# Dashing 2: genomic sketching with multiplicities and locality-sensitive hashing

Daniel N. Baker[1] and Ben Langmead[1,*]

[1]Department of Computer Science, Johns Hopkins University
[*]*corresponding author;* langmea@cs.jhu.edu

October 17, 2022

## Abstract

A genomic sketch is a small, probabilistic representation of the set of k-mers in a sequencing dataset. Sketches are building blocks for large-scale analyses that consider similarities between many pairs of sequences or sequence collections. While existing tools can easily compare 10,000s of genomes, relevant datasets can reach millions of sequences and beyond. Popular tools also fail to consider $k$-mer multiplicities, making them less applicable in quantitative settings. We describe a method called Dashing 2 that builds on the SetSketch data structure. SetSketch is related to HyperLogLog, but discards use of leading zero count in favor of a truncated logarithm of adjustable base. Unlike HLL, SetSketch can perform multiplicity-aware sketching when combined with the ProbMinHash method. Dashing 2 integrates locality-sensitive hashing to scale all-pairs comparisons to millions of sequences. Dashing 2 is free, open source software available at https://github.com/dnbaker/dashing2

## 1 Introduction

Sketching, e.g. based on MinHash or HyperLogLog, is a key building block for scaling sequence comparison. Sketches built over all the k-mers in a sequence have been applied in clustering [1], phylogenetic inference [2], strain-level profiling [3, 4], species delineation [5] and summarization of genomic collections [6, 7]. While existing tools like Mash [1] and Dashing [7] can easily cluster 10,000s of genomes, many relevant biological datasets are much larger, reaching millions of sequences and beyond. Further, these tools fail to consider multiplicities of the k-mers, limiting their applicability in settings where quantities matter, e.g. when analyzing collections of sequence reads, or summaries from quantitative sequencing assays.

Dashing 2 builds on the recent SetSketch structure [8]. SetSketch is related to HyperLogLog (HLL), but replaces the HLL's leading zero count (LZC) operation with a truncated logarithm of adjustable base. This addresses a major disadvantage of the HLL as implemented in Dashing, since the LZC wastes about 2 bits of space out of every 8-bit estimator ("register") stored. SetSketch also has similarities to multiplicity-aware approaches like BagMinHash [9] and ProbMinHash [10]. All three approaches (SetSketch, BagMinHash and ProbMinHash) make decisions about whether and how to update registers by performing a random draw from a distribution, where the draw is seeded by a hash value derived from the input item. This allows SetSketch to perform multiplicity-aware sketching in the same way as the other sketches. SetSketch also admits a simple, accurate algorithm for computing similarity between sketches in a joint fashion.

Dashing 2 uses locality-sensitive hashing (LSH) to scale all-pairs comparisons to very large inputs. It finds near neighbors by grouping samples with equal register groupings. This makes Dashing 2 particularly effective for all-pairs comparisons over large sequence collections.

Dashing 2's implementation of the SetSketch structure is efficient and versatile. Like the original Dashing software, Dashing 2 can be run in a mode that sketches a sequencing dataset and saves the result to a file. This is activated by the `dashing2 sketch` command. Also like Dashing, Dashing 2 can compare

sequences or sketches in an all-pairs fashion (`dashing2 cmp` or, equivalently, `dashing2 dist`). When combined with the `--cache` option, Dashing 2 loads pre-existing sketches from disk, making the command much faster. When the input to these commands consists of many sketches or datasets, Dashing 2 performs all-pairs comparisons and outputs tabular results. Dashing 2's new LSH-assisted all-pairs comparison mode can be activated via the `--similarity-threshold` $x$ option, where e.g. $x = 0.8$ instructs Dashing 2 to use an LSH approach to consider only pairs whose similarity is likely to be 80% or higher.

Dashing 2 supports a range of sequence alphabets, including the 2-bit DNA alphabet and a standard 20-letter amino acid protein alphabet (using `--protein` option) and compressed amino acid alphabets of size 14, 8, and 6 (`--protein14`, `--protein8`, and `--protein6`) as described by [11]. Compressed protein alphabets are more appropriate when sequence identity is low.

Dashing 2's new multiplicity-aware sketching mode can be enabled for sequencing data inputs via the `--prob` option. Dashing 2 can sketch BigWig inputs [12] encoding numerical coverage vectors using the `--bigwig` option. The more generic `--wsketch` mode can sketch inputs consisting of keys and weights.

Besides the modes discussed above, Dashing 2 has modes for computing containment coefficients, symmetric containment coefficient, and intersection size. Further, Dashing 2 has modes for computing Jaccard coefficients in an exact manner, without sketching or estimation; this is useful for evaluation but comes at the expense of longer running time and larger memory footprint compared to sketching-based approaches.

# 2 Methods

## 2.1 SetSketch

Dashing [7] used the HyperLogLog data structure and its "leading zero count" (LZC) strategy to update register values. Specifically: an incoming item is hashed and its bitwise representation is partitioned into a prefix $p$ and a suffix $q$. The prefix $p$ determines which register the item maps to, while the suffix $q$ determines how the register should be updated. Specifically, the algorithm finds the number of consecutive unset bits in the most significant digits of $q$, i.e. its LZC. If the LZC is greater than the register's current value, the value is set to the LZC. The LZC serves as a kind of order-of-magnitude estimator; many such estimators can be averaged to accurately estimate cardinalities and similarities.

Whereas Dashing's HLL registers were each 8 bits wide and able to hold a value in the range 0–255, LZCs could range only from 0 to 64. In fact, LZCs could would usually span a smaller range than this, since bits used for the prefix $p$ are not considered. An LZC would therefore fail to use at least 2 bits of an 8-bit register, leaving the structure 25% empty. Though registers could be shrunk to 6 bits, this would conflict with Dashing's use of SIMD instructions with 8-bit operands.

Ertl's SetSketch [8] addresses this issue by replacing the LZC with a logarithm of configurable base $b$. This comes with a drawback: the addition of a single item to the SetSketch potentially updates the values of *all* registers, rather than just one. Given a data item $d$, the update rule for each register $K_i$, is:

$$K_i = \max(K_i, \lfloor 1 - \log_b h_i(d) \rfloor) \tag{1}$$

where $h_i$ is an independent hash function specific to register $i$ distributed exponentially; i.e. $h_i(d) \sim \text{Exp}(a)$.

For reasons explained below, it is useful to factor this update rule into two phases, with delaying the logarithms to the second phase. We use $K_i$ to denote the register's value at update time and $K'_i$ to denote its value after the final truncation.

$$K_i = \min(K_i, h_i(d)) \qquad K'_i = \lfloor 1 - \log_b K_i \rfloor \tag{2}$$

Where again $h_i(d) \sim \text{Exp}(a)$. The subtraction in the $K'_i$ formula inverts the notion of "extremeness" from a minimum to a maximum, hence the use of min in Equation 2 versus the use of max in Equation 1.

The strategy of letting each register's value be a function of all the input items has advantages. First: since the final value is a function of all items, rather than a register-specific subset of them, the final values are statistically independent. Second: the exponential rate $a$ and logarithm base $b$ are parameters of the sketch. We can set $b$ in a way that spreads the $K_i$ values over a range of our choosing. If registers are 8 bits wide, we can choose $a$ and $b$ so that $\lfloor 1 - \log_b h_i(d) \rfloor$ ranges from 0 to 255, landing outside the range only with low (and controllable) probability. If registers are another size (e.g. Dashing 2 supports 4, 8, 16, 32 and 64-bit registers), $a$ and $b$ can be adjusted. Methods for setting $a$ and $b$ are described in section 2.2 of [8].

This comes with a potential disadvantage. Since each register is a function of every input item, each addition may require $O(m)$ work where $m$ is the number of registers. Ertl [8] proposes optimizations that ensure that the work per update quickly becomes $O(1)$ in the typical case where the input is much larger than $m$. This is accomplished by (a) maintaining a value $K_{\text{max}}$ equal to the maximum among all the registers' current values, and (b) reordering the inner loop so that iterations occur in increasing order by value of the $h_i(d) \sim \text{Exp}(a)$ draw. Note that the maximum is maintained over the original, untruncated exponential draws in the $K_i$ variables, not the truncated version eventually stored in the $K'_i$ variables. Once we reach an iteration where the draw has value $h_i(d) > K_{\text{max}}$, neither the current nor a subsequent iteration can possibly change the value of a register, and the inner loop can break.

Another disadvantage of the SetSketch is the computational cost of the exponential draws and, related to that, the computational cost of logarithms. While the strategy just described reduces the number of exponential draws, later sections describe how we reduce the cost of logarithms.

## 2.2 Similarity comparison

Dashing 1's default algorithm for estimating the Jaccard coefficient between HLLs $A$ and $B$, $J(A, B)$, used the Maximum Likelihood Estimation (MLE) method of [13]. This models register values as Poisson random variables and uses an iterative root-finding to estimate the Poisson parameter from the histogram of register values in the union ($A \cup B$) HLL. This was less accurate but substantially faster than the related Joint MLE (JMLE) method [13], which required histogramming of joint register values.

Dashing 2 uses a simpler joint estimator named $\hat{\mu}_{\text{simple}}$ [8] [1]. It has a closed-form solution and does not require an iterative root-finding procedure. Being a "joint" estimator, $\hat{\mu}_{\text{simple}}$ also does not require a union sketch; it is a function only of the input sketches $A$ and $B$. Finally, $\hat{\mu}_{\text{simple}}$ does not require a histogram of register *values*. Instead, it requires two counts: $D_+$ and $D_-$. $D_+$ is the number of registers in $A$ that are greater than their counterparts in $B$ and $D_-$ is the number of registers in $A$ that are less than their counterparts. Unlike MLE and JMLE histograms, $D_+$ and $D_-$ can be computed using only Single-Instruction Multiple-Data (SIMD) instructions. In particular, a combination of SIMD greater-than/less-than and population-count instructions enable rapid tallying of $D_+$ and $D_-$ with respect to chunks of registers at a time. As described in [8], a mathematical complication arises when the sets are mostly disjoint. We fall back on Ertl's alternative formulae $\alpha_{\text{disj}}$ and $\beta_{\text{disj}}$ from [8] in such cases.

## 2.3 Full Dashing 2 sketch update

Algorithm 1 gives the update algorithm for the full version of the Dashing 2 SetSketch. This is the default for weighted sketching, and can be enabled for non-weighted sketching using `--full-setsketch`. (Otherwise the one-permutation strategy described in the next subsection is used.) Inputs consist of the register array $K$ comprising the sketch, a MaxTree array $T$ maintaining maxima over power-of-two-sized stretches of registers, the item $X$ to be added, and the item's weight $W$. When used for unweighted sketching, $W$ equals 1. Registers in $K$ are initialized to the maximum possible value.

A single update could require modifying any number of registers, from 0 to $m$. To avoid $O(m)$ work on average, the algorithm follows the strategy of Ertl [8], examining registers in order according to the probability it will be updated, i.e. according to the extremeness that register's exponential random draw. The algorithm uses a pseudo-random number generator RNG, seeded with $X$ for deterministic updates. RNG.nextExponentialSpacing($i, m, W$) returns the value that must be added to obtain the next draw in increasing order, according to the recurrence $x_i = x_{i-1} + \frac{1}{m-i+1}\text{Exp}(aW)$. The recurrence follows from the memoryless property of the exponential distribution [10]. RNG.nextFisherYates() samples a new register randomly without replacement with Fisher-Yates shuffling, as in Algorithm 6 of [10]. By matching the increasing series of exponential draws with a random sequence of register choices, we visit registers in the desired order of most to least likely to be updated. Kahan summation [14, 15] is used to reduce numerical errors when summing exponential spacings.

Conditional statements on lines 4 and 12 of Algorithm 1 abort upon reaching a draw that is less extreme than the least extreme that could affect a register value ($K_{\text{max}}$). Accordingly, the $K_{\text{max}} \leftarrow$ T.update($K[i], i$) statements on lines 9 and 17 update a structure maintaining the current "least extreme draw" in the $K_{\text{max}}$ variable per Algorithm 4 of [10]. Updates to the $K_{\text{max}}$ structure take $O(\log(m))$ time, and are required only

[1] This estimator is described only in the "v1" version of the paper pre-print cited. Later versions of the paper describe Brent's root-finding algorithm instead.

when a register is modified. The number of loop iterations is inversely related to the number of items added, approaching 0 as the number far exceeds $m$.

```
Input: SetSketch K[0..m − 1], MaxTree T[0..m − 2], item X of weight W
Result: K and T updated according to X, W
RNG ← RandomNumberGenerator(X, m)
RV ← RNG.nextExponentialSpacing(0, m, W)
K_max ← T.max()
if RV > K_max then
    return
end
i ← RNG.nextFisherYates()
if RV < K[i] then
    K[i] ← RV
    K_max ← T.update(K[i], i)
end
for i ← 1, m − 1 do
    RV ← RV + RNG.nextExponentialSpacing(i, m, W)
    if RV > K_max then
        return
    end
    i ← RNG.nextFisherYates()
    if RV < K[i] then
        K[i] ← RV
        K_max ← T.update(K[i], i)
    end
end
```

**Algorithm 1:** Update Full Dashing 2 SetSketch

## 2.4 One-permutation SetSketch

By default, Dashing 2 computes unweighted sketches and uses an economical "one-permutation" update method [16], which modifies at most one register per update. This method uses some bits of the random draw to choose which register to update (Algorithm A2, Appendix), similarly to how the HLL update rule in Dashing uses the hash prefix $p$.

While the one-permutation approach is efficient compared to a full update, accuracy suffers when many registers are empty, i.e. when the input has few items relative to $m$. To maintain accuracy, we implement the densification approach of [17], applied after finalization. This strategy has similar accuracy compared to the full SetSketch, but is more efficient in practice. Because of this, we made this one-permutation mode the default for unweighted sketching. The full update rule is used instead when the user enables weighted sketching, or when the user enables it with the `--full-setsketch` option.

## 2.5 SetSketch parameters

Like the HyperLogLog, the number of registers $m$ is a parameter of the SetSketch. Unlike HyperLogLog, SetSketch has a related parameter, the register *width* in bits. Other key parameters are the rate for the exponential draws ($a$), and the log base used for truncation needed to fit draws into registers ($b$). As in [8], we use $q$ to denote the value that is 1 less than the maximum register value. E.g. for 8-bit registers, $q = 2^8 - 1 - 1 = 254$.

While [8] gives theoretical guidelines for choosing $a$, $b$ and other parameters, these assume foreknowledge of input cardinalities. On the other hand, multiple SetSketches are comparable only if they were built using identical parameters. This creates a tension between wishing to choose the parameters sooner, in order to make compact sketches, versus later, to delay truncation until we can ensure all relevant sketches are constructed and truncated with identical $a$, $b$ and $q$.

**Input** : $Ks$: Map from $id$'s to SetSketches, for all datasets
**Input** : $k$: Target number of nearest neighbors to find
**Input** : $id$: Identifier for this dataset
**Input** : $RNG$: Pseudo-random number generator
**Input** : $seed$: Pseudo-random seed
**Input** : $N$: Number of super-register sizes in index
**Input** : $LSH$: map from $\langle$ table id, super-register id, super-register value $\rangle$ triples to a corresponding list of datasets
**Output:** $L$: List of nearest neighbors

```
i ← N − 1
if i > 2 then
    RNG.initialize(seed)
end
// Over tables 0 .. N-1, from largest super-register size (most specific) to smallest (least specific)
while i ≥ 0 do
    P ← min(2^i, 2i)
    if i ≤ 2 then
        S ← m/N
    else
        S ← m · 8/N
    end
    j ← 0
    // Loop over super-registers
    while j < S do
        if i ≤ 2 then
            // Next non-overlapping super-register
            SuperReg ← Ks[id][K · j .. K · j + P − 1]
        else
            // Random super-register of length P
            ri ← RNG.randomInt(0, m − P)
            SuperReg ← Ks[id][ri .. ri + P − 1]
        end
        // Append this dataset to the list for this table, super-register
        // super-register value combination
        for n ∈ LSH[⟨i, j, SuperReg⟩] do
            L.append(n)
            if L.length() = k · 3 then
                return L
            end
        end
        j ← j + 1
    end
    i ← i − 1
    return L
end
```

**Algorithm 2:** Find candidates for $k$ nearest neighbors for dataset $\langle K, id \rangle$ using LSH index. Result is a list of at most $3 \cdot k$ candidates that is later refined and ordered using Jaccard similarity.

Dashing 2's default strategy is to set $a$ and $b$ according to the overall set of input datasets, and to shape and truncate the sketches according to user-configurable choices for $m$ and $q$. Dashing 2 also allows the user to delay the choices for $a$, $b$ and $q$, so that larger, untruncated sketches can be stored temporarily in preparation for future truncation and comparison with other sketches.

To select $a$ and $b$ according to the data, Dashing 2 first forms untruncated sketches, then computes $b$ and $a$ according to the expressions:

$$b = \exp\left(\frac{\ln \max(K^*)/\min(K^*)}{q}\right) \qquad a \quad = \max(K^*)/b \tag{3}$$

Where $K^*$ denotes a concatenations of all untruncated register values from all inputs. For experiments in this study, we invoked Dashing 2 with all input datasets at once, ensuring $a$ and $b$ are set identically for all.

## 2.6 Delayed logarithms

Potentially expensive logarithm calculations are used in two tasks: (a) truncation of register values, and (b) to perform the exponential draws. Dashing 2 avoids these costs in two ways. First, it uses the two-step strategy of Equation 2 to delay truncation until a finalization step, which runs only after all items are added (Algorithm A3, Appendix). Before finalization, intermediate register values are stored as untruncated 64-bit floating-point numbers. As a result, the update rule uses only a minimum, rather than both a logarithm and a maximum. The total number of logarithmic truncations performed at most $m$ regardless of the number of items added to the sketch.

This comes at the cost of requiring additional space at sketching time. For the entire algorithm up to finalization, we must store a 64-bit value for each register even if finalization will later reduce that to, e.g. 8 bits. Since practical sketches require only thousands of registers, this is not onerous in practice.

## 2.7 Approximate logarithms

The second use of logarithms is in the exponential $\text{Exp}(a)$ random draw, which is accomplished by computing $-\ln(\text{Unif}())/a$ where Unif() is a uniform random draw between 0 and 1. We observed that, once the sketch becomes quite full, many exponential draws are well above the $K_{\text{max}}$ ceiling, aborting the inner loop. While an inaccurate logarithm might cause us to miscompute whether a draw is under the $K_{\text{max}}$ ceiling, this arises only for draws near the ceiling. We use a fast, approximate logarithm first, reverting to a more accurate (and expensive) logarithm only if the first result is close to $K_{\text{max}}$.

For fast logarithms, we use an approximation computed using the floating-point number's integral representation. Specifically, we use a modified version of the algorithm of [18]. This can overestimate the result by a multiplicative factor of up to 1.42. By dividing the fast-logarithm result by this number, we can determine if the approximation is close enough to $K_{\text{max}}$ to require a full logarithm computation. This affects the computation within RNG.nextExponentialSpacing(), called on lines 2 and 11, as well as the conditional checks on lines 4 and 12 of Algorithm 1, though we omitted these details from the algorithm listing.

## 2.8 Weighted SetSketch

While [8] describes applications only to unweighted sketching, we extended SetSketch to consider weightedness using the ProbMinHash [10] strategy, i.e. by multiplying the exponential draw's rate parameter by the item's weight, represented by the argument $W$ passed to the RNG.nextExponentialSpacing() calls on lines 2 and 11 of Algorithm 1. When comparing sketches weighted in this way, the quantity being estimated is a version of the Jaccard coefficient called the "Probability Jaccard similarity" $J_P$:

$$J_P(A,B) = \sum_{d \in D} \frac{1}{\sum_{d' \in D} \max\left(\frac{w_A(d')}{w_A(d)}, \frac{w_B(d')}{w_B(d)}\right)} \tag{4}$$

Where $D$ is the item universe and $w_A$ and $w_B$ are weight functions for items in sets $A$ and $B$.

A key question is how to obtain $w_A$ and $w_B$. When the input is sequencing data, the weight of an item ($k$-mer) should equal its relative frequency. By default, Dashing 2 will use a hash table to track the exact relative frequency for each item. But Dashing 2 also supports a faster and more memory-efficient method that estimates each item's frequency using a feature hashing [19] approach, equivalent to a single-row Count-Min Sketch [20]. This mode is enabled with the `--countsketch-size` option. Non-sequencing

datasets might also come with an inherent notion of "weight;" for instance, if input items represent genes and associated expression levels, these levels could be immediately used as weights, without the need for counting or for the feature-hashing data structure.

Optimizations for logarithms described above are also used for weighted sketching.

## 2.9 Locality-Sensitive Hashing (LSH) implementation

To scale all-pairs comparisons, we implemented a filtering approach based on locality-sensitive hashing (LSH). Given a minimum similarity threshold, the filter avoids computing many below-threshold pairwise comparisons. The LSH method works by grouping SetSketch registers into "super-registers." For instance, the first four registers ($K[0...3]$) might constitute the first super-register, the next four ($K[4...7]$) the second super-register, etc. Associated with each super-register is a map from possible values to a list of all input datasets having that value in that super-register. In our example, the keys for the first super-register will consist of all combinations of the first four registers $K[0...3]$ observed in an input dataset, and the values will be the associated lists of datasets. An LSH index might consist of several such tables, each with a distinct super-register group size. Algorithm A1 shows how the index is updated with one additional dataset.

Since registers are independent, a size-$P$ super-register will match between datasets $A$ and $B$ with probability $J(A,B)^P$, where $J$ is the Jaccard coefficient. When performing a large-scale all-pairs comparison, Dashing 2 begins by computing LSH indexes for values of $P$ in a user-configurable subset of the values $\{1,2,4,6,8,10\}$. By default, Dashing 2 tries $P \in \{1,2\}$, but it can be configured to try more values for $P$ via the `--nlsh` option, corresponding to the $N$ variable in Algorithms A1 and 2. For $P \in \{1,2\}$, the super-registers are formed by partitioning registers into $m/P$ non-overlapping groups. For larger values of $P$, we select a random set of $m \cdot 8/P$ contiguous groups of registers. In this case, super-registers can overlap.

Algorithm A1 details how a single dataset is added to an LSH index. Algorithm 2 details how we query to find a list of candidate nearest neighbors for a dataset using the LSH index. In both cases, a pair of nested loops is used. The outer loop iterates over LSH tables from the most to least specific (largest to smallest $P$), while the inner loop iterates over super-registers. In the case of Algorithm A1, an iteration of the inner loop updates the LSH table with the $id$ of the current dataset. In the case of Algorithm 2, an iteration of the inner loop contains a final loop that updates a running list of candidate datasets with all other datasets having the same value for the current super-register.

The LSH tables are used in two distinct modes of Dashing 2. The mode activated with (`--topk`) builds a $k$-nearest neighbor (KNN) graph from the input datasets and follows the logic of Algorithm 2. For a given pivot genome, we use the LSH tables to generate a list of $\lceil O_s \times k \rceil$ candidates for each input genome, where $O_s > 1$ is an over-sampling rate, set to 3 by default. We then estimate the Jaccard similarity between the pivot and each of the candidates in the order they were discovered, keeping only the $k$ with the greatest Jaccard coefficients. While this can result in some misreported neighbors, e.g. because a near neighbor happened not to coincide with the pivot in any super-register, this possibility is reduced both by over-sampling and by the order in which we attempt the LSH tables, i.e. from most to least specific.

In another mode, Dashing 2 reports pairwise distances between all pairs of genomes having similarity above some threshold (`--similarity-threshold X`). In this mode, there is no additional limit on the number of "neighbors" that might be reported for a genome. While querying the index, we maintain a heap of all neighbors with similarity above/distance below a given threshold.

## 2.10 Exact similarity mode

The ability to compute exact Jaccard coefficients is useful for evaluating Dashing 2's estimates. Dashing 2 therefore implements two modes for exact computation of Jaccard coefficients. One uses sorted $k$-mer hash sets (`--set`); the other uses $k$-mer count dictionaries (`--countdict`).

## 2.11 Sketching sequencing reads

Dashing 2 can also sketch inputs consisting of sequencing reads, e.g. in FASTQ format. This involves the extra challenge of handling sequencing errors, since $k$-mers containing errors can be far more numerous than correct $k$-mers and so can dominate and bias similarity estimates. For this reason, methods for sketching sequencing-read inputs attempt to filter out $k$-mers containing sequencing errors prior to computing cardinality or similarity. Dashing 2 adapts the approach of Mash [1] for eliminating k-mers below a specific count threshold. For instance, if the target threshold is set to 2, Dashing 2's SetSketch implementations

(both one-permutation and full) maintain a dictionary of items seen fewer than 2 times so far. Once an item's count reaches 2, it is added to the final SetSketch structure.

Dashing 2 can also use a downsampling approach (`--downsample <fraction>`) to randomly keep a specified fraction of the input $k$-mers. The decision to keep or suppress a $k$-mer is made independently for each $k$-mer, rahter than for each *distinct* $k$-mer. In this way, frequent $k$-mers – i.e. occurring more than $\frac{1}{F}$ of the time – are unlikely to have all copies suppressed. The filter incurs little computational cost but greatly reduces the number of error $k$-mers ending up in the sketch. This is similar to the ideas used in sequencing error correction [21] to distinguish $k$-mers with or without sequencing errors.

Weighted sketching modes can be particularly appropriate for sequencing reads, since they have the effect of down-weighting error $k$-mers, which tend to occur infrequently compared to correct $k$-mers.

# 3 Results

We used Dashing 2 v2.1.11-10-g128c. All experiments were performed on a Lenovo ThinkSystem SR630 with 48 3.0GHz Xeon CPUs and 1.5 TB of memory.

We downloaded the Refseq database on Jun 30, 2022 [22]. Filtering to just complete genome sequences, we gathered 128,827 sequences, 729 from the "archea" category, 115,548 from "bacteria," 338 from "fungi," 2 from "human" (the GRCh38 and the CHM13 assemblies), 267 from "invertebrate," 145 from "plant," 88 from "protozoa," 183 from "vertebrate_mammalian," 274 from "vertebrate_other" and 11,253 from "viral." The compressed FASTA files occupied 475 GB. Overall, genome lengths varied from 223 bp to over 34 billion bp, with mean and median lengths of 11.1 million and 4.22 million bp respectively.

Some experiments required high-fidelity sketches for all assemblies. We computed a 1-MB sketch for each using Dashing 2 with `sketch --binary-output -S 20 -k 31`. For all 128,827 complete genomes, this process took 50m:45s using GNU parallel [23] for multiprocess parallelism.

## 3.1 Similarity estimation

We compiled a collection of pairs of assemblies covering a range of true Jaccard coefficients. If $A$ be the set of canonicalized $k$-mers from one assembly and $B$ be the same for the other, the Jaccard coefficient between then is defined as $J(A, B) = |A \cap B|/|A \cup B|$. To select genome pairs covering a range of Jaccards, we performed an all-pairs comparisons using the high-fidelity sketches described above. Specifically, we partitioned the space of Jaccard estimates into 100 buckets of equal size, plus an additional bucket for pairs with $J(A, B) = 1$. For each bucket, we selected 10 genome pairs having Jaccard coefficient within the bucket, as estimated by Dashing 2 using high-fidelity sketches. We limited our attention to genome assemblies from the "archea," "bacteria" and "viral" groups. We repeated this for $k$-mer lengths 21 and 31.

We used Dashing 2's full-accuracy mode to compute true Jaccard coefficients for all genome pairs for both $k$-mer lengths. We also computed Average Nucleotide Identities (ANIs) for all selected genome pairs using fastANI v1.33 [24]. fastANI computes an approximation, so we do not call these "true" ANIs. But these have the advantage of being calculated using a separate approach from the one used to compute the Jaccard coefficients; in particular, fastANI's approach is not $k$-mer-based.

Using these genome pairs annotated with true Jaccard coefficients and fastANI-estimated ANIs, we compared the accuracy of Dashing 2's estimates to those of Dashing 1 v1.0 [7] and Mash v2.3 [1]. We omitted BinDash initially as it would sometimes fail when comparing more distant genome pairs. We ran Dashing 2 in three configurations. **D2** used the "one-permutation" SetSketch, with each update affecting at most one register. **D2-full** used the full update rule. **D2W** used a weighted version of the full update rule and used an upstream feature-hashing sketch [19] to create a multiplicity-aware representation of the input prior to sketching (see Methods). The feature-hashing structure consisted of 5 million 64-bit counts.

Table 1 shows the sum of squared errors (SSE) between the tool-estimated Jaccard coefficient and the true Jaccard, totaled across all 1,010 genome pairs. We show these results for sketch sizes of 8 Kbits ($8 \times 1024$ bits, equivalent to 1 Kbyte), 32 Kbits, and 128 Kbits. In all cases either D2 or D2-full achieved lowest SSE, with th other achieving second-lowest. Dashing 2 and Dashing 1 both achieved lower SSE than Mash.

To additionally compare to BinDash, we also filtered the dataset to only 964 genome pairs with fastANI-estimated ANI greater than 89%. Table 2 shows these results for the same sketch sizes and values of $k$ as Table 1. In all but one case, D2 achieved the lowest SSE; the exception was the case of $k = 21$ and 8 Kbit sketch, where BinDash achieved the lowest SSE. D2 or D2-full achieved the lowest SSEs in all other cases.

| k | kbits | Mash | Dashing1 | D2-full | D2 |
|---|---|---|---|---|---|
| 21 | 8 | 1.79 | 1.10 | 0.776 | 0.814 |
| | 32 | 0.872 | 0.711 | 0.669 | 0.651 |
| | 128 | 0.765 | 0.644 | 0.628 | 0.616 |
| 31 | 8 | 1.95 | 0.938 | 0.584 | 0.595 |
| | 32 | 0.766 | 0.580 | 0.475 | 0.463 |
| | 128 | 0.544 | 0.479 | 0.461 | 0.456 |

Table 1: Sum of squared error between estimated and true Jaccard coefficients for several methods and for all genome pairs. Bright red indicates the lowest error in each row. Dark red indicates second-lowest error.

| k | kbits | Mash | BinDash | Dashing1 | D2-full | D2 |
|---|---|---|---|---|---|---|
| 21 | 8 | 1.78 | 0.750 | 1.08 | 0.773 | 0.812 |
| | 32 | 0.870 | 0.676 | 0.705 | 0.668 | 0.651 |
| | 128 | 0.764 | 0.648 | 0.642 | 0.628 | 0.615 |
| 31 | 8 | 1.94 | 0.600 | 0.912 | 0.583 | 0.594 |
| | 32 | 0.764 | 0.480 | 0.573 | 0.475 | 0.463 |
| | 128 | 0.544 | 0.473 | 0.477 | 0.461 | 0.456 |

Table 2: Sum of squared error between estimated and true Jaccard coefficients for several methods, using only genome pairs with estimated true ANI greater than 89%. Above this threshold, BinDash ran with no estimation failures. Bright red indicates the lowest error and dark red the second-lowest error in each row.

### 3.2 ANI estimation

We further assessed the Average Nucleotide Identity (ANI) estimates obtained by using the Mash distance equation and re-scaling: $ANI_{\text{est}} = 1 + 1/k \cdot \ln(2J/(1+J))$. Here $k$ is the $k$-mer length and $J$ is the estimated Jaccard coefficient. Negative $ANI_{\text{est}}$ values were rounded up to 0. We also assessed the new multiplicity-aware ("weighted") mode of Dashing 2, called D2W. In this case, the input $J$ into the $ANI_{est}$ equation was an estimate of the weighted Jaccard coefficient, rather than the typical "flat" coefficient.

We assessed SSE between $ANI_{est}$ and the fastANI-estimated ANIs for Mash, Dashing 1, D2 and D2W (Table 3). In all cases, the D2W approach achieved either the lowest or second-lowest SSE, with either D2 or Dashing 1 achieving the second-lowest SSE.

### 3.3 Refseq sketching and pairwise comparisons

We used Dashing 1 and Dashing 2 to sketch a sample of 50,000 complete genome assemblies downloaded from Refseq. Both were run using GNU parallel, allowing up to 12 sketching processes to run at a time. Both tools were configured to produce a sketches 1 MB in size. Dashing 1 took 2h:00m:34s to construct all the sketches and Dashing 2 took 50m:46s.

We then used Dashing 1 and 2 to perform exhaustive all-pairwise Jaccard similarity comparisons across the 50,000 genomes (2 human assemblies and 49,998 bacterial assemblies), comparing a total of 1.25 billion pairs of 1MB sketches. Both tools used their default similarity estimation methods. In the case of

| k | kbits | Mash | Dashing1 | D2 | D2-full | D2W |
|---|---|---|---|---|---|---|
| 21 | 8 | 6.80 | 1.94 | 2.61 | 1.41 | 0.273 |
| | 32 | 6.06 | 3.05 | 2.02 | 3.72 | 0.279 |
| | 128 | 6.06 | 3.17 | 3.66 | 3.06 | 0.276 |
| 31 | 8 | 8.20 | 4.54 | 3.88 | 4.40 | 0.538 |
| | 32 | 5.36 | 2.38 | 4.47 | 3.30 | 0.544 |
| | 128 | 5.37 | 0.787 | 3.75 | 3.64 | 0.543 |

Table 3: Sum of squared error between estimated Mash distance and the ANI as computed by fastANI. Bright red indicates the lowest error in each row. Dark red indicates second-lowest error. In all cases except D2W, Mash distance is computed as a function of the estimated "flat" Jaccard coefficient. In the case of D2W, Mash distance is a function of the Probability Jaccard Similarity from equation 4.

Dashing 1's, this was the MLE estimator of Ertl [13]. In the case of Dashing 2, this was the simple joint estimator described above under "Similarity comparison." Both Dashing 1 and Dashing were run with the `-p80 --presketched` options, enabling 80 simultaneous threads of execution and instructing both tools to use the already-computed sketches. Dashing 1 took 49h:41m to estimate all-pairwise similarities, whereas Dashing 2 took 5h:41m, over 20 times faster.

### 3.4 All-pairs comparisons using LSH

We performed all-pairs comparisons for a large collection of proteins by combining the SetSketch approach with a locality-sensitive hashing strategy to avoid comparisons unlikely to meet a minimum similarity threshold. We used the UniProtKB/Swiss-Prot collection of 565,254 protein sequences, version 2021_03. For each protein, we used Dashing 2 to create a 10-mer sketch (`-k10`) of 256 registers (`-S256`). Proteins were translated to a 14-letter reduced alphabet (`--protein14`) to capture more distant homology [11]. We ran Dashing 2 in its `sketch --topk` mode to perform all-pairs comparisons while avoiding pairings that fail to appear in the top 256 neighbors of a protein. We used the default setting of `--nLSH 2` to use two distinct sizes of super-register groupings, corresponding to the $N$ parameter of Algorithm 2. The output was a K-Nearest-Neighbor (KNN) graph in tabular (TSV) format, associating each protein to the 256 other proteins with greatest Jaccard coefficient.

Because of both estimation and LSH error, the graph may include false positives (neighbors that are not truly among the top 256) and/or false negatives (missing neighbors that are truly in the top 256). To measure the error introduced by LSH, we compared the KNN graph to another KNN generated using exhaustive all-pairs comparisons. Importantly, the exhaustive KNN was also built using Jaccard coefficient estimates, since exact computation of the Jaccard coefficient is too computationally expensive. Thus, this experiment isolates error due to the LSH filter only, and does not assess error due to the Jaccard estimate.

Relative to the exact method, Dashing 2's LSH method achieved 100% recall and precision, i.e. there were no false positives or false negatives among the neighbors found using LSH filtering. Further, using the LSH, Dashing 2 was able to generate the graph in 43 seconds, compared to 56.2 minutes for the exhaustive method, a 78-fold speedup. The exhaustive method ultimately performed 159,755,759,631 pairwise comparisons, compared to approximately 470M comparisons performed by the LSH-assisted method. The 470M comparisons performed by LSH is about 3.5 times greater than the minimum determined by the number of neighbors (256) times the number of proteins (565,254).

We also applied this approach to create a nearest-neighbor graph over the larger UniRef50 dataset, which is built by clustering UniRef90 seed sequences from the UniProt Knowledgebase v2021_03 having at least 50% sequence identity to and 80% overlap with the longest sequence in the cluster. The database contains 53,625,855 sequences totaling over 15 billion amino acids. After sketching, the KNN graph was generated in less than 10 minutes. By contrast, the exhaustive all-pairs comparisons approach required a much longer amount of time; we interrupted this computation after 2 days and extrapolated that the total time required would have exceeded a year.

## 4 Discussion

Dashing 2 combines the SetSketch with locality-sensitive hashing to bring multiplicity-aware sketching and fast filtering to genomic sketching analysis. Its all-pairs comparison method scales effectively to millions of sequences. Dashing 2 can sketch FASTA and FASTQ inputs, as well as protein-sequence inputs using a variety of protein alphabet reductions. It can compare sequences based on Jaccard coefficient, Mash distance, or containment coefficient. It can also compare sequences based on a weighted version of the Jaccard coefficient that is aware of the multiplicities of the input items. Dashing 2 is free, open source software available from `https://github.com/dnbaker/dashing2`.

Dashing 2's estimation error for the Jaccard coefficient and ANI estimates was lower than that of the previous version of Dashing, and substantially lower than that of Mash. In cases where BinDash can be run successfully, Dashing 2 had lower or comparable error to BinDash. Thus, it will be important to continue to study the SetSketch as a highly accurate alternative not only to the HyperLogLog sketch, but also the MinHash and b-bit MinHash methods.

It is remarkable that Dashing 2's ANI estimation error was lowest when taking multiplicities of the input items into account, i.e. when using a weighted Jaccard coefficient as input to the Mash distance equation.

This demonstrates that multiplicities are helpful not only for quantitative applications, but also in typical "flat" sketching scenarios. This motivates future study of sketch data structures that account for multiplicities, as well as methods – like Dashing 2's feature hashing method – for efficiently compiling multiplicity information prior to sketching.

# 5 Acknowledgements

We thank Otmar Ertl and Martin Steinneger for helpful conversations. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. In particular, we acknowledge Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. URL: `http://www.tacc.utexas.edu`.

DNB and BL were supported by NIH/NIGMS grant R35GM139602 to BL. BL was also supported by NIH/NIGMS grant R01HG012252. This work was carried out at Advanced Research Computing at Hopkins (ARCH) core facility (rockfish.jhu.edu), which is supported by the National Science Foundation (NSF) grant number OAC 1920103.

# References

1. Ondov, B. D., Treangen, T. J., Melsted, P., Mallonee, A. B., Bergman, N. H., Koren, S. & Phillippy, A. M. Mash: fast genome and metagenome distance estimation using MinHash. *Genome Biol.* **17,** 132 (June 2016).
2. Criscuolo, A. On the transformation of MinHash-based uncorrected distances into proper evolutionary distances for phylogenetic inference. *F1000Res* **9,** 1309 (2020).
3. LaPierre, N., Alser, M., Eskin, E., Koslicki, D. & Mangul, S. Metalign: efficient alignment-based metagenomic profiling via containment min hash. *Genome Biol* **21,** 242 (Sept. 2020).
4. Dilthey, A. T., Jain, C., Koren, S. & Phillippy, A. M. Strain-level metagenomic assignment and compositional estimation for long reads with MetaMaps. *Nat Commun* **10,** 3066 (July 2019).
5. Gostinčar, C. Towards Genomic Criteria for Delineating Fungal Species. *J Fungi (Basel)* **6** (Oct. 2020).
6. Ondov, B. D., Starrett, G. J., Sappington, A., Kostic, A., Koren, S., Buck, C. B. & Phillippy, A. M. Mash Screen: high-throughput sequence containment estimation for genome discovery. *Genome Biol* **20,** 232 (Nov. 2019).
7. Baker, D. N. & Langmead, B. Dashing: fast and accurate genomic distances with HyperLogLog. *Genome Biol* **20,** 265 (Dec. 2019).
8. Ertl, O. SetSketch: Filling the Gap between MinHash and HyperLogLog. *Proc. VLDB Endow.* **14,** 2244–2257. ISSN: 2150-8097 (July 2021).
9. Ertl, O. *BagMinHash - Minwise Hashing Algorithm for Weighted Sets* in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Association for Computing Machinery, London, United Kingdom, 2018), 1368–1377. ISBN: 9781450355520.
10. Ertl, O. ProbMinHash – A Class of Locality-Sensitive Hash Algorithms for the (Probability) Jaccard Similarity. *IEEE Transactions on Knowledge and Data Engineering* **34,** 3491–3506 (2022).
11. Edgar, R. C. Local homology recognition and distance measures in linear time using compressed amino acid alphabets. *Nucleic Acids Research* **32,** 380–385. ISSN: 0305-1048 (Jan. 2004).
12. Kent, W. J., Zweig, A. S., Barber, G., Hinrichs, A. S. & Karolchik, D. BigWig and BigBed: enabling browsing of large distributed datasets. *Bioinformatics* **26,** 2204–2207 (Sept. 2010).
13. Ertl, O. New cardinality estimation algorithms for HyperLogLog sketches. *CoRR* **abs/1702.01284.** arXiv: `1702.01284` (2017).
14. Kahan, W. Pracniques: Further Remarks on Reducing Truncation Errors. *Commun. ACM* **8,** 40. ISSN: 0001-0782 (Jan. 1965).

15. Babuska, I. *Numerical stability in mathematical analysis* in (IFIP Congress, North-Holland, Amsterdan, 1969), 11–23.

16. Wang, J., Zhang, T., Sebe, N., Shen, H. T., *et al.* A survey on learning to hash. *IEEE transactions on pattern analysis and machine intelligence* **40,** 769–790 (2017).

17. Shrivastava, A. *Optimal Densification for Fast and Accurate Minwise Hashing* in *Proceedings of the 34th International Conference on Machine Learning* (eds Precup, D. & Teh, Y. W.) **70** (PMLR, June 2017), 3154–3163.

18. Schraudolph, N. N. A fast, compact approximation of the exponential function. *Neural Computation* **11,** 853–862 (1999).

19. Moody, J. *"Fast Learning in Multi-Resolution Hierarchies"* in *Proceedings of the 1st International Conference on Neural Information Processing Systems* (MIT Press, Cambridge, MA, USA, 1988), 29–39.

20. Cormode, G. & Muthukrishnan, S. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* **55,** 58–75. ISSN: 0196-6774 (2005).

21. Song, L., Florea, L. & Langmead, B. Lighter: fast and memory-efficient sequencing error correction without counting. *Genome Biol* **15,** 509 (2014).

22. O'Leary, N. A., Wright, M. W., Brister, J. R., Ciufo, S., Haddad, D., McVeigh, R., Rajput, B., Robbertse, B., Smith-White, B. & Ako-Adjei, D. e. a. Reference sequence (RefSeq) database at NCBI: current status, taxonomic expansion, and functional annotation. *Nucleic Acids Res.* **44,** D733–745 (Jan. 2016).

23. Tange, O. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine* **36,** 42–47 (Feb. 2011).

24. Jain, C., Rodriguez-R, L. M., Phillippy, A. M., Konstantinidis, K. T. & Aluru, S. High throughput ANI analysis of 90K prokaryotic genomes reveals clear species boundaries. *Nat Commun* **9,** 5114 (Nov. 2018).

# Appendix

**Input:** $K[0..m-1]$: SetSketch registers for this dataset
**Input:** $id$: Identifier for this dataset
**Input:** $RNG$: Pseudo-random number generator
**Input:** $seed$: Pseudo-random seed
**Input:** $N$: Number of super-register sizes in index
**Input:** $LSH$: map from $\langle$ table id, super-register id, super-register value $\rangle$ triples to a corresponding list of datasets
**Result:** $LSH$ updated to include dataset $\langle K, id \rangle$

```
i ← N − 1
if i > 2 then
    RNG.initialize(seed)
end
// Loop over tables 0 .. N-1, from largest super-register size (most specific)
// to smallest (least specific)
while i ≥ 0 do
    P ← min(2^i, 2i)
    if i ≤ 2 then
        S ← m/N
    else
        S ← m · 8/N
    end
    j ← 0 // Loop over super-registers
    while j < S do
        if i ≤ 2 then
            // Next non-overlapping super-register
            SuperReg ← K[P · j .. P · j + P − 1]
        else
            // Get uniform random integer in [0, m − P]
            ri ← RNG.randomInt(0, m − P)
            SuperReg ← K[ri .. ri + P − 1]
        end
        // Append this dataset to the list for this table, super-register
        // super-register value combination
        LSH[⟨i, j, SuperReg ⟩].append(id)
        j ← j + 1
    end
    i ← i − 1
end
```

**Algorithm A1:** Add dataset $\langle K, id \rangle$ to LSH index

**Input:** SetSketch $K[0..m-1]$, item $X$
**Result:** $K$ updated according to $X$

```
RI ← hash(X)
q ← RI mod m
p ← ⌊RI/m⌋
K[i] ← min(K[i], p)
```

**Algorithm A2:** Update one-permutation Dashing 2 SetSketch. Each $K_i$ gets the minimal 64-bit random draw among draws that map there.

**Input :** SetSketch $K[0..m-1]$, item $X$
**Output:** $K'$ set to a truncated form of $K$

```
for i ← 0, m − 1 do
    K'[i] ← ⌊1 − log_b(K[i])⌋
end
```

**Algorithm A3:** Finalize Full Dashing 2 SetSketch

**Input :** Temporary $K[0..m-1]$ from Algorithm A2
**Output:** Finalized $K'$

```
for i ← 0, m − 1 do
    RV ← −ln(K[i])
    K'[i] ← ⌊1 − log_b(RV)⌋
end
```

**Algorithm A4:** Finalize one-permutation Dashing 2 SetSketch