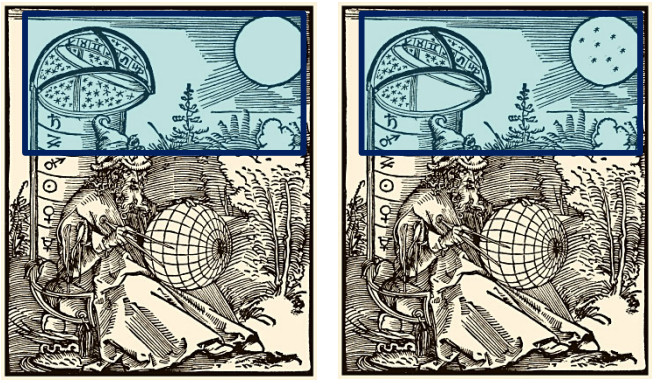
Set-Difference Range Queries

David Eppstein * Michael T. Goodrich † Joseph A. Simons ‡

Abstract

We introduce the problem of performing set-difference range queries, where answers to queries are set-theoretic symmetric differences between sets of items in two geometric ranges. We describe a general framework for answering such queries based on a novel use of data-streaming sketches we call *signed symmetric-difference* sketches. We show that such sketches can be realized using invertible Bloom filters (IBFs), which can be composed, differenced, and searched so as to solve set-difference range queries in a wide range of scenarios.

*Dept. of Comp. Sci., U. of CA, Irvine, eppstein(at)uci.edu
†Dept. of Comp. Sci., U. of CA, Irvine, goodrich(at)uci.edu
‡Dept. of Comp. Sci., U. of CA, Irvine, jsimons(at)uci.edu



(a) (b)

Figure 1: Illustrating the set-difference range query problem. The images in (a) and (b) have four major differences, three of which are inside the common query range. The image (a) is a public-domain engraving of an astronomer by Albrecht Dürer, from the title page of *Messahalah, De scientia motus orbis* (1504).

1 Introduction

Efficiently identifying or quantifying the differences between two sets is a problem that arises frequently in applications, for example, when clients synchronize the calendars on their smart phones with their calendars at work, when databases reconcile their contents after a network partition, or when a backup service queries a file system for any changes that have occurred since the last backup. Such queries can be global, for instance, in a request for the differences across all data values for a pair of databases, or they can be localized, requesting differences for a specific range of values of particular interest. For example, clients might only need to synchronize their calendars for a certain range of dates or a pair of databases may need only to reconcile their contents for a certain range of transactions. We formalize this task by a novel type of range searching problem, which we call *set-difference range queries*.

We assume a collection $\mathcal{X}$ of sets $\{X_1, X_2, \ldots, X_N\}$, containing *data items* that are each associated with a geometric point and with a member of universe, $\mathcal{U}$, of size $U = |\mathcal{U}|$. A set-difference range query is specified by the indices of a pair of data sets, X_i and X_j, and by a pair of *ranges*, R_1 and R_2, which are each a constant-size description of a set of points such as a hyper-rectangle, simplex, or half-space. The answer to this set-difference range query consists of the elements of X_i and X_j whose associated points belong to the ranges R_1 and R_2 respectively, and whose associated elements in U are contained in one of the two data sets but not both. Thus, we preprocess $\mathcal{X}$ so that given ranges, R_1 and R_2, and two sets, X_1 and X_2, we can quickly report (or count) the universe elements in the set-theoretic symmetric difference $(R_1 \cap X_1) \triangle (R_2 \cap X_2)$. The performance goal in answering such queries is to design data structures having low space and preprocessing requirements that support fast set-difference range queries whose time depends primarily on the size of the difference, not the number of items in the range (see Figure 1). Examples of such scenarios include the following.

- Each set contains readings from a group of sensors in a given time quantum (e.g., see [5]). Researchers may be interested in determining which sensor values have changed between two time quanta in a given region.

Query Type		Query Time	Space
Orthogonal:	Standard [25, 7]	$O(\log^{d-1} n)$	$O(n \log^{d-1} n)$
	SD fixed m	$O(m \cdot \log^{d} n)$	$O(m \cdot n \log^{d-1} n)$
	SD variable m	$O(m \cdot \log^{d} n)$	$O(n \log^{d} n)$
	SD size est.	$O(\log^{d+1} n \log U)$	$O(n \log^{d} n \log U)$
Simplex:	Standard [26]	$O(n^{1-1/d}(\log n)^{O(1)})$	$O(n)$
	SD fixed m	$O(m \cdot n^{1-1/d}(\log n)^{O(1)})$	$O(m \cdot n)$
	SD variable m	$O(m \cdot n^{1-1/d}(\log n)^{O(1)})$	$O(n \log \log n)$
	SD size est.	$O(n^{1-1/d}(\log n)^{O(1)} \log U)$	$O(n \log n \log U)$
Stabbing:	Standard [6]	$O(\log n)$	$O(n \log n)$
	SD fixed m	$O(m \cdot \log n)$	$O(m \cdot n \log n)$
	SD variable m	$O(m \cdot \log n)$	$O(n \log n)$
	SD size est.	$O(\log^2 n \log U)$	$O(n \log n \log U)$
Partial Sum:	Standard[1]	$O(1)$	$O(n)$
	SD fixed m	$O(m)$	$O(m \cdot n)$
	SD variable m	$O(m)$	$O(n^2)$
	SD size est.	$O(\log n \log U)$	$O(n \log n \log U)$

Table 1: The results labeled "Standard" are previously known results for each data structure. Results labeled "SD" indicate bounds for set-difference range queries. Here d is the dimension of the query, m is the output size, and we assume the approximation factor $(1 \pm \epsilon)$ and failure probability δ are fixed.

- Each set is a catalog of astronomical objects in a digital sky survey. Astronomers are often interested in objects that appear or disappear in a given rectangular region between pairs of nightly observations. (E.g., see [36].)
- Each set is an image taken at a certain time and place. Various applications may be interested in pinpointing changes that occur between pairs of images (e.g., see [29]), which is something that might be done, for instance, via two-dimensional binary search and repeated set-difference range queries.

1.1 Related Work

We are not aware of prior work on set-difference range queries. However, Suri *et al.* [32] consider approximate range counting in data streams. Shi and JaJa [31] present a data structure for range queries indexed at a specific time for data that is changing over time, achieving polylogarithmic query times. If used for set-difference range queries, however, their scheme would not produce answers in time proportional to the output size. Such queries are also related to kinetic range-query data structures, such as those given by Agarwal *et al.* [2] or Šaltenis *et al.* [30], which can answer range queries for a given moment in time for points moving along known trajectories. For a survey of various schemes for indexing and search temporal data, please see Mokbel *et al.* [27]. For a survey of general schemes for range searching data structures, see Agarwal [1].

[1] A standard solution to the partial sum problem is illustrated in Figure 3

1.2 Our Results

We provide general methods for supporting a wide class of set-difference range queries by combining signed-symmetric difference sketches with any canonical group or semigroup range searching structure. Our methods solve range difference queries where the two sets being compared may be drawn from the same or different data sets, and may be defined by the same or different ranges. In our data structures, sets are combined in the *multiset model*: two data items may be associated with the same element of U, and if they belong to the same query range they are considered to have multiplicity two, while if they belong to the two different query ranges defining the set difference problem then their cardinalities cancel. The result of a set difference query is the set of all elements whose total cardinality defined in this way is nonzero.

Our data structures are probabilistic and return the correct results with high probability. Our running times depend on the size of the output, but only weakly depend on the size of the original sets. In particular, we derive the results shown in Table 1 for the following range-query problems (see Section 7 for details):

- Orthogonal: Preprocess a set of points in R^d such that given a query range defined by an axis-parallel hyper rectangle, we can efficiently report or estimate the size of the set of points contained in the hyper-rectangle
- Simplex: Preprocess a set of points in R^d such that given a query range defined by a simplex in R^d, we can efficiently report or estimate the size of set of points contained in the simplex
- Stabbing: Preprocess a set of intervals such that given a query point x, we can efficiently report or estimate the size of the subset which intersects x.
- Partial Sum: Preprocess a grid of total size $O(n)$ (e.g. an image with n pixels, or a d dimensional array with $O(n)$ entries for any constant d) such that given a query range defined by a hyper-rectangle on the grid, we can efficiently report or estimate the sum of the values contained in that hyper-rectangle.

In the remainder of the paper, we show how to perform set-difference range reporting and counting, based on the framework of using signed-symmetric difference reporting (SDR) and signed-symmetric difference counting (SDC) data structures. Finally, we discuss the underlying methods for implementing SDR and SDC sketches in more detail in Section 4 and Section 6, respectively, based on the use of invertible Bloom filters and frequency moments.

2 The Abstract Range Searching Framework

In order to state our results in their full generality it is necessary to provide some general definitions from the theory of range searching (e.g., see [1]).

A *range space* is a pair (X, R) where X is a universe of objects that may appear as data in a range searching problem (such as the set of all points in the Euclidean plane) and R is a family of subsets of X that may be used as queries (such as the family of all disks in the Euclidean plane). A *range searching problem* is defined by a range space together with an *aggregation function* f that maps subsets of X to the values that should be returned by a query. For instance, in a *range reporting problem*, the aggregation function is the identity; in a *range counting problem*, the aggregation function maps a subset of X to its cardinality. A data structure for the range searching

problem must maintain a finite set $Y \subset X$ of objects, and must answer queries that take a range $r \in R$ as an argument and return the value $f(r \cap Y)$. The goal of much research in range searching is to design data structures that are efficient in terms of their space usage, preprocessing time, update time (if changes to Y such as insertions and deletions are allowed), and query time. In particular, it is generally straightforward to answer a query in time $O(|Y|)$, by testing each member of Y for membership in the query range r; the goal is to answer queries in an amount of time that may depend on the output size but that does not depend so strongly on the size of the data set.

A range searching problem is said to be *decomposable* if there exists a commutative and associative binary operation $\oplus$ such that, for any two disjoint subsets A and B of X, $f(A \cup B) = f(A) \oplus f(B)$. In this case, the operation $\oplus$ defines a *semigroup*.

Many range searching data structures have the following form, which we call a *canonical semigroup range searching data structure*: the data structure stores the values of the aggregation function on some family F of sets of data values, called *canonical sets*, with the property that every data set $r \cap Y$ that could arise in a query can be represented as the disjoint union of a small number of canonical sets r_1, r_2, ..., r_K. To answer a query, a data structure of this type performs this decomposition of the query range into canonical sets, looks up the values of the aggregation function on these sets, and combines them by the $\oplus$ operation to yield the query result. Note that this is sometimes called a *decomposition scheme*. For instance, the order-statistic tree is a binary search tree over an ordered universe in which each node stores the number of its descendants (including itself) in the tree, and it can be used to quickly answer queries that ask for the number of data values within any interval of the ordered universe. This data structure can be seen as a canonical semigroup range searching data structure in which the aggregation function is the cardinality, the combination operation $\oplus$ is integer addition, and the canonical sets are the sets of elements descending from nodes of the binary search tree. Every intersection of the data with a query interval can be decomposed into $O(\log n)$ canonical sets, and so interval range counting queries can be answered with this data structure in logarithmic time.

In general, there is a tradeoff between $|F|$, the number of canonical subsets stored by the data structure, and the number of canonical subsets required to answer a query. To illustrate, we consider a range searching problem in one dimension. If we have n canonical subsets, then in order to cover all possible queries we need to have each subset be a singleton set, and to answer a query we may need to use $O(n)$ subsets. If we use $O(n \log n)$ canonical subsets, then we can answer a query using only $O(\log n)$ of them as described above. If we use $O(n^2)$ canonical subsets, we only need to use $O(1)$ canonical subsets to answer a query, because there are only $O(n^2)$ unique answers to a range query in one dimension, and by using $O(n^2)$ canonical subsets we can precompute all of them. When the combination operation $\oplus$ has additional properties, they may sometimes be used to obtain greater efficiency. In particular, if $\oplus$ has the structure of a group, then we may form a *canonical group range searching data structure*. Again, such a data structure stores the values of the aggregation function on a family of sets of data values, but in this case it represents the query value as an expression $(\pm f(r_1)) \oplus (\pm f(r_2)) \oplus \cdots$, where the canonical sets r_i and their signs are chosen with the property that each element of the query range r belongs to exactly one more positive set than negative set. Again, in the interval range searching problem, one may store with each element its *rank*, the number of elements in the range from $-\infty$ to that element, and answer a range counting query by subtracting the rank of the right endpoint from the rank of the left endpoint. In this example, the ranks are not easy to maintain if elements are inserted and deleted, but they allow interval range queries to be answered by combining only two canonical sets instead

of logarithmically many.

2.1 Signed Symmetric-Difference Sketches

Suppose that we want to represent an input set S in space sub-linear in the size of S such that we can compute some function $f(S)$ on our compressed representation. This problem often comes up in the streaming literature, and common solutions include dimension reduction (e.g. by a Johnson-Lindenstrauss transform [21]) and computing a *sketch* of S (e.g. Count-Min Sketch [12]).

A *sketch* σ_S of a set S is a randomized compressed representation of S such that we can approximately and probabilistically compute $f(S)$ by evaluating an appropriate function $f'(\sigma_S)$ on the sketch σ_S. This construct also comes up when handling massive data data sets, and in this context the compressed representation is sometimes called a synopsis [19].

A sketch algorithm σ is called *linear* if it has a group structure. That is, there exist two operators $\oplus$ and $\ominus$ on sketches σ such that given two multi-sets S and T,

$$\sigma_{S \uplus T} = \sigma_S \oplus \sigma_T \text{ and } \sigma_{S \setminus T} = \sigma_S \ominus \sigma_T$$

where $\uplus$ and $\setminus$ are the multi-set addition and subtraction operators respectively.

For our results, we define two different types of linear sketches. A *Signed Symmetric-Difference Reporting* (SDR) sketch is a linear sketch that supports a function `report`: given a pair of sketches σ_S and σ_T for two sets S and T respectively, probabilistically compute $S \setminus T$ and $T \setminus S$ using only information stored in the sketches σ_S and σ_T in $O(1 + m)$ time, where m is the cardinality of the output. A *Signed Symmetric-Difference Cardinality* (SDC) sketch is a linear sketch that supports a function `count`: given a pair of sketches σ_S and σ_T for two sets S and T respectively, probabilistically approximate $|S \triangle T|$ using only information stored in the sketches σ_S and σ_T in time linear in the size of the sketches.

3 Main Results

The main idea of our results is to represent each canonical set by an SDR or an SDC sketch. We implement our signed symmetric-difference counting sketches using a linear sketch based on the frequency moment estimation techniques of Thorup and Zhang [34]. We implement our signed symmetric-difference reporting sketches via an *invertible Bloom filter* (IBF), a data structure introduced for straggler detection in data streams [16]. IBFs can be added and subtracted, giving them a group structure and allowing an IBF for a query range to be constructed from the IBFs for its constituent canonical sets. The difference of the IBFs for two query ranges is itself an IBF that allows the difference elements to be reported when the difference is small. To handle set differences of varying sizes we use a hierarchy of IBFs of exponentially growing sizes, together with some special handling for the case that the final set difference size is larger than the size of some individual canonical set.

Further details of the SDR and SDC sketches are given in later sections. In this section, we assume the existence of SDR and SDC sketches as defined above in order to prove the following three theorems which are the crux of our results listed in Table 1.

Theorem 1: *Suppose that a fixed limit m on the cardinality of the returned set differences is known in advance of constructing the data structure, and our queries must either report the difference if*

it has cardinality at most m, or otherwise report that it is too large. In this case, we can answer set-difference range queries with probability at least $1-\epsilon$ for any range space that can be modeled by a canonical group or semigroup range searching data structure. Our solution stores $O(m)$ words of aggregate information per canonical set, uses a combination operation $\oplus$ that takes time $O(m)$ per combination, and allows the result of this combination to be decoded to yield the query results in $O(m)$ time. If the data structure is updated, the aggregate information associated with each changed canonical set can itself be updated in constant time per change.

Proof: We associate with each canonical set an SDR sketch for its elements that supports the `report` function for outputs of size up to m, and we let $\oplus$ be defined by the operations of adding SDR sketches as described in the previous section. The result of applying $\oplus$ to the SDR sketches stored at each of the canonical sets will be a single sketch for each of the queried ranges. Then, to report the set-difference, we call the `report` function which outputs the elements if there are fewer than m of them. The SDR sketch is implemented as an IBF, and the `report` function is supported via an IBF subtraction followed by the listItems operation. See Section 4 for details of the implementation. If the listItems operation fails to decode the IBF, or if it lists more than m items, we report that the cardinality bound is exceeded. ■

Theorem 2: *Suppose that we wish to report set differences that may be large or small, without the fixed bound m, in a time bound that depends on the size of the difference but that may depend only weakly on the size of the total data set. In this case, we can answer range difference queries with probability at least $1-\epsilon$ for any range space that can be modeled by a canonical group or semigroup range searching data structure. Our solution stores a number of words of aggregate information per canonical set that is $O(1)$ per element of the set, uses a combination operation $\oplus$ that takes time $O(m)$ (where m is the cardinality of the final set-theoretic difference) and allows the result of this combination to be decoded to yield the query results in $O(m)$ time. If the data structure is updated, the aggregate information associated with each changed canonical set can itself be updated in logarithmic time per change.*

Proof: For each canonical set S of the range query data structure, we store an SDR sketch consisting of a hierarchy of $\log_2 |S|$ IBFs, capable of successfully performing the listItems operation on sets of sizes that are powers of two up to the size of S itself. The total size of these IBFs adds in a geometric series to be proportional to the size of S. The time to update the data structure then follows since each element of each canonical set is stored in a logarithmic number of IBFs.

To answer a range query, suppose first that m is known. In this case, we choose $j = \lceil \log_2 d \rceil$, so that $2^j > m$ but $2^j = O(m)$, and we collect and combine the IBFs of size 2^j exactly as in Theorem 1. The only possible complication is that, for some canonical sets, the number 2^j may be larger than the number of elements in the canonical set, so that we have no pre-stored IBF of the correct size to collect. In this case, though, we can construct an IBF of the correct size from the list of elements of the canonical set, in time $O(2^j)$. Finally, if m is unknown, we try the same procedure for successive powers of two until finding a set of powers for which the result succeeds. ■

Theorem 3: *Suppose that we wish to report the cardinality of the set difference rather than its elements, and further that we allow this cardinality to be reported approximately, within a*

fixed approximation ratio that may be arbitrarily close to one. In this case, we can answer range difference queries with probability at least $1 - \epsilon$ *for any range space that can be modeled by a canonical group or semigroup range searching data structure. Our solution stores a number of words of aggregate information per canonical set that has size* $O(\log n \log U)$*, uses a combination operation* $\oplus$ *that takes time* $O(\log n \log U)$ *and allows the result of this combination to be decoded to yield the query results in* $O(\log n \log U)$ *time.*

Proof: We associate with each canonical set an SDC sketch for its elements that supports the `count` function with probability $1-\epsilon$. and we let $\oplus$ be defined by the operations of adding the SDC sketches. The result of applying $\oplus$ to these sketches will be a single sketch for each of the queried ranges. Then to approximate the cardinality of the difference, we call the `count` function. We implement each SDC sketch as set of $O(\log(1/\epsilon))$ independent linear frequency moment estimation sketches. The space and time bounds are the same as those for computing these sketches, and further details of the implementation are given in Section 6. ■

4 Invertible Bloom Filters

We implement our SDR sketches using the invertible Bloom filter (IBF) [16], a variant of the Bloom filter [8] for maintaining sets of items that extends it in three crucial ways that are central to our application. First, like the counting Bloom filter [9, 18], the IBF allows both insertions and deletions, and it allows the number of inserted elements to far exceed the capacity of the data structure as long as most of the inserted elements are deleted later. Second, unlike the counting Bloom filter, the IBF allows the elements of the set to be listed back out. And third, again unlike the counting Bloom filter, the IBF allows *false deletions*, deletions of elements that were never inserted, and again it allows the elements involved in false deletion operations to be listed back out as long as their number is small. These properties allow us to represent large sets as small IBFs, and to quickly determine the elements in the symmetric difference of the sets, as we now detail.

We assume that the items being considered belong to a countable universe, U, and that given any element, x, we can determine a unique integer identifier for x in $O(1)$ time (e.g., either by x's binary representation or through a hash function). Thus, for the remainder of this paper, we assume w.l.o.g. that each element x is an integer.

The main components of an IBF are a table, T, of t cells, for a given parameter, t; a set of k random hash functions, $h_1, h_2, \ldots, h_k$, which map any element x to k distinct cells in T; and a random hash function, g, which maps any element x to a random value in the range $[1, 2^\lambda]$, where λ is a specified number of bits.

Each cell of T contains the following fields:

- count: an integer count of the number of items mapped to this cell
- idSum: a sum of all the items mapped to this cell
- gSum: a sum of $g(x)$ values for all items mapped to this cell.

The gSum field is used for checksum purposes. An IBF supports several simple algorithms for item insertion, deletion, and membership queries, as shown in Figure 2.

In addition, we can take the difference of one IBF, A, with a table T_A, and another one, B, with table T_B, to produce an IBF, C, with table T_C, representing their signed difference, with the

items in $A \setminus B$ having positive signs for their cell fields in C and items in $B \setminus A$ having negative signs for their cell fields in C (we assume that C is initially empty). This simple method is also shown in Figure 2.

Finally, given an IBF, which may have been produced either through insertions and deletions or through a subtract operation, we can list out its contents by repeatedly looking for cells with counts of $+1$ or -1 and removing the items for those cells if they pass a test for consistency. This method therefore produces a list of items that had positive signs and a list of items that had negative signs, and is shown in Figure 2. In the case of an IBF, C, that is the result of a $\mathsf{subtract}(A, B, C)$ operation, the positive-signed elements belong to $A \setminus B$ and the negative-signed elements belong to $B \setminus A$.

Remark 4: *The* Strata Estimator *[17] combines invertible Bloom filters with a hierarchical sampling technique. The strata estimator can also be used to implement the SDC sketch. However, compared to the frequency moment estimation technique outlined below, the strata estimator has a slightly worse dependence on ϵ and δ, and therefore we do not incorporate it as a component in our results.*

4.1 Analysis

In this section, we extend previous analyses [16, 17] to bound the failure probability for the functioning of an invertible Bloom filter to be less than a given parameter, $\epsilon > 0$, which need not be a constant (e.g., its value could be a function of other parameters).

Theorem 5: *Suppose X and Y are sets with m elements in their symmetric difference, i.e., $m = |X \triangle Y|$, and let $\epsilon > 0$ be an arbitrary real number. Let A and B be invertible Bloom filters built from X and Y, respectively, such that each IBF has $\lambda \geq k + \lceil \log k \rceil$ bits in its gSum field, i.e., the range of g is $[1, 2^\lambda)$, and each IBF has at least $2km$ cells, where $k > \lceil \log(m/\epsilon) \rceil + 1$ is the number of hash functions used. Then the $\mathsf{listItems}$ method for the IBF C resulting from the $\mathsf{subtract}(A, B, C)$ method will list all m elements of $X \triangle Y$ and identify which belong to $X \setminus Y$ and which belong to $Y \setminus X$ with probability at least $1 - \epsilon$.*

Proof: There are at least km empty cells in C; hence, the probability that an element $z \in X \triangle Y$ collides with another element in $X \triangle Y$ in each of its k cells is at most 2^{-k}, independent of the locations chosen in C by the other elements in $X \triangle Y$. Thus, the probability that any element in $X \triangle Y$ has k collisions is at most $m\, 2^{-k} \leq \epsilon/2$. Therefore, with probability at least $1 - \epsilon/2$, every element in $X \triangle Y$ has at least one cell in C where it is the only occupant.

Next, consider the probability that a cell has $|\mathsf{count}| = 1$ and $|\mathsf{gSum}| = g(|\mathsf{idSum}|)$ but nevertheless has two or more occupants. Since the gSum field has at least k bits, the probability that a cell is "bad" in this way is at most $2^{-\lambda}$. Thus, since there are at most km non-zero cells, the probability we have any such bad cell in C is at most

$$km\, 2^{-\lambda} \leq m 2^{-k} \leq \epsilon/2.$$

This completes the proof. ∎

To avoid infinite loops, we make a small change to $\mathsf{listItems}$, forcing it to stop decoding after m items have been decoded regardless of whether there remain any decodable cells. This change does not affect the failure probability and with it the running time is always $O(mk)$.

For example, we can achieve an approximation factor of $1 \pm o(1)$ with probability $1 - U^{-c}$, for any constant $c > 0$, with a strata estimator that has size polylogarithmic in U.

Remark 6: *Suppose we only need to answer the decision version of the problem: "Is* $|X \triangle Y|(1 \pm \delta) \leq \tau$*?" for some threshold parameter* τ*. Then we can save a* $\log U$ *factor in both time and space by only constructing a single layer of the strata estimator, chosen according to* τ*.*

5 Measuring and Estimating Set Dissimilarity

Many distances have been defined between pairs of sets, and different choices may be appropriate in different applications. For instance, in information retrieval, it is useful to normalize the distance between sets by weighting distance by the size of the sets, so that documents on similar topics may count as similar even if their lengths may greatly differ. In applications where the sets being compared describe the presence or absence of a fixed collection of binary attributes, Hamming distance may be more appropriate.

In more detail, suppose we are given finite sets X and Y either by listings of their elements or by bit vectors $\vec{X}$ and $\vec{Y}$ that represent their characteristic functions as subsets of a fixed universe U. There are a host of ways to define notions of distance between X and Y, with the following being some of the most well-known:

- *Set-theoretic Hamming distance*: $H(X, Y) = |X \triangle Y| = |X \setminus Y| + |Y \setminus X|$, that is, the number of positions where $\vec{X}$ and $\vec{Y}$ differ.
- *Jaccard/Tanimoto distance*: $J(X, Y) = |X \triangle Y|/|X \cup Y| = 1 - j(X, Y)$, where $j(X, Y)$ is the related *Jaccard index* [20], defined as $j(X, Y) = |X \cap Y|/|X \cup Y|$. The *Tanimoto coefficient* [24], $t(\vec{X}, \vec{Y}) = (\vec{X} \cdot \vec{Y})/(||\vec{X}||^2 + ||\vec{Y}||^2 - \vec{X} \cdot \vec{Y})$ generalizes this distance to vectors.
- *Dice dissimilarity*: $D(X, Y) = |X \triangle Y|/(|X| + |Y|) = 1 - \gamma(X, Y)$, where $\gamma(X, Y)$ is *Dice's coefficient* $\gamma(X, Y) = 2|X \cap Y|/(|X| + |Y|)$ [13].
- *Tversky dissimilarity*: $T(X, Y) = 1 - t(X, Y)$, where $t(X, Y)$ is the *Tversky index* $t(X, Y) = |X \cap Y|/(|X \cap Y| + \alpha|X \setminus Y| + \beta|Y \setminus X|)$ parameterized by $\alpha, \beta \geq 0$ [35]. For instance, setting $\alpha = \beta = 1$ is equivalent to the Jaccard/Tanimoto index and setting $\alpha = \beta = 1/2$ is equivalent to Dice's coefficient. Here, we are interested in any constant non-zero setting of α and β.

For a detailed discussion of these metrics, indexes, and coefficients, as well as others, please see [23, 33].

The Hamming and Jaccard/Tanimoto distances are metrics [4, 23, 24], whereas Dice dissimilarity is a semimetric that doesn't always satisfy the triangle inequality [11, 23]. Depending on α and β, Tversky dissimilarity could be a metric, semimetric, quasimetric, or quasisemimetric, in that it may or may not satisfy the triangle inequality and/or the symmetry axiom. However, this difference is not significant for our distance estimation algorithms.

5.1 Estimates for Other Dissimilarities

Given $|A|$ and $|B|$ and an accurate estimate for $H(A, B)$, we can also estimate the cardinalities of the intersection and union using the identities

$$|A \cap B| = (|A| + |B| - H(A, B))/2$$
$$|A \cup B| = (|A| + |B| + H(A, B))/2$$

The estimate for the cardinality of the union preserves the multiplicative error of the estimate for the Hamming distance. We cannot bound the multiplicative error of the estimated intersection cardinality, however, unless it is known to be at least proportional to the Hamming distance.

Once we have an accurate estimate for the Hamming distance $H(A, B)$ and the cardinality of the union $A \cup B$, we can compute similarly accurate estimates for each of the set-dissimilarity metrics and semimetrics mentioned in the introduction.

- For the Dice dissimilarity, estimate $D(A, B)$ as $H(A, B)/(|A| + |B|)$. Since $|A|$ and $|B|$ are exact, the accuracy does not change compared to that for H.
- For the Jaccard distance, estimate $J(A, B)$ as $2H(A, B)/(|A|+|B|+H(A, B))$. The resulting estimate has a multiplicative error that is at least as good as the error in estimating H.
- For the Tversky dissimilarity, assume without loss of generality that (as estimated) $|A \setminus B| > |B \setminus A|$ (else we apply a symmetric formula with the roles of A and B reversed), let p be our estimate of the ratio $|B \setminus A|/|A \setminus B|$, and let q be our estimate of the ratio $|A \cap B|/|A \setminus B|$. We then estimate $T(A, B) = (\alpha+\beta p)/(\alpha+\beta p+1)$. Both p and q are estimated with additive error $O(\epsilon)$, and since both the numerator and denominator of our estimate of $T(A, B)$ are $\Omega(1)$ and are the sum of additively-accurate estimators, they are also estimated to within multiplicative error $1 + O(\epsilon)$. Therefore, their ratio also has multiplicative error $1 + O(\epsilon)$.

6 Frequency Moment Estimation

We implement our SDC sketches using frequency moment estimation techniques. Let x be a vector of length U, and suppose we have a data stream of length m, consisting of a sequence of updates to x of the form $(i_1, v_1), \ldots, (i_m, v_m) \in [U] \times [-M, M]$ for some $M > 0$. That is, each update is a pair (i, v) which updates the ith coordinate of x such that $x_i \to x_i + v$.

The frequency moment of a data stream is given by

$$F_p = \sum_{i \in [U]} x_i^p = \|x\|_p^p.$$

Since the seminal paper by Alon *et al.* [3], frequency moment estimation has been an area of significant research interest. Indeed the full literature on the subject is too rich to survey here. Instead, see e.g. the recent work by Kane *et al.* [22] and the references therein. Kane *et al.* [22] gave algorithms for estimating F_p, $p \in (0, 2)$. Their algorithm requires $O(\log^2(1/\delta))$ time per update and $O(\delta^{-2} \log(mM))$ space. However, faster results are known for estimating the second frequency moment F_2 with constant probability. Thorup and Zhang [34] and Charikar *et al.* [10] independently improve upon the original result of Alon *et al.* [3], to achieve an optimal $O(1)$ update time using $O(\delta^{-2} \log(mM))$ space.

Given a sparse vector X of length m with coordinates bounded by $[-M, M]$, we can estimate $\|X\|_2^2$ by treating it as a data stream and running the algorithm of Thorup and Zhang. The algorithm computes a *sketch* S_X of X, of size $O(\delta^{-2} \log mM)$ such that $\|S_X\|_2^2$ is within a factor of $O(1 \pm \delta)$ of $\|X\|_2^2$ with constant probability. We can improve the probability bound to any arbitrary $O(1-\epsilon)$ by running the algorithm $O(\log(1/\epsilon))$ times independently to produce $O(\log(1/\epsilon))$ independent sketches S_{Xi}, and taking the median of $\|S_{Xi}\|_p^p$. This strategy takes $O(\log(1/\epsilon))$ time to process each non-zero element in X, and the space required is $O(\delta^{-2} \log(1/\epsilon) \log(mM))$.

Furthermore, each sketch is linear, and therefore we can estimate the frequency moment of the difference of two sparse vectors X and Y by subtracting their sketches; $\|S_X - S_Y\|_2^2$ is within a $O(1 \pm \delta)$ factor of $\|X - Y\|_2^2$ with constant probability, and we can maintain $O(\log(1/\epsilon))$ independent sketches to achieve probability $O(1 - \epsilon)$.

Now, suppose we want to estimate the Hamming distance between two sets. We treat each set as a sparse bit-vector and use the fact that the Hamming distance between two bit vectors is equivalent to the squared Euclidean distance, which is just the second frequency moment of the difference of the vectors. Then we can apply the above strategy for sparse vectors to produce $O(\log(1/\epsilon))$ sketches for each set in $O(\log(1/\epsilon))$ time per element, and we subtract all $O(\log 1/\epsilon)$ pairs of sketches in $O(\delta^{-2} \log(1/\epsilon) \log U)$ time. Finally we compute the second frequency moment of each sketch and take the median over all estimations in $O(\log 1/\epsilon)$ time.

We summarize this strategy in the following theorem.

Theorem 7: *Let $0 < \epsilon < 1$ and $0 < \delta < 1$ be arbitrary real numbers. Given two sets, X and Y, taken from a universe of size U, we can compute an estimate $\hat{m}$ such that*

$$(1 - \delta)|X \triangle Y| \leq \hat{m} \leq (1 + \delta)|X \triangle Y|,$$

with probability $1-\epsilon$, using a sketch of size $O(\delta^{-2} \log(1/\epsilon) \log U)$ The preprocessing time, including the time required to initialize the sketch is $O(\delta^{-2} \log(1/\epsilon) \log U + (|Y| + |X|) \log(1/\epsilon))$ and the time to compute the estimate $\hat{m}$ is $O(\delta^{-2} \log(1/\epsilon) \log U)$.

7 Set-Difference Range Searching Details

In this section, we look at several examples of canonical semigroup range searching structures and show how to transform them into data structures that can efficiently answer set-difference range queries. Specifically, we give more detail for the results which were summarized in Table 1.

7.1 Orthogonal Range Searching

In the orthogonal range searching problem, our task is to preprocess a set of points in R^d such that given a query range defined by an axis-parallel hyper rectangle, we can efficiently report or count the set of points contained in the hyper-rectangle The range tree is a classic canonical semigroup range searching data structure that solves this problem.

In one dimension, a range tree is a binary tree in which each leaf represents a small range containing a single point, and an in-order traversal of the leaves corresponds to the sorted order of those points. The range of any interior node is the union of the ranges of its descendant leaves. Each node stores the value of the aggregate function on its canonical set—the set of points in its range.

We can easily build a $d \geq 2$ dimensional range tree by building a range tree over the d^{th} dimension, and at each internal node we store a $(d-1)$-dimensional dimensional range tree over the points in that node's range. To answer a query using a range tree, we find a set of $O(\log^d n)$ canonical ranges which exactly cover the query range and report the sum of the aggregate functions of each of these canonical ranges.

Thus we can apply Theorem 1, 2, or 3 to obtain a data structure for set-difference orthogonal range queries. We preprocess each canonical set into an appropriate sketch, and instead of storing the value of the aggregate function in each node, instead we store a pointer to the corresponding sketch for that node's canonical set. When answering a query, instead of summing the aggregate functions, we simply add or subtract the sketches.

7.2 Simplex Range Searching

Partition trees [26] are a linear space data structure for the simplex range searching problem. In this problem we preprocess a set of points P in $\mathbb{R}^d$ so that given a d-dimensional query simplex ρ we can return the set of points $\{p \in P | p \in \rho\}$. Partition trees are based on recursively partitioning the plane into a set of regions such that each region contains at least a constant fraction of the points in P. These regions form canonical subsets and thus partition trees are a canonical semigroup range searching data structure Therefore, we can apply the transformation in Theorem 1, 2, 3 to obtain a data structure for set-difference simplex range queries. Note that although there has been significant research on how to choose the partitions, Theorems 1, 2 and 3 can be applied independent of these details. We only require that the data structure precomputes answers for each member of a family of canonical subsets and answers queries by a combination of these precomputed answers. We simply replace the precomputed answe in the standard version of the data structure for each canonical subset with the appropriate SDR or SDC sketching data structure. Then, when answering a query, instead of summing the standard precomputed answers, we simply add the sketches of all the canonical subsets in each query range, and then subtract the two resulting sketches.

7.3 Stabbing Queries

Given a set S of line segments in the plane and a vertical query line, we want to report or count the line segments which intersect the query line. This is known as a vertical line stabbing query. One common solution to this problem is the segment tree. A segment tree is an augmented binary tree, similar to the range tree. Each leaf represents a small range containing a single endpoint of one line segment, and the range r_v of any interior node v is the union of the ranges of its descendant leaves. Each segment $s = (x_s, y_s), (x_e, y_e)$ corresponds to the range $r(s) = [x_s, x_e]$. Let $p(v)$ denote the parent of v. We store a segment s in each node v such that $r(s)$ spans r_v but does not span $r_{p(v)}$. To answer a query we search in the segment tree for the x coordinate of the vertical query line r. This root to leaf path in the tree defines a $O(\log n)$ size sub-family of canonical subsets of S, which is exactly those canonical subsets which cover the query point x.

Hence, the segment tree is a canonical semigroup range searching data structure Thus, given a segment tree for vertical line stabbing (counting) queries we can apply Theorems 1, 2 or 3 to transform it into a data structure for set-difference stabbing queries. The details are similar to the previous range searching data structures.

7.4 Partial Sum Queries

Let A be a d-dimensional array for constant d. A partial sum query has as input a pair of d-dimensional indices $([i_1, \ldots, i_d], [j_1, \ldots j_d])$ which define a d-dimensional (rectangular) range, and to answer the query we must return the sum of elements whose indices fall in that range. This is similar to the orthogonal range query problem except that the points in the set are restricted to a grid.

Although the following techniques extend well to any (constant) dimension, for clarity of exposition we describe the one and two-dimensional case. In the one dimensional case, we must preprocess an array of length n such that for any pair of indices $1 \leq i \leq j \leq n$ we can return the sum of elements with indices between i and j in constant time. In linear time we build an array B such that for any index k, $B[k] = \sum_{i=1}^{k} A[i]$. Then to answer a query $Q((i,j), A)$ we return $B[j] - B[i]$.

In the two dimensional case, we preprocess by initializing B with the values $B[k,\ell] = \sum_{i=1}^{k} \sum_{j=1}^{\ell} A[k,\ell]$. This also runs in linear time (with respect to the size of the input array A). To answer a query $Q(([i,j],[k,\ell]), A)$ we return $B[k,\ell] - B[i,\ell] - B[k,j] + B[i,j]$. See Figure 3 for an illustration.

In the above data structure, we can think of the array A as a set of grid cells, and each cell of B represents a canonical subset of grid cell of A. The data structure is a canonical group range searching structure which requires us to take set differences of canonical subsets. This is not a problem when the elements corresponding to the grid cells are drawn from a group, since we can just subtract the sum of the subsets which we want to subtract. We can apply Theorems 1, 2 or 3 to obtain an efficient structure for set-difference queries on a contiguous sub-array.

8 Conclusion and Open Problems

We have given a general framework for converting canonical range-searching data structures into data structures for answering set-difference range queries, with efficient time and space bounds.

Some interesting open problems include the following:

- Is it possible to extend the results of this paper to difference range queries on strings where distance is measured using Smith-Waterman (edit) distance?
- What are some time and space lower bounds for set-difference range queries?
- Can set-difference range queries be efficiently constructed for data structures, like BBD-trees [28] and BAR-trees [14, 15], which can be used to answer approximate range queries?

Acknowledgments

The authors' research was supported in part by NSF grant 0830403 and by the Office of Naval Research under grant N00014-08-1-1015.

insert(x):

for each $h_i(x)$ value, for $i = 1, \ldots, k$ **do**
  add 1 to $T[h_i(x)]$.count
  add x to $T[h_i(x)]$.idSum
  add $g(x)$ to $T[h_i(x)]$.gSum
end for

delete(x):

for each $h_i(x)$ value, for $i = 1, \ldots, k$ **do**
  subtract 1 from $T[h_i(x)]$.count
  subtract x from $T[h_i(x)]$.idSum
  subtract $g(x)$ from $T[h_i(x)]$.gSum
end for

isMember(x):

for each $h_i(x)$ value, for $i = 1, \ldots, k$ **do**
  if $T[h_i(x)]$.count $= 0$ **and** $T[h_i(x)]$.idSum $= 0$ **and** $T[h_i(x)]$.gSum $= 0$ **then**
    return false
  else if $T[h_i(x)]$.count $= 1$ **and** $T[h_i(x)]$.idSum $= x$ **and** $T[h_i(x)]$.gSum $= g(x)$ **then**
    return true
  end if
end for
return "not determined"

subtract(A, B, C):

for $i = 0$ to $t - 1$ **do**
  $T_C[i]$.count $= T_A[i]$.count $- T_B[i]$.count
  $T_C[i]$.idSum $= T_A[i]$.idSum $- T_B[i]$.idSum
  $T_C[i]$.gSum $= T_A[i]$.gSum $- T_B[i]$.gSum
end for

listItems():

while there is an $i \in [1, t]$ such that $T[i]$.count $= 1$ **or** $T[i]$.count $= -1$ **do**
  if $T[h_i(x)]$.count $= 1$ **and** $T[h_i(x)]$.gSum $= g(T[h_i(x)]$.idSum$)$ **then**
    add the item, $(T[i]$.idSum$)$, to the "positive" output list
    call delete($T[i]$.idSum)
  else if $T[h_i(x)]$.count $= -1$ **and** $-T[h_i(x)]$.gSum $= g(-T[h_i(x)]$.idSum$)$ **then**
    add the item, $(-T[i]$.idSum$)$, to the "negative" output list
    call insert($-T[i]$.idSum)
  end if
end while

Figure 2: Operations supported by an invertible Bloom filter.

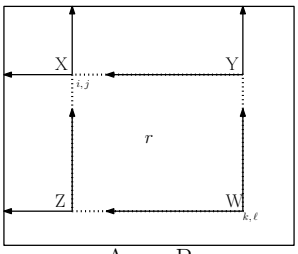


Figure 3: A data structure for partial sum queries when subtraction is allowed. The prefix sums required to answer any partial sum query can be precomputed in constant time per array element. Given these precomputed canonical subsets, we can answer a partial sum query in constant time (assuming d is constant). We illustrate this concept in the case where $d = 2$. The region X is all the elements of B with indices $< i, < j$. Y is all elements of B with indices $< i, < k$. Z is all elements with indices $< k, < j$. Z is all elements with indices $\leq k, \leq \ell$. The query rectangle r is all elements between indices i, j and k, ℓ inclusive. Thus $\sum w(r) = \sum w(W) - \sum w(Z) - \sum w(Y) + \sum w(X)$.

References

[1] P. K. Agarwal. Range Searching. *Handbook of Discrete and Computational Geometry*, pp. 575–598. CRC Press, Inc., 1997.

[2] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. *J. Comput. Syst. Sci.* 66(1):207–243, February 2003, doi:10.1016/S0022-0000(02)00035-1.

[3] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.* 58(1):137–147, 1999.

[4] P. Baldi and D. S. Hirschberg. An intersection inequality sharper than the Tanimoto triangle inequality for efficiently searching large databases. *J. Chemical Information and Modeling* 49(8):1866–1870, 2009, doi:10.1021/ci900133j.

[5] M. Basseville. Detecting changes in signals and systems—A survey. *Automatica* 24(3):309–326, 1988, doi:10.1016/0005-1098(88)90073-8.

[6] J. L. Bentley. Solution to Klee's Rectangle Problem. Unpublished manuscript, 1977.

[7] J. L. Bentley and J. B. Saxe. Decomposable searching problems I. Static-to-dynamic transformation. *J. Algorithms* 1(4):301–358, 1980, doi:10.1016/0196-6774(80)90015-2.

[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13(7):422–426, 1970, doi:10.1145/362686.362692.

[9] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting Bloom filters. *Proc. 14th Eur. Symp. on Algorithms*, pp. 684–695. Springer-Verlag, LNCS 4168, 2006, doi:10.1007/11841036_61.

[10] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *ICALP*, pp. 693-703. Springer, Lecture Notes in Computer Science 2380, 2002.

[11] M. S. Charikar. Similarity estimation techniques from rounding algorithms. *Proc. 34th ACM Symp. on Theory of Computing*, pp. 380–388, 2002, doi:10.1145/509907.509965.

[12] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms* 55(1):58–75, April 2005, doi:10.1016/j.jalgor.2003.12.001.

[13] L. R. Dice. Measures of the amount of ecologic association between species. *Ecology* 26(3):297–302, July 1945, doi:10.2307/1932409.

[14] C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Balanced aspect ratio trees and their use for drawing very large graphs. *J. Graph Algorithms and Applications* 4(3):19–46, 2000, `http://jgaa.info/accepted/00/Duncan+00.4.3.pdf`.

[15] C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Balanced aspect ratio trees: combining the benefits of k-d trees and octrees. *J. Algorithms* 38:303–333, 2001, doi:10.1006/jagm.2000.1135.

[16] D. Eppstein and M. T. Goodrich. Straggler Identification in Round-Trip Data Streams via Newton's Identities and Invertible Bloom Filters. *IEEE Trans. on Knowledge and Data Engineering* 23:297–306, 2011, doi:10.1109/TKDE.2010.132.

[17] D. Eppstein, M. T. Goodrich, F. Uyeda, and G. Varghese. What's the difference? Efficient set reconciliation without prior context. *Proc. SIGCOMM*, 2011, `http://www.ics.uci.edu/~eppstein/pubs/EppGooUye-SIGCOMM-11.pdf`.

[18] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.* 8(3):281–293, 2000, doi:10.1109/90.851975.

[19] P. B. Gibbons and Y. Matias. Synopsis data structures for massive data sets. *SODA*, pp. 909-910. ACM/SIAM, 1999.

[20] L. Hamers, Y. Hemeryck, G. Herweyers, M. Janssen, H. Keters, R. Rousseau, and A. Vanhoutte. Similarity measures in scientometric research: The Jaccard index versus Salton's cosine formula. *Information Processing & Management* 25(3):315–318, 1989, doi:10.1016/0306-4573(89)90048-4.

[21] D. M. Kane and J. Nelson. Sparser johnson-lindenstrauss transforms. *SODA*, pp. 1195-1206. SIAM, 2012.

[22] D. M. Kane, J. Nelson, E. Porat, and D. P. Woodruff. Fast moment estimation in data streams in optimal space. *43rd ACM Symp. on Theory of Computing (STOC)*, pp. 745–754, 2011.

[23] A. R. Leach and V. J. Gillet. Similarity Methods. *An Introduction To Chemoinformatics*, pp. 99–117. Springer-Verlag, 2007, doi:10.1007/978-1-4020-6291-9_5.

[24] A. Lipkus. A proof of the triangle inequality for the Tanimoto distance. *J. Mathematical Chemistry* 26(1):263–265, 1999, doi:10.1023/A:1019154432472.

[25] G. S. Lueker. A data structure for orthogonal range queries. *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pp. 28–34. IEEE Computer Society, 1978, doi:10.1109/SFCS.1978.1.

[26] J. Matoušek. Efficient partition trees. *Discrete Comput. Geom.* 8(3):315–334, October 1992, doi:10.1007/BF02293051.

[27] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-temporal access methods. *IEEE Data Engineering Bulletin* 26:40–49, 2003.

[28] D. M. Mount and E. Park. A dynamic data structure for approximate range searching. *Proc. ACM Symp. on Computational Geometry (SoCG)*, pp. 247–256, 2010, doi:10.1145/1810959.1811002.

[29] R. Radke, S. Andra, O. Al-Kofahi, and B. Roysam. Image change detection algorithms: a systematic survey. *IEEE Trans. Image Processing* 14(3):294–307, March 2005, doi:10.1109/TIP.2004.838698.

[30] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 331–342, 2000, doi:10.1145/342009.335427.

[31] Q. Shi and J. JaJa. A new framework for addressing temporal range queries and some preliminary results. *Theor. Comput. Sci.* 332(1-3):109–121, February 2005, doi:10.1016/j.tcs.2004.10.013.

[32] S. Suri, C. D. Tóth, and Y. Zhou. Range counting over multidimensional data streams. *Discrete Comput. Geom.* 36(4):633–655, 2006, doi:10.1007/s00454-006-1269-4.

[33] S. J. Swamidass and P. Baldi. Bounds and algorithms for fast exact searches of chemical fingerprints in linear and sublinear time. *J. Chemical Information and Modeling* 47(2):302–317, 2007, doi:10.1021/ci600358f.

[34] M. Thorup and Y. Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. *SODA*, pp. 615-624. SIAM, 2004.

[35] A. Tversky. Features of similarity. *Psychological Review* 84(4):327–352, 1977, doi:10.1037/0033-295X.84.4.327.

[36] D. G. York, J. Adelman, J. John E. Anderson, S. F. Anderson, et al. The Sloan digital sky survey: technical summary. *The Astronomical Journal* 120(3):1579, 2000, `http://stacks.iop.org/1538-3881/120/i=3/a=1579`.