoff bound gives:

$$Pr[y > L] \quad < \quad \frac{e^L}{(L2^L/D)^L}$$

Using the fact $\binom{m}{L} \leq (em/L)^L$, the right hand side of the above is bounded from above by $(1/2 - \epsilon)/\binom{m}{L}$ when

$$\begin{aligned} e^{2L}(mD)^L \quad &\leq \quad (1/2 - \epsilon)L^{2L}(2^L)^L \\ \Leftrightarrow 2L \log_2 e + L \log_2(mD) \quad &\leq \quad \log_2(1/2 - \epsilon) + \\ &\quad\ 2L \log_2 L + L^2 \\ \Leftrightarrow 2L \log_2 e + L^2 \quad &\leq \quad \log_2(1/2 - \epsilon) + \\ &\quad\ 2L \log_2 L + L^2 \end{aligned}$$

where the last step applied $L = \log_2(mD)$. The final inequality is true as long as mD is greater than a certain constant. As there are $\binom{m}{L}$ sets of $\{i_1, ..., i_L\}$, by the union bound, Condition $C2$ in the lemma holds with probability at least $1/2 + \epsilon$.

In summary, when $m \leq 2^{cD}$ for some constant c, Conditions $C1$ and $C2$ are satisfied simultaneously with probability at least 2ϵ (applying union bound), i.e., a non-zero probability. We thus have concluded the proof. □

Our dataset contains exactly the $S_1, ..., S_m$ in the above lemma. Note that Condition $C1$ implies that each of $S_1, ..., S_m$ must have at least $D/8$ objects. Hence, $n = \sum_{i=1}^{m} |S_i| = \Omega(mD)$.

Tradeoff between space and query time. As mentioned earlier, an index may store pre-aggregates $r' = \text{AGG}(R')$ where R' is a set of objects. We say that r' is *heavy* if $|R'| > L$. The lemma below gives the relationship between the space and query time:

4. Let $x_1, ..., x_m$ be independent Bernoulli random variables, each of which has success probability p. Set $X = \sum_{i=1}^{m} x_i$. The Chernoff bound says that, for any $\delta > 0$, (i) $\mathbf{Pr}[X < (1 - \delta)mp] < \exp(\delta(mp)^2/2)$ and (ii) $\mathbf{Pr}[X > (1 + \delta)mp] < (\exp(\delta)/(1 + \delta)^{1+\delta})^{mp}$.

Lemma 3: Assuming $D > L$, if every query must be answered within α time, the number s of heavy pre-aggregates stored in the structure must satisfy

$$s \geq \frac{(D/8 - \alpha L)m^2}{2(D-L)L^2}.$$

Proof: We first establish a fact about heavy pre-aggregates:

Lemma 4: A heavy pre-aggregate can be used by less than $(L-1)^2$ infinite-interval queries.

Proof: Consider a query that specifies S_i, S_j. If the algorithm uses a heavy pre-aggregate $r' = \text{AGG}(R')$ to answer the query, it implies that R' must belong to $S_i \cap S_j$, namely, R' is a subset of both S_i and S_j. Since $|R'| > L$, by Condition $C2$ of Lemma 2, R' can belong to at most $L-1$ sets among $S_1, ..., S_m$ (otherwise, we have found L sets among them with an intersection of more than L objects, violating $C2$). $L-1$ sets can only define $(L-1)(L-2) < (L-1)^2$ infinite-interval queries. □

Now we resume the proof of Lemma 3. Set $M = \binom{m}{2}$, namely, the number of possible (infinite-interval) queries. Suppose that, in answering the i-th ($1 \leq i \leq M$) query, the algorithm uses x_i heavy pre-aggregates. Hence, at most $\alpha - x_i$ non-heavy pre-aggregates can have been used to answer the query (otherwise, the query time would exceed α). By Condition $C1$ of Lemma 2, each query must aggregate at least $D/8$ objects. On the other hand, by definition, a non-heavy pre-aggregate can aggregate at most $L-1$ objects. So, even if each heavy pre-aggregate is used to aggregate D objects, we still have:

$$L(\alpha - x_i) + Dx_i \quad > \quad D/8.$$

As the above holds for each i, we have

$$\sum_{i=1}^{M} \Big(L(\alpha - x_i) + Dx_i\Big) \quad > \quad DM/8$$

$$\Rightarrow \alpha LM + (D-L)\sum_{i=1}^{M} x_i \quad > \quad DM/8. \qquad (5)$$

By Lemma 4, a heavy pre-aggregate can be used to process less than L^2 queries. As there are s heavy pre-aggregates, we know

$$\sum_{i=1}^{M} x_i \quad < \quad sL^2.$$

With the above, Inequality 5 gives

$$\alpha LM + (D-L)sL^2 \quad > \quad DM/8$$

$$\Rightarrow s \quad > \quad \frac{(D/8 - \alpha L)M}{(D-L)L^2}.$$

The lemma then follows from the fact that $M = m(m-1) \geq m^2/2$ for $m \geq 2$. □

The previous lemma indicates

$$\alpha \quad \geq \quad \frac{m^2 D - 16sL^2(D-L)}{8m^2 L}. \qquad (6)$$

Finally, we fix the parameters s, m, D, L to obtain a tradeoff good enough to prove Theorem 2. As the space consumption must be linear, we let $s = cn$ for some constant $c > 0$. Accordingly, set:

$$\begin{aligned} m &= \sqrt{n}\log_2^2 n \\ D &= \sqrt{n}/\log_2^2 n \\ L &= \log_2 n \end{aligned}$$

Inequality 6 gives (all logarithms have base 2):

$$\begin{aligned} \alpha \quad &\geq \quad \frac{n\sqrt{n}\log^2 n - 16cn\log^2 n\left(\frac{\sqrt{n}}{\log^2 n} - \log n\right)}{8\log n \cdot n\log^4 n} \\ &= \quad \frac{\sqrt{n} - 16c\left(\frac{\sqrt{n}}{\log^2 n} - \log n\right)}{8\log^3 n} > \frac{\sqrt{n}}{9\log^3 n} \end{aligned}$$

when n is large enough. This completes the proof of Theorem 2.

5 Related Work

Range aggregation is a classic topic in the database area, and has been studied in a large variety of contexts: relational [19], temporal [15], [21], spatial databases [8], [12], [14], OLAP [9], [10], [13], etc. Towards enabling this fundamental operation in a new context, the proposed RASS problem permits a query to intersect a number of sets that are *arbitrarily* selected from a large collection of candidate sets. Currently, no system is able to support RASS queries effectively, which has been a serious problem due to their vast importance in practice.

The work of [4], [20] considers a special instance of the RASS problem where objects have no index values, and a query's interval I is thus irrelevant. Hence, the solutions in [4], [20] are inapplicable to RASS queries, whereas all our algorithms and analysis (including the lower bound result) apply to the specialized problem of [4], [20]. It is also worth noting that, the focus of [4], [20] is on the tradeoff between query and update costs, while our emphasis rests on the tradeoff between space and query time.

Range aggregation is always accompanied by range *reporting*. In the reporting version of the RASS problem, the data input is still m sets $S_1, ..., S_m$ of objects, where each object has an index value. Given an interval I and set ids i, j, a query reports all the objects in $S_i \cap S_j$ whose index values are covered by I. A RASS query can be answered by first retrieving all the objects satisfying the corresponding reporting query, and then aggregating those objects. The reporting version of RASS is essentially the one-dimensional variant of *spatial keyword search* [1], [6]. Furthermore, if the goal is to return only $S_i \cap S_j$ (i.e., ignoring objects' index values and the query interval I), the problem degenerates into *set intersection*, which is a well-studied problem with numerous interesting solutions [2], [5], [22].

In general, although range aggregation can be supported by range reporting, performance can be significantly improved by deploying a technique specially designed for aggregation. This is because the specialized technique typically computes the

final answer directly, without fetching the aggregated objects. As shown in the next section, our *RASS* system considerably outperforms alternative solutions adapted from methods of the reporting problem.

Finally, query processing on sets has been actively investigated in the database community. The previous work has addressed numerous problems such as *similarity selection* on a single collection of sets [18], *similarity join* on multiple collections [17], *set containment search* [16], to mention just a few. The solutions under those topics, however, are specific to their own contexts, and cannot be adapted for the problem considered in this paper.

6 EXPERIMENTS

We now proceed to experimentally evaluate the proposed solutions, and compare their efficiency against alternative methods for RASS queries. All our experiments were carried out on a machine running an Intel Core 2 Duo CPU at 3GHz with 8 GB memory. The operating system was 64-bit Linux.

Datasets. We deployed two real datasets:

- *Flickr*. *Flickr* (*www.flickr.com*) is a website where people share photos they have taken. After uploading a picture, a user can associate it with *labels* (e.g., `European`, `landscape`, `summer`, etc.) to describe the picture. Our dataset contains 28.1 million pictures (i.e., objects). They belong to $m = 1000$ sets, where each set includes all the pictures having a specific label (i.e., there are 1000 distinct labels). Each picture has an upload timestamp that serves as its index value. The total size of all the sets equals $n = 48.3$ million. The dataset occupies 1.3 GB.

  A d-RASS query has the semantics of counting the number of pictures that were uploaded during a period of time (i.e., interval I), and carry all the d labels designated by the query. An example with $d = 2$ is: *find the number of pictures uploaded during 1 Jun 2006 and 31 Aug 2006, and having labels* `European` *and* `landscape`.
- *Delicious*. *Delicious* (*delicious.com*) is a website where people publish their Internet bookmarks, and thereby share the online resources they have found. For example, a user can create a bookmark referencing a painting of Leonardo da Vinci in the Internet, and label the bookmark with `art`, `design` and `history`. Our dataset has $n = 5.8$ million bookmarks (i.e., objects). They belong to $m = 1000$ sets, where each set contains all the bookmarks having a specific label (i.e., 1000 distinct labels). Each object has an index value that is the creation timestamp of the corresponding bookmark. The total size of all the sets equals $n = 16.1$ million. The dataset occupies 411 MB.

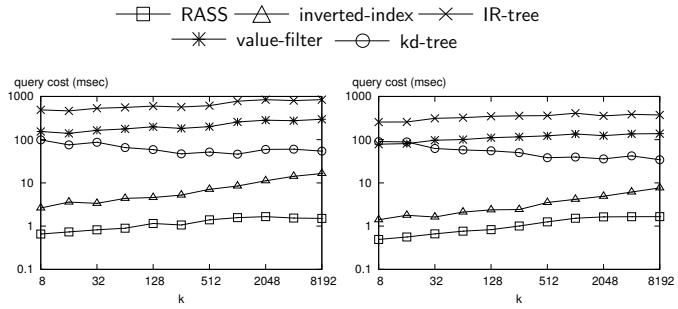
  A d-RASS query has the semantics of counting the number of bookmarks that were created during a period of time, and carry all the d labels designated by the query. An example with $d = 2$ is: *find the number of bookmarks created during 1 Jan 2004 and 31 Dec 2004, and having labels* `art` *and* `design`.

Competitors. We examined the following methods:

- *RASS*: The proposed system.
- *Inverted-index*: This method adapts a querying mechanism commonly found in search engines [22]. We explain how it works for $d = 2$ because the generalization to higher d is straightforward. In preprocessing, the objects of each set S_i ($1 \leq i \leq m$) are sorted by their ids. Given a query designating an interval I and distinct set ids i, j, the query algorithm computes $S_i \cap S_j$ by synchronously scanning S_i and S_j once. An object in $S_i \cap S_j$ contributes 1 to the query answer if its index value falls in I.
- *Value-filter*: This method first obtains (using a binary search tree) all the objects whose index values are covered by the query's interval I. Then, for each object fetched, perform membership tests to check whether it belongs to all the sets specified by the query.
- *IR-tree*: This is a well-known structure [6] for spatial keyword search, which captures the reporting version of the *RASS* problem as a special case (see the previous section). The query answer is then obtained by counting the number of objects reported.
- *kd-tree*: This method is included to show that the RASS problem cannot be efficiently solved using a multidimensional index. Each object o is regarded as an $(m + 1)$-dimensional point p. Specifically, the i-th ($1 \leq i \leq m$) coordinate of p is 1 if $o \in S_i$, or 0 otherwise. The $(m+1)$-th coordinate of p, on the other hand, equals $val(o)$. Furthermore, p carries a weight which equals $weight(o)$. We build a kd-tree on the set of points converted from all the objects. Each internal node of the kd-tree is associated with the total weight of the points in its subtree. We include the optimization that, at each leaf node – suppose that it corresponds to point p – we store only its 1 coordinates so that the overall space is $O(n)$. Give a 2-RASS query with interval I and set ids i, j, we can see that its result is the total weight of the points that fall in an $(m + 1)$-dimensional rectangle ρ: (i) the extent of ρ is $[0, 1]$ for every dimension $k \in [1, m]$ such that $k \neq i$ and $k \neq j$, (ii) the extent of ρ is $[1, 1]$ on dimensions i and j, and (iii) the extent of ρ is I on dimension $m+1$. Finding the total weight (of the points in ρ) is a standard *aggregate range query* [12] that is answered by the kd-tree.

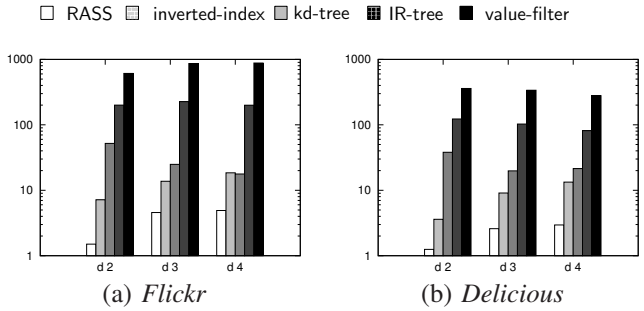
Query workload. In general, an algorithm geared for range aggregation has an advantage over an algorithm for range *reporting* when the query answer is large – after all, if only few objects qualify the query conditions, little could be gained from intersection values, in which case range aggregation would end up fetching all those objects, and thus degenerate into range reporting. A main objective of our experiments is to find out how different approaches work as the query answer changes.

For this purpose, given a target query answer $k > 0$, we generated a *k-workload* of queries as follows. Let us first consider $d = 2$. Given distinct set ids i and j, we say that the pair (i, j) is *k-meaningful* if the dataset has at least k objects in $S_i \cap S_j$. A *k-workload* contains 100 queries, each of which



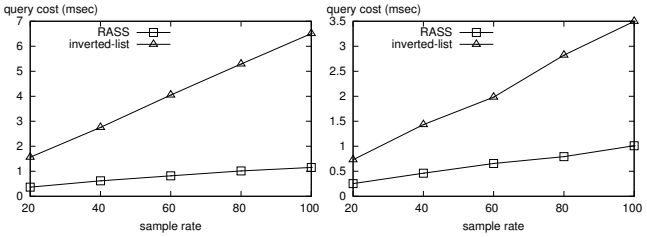
(a) *Flickr* (b) *Delicious*

Fig. 3. Query time vs. k ($d = 2$)



(a) *Flickr* (b) *Delicious*

Fig. 4. Query time vs. d ($k = 512$)



(a) *Flickr* (b) *Delicious*

Fig. 5. Query time vs. dataset size ($k = 512, d = 2$)

is obtained in two steps. First, its set ids i, j are determined such that (i, j) can be any of the k-meaningful pairs with the same probability. Second, having fixed (i, j), the query interval I is a range that makes the number of qualifying objects exactly k. Furthermore, it is ensured that all such ranges have equal likelihood to be selected as I. Interestingly, the above approach has the benefit of *avoiding* queries that are semantically void (e.g., picking S_{male} and S_{female} at the same time) because $k > 0$ objects must be in the query answer. Straightforward adaptation of the above method was used to generate a k-workload of $d > 2$.

We will vary k from 8 to 8192 with $k = 512$ being the default value. The value of d will be varied from 2 to 4, with $d = 2$ as the default. Unless otherwise stated, each parameter is set to its default value in the subsequent presentation. It is worth mentioning that, for $d \geq 5$, almost all queries we examined have exceedingly small answers (namely, few objects exist in 5 or more *common* sets simultaneously). Therefore, range aggregation nearly always degenerated into range reporting, due to reasons explained earlier.

Query efficiency. We started by assessing the query performance of various methods when k doubled from 8 to 8192. For each method, we measured the averaged time it required to answer a query in a k-workload. The results are illustrated in Figures 3a and 3b for datasets *Flickr* and *Delicious*, respectively. Note that the y-axes are in log-scale.

RASS exhibited by far the best efficiency. It outperformed the closest competitor *inverted-index* by a factor up to an order of magnitude, whereas *value-filter* and *IR-tree* were much more expensive. The cost of *RASS* initially grew with k, and then stabilized even when k increased further. To explain this, first notice that a large k typically corresponds to queries with long intervals I. As analyzed in Section 3.4, the query time of *RASS* is sensitive to the number of canonical nodes. If $I = [x, y]$, the canonical nodes in a BST (on a set chosen by the query) are those along two root-to-leaf paths: one towards x, and the other towards y. When I is short, the two paths share many nodes, in which case there are fewer canonical nodes. This explains why the query time was lower for small k. As I becomes longer, the number of canonical nodes increases, until the two paths share no common node (except the root). This was the point when the query time started to stabilize.

The next experiment studied the influence of the number d of sets chosen by the query. Figure 4a (4b) shows the average query time of each method in processing a 512-workload on *Flickr* (*Delicious*), as d increased from 2 to 4. It is evident that *RASS* remained significantly faster than all of its competitors. *RASS* demanded longer time answering a d-RASS query when d was larger. Recall that its query algorithm needs to traverse the subtree of each small canonical node. When d increases, the subtree of a small node has more leaves, because the threshold τ in Definition 3 equals $n^{1-1/d}$ in general, namely, greater for a larger d. This is the reason for the increase of the query cost with d. In fact, this also explains why the increase was (much) more obvious when d went from 2 to 3, compared to when d went from 3 to 4. Specifically, the values of τ for $d = 2, 3, 4$ are $n^{1/2}$, $n^{2/3}$, $n^{3/4}$ respectively. In other words, from $d = 2$ to 3, τ escalated by a factor of $n^{1/6}$, whereas the factor was merely $n^{1/12}$ from $d = 3$ to 4.

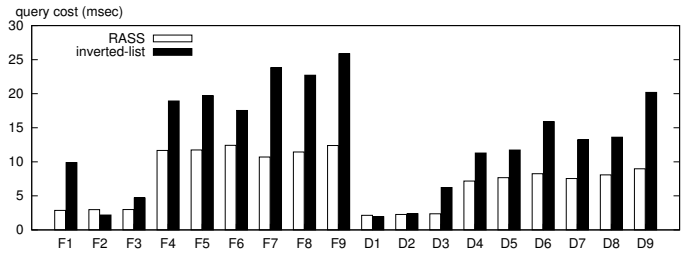
We continued with an experiment to inspect how query cost scales with the dataset cardinality. Towards this purpose, for each of the two datasets, we created 4 *miniatures*, each of which was obtained by sampling (without replacement) a certain percentage p of the records from the underlying dataset. Naturally, we refer to p as the *sample rate*. It is obvious that each miniature has the same data distribution. Therefore, cardinality becomes the chief factor affecting query efficiency. Regarding the miniatures of *Flickr*, Figure 5a plots, as a function of p, the average query time of *RASS* and *inverted-index* in processing a workload. Clearly, a 100% miniature is simply the original dataset. Figure 5b gives the results of the same experiment on *Delicious*. In both figures, we have omitted *IR-tree*, *value-filter*, and *kd-tree* because their performance was far worse, as should have become obvious from the previous experiments. The cost of both *RASS* and *inverted-index* scaled well with the dataset size; *RASS* once again was the clear winner.

	interval I	set ids	d
$F1$	05-10-25,22:38:11 - 05-12-28,9:20:17	london, red	2
$F2$	04-11-19,13:20:09 - 05-12-11,10:16:51	boy, girls	2
$F3$	05-03-11,20:36:17 - 05-12-29,7:48:36	macro, photoshop	2
$F4$	05-01-31,15:35:59 - 05-12-18,21:22:46	dog, vacation, sky	3
$F5$	04-05-23,10:47:46 - 05-12-18,21:19:42	blue, friends, people	3
$F6$	04-09-15,10:22:05 - 05-12-18,21:02:41	art, portrait, cameraphone	3
$F7$	04-05-13,17:07:18 - 05-12-31,11:12:22	vacation, trip, sky, tree	4
$F8$	04-12-06,17:56:09 - 05-12-18,21:16:41	art, sky, usa, architecture	4
$F9$	05-03-16,8:17:04 - 05-12-24,0:23:01	friends, water, blue, green	4
$D1$	06-08-18,22:47:42 - 06-11-22,4:40:31	code, downloads	2
$D2$	06-05-31,08:45:08 - 06-11-13,9:11:31	personal, entertainment	2
$D3$	06-06-12,20:18:18 - 06-11-29,15:00:45	jobs, art	2
$D4$	06-06-06,03:33:57 - 06-11-19,13:15:52	search, research, blogs	3
$D5$	06-07-03,13:44:34 - 06-11-20,03:38:58	free, internet, games	3
$D6$	06-06-04,01:40:27 - 06-11-21,17:01:47	books, art, web	3
$D7$	06-06-13,07:50:46 - 06-11-20,12:36:10	tutorial, development, work, web2.0	4
$D8$	06-06-06,08:45:31 - 06-11-20,16:20:29	tools, online, toread, technology	4
$D9$	06-07-21,03:59:03 - 06-11-19,23:26:22	web, news, blogs, fun	4

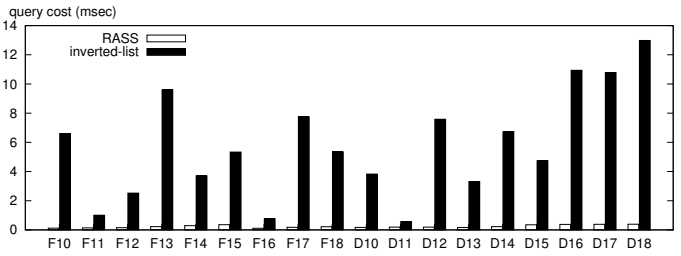
(a) Slowest queries for *RASS*

	interval I	set ids	d
$F10$	05-04-04,11:00:24 - 05-05-03,16:07:01	california, san	2
$F11$	05-04-03,11:26:32 - 05-05-27,20:21:38	diego, ski, skiing	2
$F12$	05-06-08,11:57:05 - 05-08-01,4:38:53	photography, september	2
$F13$	05-06-04,3:10:18 - 05-09-06,10:13:24	vacations, holiday, europe	3
$F14$	05-07-01,18:54:44 - 05-08-01,04:38:47	day, portraits, women	3
$F15$	05-08-27,3:37:25 - 05-10-29,17:37:56	cathedral, sunny, germany	3
$F16$	05-06-21,13:54:07 - 05-07-30,17:03:52	stuttgart, kii, campuswg, idwebcamkii	4
$F17$	05-06-20,20:55:27 - 05-08-01,4:38:26	october, september, may, march	4
$F18$	05-06-19,1:17:49 - 05-08-01,4:37:18	day, april, march, january	4
$D10$	06-11-20,3:07:47 - 06-11-28,5:23:55	blogs, trends	2
$D11$	06-06-21,14:54:24 - 06-08-22,16:19:17	lesbian, galleries	2
$D12$	06-10-04,12:23:00 - 06-10-09,0:52:02	inspiration, design	2
$D13$	06-08-20,19:18:18 - 06-09-03,15:40:59	social, discussion, networking	3
$D14$	06-11-21,3:15:35 - 06-11-27,20:26:37	computing, tech, tools	3
$D15$	06-11-20,16:10:49 - 06-12-01,3:50:51	resource, service, cool	3
$D16$	06-11-17,6:22:25 - 06-12-05,23:20:45	internet, online, technology, webservice	4
$D17$	06-11-16,22:15:42 - 06-12-18,12:14:14	web, webdesign, css, developer	4
$D18$	06-11-21,12:55:09 - 06-12-06,5:46:11	reference, tools, work, utilities	4

(b) Fastest queries for *RASS*



(c) Comparison on the queries in (a)



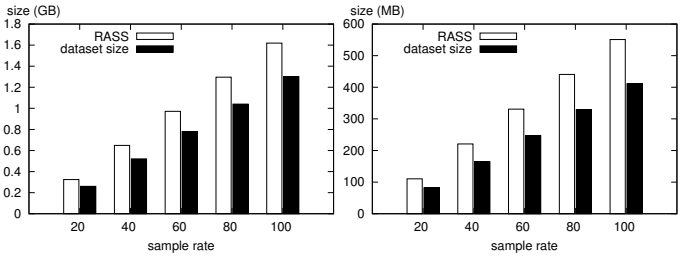
(d) Comparison on the queries in (b)

Fig. 6. Efficiency of individual queries in the 512-workloads of $d = 2, 3, 4$

We now delve into query workloads to examine the behavior of *RASS* on individual queries. For each dataset, we looked at the 512-workloads for $d = 2, 3$ and 4, respectively. In each workload, we identified the 3 slowest and 3 fastest queries for *RASS*, respectively. As 6 queries were identified from each workload, altogether 18 queries were extracted this way for each dataset, i.e., 36 queries in total. We provide the full details of all these queries in Figures 6a and 6b. Specifically, Figure 6a includes the slowest queries for *RASS*: $F1, ..., F9$ from *Flickr*, and $D_1, ..., D_9$ from *Delicious*. Take $F1$ as an example. It is from the workload of $d = 2$. Its interval I is a period of time, whose starting and ending timestamps are as shown (each timestamp has the format: yy-mm-dd,hr-mn-sc). It specifies labels `london` and `red` (namely, it concerns pictures carrying these labels at the same time; see the query semantics explained earlier). Figure 6b lists the fastest queries in the same style.

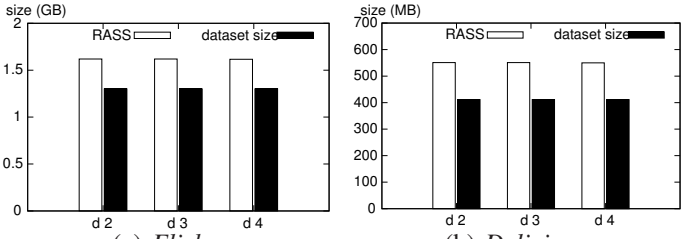
In Figure 6c (6d), we compare the cost of *RASS* and *inverted-index* on the queries in Figure 6a (6b). Remember that $F1$-$F9$ and $D1$-$D9$ are the "hardest" queries for *RASS* in their workloads. Even regarding these queries, *RASS* beat *inverted-index* on 16 of them, and narrowly lost only in 2 (i.e., $F2$ and $D1$). On the other hand, on the "easiest" queries in Figure 6d, *RASS* was extremely fast, and outperformed *inverted-index* by more than 10 times in most of them. It was such vast performance discrepancy that determined the dominant superiority of *RASS* in the previous experiments.

Space consumption and construction time. We now proceed to evaluate the space overhead of *RASS*. We began by assessing its scalability with the dataset size. Figure 7a (7b) plots the space usage of *RASS* for the 5 miniatures of *Flickr* (*Delicious*) in the experiment of Figure 5a (5b). For comparison, the space occupied by the corresponding datasets is also included. We can see that, thanks to its linear space complexity, *RASS* demonstrates excellent scalability in its space consumption. In particular, notice that it uses only about 20% more space than the dataset itself. Given the significant benefits brought by *RASS* in query efficiency, we believe that such space overhead is fairly reasonable and worthwhile.



(a) *Flickr* (b) *Delicious*

Fig. 7. Space vs. dataset size ($k = 512, d = 2$)

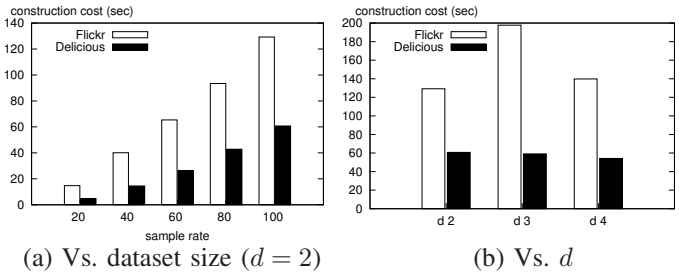


(a) *Flickr* (b) *Delicious*

Fig. 8. Space vs. d ($k = 512$)

Next, we turned attention back to the original *Flickr* and *Delicious* (i.e., 100% sample rate), and measured the space consumption of *RASS* as a function of d. The results are presented in Figure 8. Evidently, the space is hardly affected by this parameter. This is expected because, for different d, the value of τ (see Definition 3) is adjusted, so that the intersection array consumes roughly the same amount of space. Also recall that, the intersection array is the only component that requires "extra" space – all the other space is devoted to basic structures (i.e., hash tables and BSTs).

The last set of experiments studied the time required to build a RASS index. Figure 9a gives the construction cost as a function of the dataset size (using the 5 miniatures of each dataset), demonstrating graceful scalability. Figure 9b plots the cost with respect to d (using the original datasets). It is worth pointing out that the cost does *not* need to be monotonic to d. This is because, as d grows, even though the number of large nodes decreases, the number of their *combinations* remains roughly the same, i.e., equal to the size of the intersection array A. The building time may actually increase if more cells of A need to be modified, which in turn depends on the data distribution. In any case, even for the largest dataset *Flickr* (1.3GB in size), the entire construction finished within only 3 and a half minutes.



(a) Vs. dataset size ($d = 2$) (b) Vs. d

Fig. 9. RASS index construction time ($k = 512$)

7 Conclusions

This paper proposes the RASS problem, which aims at extracting aggregate information on the intersection of sets that are arbitrarily selected at query time from a large number of possible sets. Motivated by the vast importance of this problem, we have developed the *RASS* system, which is able to answer RASS queries significantly faster than alternative methods, by a factor up to an order of magnitude. Besides its excellent practical efficiency, *RASS* is designed based on solid theoretical foundation. Not only that it enjoys linear space complexity, but also its query time nearly matches the theoretically optimal performance. *RASS* involves only binary search trees and a multi-dimensional array, and therefore can be easily incorporated into many existing commercial systems. Finally, our theoretical analysis has revealed valuable insight into the characteristics of the RASS problem, and thus paves a solid foundation for further research on this topic.

Acknowledgements

This work was supported in part by the WCU (World Class University) program under the National Research Foundation of Korea, and funded by the Ministry of Education, Science and Technology of Korea (Project No: R31-30007). Yufei Tao and Cheng Sheng were also supported in part by projects GRF 4166/10, 4165/11, and 4164/12 from HKRGC. Chin-Wan Chung and Jong-Ryul Lee were also supported in part by the National Research Foundation of Korea grant funded by the Korean government (MSIP) (No. NRF-2009-0081365).

References

[1] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD*, pages 277–288, 2006.
[2] H. Cohen and E. Porat. Fast set intersection and two-patterns matching. In *LATIN*, pages 234–242, 2010.
[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.
[4] P. F. Dietz, K. Mehlhorn, R. Raman, and C. Uhrig. Lower bounds for set intersection queries. *Algorithmica*, 14(2):154–168, 1995.
[5] B. Ding and A. C. Konig. Fast set intersection in memory. *PVLDB*, 4(4):255–266, 2011.
[6] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, pages 656–665, 2008.
[7] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.
[8] S. Govindarajan, P. K. Agarwal, and L. Arge. CRB-tree: An efficient indexing scheme for range-aggregate queries. In *ICDT*, pages 143–157, 2003.
[9] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE*, pages 152–159, 1996.
[10] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in olap data cubes. In *SIGMOD*, pages 73–88, 1997.
[11] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
[12] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP operations in spatial data warehouses. In *SSTD*, pages 443–459, 2001.
[13] C. K. Poon. Optimal range max datacube for fixed dimensions. In *ICDT*, pages 158–172, 2003.
[14] C. Sheng and Y. Tao. New results on two-dimensional orthogonal range aggregation in external memory. In *PODS*, pages 129–139, 2011.
[15] Y. Tao and D. Papadias. Range aggregate processing in spatial databases. *TKDE*, 16(12):1555–1570, 2004.

[16] M. Terrovitis, P. Bouros, P. Vassiliadis, T. K. Sellis, and N. Mamoulis. Efficient answering of set containment queries for skewed item distributions. In *EDBT*, pages 225–236, 2011.
[17] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, pages 495–506, 2010.
[18] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD*, pages 85–96, 2012.
[19] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. *VLDB J.*, 12(3):262–283, 2003.
[20] D. M. Yellin. Algorithms for subset testing and finding maximal sets. In *SODA*, pages 386–392, 1992.
[21] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *ICDE*, pages 688–699, 2009.
[22] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comp. Surv.*, 38(2), 2006.

Yufei Tao is a full professor at the Chinese University of Hong Kong. From Sep 2011 to Jun 2013, he also held a Visiting Professor position, under the World Class University (WCU) program of the Korean government, at the Korea Advanced Institute of Science and Technology (KAIST). He is an associate editor of ACM Transactions on Database Systems (TODS), and of IEEE Transactions on Knowledge and Data Engineering (TKDE). He is/was a PC co-chair of ICDE 2014, a PC co-chair of SSTD 2011, an area PC chair of ICDE 2011, and a senior PC member of CIKM 2010, 2011, 2012. He received the best paper award at SIGMOD 2013, and the Hong Kong Young Scientist Award in 2002.

Cheng Sheng received his Ph.D. degree in computer science from the Chinese University of Hong Kong in 2012 and BSc degree in computer science from Fudan University in 2008. He is currently a software engineer at Google Switzerland. His research focuses on algorithms in database systems with nontrivial theoretical guarantees.

Chin-Wan Chung received a B.S. degree in electrical engineering from Seoul National University, Korea, in 1973, and a Ph.D. degree in computer engineering from the University of Michigan, Ann Arbor, USA, in 1983. From 1983 to 1993, he was a Senior Research Scientist and a Staff Research Scientist in the Computer Science Department at the General Motors Research Laboratories (GMR). While at GMR, he developed Dataplex, a heterogeneous distributed database management system integrating different types of databases. Since 1993, he has been a professor in the Department of Computer Science at the Korea Advanced Institute of Science and Technology (KAIST), Korea. At KAIST, he developed a full-scale object-oriented spatial database management system called OMEGA, which supports ODMG standards. His current major project is about mobile social networks in Web 3.0. He was in the program committees of major international conferences including ACM SIGMOD, VLDB, IEEE ICDE, and WWW. He was an associate editor of ACM TOIT, and is an associate editor of WWW Journal. He will be the General Chair of WWW 2014. His current research interests include the semantic Web, mobile Web, social networks, and spatio-temporal databases. More information is available at http://islab.kaist.ac.kr/chungcw.

Jong-Ryul Lee received the B.S. degree in computer science from Korea Advanced Institute of Science and Technology (KAIST). Currently, he is working toward the Ph.D. degree in computer science at KAIST. His research interests include data management, spatio-temporal data mining, and social network analysis.