Answering Spatial Multiple-Set Intersection Queries Using 2-3 Cuckoo Hash-Filters

Michael T. Goodrich
Department of Computer Science, Univ. of California, Irvine
goodrich@uci.edu

ABSTRACT

We show how to answer spatial multiple-set intersection queries in $O(n(\log w)/w + kt)$ expected time, where n is the total size of the $t \leq w^c$ sets involved in the query, w is the number of bits in a memory word, k is the output size, and $c \geq 1$ is any fixed constant.

CCS CONCEPTS

• **Theory of computation → Parallel computing models**;

KEYWORDS

set intersection, 2-3 cuckoo filters, range searching, GIS

1 INTRODUCTION

In this paper, we are interested in asymptotic improvements to spatial multiple-set intersection queries by taking advantage of bit-level parallelism, in a computational model known as the ***practical RAM*** model [11] or ***word-RAM*** model [8]. By "bit-level parallelism," we are referring to an ability to compute bit-parallel operations on pairs of binary words of w bits in constant time, e.g., using operations built into modern CPUs and GPUs. There are actually different versions of the word-RAM model (e.g., see [8, 9]). For example, we can consider a ***restricted word-RAM*** model [8], where bit-parallel operations are limited to addition, subtraction, and the bit-parallel operations AND, OR, NOT, XOR, shift, and MSB (most-significant set bit). In this paper, we provide results for the restricted word-RAM model and also for a ***permutation word-RAM*** model (e.g., see [1]), which can perform a fixed permutation of w/m subwords, each of size m, in constant time. Formally, we assume we have a computational setting that consists of the following items:

- A collection of data sets, $D_1, D_2, \ldots$, that store ***items***, such that each item is associated with a point along a one-dimensional curve, C, which is the same for all the data sets. In the simplest case, the curve, C, is just a straight line, but we also allow C to be a space-filling curve, such as a Hilbert curve or z-order curve.
- A ***spatial multple-set intersection query*** consists of an interval range, $\mathcal{R} \subseteq C$, and a set of indices, $\mathcal{I} = \{i_1, i_2, \ldots, i_t\}$. Each i_j identifies a specific data set, D_{i_j}, e.g., based on some keyword of interest. The response to this query should be every item that has a point in the interval range, $\mathcal{R}$, and belongs to the common intersection, $D_{i_1} \cap D_{i_2} \cap \cdots \cap D_{i_t}$.

SIGSPATIAL'17, November 7–10, 2017, Los Angeles Area, CA, USA
© 2017 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5490-5/17/11.
https://doi.org/10.1145/3139958.3140021

Related Work. In addition to work by Miltersen [11] introducing the practical RAM model and work by Hagerup [8] introducing the word-RAM model, researchers have explored various algorithms for versions of the practical RAM and word-RAM models, e.g., see [12]. There is, for instance, considerable previous work on algorithms for answering set-intersection queries in the word-RAM model. Ding and König [3] show how to compute the common intersection of t sets of total size n in expected time $O(n/\sqrt{w} + kt)$, where w is the word size in bits and k is the size of the output in words. Bille *et al.* [2] present a data structure that can compute the intersection of t sets of total size n in $O(n(\log^2 w)/w + kt)$ expected time in the permutation word-RAM model. In addition, Kopelowitz *et al.* [10] introduce a data structure for computing set intersections for two sets of roughly the same size, n, in $O(n(\log^2 w)/w + \log w + k)$ expected time in this model. Eppstein *et al.* [6] improve this bound to $O(n(\log w)/w + k)$ expected time for the restricted word-RAM model, using a data structure they call "2-3 cuckoo hash-filters," which consist of a combination of 2-3 cuckoo hash tables and 2-3 cuckoo filters. Eppstein and Goodrich [5] extend this contruction for pairs of sets of different sizes. Unfortunately, all of these previous uses of 2-3 cuckoo hash-filters are limited to pairwise intersections of sets and they do not extend to intersections of three or more sets or spatial queries. Thus, the best previous algorithm for answering (standard) multiple-set intersection queries in the word-RAM model is due to Bille *et al.*, as mentioned above. We are not familiar with any previous work in the word-RAM model for answering spatial multiple-set intersection queries.

Our Results. We show how to answer spatial multiple-set intersection queries in $O(n(\log w)/w + kt)$ expected time in the permutation word-RAM model, or $O(n(\log^2 w)/w + kt)$ expected time in the restricted word-RAM model, where n is the total size of the $t \leq w^c$ location-constrained subsets involved in the query, where $c \geq 1$ is a fixed constant and k is the output size.

Our data structures and algorithms take advantage of a simple approach that exploits bit-level parallelism by packing information into memory subwords, which allows us to represent small sets of size $O(w/\log w)$ with 2-3 cuckoo filters occupying $O(1)$ memory words, and intersect pairs of sets represented this way in expected time $O(1 + k)$, where k is the output size. The main computational difficulty of using this approach is that the result of such an operation is itself ***not*** a 2-3 cuckoo filter. Nevertheless, unlike previous results that exploited 2-3 cuckoo filters for intersection queries [5, 6], we show in this paper how to restore the result of a pairwise intersection query back to being a 2-3 cuckoo filter. We then show that such restoration operations allow us to achieve our results for spatial multiple-set intersection queries. For full details and proofs, please see the full version of this paper [7].

2 A REVIEW OF 2-3 CUCKOO HASH-FILTERS

Let us review the 2-3 cuckoo hash-filter data structure of Eppstein *et al.* [6]. Suppose we wish to represent a set, S, of n items taken from a universe such that each item can be stored in a single memory word. We use the following components, T, M, C, and F.

- A hash table T of size $O(n)$, using three pseudo-random hash functions h_1, h_2, and h_3, which map items of S to triples of distinct integers in the range $[0, n-1]$. Each item, x in S, is stored, if possible, in two of the three possible locations for x based on these hash functions. We refer to each T as a 2-3 ***cuckoo hash table***.
- We also store a stash cache, C, of size λ, where λ is bounded by a constant. C stores items for which it was not possible to store properly in two distinct locations in T. We maintain each C as an array of size $O(\lambda)$.
- A table, F, having $O(n)$ cells, that parallels T, so that $F[j]$ stores a non-zero fingerprint digest, $f(x)$, for an item, x, if and only if $T[j]$ stores a copy of x. The digest $f(x)$ is a non-zero random hash function comprising δ bits, where $\delta = c \log w$ is a parameter chosen to achieve a small false positive rate. The table F is called a 2-3 ***cuckoo filter***, and it is stored in a packed format, so that we store $O(w/\log w)$ cells of F per memory word. In addition to the vector F, we store a bit-mask, M, that is the same size as F and has all 1 bits in the corresponding cell of each occupied cell of F.

One can show that one can build a 2-3 cuckoo hash-table of size, n, with a stash of size s, in expected $O(n)$ time and that the probability that this construction fails is at most $\tilde{O}(n^{-s})$, where the $\tilde{O}(\cdot)$ notation ignores polylogarithmic factors [6].

3 DATA STRUCTURES

As a starting point for our data structure construction, we subdivide each data set, D_i, into a sequence of interval regions along the curve, C, such that each region stores $\Theta(w/\log w)$ points. Our data structure construction, then, is as follows. For each interval region, R, in one of our structures, D_i, use the algorithm given above to construct a 2-3 cuckoo hash-filter for the $\Theta(w/\log w)$ points in R with a stash of constant size, λ. If the construction for R fails, then simply fallback to representing this subset as a sorted listing of its items (according to some canonical ordering). The resulting set of interval regions and the 2-3 cuckoo hash-filter or sorted list for each region comprises our representation for this structure. In spite of this failure possibility (which amounts to our using a standard list-based subset representation as a fallback), our construction has the following property.

Lemma 3.1. *Let S and T be sets of consecutive interval regions from two data sets, D_i and D_j, and let n denote the number of points in all the regions of S and T. Let α denote the number of interval region pairs, (R_i, R_j), such that R_i and R_j overlap and our 2-3 cuckoo hash-filter construction failed for either R_i or R_j. Then $\mathbf{E}[\alpha] \le \frac{2n \log w}{w^{s+1}}$, where $s \ge 2$ is a chosen constant.*

That is, the total expected number of points summed across all pairs of overlapping regions where one of the regions has a failed 2-3 cuckoo hash-filter is $O(n/w^s)$.

4 INTERSECTION ALGORITHMS

In this section, we describe our algorithm for performing spatial multiple-set intersection queries.

Intersecting Two 2-3 Cuckoo Hash-Filters. Let us begin by reviewing the method of Eppstein *et al.* [6] for intersecting a pair of 2-3 cuckoo hash-filters, which in our case will always be represented using $O(1)$ memory words for the filter component, since interval regions in our structures hold $\Theta(w/\log w)$ items.

Suppose then that we have two subsets, S_i, and S_j, of size $O(w/\log w)$ each for which we wish to compute a representation of the intersection $S_i \cap S_j$. Suppose further that S_i and S_j are each represented with 2-3 cuckoo hash-filters of the same size and using the same three hash functions and fingerprint function. We begin our set-intersection algorithm for this pair of 2-3 cuckoo hash-filters by computing a vector of $O(1)$ words that identifies the matching non-empty cells in F_i and F_j. For example, we could compute the vector defined by the following bit-wise vector expression:

$$A = (M_i \text{ AND NOT } (F_i \text{ XOR } F_j)). \tag{1}$$

We view A as being a parallel vector to F_i and F_j. Note that a cell, $A[r]$, consists of δ bits and this cell is all 1s if and only if $F_i[r]$ stores a fingerprint digest for some item and $F_i[r] = F_j[r]$, since fingerprint digests are non-zero. Thus, with standard operations in the word-RAM we can compute a compact representation of $O(1)$ words that stores each subword for each fingerprint that matches in F_i and F_j, with the matching locations identified by all 1's in a mask, M_i'. The crucial insight is that since each item is stored in two-out-of-three locations in a 2-3 cuckoo hash-filter, if the same item is stored in two different cuckoo hash-filters, then it will be stored in one of its three locations in both hash-filters. It is this common location that then stores the filter for this item after we perform the bit-parallel operations to compute this intersection.

Thus, if we are interested in just this pairwise intersection, we can create a list, L, of members of the common intersection of S_i and S_j, by visiting each word of A and storing to L the item in $T_i[r]$ corresponding to each cell, $A[r]$, that is all 1s, but doing so only after confirming that $T_i[r] = T_j[r]$. In addition, since we are assuming that stashes are of constant size, we can do a lookup in the other hash table for each item in a stash for S_i or S_j, which takes an additional time that is $O(1)$. Therefore, the listing of the members in $S_i \cap S_j$ can be done in time $O(1 + k + p)$, where k is the number of items in the intersection and p is the number of false positives (i.e., places where fingerprints match but the item is not actually in the common intersection), by using standard and bit-parallel operations in the word-RAM model (e.g., see also [4]).

Note that the additional step of weeding out false positives can be done at the end, which involves a constant-time operation per item in the list, that itself involves a lookup in the two cuckoo hash tables, to remove items that map to the same locations and have the same fingerprint digests but are nevertheless different items. Note that by requiring $\delta = c \log w$, we can guarantee that the probability of such false positives is at most $1/w^c$.

The running time of our entire algorithm for computing the intersection of S_i and S_j, therefore, is $O(1 + k + p)$, where k is the size of the output and p is the number of false positives. In addition, by choosing $\delta = c \log w$, we can bound the probability that two different items have the same fingerprint value as being at most $1/w^c$. Thus, $\mathbf{E}[p] \leq n/w$, since each item is stored in at most two places in a 2-3 cuckoo hash table.

Answering Pairwise Spatial Set-Intersection Queries. Let us next describe our algorithm for answering a spatial two-set intersection query, asking for the intersection of two sets, S_1 and S_2, of possibly different sizes. Let us assume that S_1 and S_2 are each represented using the structures as described above, that is, S_1 and S_2 are subdivided into interval regions, such that, for each interval region, we have done our construction of a 2-3 cuckoo hash-filter (or, if that failed, then we have a sorted list of the items in that interval). In addition, we assume that we also are representing each entire set, S_i, for $i = 1, 2$, using a standard hash table, H_i, such as a cuckoo hash table. This hash table, H_i, will allow us to cull false positives as a post-processing step.

Given a query interval range, $\mathcal{R} \subseteq C$, we first cull from S_1 and S_2 all the intervals that do not intersect $\mathcal{R}$. To allow for easier analysis of our method for computing the intersection of S_1 and S_2, let n_1 denote the number of items remaining in S_1 and let n_2 denote the number of items remaining in S_2, and let $n = n_1 + n_2$. Note that, since the number if items in each remaining interval region in either S_1 or S_2 is $O(w/\log w)$, the number of intervals in each S_i is $O(n_i(\log w)/w)$, for $i = 1, 2$. In addition, we assume that we are representing these interval regions so that have an easy way of identifying when two regions overlap.

The goal of our algorithm is to compute a cuckoo hash-filter representation that contains all the items in $S_1 \cap S_2$, plus possibly some false positives that our algorithm identifies as high-probability items belonging to this common intersection.

Because of the way our algorithm works, it produces a cuckoo-filter representation for the common intersection, where $S_1 \cap S_2$ is represented in terms of the intervals in S_2 (or, alternatively, we can swap the two sets and our output will be in terms of the intervals of S_1). Namely, for each such interval, I, we will have a cuckoo filter, F_I, and a backing 2-3 cuckoo table, T_I, where, for each non-zero fingerprint, $F_I[j]$, there is a corresponding item, $x = T_I[j]$, with x being a confirmed item in S_2. The cuckoo filter, F_I, may not be a 2-3 cuckoo filter, however, because each item might be stored in just one location in F_i, not two. (We explain later how we can repair this situation to quickly build a 2-3 cuckoo filter representing the pairwise intersection, albeit possibly with a small number of false positives that can be removed in a post-processing step.) Nevertheless, after our core algorithm completes, our representation allows us to examine each non-zero cuckoo filter fingerprint in a filter F_I, lookup its corresponding item, x, in a backing 2-3 cuckoo table, T_I, and then perform a search for x in the hash table for the other set (e.g., S_1) to verify that it belongs to the common intersection or is a false positive. That is, after one additional lookup for each such candidate intersection item, x, we can confirm or discard x depending on whether it is or isn't in the common intersection and with one more comparison.

There is also a possibility that our construction of a 2-3 cuckoo hash-filter fails for some interval, in which case we fallback to a standard intersection algorithm, such as merging two sorted lists, or looking up each item in one set in a hash table for the other. Since such failures occur with probability at most $1/w^s$, for some constant s, the time spent on such fallback computations is dominated by the time for our other steps; hence, let us ignore the time spent on such fallback computations.

Our algorithm for constructing the representation of $S_1 \cap S_2$, using 2-3 cuckoo hash-filters, and then optionally culling out false positives, is as follows.

(1) Merge the interval regions of S_1 and S_2, to identify each pair of overlapping interval regions. This step can be done in $O(n(\log w)/w)$ time, by Lemma 3.1.
(2) For each overlapping interval, $I_{1,j}$ and $I_{2,k}$, where $I_{1,j}$ is from S_1 and $I_{2,k}$ is from S_2, intersect these two subsets using the bit-parallel intersection algorithm for 2-3 cuckoo filters derived using Equation 1 above (but skipping the lookups in the corresponding 2-3 cuckoo table) Also perform lookups for any items in stashes (which are confirmed as intersections; hence, we are done with our computations for them). Let $F_{j,k}$ denote the resulting (now partial) 2-3 cuckoo filter. This step can be implemented in $O(n(\log w)/w)$ time, since the total number of overlapping pairs of intervals is $O(n(\log w)/w)$.
(3) For each interval, $I_{1,j}$, collect all the partial 2-3 cuckoo hash-filters, $F_{j,k}$, computed in the previous step for $I_{1,j}$. Compute the bit-wise OR of these filters. Let F_j denote the resulting partial 2-3 cuckoo filter for $I_{1,j}$, and let $\mathbf{F}$ denote the collection of all such filters (note that there is potentially a non-empty partial cuckoo filter, F_j, for each interval in S_1). Also note that there are no collisions in the bit-wise OR of all these cuckoo filters, since each is computed as an intersection of a disjoint set of other items with the items in this interval. This step can be implemented in $O(n(\log w)/w)$ time, since the total number of overlapping pairs of intervals is $O(n(\log w)/w)$.
(4) For each interval $I_{1,j}$, and each item, x, in the 2-3 cuckoo hash table for $I_{1,j}$ that has a corresponding fingerprint belonging to F_j in $\mathbf{F}$, do a lookup in each of corresponding backing hash tables (we assume we have global lookup tables for S_1 and S_2) to determine if x is indeed a common item in the sets S_1 and S_2. Let Z denote the set of all such items so determined to belong to this common intersection. This step can be implemented in $O(n(\log w)/w + k + p)$ time, where k is the size of the output and p is the number of false positives, that is, items that have a non-zero fingerprint in some F_j but nevertheless are not in the common intersection, $S_1 \cap S_2$.
(5) Output the members of the set, Z, as the answer.

Let us analyze the running time of this algorithm. We have already accounted for the time bounds for each step above. Furthermore, note that the parameter p is $O(n/w)$, since the probability of two fingerprints of size $2\log w$ collide is at most $1/w$. Therefore, the total expected time for our algorithm is $O(n(\log w)/w + k)$.

Answering Spatial Multiple-Set Intersection Queries. The main bottleneck for extending our method from the previous section to spatial multiple-set intersection queries is that the partial result of performing the bit-parallel intersection of a pair of 2-3 cuckoo filters is itself not a 2-3 cuckoo filter, because after the intersection computation is performed some fingerprints might be stored in just one location, not two. We can assume, however, that every subword in a fingerprint vector, F, is either all 0's or it holds a complete fingerprint of $O(\log w)$ bits.

We can to augment our 2-3 cuckoo filter representations so that we can restore them to be proper 2-3 cuckoo filters even after a pairwise intersection computation, as follows:

(1) For each fingerprint vector, F, create and store ***cuckoo-restore*** information, which encodes a permutation, π_F, which routes every subword fingerprint, f, in F to the location of its other location, which we call f's ***twin***. That is, if the fingerprint f for some item x is stored at subword locations i and j in F, then π_F moves the copy of f at i to position j and the copy of f at j to position i. This step can be done in $O(1)$ time in the permutation word-RAM model.
(2) After an intersection operation occurs, so that each item in the intersection of a pair of 2-3 cuckoo filters (including some possible false positives) may be is stored in a fingerprint vector, F, in just one location instead of two, apply π_F to create a copy, F' of F such that each fingerprint subword in F is routed to its twin location.
(3) Restore F to be a 2-3 cuckoo filter by computing the bit-wise OR of F and F'. Note that there can be no collisions occurring as a result of this bit-wise OR, because every fingerprint (even the false positives) is the same as its twin and all empty locations are all 0's. Thus, this bit-wise OR will copy each surviving fingerprint to its twin location and then OR this fingerprint with itself (causing no change) or with a subword of all 0's (restoring the fingerprint to its original two locations).

Thus, to answer a spatial multiple-set intersection query, for sets, $S_1, S_2, \ldots, S_t$, we perform the above pairwise intersection operations iteratively, first with S_1 and S_2, and then with the result of this intersection with S_3, and so on, postponing until the very end our culling of false positives by doing a lookup in global hash tables for $S_1, S_2, \ldots, S_t$, which we assume we have available, to check for each item x in the final 2-3 cuckoo hash-table whether x is indeed a member of every set. Each intersection step, for an iteration i, takes $O(1)$ time per cuckoo-filter, in the permutation word-RAM model, for which we can charge this cost in iteration i to the size of the subset in a region of the set S_i.

We can force the expected total number of false positives to be at most n/w^{c+1}, for a fixed constant $c \geq 1$, by defining the fingerprints to have size at least $(c+1)\log w$. Thus, testing each surving candidate at the end to see if it really belongs to the common intersection, by looking up each such element x in the t hash tables for each set, takes time $O(kt)$ plus a term that is dominated by $O(n/w)$.

Note that in order to implement our algorithm in the restricted word-RAM model, the only part of this computation left as of yet unspecified is how to create a representation of the permutation, π_F, and perform the routing of subwords defined by π_F. For this part, we follow the approach of Yang *et al.* [13], who define subword permutation micro-code instructions and show how to implement them using a double butterfly network, which is also known as a Benes network.

It is known that a Benes network can route any permutation and that it is fairly straightforward to set the switches in a Benes network for this purpose for any given permutation. So let N be such a network, which in the case of routing $O(w/\log w)$ subwords of size $O(\log w)$ will have depth $O(\log w)$. Depending on how the switches are set, we note that each stage of N involves keeping some subword in place and moving others by shifting them all the same distance. Thus, each stage of a Benes network can be implemented in $O(1)$ steps in the word-RAM model by simple applications of AND, OR, and shift operations (plus either the use of mask vectors to encode the switch settings or using other built-in word-RAM operations based on an encoding of π_F. Thus, we can store an encoding of a Benes network implementing π_F for each fingerprint vector, F, as our cuckoo-restore information, and this will allow us to restore any cuckoo filter to be a 2-3 cuckoo filter in $O(\log w)$ time in the word-RAM model. This gives us the following result.

THEOREM 4.1. *Suppose $t \leq w^c$ sets are from interval regions identified through a spatial multiple-set intersection query for a range $\mathcal{R} \subseteq C$, in structures constructed as described above. We can compute the result of such a query in $O(n(\log w)/w + kt)$ expected time in the permutation word-RAM model, or $O(n(\log^2 w)/w + kt)$ expected time in the restricted word-RAM model, where n is the total size of all the sets involved, k is the size of the output, and $c \geq 1$ is a constant.*

Acknowledgments. This article reports on work supported by the DARPA under agreement no. AFRL FA8750-15-2-0092. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. This work was also supported in part from NSF grants 1228639, 1526631, 1217322, 1618301, and 1616248. We thank David Eppstein for several helpful discussions.

REFERENCES

[1] Susanne Albers and Torben Hagerup. 1997. Improved Parallel Integer Sorting without Concurrent Writing. *Inf. & Comput.* 136, 1 (1997), 25–51.
[2] Philip Bille, Anna Pagh, and Rasmus Pagh. 2007. Fast evaluation of union-intersection expressions. In *Int. Symp. Algorithms and Computation (LNCS)*, Vol. 4835. Springer, 739–750.
[3] Bolin Ding and Arnd Christian König. 2011. Fast Set Intersection in Memory. *Proc. VLDB Endow.* 4, 4 (Jan. 2011), 255–266.
[4] David Eppstein. 2016. Cuckoo Filter: Simplification and Analysis. In *15th Scand. Workshop on Algorithm Theory (LIPIcs)*, Vol. 53. 8:1–8:12.
[5] David Eppstein and Michael T. Goodrich. 2017. Brief Announcement: Using Multi-Level Parallelism and 2-3 Cuckoo Filters for Set Intersection Queries and Sparse Boolean Matrix Multiplication. In *29th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*.
[6] David Eppstein, Michael T. Goodrich, Michael Mitzenmacher, and Manuel R. Torres. 2017. 2-3 Cuckoo Filters for Faster Triangle Listing and Set Intersection. In *36th ACM Symp. on Principles of Database Systems*. 247–260.
[7] Michael T. Goodrich. 2017. Answering Spatial Multiple-Set Intersection Queries Using 2-3 Cuckoo Hash-Filters. *ArXiv ePrint* 1708.09059 (2017). arxiv.org/abs/1708.09059
[8] Torben Hagerup. 1998. Sorting and searching on the word RAM. In *15th Symp. on Theor. Aspects of Comp. Sci.* 366–398.
[9] Torben Hagerup. 2000. Improved Shortest Paths on the Word RAM. In *27th Int. Colloqium on Automata, Languages and Programming*. Springer, 61–72.
[10] Tsvi Kopelowitz, Seth Pettie, and Ely Porat. 2015. Dynamic Set Intersection. In *14th Symp. on Algorithms and Data Structures (SODA)*. 470–481.
[11] Peter Bro Miltersen. 1996. Lower bounds for static dictionaries on RAMs with bit operations but no multiplication. In *23rd Int. Colloq. on Automata, Languages and Programming (ICALP) (LNCS)*, Vol. 1099. Springer, 442–453.
[12] Dan E. Willard. 2000. Examining Computational Geometry, Van Emde Boas Trees, and Hashing from the Perspective of the Fusion Tree. *SIAM J. Comput.* 29, 3 (2000), 1030–1049.
[13] Xiao Yang, Manish Vachharajani, and Ruby B. Lee. 1999. Fast subword permutation instructions based on butterfly network. In *Proc. SPIE*, Vol. 3970. 80–86.