Improved Range-Summable Random Variable Construction Algorithms

A. R. Calderbank* A. Gilbert† K. Levchenko‡ S. Muthukrishnan§
M. Strauss¶

October 31, 2004

Abstract

Range-summable universal hash functions, also known as range-summable random variables, are binary-valued hash functions which can efficiently hash single values as well as ranges of values from the domain. They have found several applications in the area of data stream processing where they are used to construct sketches—small-space summaries of the input sequence.

We present two new constructions of range-summable universal hash functions on n-bit strings, one based on Reed-Muller codes which gives k-universal hashing using $O(n^{\log k})$ space and time for point operations and $O(n^{2\log k})$ for range operations, and another based on a new subcode of the second-order Reed-Muller code, which gives 5-universal hashing using $O(n)$ space, $O(n\log^3 n)$ time for point operations, and $O(n^3)$ time for range operations.

We also present a new sketch data structure using the new hash functions which improves several previous results.

*Princeton University, Princeton, New Jersey. E-mail: calderbk@math.princeton.edu.

†University of Michigan, Ann Arbor, Michigan. Most of this work was done while the author was at AT&T Labs, Florham Park, New Jersey. E-mail: annacg@umich.edu.

‡University of California San Diego, La Jolla, California. Most of this work was done while the author was at AT&T Labs, Florham Park, New Jersey. E-mail: klevchen@cs.ucsd.edu.

§Rutgers University, Piscataway, New Jersey. E-mail: muthu@cs.rutgers.edu. Work supported by NSF ITR 0220280.

¶University of Michigan, Ann Arbor, Michigan. Most of this work was done while the author was at AT&T Labs, Florham Park, New Jersey. E-mail: martinjs@eecs.umich.edu.

1 Introduction

The role of hashing in Computer Science needs little introduction, for it can be seen in many areas from the practical considerations of compiler design to our latest understanding of randomness in the theory of complexity. After the early advancements, described, for example, in Knuth [12], the seminal work of Carter and Wegman deserves special mention. In [4], they introduced the concept of universal hashing, showing that pair-wise independence of hash values suffices for most practical applications, allowing efficient implementations with strong probabilistic guarantees. Since then, pair-wise, and more generally, exact and approximate k-wise independent hashing has found many applications (see, e.g., [13, 21]).

In this paper, we study exact k-wise independent—also known as k-universal—hash functions with added resource constraints motivated by data stream algorithmics. In addition to efficiently computing a hash value at a point, we also want to be able to compute the distribution of hash values for a given range of points in the domain and a fixed choice of the underlying randomness. These *range-summable* universal hash functions were first introduced in the work of Feigenbaum *et al.* [8] in the context of data stream algorithms, where they gave a polylogarithmic (in the size of the domain) time and space construction. Since then, k-universal hashing has been used to solve a variety of problems in the stream setting, including: estimating norms [6, 10, 3, 8], computing approximate wavelet summaries [9] and histograms [16, 10, 17], and estimating subgraph counts in graphs [3]. Other applications are yet to be explored, eg., maintaining statistics on geometric such as intervals and boxes in spatial databases. Of special interest is the work of Bar-Yossef *et al.* [3] which uses range-summable hash functions extensively, and introduces a general framework of "list reductions" for reducing new data stream problems to known ones and applies it to counting triangles in a graph in the data stream setting, which is of interest in Web graph analysis.

Our contributions are:

1. We present a new range-summable construction of k-universal hash functions based on Reed-Muller error-correcting codes, which use polylogarithmic in time and space for any fixed k, whereas previously known constructions were only available in flavors of 3-universal and 7-universal. In other words, applications that deal with, say, finding large frequency moments (F_3 and higher) range-efficiently or counting constant-sized cliques (of size more than 3) in graph data streams needed k-universal range-summable constructions for $k > 7$. So even though in principle the use of range-summable random variables is clear, there were no known efficient solutions prior to this work.

2. We present a new, near-optimal range-summable 5-universal construction which immediately improves existing algorithms using the previously known, but more expensive, 7-universal construction, bringing range-summable hashing within the realm of practical implementation. Our result uses a new subcode of the second-order Reed-Muller error-correcting code, and may be of independent interest.

The rest of the paper is organized as follows. In Section 2 we formally define range-summable hash functions with a brief and more technical survey of related work. In Section 3 we present our first result, a k-universal range-summable construction based on Reed-Muller codes. In Section 4 we present the Hankel code and the corresponding range-summable 5-universal hash function. In Section 5 we describe a data stream sketch—a kind of synopsis data structure—using the results of Section 4, which improves several existing algorithms. Section 6 concludes the paper.

2 Definitions and Preliminaries

Range-summable hash functions were first introduced by Feigenbaum *et al.* [8] under the name of range-summable random variables. In this paper, we use the equivalent formulation using hash functions.

Universal Hashing. Recall that a *k-universal family of hash functions* is a set $\mathcal{H}$ of functions $h : A \to B$ such that for all distinct $x_1, \ldots, x_k \in A$ and all (not necessarily distinct) $b_1, \ldots, b_k \in B$,

$$\Pr_{h\in\mathcal{H}}[h(x_1) = b_1 \wedge \cdots \wedge h(x_k) = b_k] = |B|^{-k}.$$

The original definition of Carter and Wegman [4] differs slightly from the above, which is what they call *strongly k-universal* in [20]. In this paper, we use a uniform definition and restrict the domain to fixed-size binary strings and the range to $\{0, 1\}$. Thus, a *k-universal hash function* is a function $h : \{0, 1\}^n \times \{0, 1\}^s \to \{0, 1\}$ such that for all distinct $x_1, \ldots, x_k \in \{0, 1\}^n$ and all (not necessarily distinct) $b_1, \ldots, b_k \in \{0, 1\}$,

$$\Pr_{\sigma\in\{0,1\}^s}[h(x_1; \sigma) = b_1 \wedge \cdots \wedge h(x_k; \sigma) = b_k] = 2^{-k}.$$

We call the parameter σ, chosen uniformly at random, the *seed* of the function.

The Range-Summable Property. We are interested in one more property of hash functions, which has appeared in several forms in [3, 9, 8]. We will give our definition first, and then the prior definitions, which it subsumes. Call a hash function *bitwise range-summable*, if there is a polynomial-time (in n) function g that given a pair of vectors α and β in the domain $\{0, 1\}^n$, and a seed σ,

$$g(\alpha, \beta; \sigma) = \sum_{\alpha\leq x\leq\beta} h(x; \sigma). \tag{2.1}$$

Note that α, β, and x are binary vectors, so the bitwise restriction $\alpha \leq x \leq \beta$ has the effect of fixing some coordinate positions and leaving others to range over $\{0, 1\}$. Certainly the range-sum 2.1 can be evaluated directly (in $\Omega(2^n)$ time), however we would like to compute g more efficiently, using time polynomial in n.

The notion "range-summable" first appeared in the work of Feigenbaum *et al.* [8], which here we call *numerically range-summable*. Their definition has the restriction $\alpha \leq x < \beta$, where α, β, and x are natural numbers. However since every such numeric interval can be decomposed into a union of $O(n)$ dyadic intervals[1],

[1] An interval $[\alpha, \beta)$ is *dyadic* if $\alpha = i2^j$ and $\beta = (i+1)2^j$ for some non-negative integers i and j.

which in turn, can be expressed as our restriction on binary vectors, fixing some number of consecutive high-order coordinates and letting the remaining (low-order) coordinates range over $\{0, 1\}$. Bitwise range-summable includes numerically range-summable. Their construction was based on the extended Hamming error-correcting code and gave 3-wise independence. In a follow-up, Gilbert *et al.* [9] give a 7-universal construction based on the second-order Reed-Muller code. These hash functions were used to support point updates and range queries on a sketch data structure (see Sec. 5).

The concept of "range-summable" also appears under the name of "range-efficient" in the work of Bar-Yossef, Kumar, and Sivakumar [3] where it has the meaning of numerically range-summable described in the preceding paragraph. They also consider a multi-dimensional extension of this idea, where the domain is bounded integer tuples, and the sum is taken over all vectors where all but one of the coordinates are fixed, and the remaining coordinate is restricted to a given interval. They call such a construction "range-efficient in every coordinate," which we call *numerically range-summable in every coordinate.* It is easy to verify that bitwise range-summable also includes this definition. Bar-Yossef *et al.* used the 7-universal hash function to support range updates on sketches (again, see Sec. 5).

Reingold and Naor have also constructed a range-summable family of hash functions; for description, see [10]. However their construction, while conceptually elegant and of theoretical interest, is computationally costly, as it requires strong pseudo-random generators and sampling Gaussian random variables. Also see [11] for range-summable hash functions in Complexity Theory.

It is worth noting that our definition is not technically new, as it appears implicitly in [8], where the less general "range-summable" was more in line with the paper's application. Finally, we note that the most general notion of "range-summable" achieved by our constructions is what we call *range-summable over affine spaces*, which is a restriction on x of the form $Ax = b$, for $A \in \mathbb{F}_2^{m\times n}$ and $b \in \mathbb{F}_2^m$ for some integer m. However this seems too general for practical applications, and we only use this stronger property in Section 5.

3 Reed-Muller Hash Functions

In this section, we give explicit constructions of range-summable universal hash functions based on Reed-Muller codes.

An order-r Reed-Muller code with block length 2^n consists of all n-variable binary polynomials of degree at most r. The binary codeword corresponding to a polynomial is the binary 2^n-vector resulting from evaluating the polynomial at all 2^n possible values. For example, the second-order Reed-Muller code with block length 2^3 is consists of all linear combinations of polynomials $\mathbf{1}$, x_1, x_2, x_3, x_1x_2, x_1x_3, and x_2x_3. The codeword corresponding to the polynomial $\mathbf{1} + x_3 + x_1x_2$ is

$$\begin{array}{rcl}
x_1 & = & 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1 \\
x_2 & = & 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1 \\
x_3 & = & \underline{0\ 1\ 0\ 1\ 0\ 1\ 0\ 1} \\
\mathbf{1} + x_3 + x_1x_2 & & 1\ 0\ 1\ 0\ 1\ 0\ 0\ 1
\end{array}$$

The generator matrix for the above code is

$$\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1
\end{bmatrix},$$

where the rows correspond to the basis polynomials above. The *dual* of a code $\mathcal{C}$ is the set of vectors y such that $x^T y = 0$ for all $x \in \mathcal{C}$. The following is known about the Reed-Muller family of codes, and can be found in, for example [14]:

LEMMA 3.1. *The order-r Reed-Muller code has distance 2^{n-r} and its dual is the order-$(n-r-1)$ Reed-Muller Code.*

We can naturally construct a hash function from a linear code by choosing a random codeword and mapping x to bit x of the codeword. The resulting hash function has independence properties given by the following Lemma.

LEMMA 3.2. *Let $\mathcal{C}$ be a linear code whose dual has distance $k + 1$. Then the hash function constructed using $\mathcal{C}$ as above is k-universal.*

Thus, the hash function based on the order-r Reed-Muller code, given by an n-variable degree-r polynomial over $\mathbb{F}^2$ chosen uniformly at random, is $(2^r - 1)$-universal. Specifically,

$$h(x; \sigma) = \bigoplus_{\substack{I\subset[n] \\ |I|\le r}} \sigma_I \prod_{i\in I} x_i, \tag{3.2}$$

where $\sigma_I \in \{0,1\}$ is the seed vector indexed by subsets of $[n] = \{1,\ldots,n\}$. We use $\oplus$ to denote modulo-2 summation where there is a possibility of ambiguity. It follows that $|\sigma| = \sum_{i=0}^{r} \binom{n}{i} = O(n^r)$.

We now give an algorithm for evaluating the range-sum function g defined earlier as

$$g(\alpha,\beta;\sigma) = \sum_{\alpha\le x\le\beta} h(x;\sigma),$$

where α, β, and x are binary vectors. We begin by showing how the restriction $\alpha \le x \le \beta$ reduces to the case where x ranges over $\{0,1\}^{n'}$, for some $n' \le n$.

Reduction to the Unrestricted Case. Given a restriction of the form $\alpha \le x \le \beta$, we note that because α and β are binary vectors, the restriction effectively fixes some positions i (where $\alpha_i = \beta_i$) and leaves the remaining positions free to range over $\{0,1\}$. Let R be the set of fixed (restricted) positions and U be the set of free (unrestricted) positions. That is,

$$R = \{i : \alpha_i = \beta_i\} \quad \text{and} \quad U = \{i : \alpha_i < \beta_i\}.$$

We can reduce the restricted instance to a smaller unrestricted instance where the sum is over all x in $\{0,1\}^{|U|}$ by making the substitution $x_i \leftarrow \alpha_i$ for every $i \in R$. Thus, every term $x_{i_1}\cdots x_{i_k}\cdots x_{i_\ell}$ in $h(x;\sigma)$ where $i_1,\ldots,i_k \in R$ and $i_{k+1},\ldots,i_\ell \in U$ becomes $\alpha_{i_1}\cdots\alpha_{i_k} \cdot x_{k+1}\cdots x_k$; equivalently $\sigma_{\{i_1,\ldots,i_\ell\}}$ becomes 0, and $\sigma_{\{i_1,\ldots,i_k\}}$ becomes $\sigma_{\{i_1,\ldots,i_k\}} + \alpha_{i_1}\cdots\alpha_{i_k}$. From now on, because the sum restricted to $\alpha \le x \le \beta$ can be reduced to an unrestricted sum, we will only consider unrestricted sums.

Seed Sparsification. Intuitively, computing the range-sum efficiently is complicated by that fact that the value of the hash function may depend on arbitrary combinations of variables $x_1,\ldots,x_n$. If we could limit the dependence to small groups of variables, such that one group is independent of another, we would be able to perform the range-sum of each small group independently by brute force, and then combine it with the result of range-summing other blocks. We show how to convert an arbitrary Reed-Muller seed (corresponding to an instance of the hash function) into a seed which has the aforementioned type of independence, called a *sparse* seed, and then how to perform the range-sum efficiently using such a seed.

Later we will show how to efficiently compute the unrestricted range-sum for a certain class of seeds σ we call *sparse*, to be defined shortly; here, we will show how to transform a seed into a sparse seed.

For this procedure, it is more convenient to think of σ as a characteristic function for a set Σ. Therefore, let Σ be a family of subsets of $[n]$ such that $I \in \Sigma$ if $\sigma_I = 1$. Call a set I *maximal* (with respect to a given seed) if for all $J \supset I$, $J \notin \Sigma$. A seed is *sparse* if every maximal set of the seed is disjoint. For example, let $\Sigma = \{\{1,2\},\{2,3\}\}$, corresponding to $h(x) = x_1x_2 + x_2x_3$. Then $\{1,2\}$ and $\{2,3\}$ are both maximal, but the seed is not sparse because they intersect.

Making a seed sparse crucially depends on the following observation. Because we are summing over all x in $\{0,1\}^n$ (after the reduction in the previous section), we can change the order of summation, which includes any invertible linear transformation such as a change of variables of the form $x_i \leftarrow x_i + x_j$ for $i \ne j$ or $x_i \leftarrow x_i + 1$. Furthermore, we can view this as a way to introduce a set J into Σ, given a set $I \in \Sigma$ no smaller than J. Let $\Sigma = \{I\}$, then: pair each element i in $I\backslash J$ with an element j in $J\backslash I$, and make the substitution $x_i \leftarrow x_i + x_j$. For each unpaired element i in $I\backslash J$, make the substitution $x_i \leftarrow x_i + 1$. It is easy to verify, via the corresponding polynomials, that $I, J \in \Sigma$. (Note that Σ now also contains additional sets, equivalently, $h(x)$ contains additional terms, as a result of the substitutions.) Define a function $\Delta(I,J)$ to be the number of variable changes, as above, necessary to introduce J starting with $\Sigma = \{I\}$, or ∞ if $|I| < |J|$.

LEMMA 3.3. *Fix $I \in \Sigma$, and $J \notin \Sigma$. If $\Delta(I,J) \ne \infty$ then there is a sequence of $\Delta(I,J)$ or fewer variable changes of the form $x_i \leftarrow x_i + x_j$ and $x_i \leftarrow x_i + 1$ where $i \in I$ and $j \in J$ that results in $I, J \in \Sigma$.*

Proof. The proof is by induction on $\Delta(I,J)$. Consider the sequence of $\Delta(I,J)$ variable changes described above (for the case when $\Sigma = \{I\}$). The k-th change of variables introduces a set I_k such that $\Delta(I_k,J) = \Delta(I,J) - k$; i.e., we're getting "closer" to J. This process only fails if I_k is already in Σ, for then the introduced set I_k cancels with the existing I_k, since arithmetic is over $\mathbb{F}_2$. But then, we can omit the first k variable changes and, by induction, start from $I_k \in \Sigma$.

We are now ready to sparsify a seed using a sequence of variable changes as above. The algorithm follows.

1. Set $i = n$.
2. Let $I = \{i - r + 1, \ldots, i\}$; note that $|I| = r$. If $I \notin \Sigma$, introduce I as described above via a set $J \subseteq [i - r]$ already in Σ, unless all such sets in Σ are smaller then r, in which case, decrease r and start over from step 2.
3. Arrange the elements of Σ in the following order. For a set $J \in \Sigma$, consider $\{j_1, \ldots, j_k\} = J$ as an r-tuple where $j_1 > j_2 > \cdots > j_k$, padding with 0 as neccessary. E.g., if $I = \{1, 2, 3\}$ (thus $r = 3$), then $\{3, 5\} \in \Sigma$ becomes $(5, 3, 0)$. Now arrange the sets and corresponding tuples in decreasing lexicographic order of the tuples. So $(5, 3, 0)$ comes before $(5, 2, 1)$ which comes before $(4, 3, 0)$.
4. Let $\Pi = \{J : J \in \Sigma \text{ and } I \cap J \neq \emptyset \text{ and } I \setminus J \neq \emptyset\}$, the set of all elements of Σ that intersect with I but are not a subset of I. Note that for all $J \in \Pi$, we have $|J| \geq 2$. Informally, these are the sets that stand in the way of "sparseness" with respect to the maximal set I.
5. Pick the first (in the order of step 3) set in Π; call it J. Choose $I' \in \Sigma$ that minimizes $\Delta(I', J)$. (Note that I' may be I.) Now perform the variable changes of Lemma 3.3. Repeat steps 4 and 5 until $\Pi = \emptyset$.
6. Set $i \leftarrow i - r$. If $i < r$, we're almost done, so set i and r so that I is the remaining interval. Go to step 2.

We now establish the correctness of the algorithm.

LEMMA 3.4. *The algorithm above makes an arbitrary seed sparse.*

Proof. There are two parts to the proof: We show that once a set J is eliminated in step 5, it does not reappear, and therefore the algorithm terminates. We then argue that the resulting seed is sparse.

We argue by induction. First, we assert that in steps 4 and 5, for all $j \in J \in \Pi$, it is the case that $j \leq i$. Clearly this is true the first time steps 4 and 5 are completed ($i = n$). In subsequent executions of steps 4 and 5, if $j > i$, then by induction J was either the set I in a prior iteration of the outer loop (step 6) or was eliminated in a prior iteration, and by induction, did not reappear.

Now note that in the change of variables $x_i \leftarrow x_i + x_j$, we have that $j < i$ by the claim above and that $j \notin I$ (by construction of the sequence of variable changes). But any sets introduced as a result of these variable changes must occur *after* J in our ordering. This is because if J' is a counterexample, with corresponding tuple $(j'_1, j'_2, \ldots, j'_k)$, then there must be some ℓ such that $j'_\ell > j_\ell$; let it be the first such. But j'_ℓ must result from a variable change of the form $x_i \leftarrow x_i + x_{j'_\ell}$, which implies $j'_\ell \in J$, a contradiction.

Since in step 6 i is decremented so that the maximal sets I do not overlap, it is easy to verify that after termination the sets I are mutually disjoint.

LEMMA 3.5. *An order-r Reed-Muller seed can be made sparse in time $O(rn^{2r-1})$.*

Proof. The seed has $\sum_{i=0}^{r} \binom{n}{i} = O(n^r)$ elements, each of which requires up to r variable substitutions to eliminate, with each variable substitution affecting up to $\sum_{i=0}^{r-1} \binom{n}{i} = O(n^{r-1})$ terms.

Range-Sum with Sparse Seeds. Let Σ be a sparse seed, and, without loss of generality, let $[k] = \{1, \ldots, k\}$ be a maximal set. Since $[k]$ is mutually disjoint with respect to the other maximal sets, we can consider all 2^k possible assignments to the variables $x_1, \ldots, x_k$ independently of the remaining variables:

$$
\begin{aligned}
g(\sigma) &= \sum_{x \in \{0,1\}^n} \bigoplus_{\substack{I \subseteq [n] \\ |I| \leq r}} \sigma_I \prod_{i \in I} x_i \\
&= \sum_{x \in \{0,1\}^n} \Big[\bigoplus_{\substack{I \subseteq [1,k] \\ |I| \leq r}} \sigma_I \prod_{i \in I} x_i \oplus \bigoplus_{\substack{I \subseteq [k+1,n] \\ |I| \leq r}} \sigma_I \prod_{i \in I} x_i \Big] \\
&= \underbrace{\Big[\sum_{x \in \{0,1\}^k} \bigoplus_{\substack{I \subseteq [1,k] \\ |I| \leq r}} \sigma_I \prod_{i \in I} x_i \Big]}_{\mathcal{A}} \times \\
&\quad \underbrace{\Big[\sum_{x \in \{0,1\}^{n-k}} \bigoplus_{\substack{I \subseteq [k+1,n] \\ |I| \leq r}} \sigma_I \prod_{i \in I} x_{k+i} \Big]}_{\mathcal{B}}.
\end{aligned}
$$

The second equality follows by the sparseness. Now $\mathcal{A}$ can be evaluated directly, and $\mathcal{B}$ recursively as a smaller range-sum instance.

THEOREM 3.1. *Given k and $n > k$, there is a k-universal binary hash function with a seed* *of length* $\sum_{i=0}^{r}\binom{n}{i}$*, where* $r = \lceil \lg(k+1) \rceil - 1$*, that takes* $O(n^r)$ *time to compute the value at a point and* $O(rn^{2r-1})$ *time to compute the bitwise range-sum.*

Proof. The time to compute the hash function follows directly from the seed length. The range-sum cost follows from Lemma 3.5, since the sparsification dominates the computation.

Theorem 3.1 extends the bitwise range-summable property to arbitrary degrees of independence. Since for $r = 1$ and $r = 2$ the constructions are effectively the same as existing ones, it matches the running time for these cases. However when $r = 2$ and the set of free positions U is known *a priori*, we can do slightly better:

PROPOSITION 3.1. *Let* $r = 2$*, and fix* n *as well a set* U *of free positions (but not the assignments to the fixed positions); that is, for all* $i \in U$ *and all range-sum restrictions* $\alpha \leq x \leq \beta$*, we have* $0 = \alpha_i < \beta_i = 1$*, but for* $i \notin U$*, either* $\alpha_i = \beta_i = 0$ *or* $\alpha_i = \beta_i = 1$*. Then the bit-wise range-sum can be computed in time* $O(n^2)$ *after a one-time per-seed* $O(n^3)$ *preprocessing step for each seed.*

Proof. The idea is to compute the sparsification transformation *before* applying the restriction, allowing the sparsification step to be done once for each seed. However applying the restriction changes the seed, so we must take care to ensure that we can still apply the precomputed sparsification after we do learn the restriction. Fortunately, for $r = 2$, this is not a problem, for then the seed Σ contains sets of size 0, 1, and 2, corresponding to coefficients of the polynomial $\mathbf{1}$, coefficients of linear terms, and quadratic terms, respectively. Furthermore, note that the set Π formed in the sparsification procedure contains only sets of size 2, since a singleton or the empty set cannot both intersect with I and contain an element not in I. Since applying the restriction will leave unchanged only those sets of size 2 in Σ whose elements are both in U, the set Π is unaffected by applying the restriction. It follows that we can compute the sparsification, storing the variable changes as an affine transformation of size $O(n^2)$. After applying the restriction, we can apply the precomputed sparsification transformation in time $O(n^2)$.

Note that both the numeric range-sum operation and the numeric range-sum operation in all coordinates have a small number of possible free positions for which the seed can be pre-processed. Therefore, these two operations can be performed in time $O(n^2)$ compared to the $O(n^3)$ used previously.

$$\begin{bmatrix} 0 & q_1 & q_2 & q_3 & q_4 \\ 0 & 0 & q_3 & q_4 & q_5 \\ 0 & 0 & 0 & q_5 & q_6 \\ 0 & 0 & 0 & 0 & q_7 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 1: A 5×5 strictly upper triangular Hankel matrix. The matrix entries (i, j) is 0 if $i < j$ and identical on the skew diagonals. The portion of the corresponding polynomial made up of quadratic terms is $q_1x_1x_2 + q_2x_1x_3 + q_3(x_1x_4 + x_2x_3) + q_4(x_1x_5 + x_2x_4) + q_5(x_2x_5 + x_3x_4) + q_6x_3x_5 + q_7x_4x_5$.

4 Hankel Hash Functions

The second-order Reed-Muller code consists of all n-variable degree-2 polynomials over $\mathbb{F}_2$, which can be written as x^TQx+w^Tx+c (over $\mathbb{F}_2$), where the upper-triangular matrix $Q \in \{0,1\}^{n\times n}$, the vector $w \in \{0,1\}^n$ and the scalar $c \in \{0,1\}$ together define the codeword. We introduce a new code we call the Hankel code (because of the form of the matrix), which is a subcode of the second-order Reed-Muller code where Q is a strictly upper-triangular binary Hankel matrix, that is, a matrix identical on the skew-diagonal (see Fig. 1). We show that the dual of the Hankel code defined above has distance 5 (see the Appendix), so the hash function $h(x; Q, w, c) = x^TQx + w^Tx + c \mod 2$, where Q is strictly upper-triangular Hankel and Q, w, and c are chosen uniformly at random, is 5-universal, using the construction of Lemma 3.2. Because the Hankel code is a special subcode of the second-order Reed-Muller code, we can use it as we would a Reed-Muller-based hash function. However because of its special structure, we can evaluate it more efficiently than by straight matrix-vector multiplication suggested by $x^TQx + w^Tx + c$.

Evaluation. It turns out that the product x^TQx, which dominates the cost of evaluation, can be computed as a convolution using the Fast Fourier Transform. Recall that a convolution of two n-vectors x and y, denoted $x * y$, is a $2n$-vector z where $z_i = \sum_j x_jx_{i-j+1}$.

LEMMA 4.1. *There is an evaluation algorithm for Hankel dual hash functions that runs in time* $O(n \log^3 n)$.

Proof. Let q' be the $(2n-1)$-vector of the $2n-3$ unique entries of the upper-triangular Hankel matrix Q, prefixed with a zero entry and suffixed with a zero entry. Let $x' = (x_1, x_2, \ldots, x_n, 0, \ldots, 0)^T$ and $x'' = (x_1, 0, x_2, 0, \ldots, x_{n-1}, 0, x_n)^T$. Now it can be verified via direct calculation that $x^T Q x = \frac{1}{2}(x' * x' - x'')^T q'$. The convolution $x' * x'$ can be computed using the Fast Fourier Transform over a properly-chosen finite field of size $O(n)$, giving the desired result.

Range-Sum. The range-sum for a Hankel hash function can be computed using the algorithm in Section 3 for the second-order Reed-Muller code, because the Hankel code is a subcode of the Reed-Muller code.

THEOREM 4.1. *Given n, there is a 5-wise independent hash function with seed length* $3n-2$ *that takes* $O(n \log n \log\log n)$ *time to compute the value at a point and* $O(n^3)$ *time to compute the bitwise range-sum.*

Theorem 4.1 gives a more efficient way to achieve the 4-wise independence needed in many applications in stream processing. In the next section, we describe one such case.

5 Applications

In this section, we describe a *sketch*, a kind of synopsis data structure (see, for example, Muthukrishnan [15]) which supports queries and updates of ranges and points, as well as dot product estimation, including second-frequency moment estimation (due to space constraints, an analysis of the latter is omitted). Our sketch first appeared in [5], however it can be used in [1, 2, 3, 5, 7, 9], among others. Table 1 summarizes the improvements realized as a result of the new data structure.

The sketch consists of an array of integer counters, initially zero. A point update consists of hashing the input point x to one of 2^m counters using a 2-universal hash function, then hashing x to $\{1, -1\}$ using a 4-universal hash function and adding the result to the counter. Point queries/updates then follow as in [5]; however, using the analysis of [7] yields somewhat better error bounds. Range operations generally need to update every counter, however careful analysis reveals that some parts of the range-sum computation need only be performed once, rather than for each counter:

THEOREM 5.1. *For the sketch described above, using a 2-universal hash function to hash to counters and a 4-universal sign hash function, a point query/update can be computed in* $O(n \log^3 n)$*, a bitwise range query/update in* $O(n^3 + 2^m n^2)$ *time.*

Proof. We need to specify two hash functions: h_1 for hashing an element to a counter, and h_2 for hashing an element to $\{1, -1\}$. Hashing to a counter need only be 2-universal, so choose it to be a concatenation of first-order Reed-Muller hash functions: $h_1(x) = Px + q \mod 2$ where P is a binary $m \times n$ matrix and q is a binary vector of size m, both chosen uniformly at random. Choose h_2 to be a Hankel hash function, with the range suitably remapped to $\{1, -1\}$. This gives the desired complexity for the point operations. We now turn to the range operations.

For each counter, we would like to compute the bitwise range-sum of all points that hashed to that counter. The additional restriction that the points hash to a given bucket b (expressed as a binary m-vector) is just a restriction $Px = q + b$. Through Gaussian elimination, we can compute binary G and d such that $Gy + d$ enumerates all x satisfying the above. Substituting $Gy + d$ in for x in $x^T Q x + w^T x + c$ gives the desired unrestricted hash function, which we can then evaluate as before. This would require performing the above computation for every counter, however we notice that in the above, only the vector b changes between counters, and this does not affect the quadratic form $x^T Q x$, so we can compute the sparsification of Q once, and then apply the transformation to the linear and constant parts, which has cost $O(n^2)$ per counter.

Remark. The above construction achieves the desired error with some constant failure probability. In typical applications, the sketch is duplicated $O(\log \delta^{-1})$ times (using independent hash function seeds) to bring the failure probability down to δ. Since the convolution computation, which dominates the cost of the point operations, does not depend on the seed, it can be done once, bringing the total point update/query cost down to $O(n \log^3 n + nm \log \delta^{-1})$. However, the same optimization does not apply to range operations because their cost is dominated by com-

	Previous	New
Estimating F_2 (Prop. 7 of [3]): Range update time*	$O(\log\delta^{-1}\epsilon^{-2}dn^4)$	$O\big(\log\delta^{-1}(\epsilon^{-2}+n^3)\big)$
Wavelet decomposition ([9]): Point update time	$O(\log\delta^{-1}\epsilon^{-2}n^2)$	$O(n\log^3 n+\log\delta^{-1}n)$
Range query time	$O(\log\delta^{-1}\epsilon^{-2}n^3)$	$O\big(\log\delta^{-1}(n^3+\epsilon^{-2}n^2)\big)$

Table 1: Improvements to previous work realized by our 5-universal hash function (Sec. 4), where operations on integers up to the size of the data stream in magnitude have unit cost. *Here, d is the dimension of the input tuple; the improved update time for F_2 yields an improved update time for the triangle counting algorithms of [3].

putations involving the independent seeds themselves.

6 Conclusion

We introduced two new constructions of range-summable hash functions: a general k-universal construction using Reed-Muller codes, and a 5-universal one using a subcode of the second-order Reed-Muller code we call a Hankel code. The general construction gives $O(n^r)$ point operation time complexity and $O(n^{2r-1})$ range operation time complexity using $O(n^r)$ space, where $r = \lceil \log(k+1)\rceil - 1$. The Hankel code construction gives $O(n\log^3 n)$ point operation time complexity and $O(n^3)$ range operation time complexity, using $O(n)$ space. Because only 3-universal and 7-universal range-summable constructions were available previously, our contributions advance the development of range-summable hash functions, leading to immediate improvements to existing algorithms by using the sketch presented in Section 5 (see Table 1). Our more general construction for arbitrary k makes possible sublinear algorithms for many other problems such as estimating F_3 and higher frequency moments range-efficiently, counting subgraphs on 4 or more nodes in massive graphs, and so on.

Future Work. We note that our Hankel range-sum computation defers to the generic Reed-Muller range-sum algorithm; however it seems natural that the special structure of the Hankel code could be exploited to improve the time complexity of this operation.

More broadly, the full potential of range-summable universal hashing is yet to be fully explored, and we hope our work will stimulate further research in this area.

References

[1] N. Alon, P. Gibbons, Y. Matias, and M. Szegedy. Tracking Join and Self-Join Sizes in Limited Storage. *ACM Principles of Database Systems conference*, 10–20 (1999).

[2] N. Alon, Y. Matias, and M. Szegedy. The Space Complexity of Approximating the Frequency Moments. *Proc. of the* 28^{th} *Annual ACM Symposium on the Theory of Computing*, 20–29 (1996).

[3] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in Streaming Algorithms, with an Application to Counting Triangles in Graphs. *SODA 2002.*

[4] J. L. Carter and M. N. Wegman. Universal Classes of Hash Functions. *J. of Comp. and Syst. Sci.* 18, 143–154 (1979).

[5] M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. *Proc. of the 29th International Colloquium on Automata Languages and Programming* (2002).

[6] G. Cormode, S. Muthukrishnan. Estimating Dominance Norms of Multiple Data Streams. *ESA* 2003: 148-160.

[7] G. Cormode and S. Muthukrishnan. Improved Data Stream Summary: The Count-Min Sketch and its Applications. *DIMACS Technical Report 2003-20*, June 2003. LATIN 2004. To appear in *J. Algorithms.*

[8] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An Approximate L^1-Difference Algorithm for Massive Data Streams. *IEEE Symp. on Foundations of Computer Science*, 1999.

[9] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. One-Pass Wavelet Decompositions of Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 15 (3), 2003.

[10] A. C. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, M. Strauss. Fast,

small-space algorithms for approximate histogram maintenance. *STOC* 2002: 389-398.

[11] O. Goldreich, S. Goldwasser and A. Nussboim. On the implementation of huge random objects. ECCC eport TR03-045.

[12] D. Knuth. *Art of computer programming: Sorting and Searching*. Vol3. Addison Wesley. 1st Edition.

[13] M. Luby and A. Wigderson. Pairwise independence and derandomization. Survey. ICSI TR-95-035 UC Berkeley UCB/CSD-95-880 Ecole Normale Superieure LIENS-95-22. 1995.

[14] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. North-Holland, Amsterdam, 1977.

[15] S. Muthukrishnan. *Data stream algorithms*. `http://www.cs.rutgers.edu/~muthu/stream-1-1.ps`

[16] S. Muthukrishnan, M. Strauss. Maintenance of Multidimensional Histograms. *FSTTCS* 2003: 352-362

[17] S. Muthukrishnan, M. Strauss. Rangesum histograms. *SODA* 2003: 233-242.

[18] J. Naor, M. Naor. Small-Bias Probability Spaces: Efficient Constructions and Applications, SIAM J. Comput. 22(4): 838-856 (1993).

[19] M. Sipser. A Complexity Theoretic Approach to Randomness. *Proc. of the 15^{th} Annual ACM Symposium on the Theory of Computing*. 330–335 (1983).

[20] M. N. Wegman and J. L. Carter. New Hash Functions and Their Use in Authentication and Set Equality. *J. of Comp. and Syst. Sci.* 22, 265–279 (1981).

[21] A. Wigderson. The amazing power of pairwise independence (abstract). *STOC* 1994: 645-647

Appendix

The Hankel code consists of all n-variables polynomials in of the form $x^TQx + w^Tx + c$, where Q is a strictly upper-triangular binary Hankel matrix, w is a binary vector, and c is a binary scalar. A Hankel matrix that is identical on the skew diagonals (see Fig. 1). Note that all arithmetic is modulo 2.

THEOREM 6.1. *The dual of the Hankel code has minimum distance 5.*

Proof. Recall that the generator matrix for the second-order Reed-Muller code consists of the basis monomials 1, x_i for all i between **1** and n, and x_ix_j for all $i < j$ between 1 and n, evaluated at values $\{0,1\}^n$. (Note that this corresponds to the form $x^TQx + w^Tx + c$ where Q are the coefficients of the terms x_ix_j, w are the coefficients of the x_i, and c is the coefficient of 1.) From the form of the Hankel matrix, the basis of the Hankel code is given by the monomial $\mathbf{1}$, x_i for all i between 1 and n, and

$$\sum_{\substack{i<j\\ i+j=k}} x_ix_j,$$

for k between 2 and $2n$; each of the basis elements corresponds to a row of the generator matrix of the code consisting of the basis function evaluated at the 2^n possible inputs. The generator matrix of the Hankel code forms the parity check matrix of its dual, and we will show that every set of 5 columns of this matrix is linearly independent, so the distance of the corresponding code, namely the dual of the Hankel code, is 5.

First, note that the rows corresponding to the linear terms x_i for all $i \in [n]$ are exactly the rows of parity check matrix for the Hamming code, which has distance 3, and therefore, any two columns are linearly independent. The first row, a row of all ones, makes any odd number of columns linearly independent, so it remains to show that any four columns of this matrix are linearly independent, or, because we are working in $\mathbb{F}_2$, that the sum of any four distinct columns is non-zero.

Assume to the contrary, that is, that some four distinct vectors $x^{(1)}$, $x^{(2)}$, $x^{(3)}$, and $x^{(4)}$, such that the corresponding columns, consisting of the basis polynomials evaluated at those vectors, sum to zero. Consider the parity check matrix restricted to these columns. Since the rows sum to zero, each row must have an even number of ones. Now since $x^{(1)}$ and $x^{(3)}$ are distinct, they must differ in some position, say $\hat{\imath}$, and let it be the first such. Without loss of generality, say $x^{(1)}_{\hat{\imath}} = 0$ and $x^{(1)}_{\hat{\imath}} = 1$. Since the sum of the row corresponding to the basis monomial $x_{\hat{\imath}}$ is zero, either $x^{(2)}_{\hat{\imath}} = 1$ or $x^{(4)}_{\hat{\imath}} = 1$, but not both (because the number of ones must be even). Again, without loss of generality, say $x^{(4)}_{\hat{\imath}} = 1$. But $x^{(1)}$ and $x^{(2)}$ must also differ, say at position $\hat{\jmath} > \hat{\imath}$, and let $\hat{\jmath}$ be the first such. At the same position, $x^{(3)}$ and $x^{(4)}$ must also differ. Without loss of generality, say $x^{(1)}_{\hat{\jmath}} = 0$, $x^{(2)}_{\hat{\jmath}} = 1$, $x^{(3)}_{\hat{\jmath}} = 0$, and

$x_{\hat{\jmath}}^{(4)} = 1$. Thus we have

$$\begin{array}{cccccc} & & x^{(1)} & x^{(2)} & x^{(3)} & x^{(4)} \\ & & \vdots & \vdots & \vdots & \vdots \\ \hat{\imath} & \rightarrow & 0 & 0 & 1 & 1 \\ & & \vdots & \vdots & \vdots & \vdots \\ \hat{\jmath} & \rightarrow & 0 & 1 & 0 & 1 \\ & & \vdots & \vdots & \vdots & \vdots \end{array}$$

Now consider the row corresponding to the basis polynomial

$$\sum_{\substack{i<j \\ i+j=k}} x_i x_j$$

where $k = \hat{\imath} + \hat{\jmath}$. Since the rows sum to zero, we must have (modulo 2),

$$\sum_{\substack{i<j \\ i+j=k}} x_i^{(1)} x_j^{(1)} + x_i^{(2)} x_j^{(2)} + x_i^{(3)} x_j^{(3)} + x_i^{(4)} x_j^{(4)} = 0.$$

The rows before row $\hat{\imath}$ are either all zero or all one, the rows between row $\hat{\imath}$ and row $\hat{\jmath}$ may be all zero, all one, or like row $\hat{\imath}$. When $\hat{\imath} < i < j < \hat{\jmath}$, all rows have the form of row $\hat{\imath}$, so the terms of the summation cancel modulo 2. Similarly, when $i < \hat{\imath} < \hat{\jmath} < j$, the entries $x_i^{(1)}$, $x^{(2)_i}$, $x_i^{(3)}$, and $x_i^{(4)}$ are either all zero or all one, and an even number of the entries $x_j^{(1)}$, $x^{(2)_j}$, $x_j^{(3)}$, and $x_j^{(4)}$ are one, so those terms of the summation cancel. This leaves $x_{\hat{\imath}}^{(1)} x_{\hat{\jmath}}^{(1)} + x_{\hat{\imath}}^{(2)} x_{\hat{\jmath}}^{(2)} + x_{\hat{\imath}}^{(3)} x_{\hat{\jmath}}^{(3)} + x_{\hat{\imath}}^{(4)} x_{\hat{\jmath}}^{(4)}$, of which $x_i^{(1)} x_j^{(1)} = x_i^{(2)} x_j^{(2)} = x_i^{(3)} x_j^{(3)} = 0$, and $x_{\hat{\imath}}^{(1)} x_{\hat{\jmath}}^{(1)} = 1$, so the corresponding row sums to 1, a contradiction.