now
the essence of knowledge

Sketch Techniques for Approximate Query Processing

Graham Cormode[1]

[1] *180 Park Avenue, Florham Park, NJ, 07932, USA, graham@research.att.com*

Abstract

Sketch techniques have undergone extensive development within the past few years. They are especially appropriate for the data streaming scenario, in which large quantities of data flow by and the the sketch summary must continually be updated quickly and compactly. Sketches, as presented here, are designed so that the update caused by each new piece of data is largely independent of the current state of the summary. This design choice makes them faster to process, and also easy to parallelize.

"Frequency based sketches" are concerned with summarizing the observed frequency distribution of a dataset. From these sketches, accurate estimations of individual frequencies can be extracted. This leads to algorithms to find the approximate heavy hitters (items which account for a large fraction of the frequency mass) and quantiles (the median and its generalizations). The same sketches are also used to estimate (equi)join sizes between relations, self-join sizes and range queries. These can be used as primitives within more complex mining operations, and to extract wavelet and histogram representations of streaming data.

A different style of sketch construction leads to sketches for distinct-value queries. As mentioned above, using a sample to estimate the answer to a COUNT DISTINCT query does not give accurate results. In contract, sketch-

ing methods which can make a pass over the whole data can provide guaranteed accuracy. Once built, these sketches estimate not only the cardinality of a given attribute or combination of attributes, but also the cardinality of various operations performed on them, such as set operations (union and difference), and selections based on arbitrary predicates.

Contents

1

Sketches

1.1 Introduction

Of all the methods for approximate query processing presented in this volume, *sketches* have the shortest history, and consequently have had the least direct impact on real systems thus far. Nevertheless, their flexibility and power suggests that they will surely become a fixture in the next generation of approximate query processors. Certainly, they have already had significant impact within various specialized domains that process large quantities of structured data, in particular those that involve the *streaming* processing of data.

The notion of *streaming* has been popularized within recent years to capture situations where there is one chance to view the input, as it "streams" past the observer. For example, in processing financial data streams (streams of stock quotes and orders), many such transactions are witnessed every second, and a system must process these as they are seen, in real time, in order to facilitate real time data analysis and decision making. Another example that is closer to the motivating applications discussed so far is the sequence of updates to a traditional database—insertions and deletions to a given table—which also constitute a stream to be processed. Streaming algorithms typi-
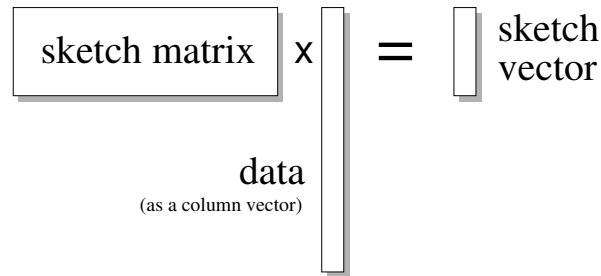
Fig. 1.1 Schematic view of linear sketching

cally create a compact synopsis of the data which has been observed, which is usually vastly smaller than the full data. Each update observed in the stream potentially causes this synopsis to be modified, so that at any moment the synopsis can be used to (approximately) answer certain queries over the original data. This fits exactly into our model of approximate query processing: provided we have chosen the right synopsis, the summary becomes a tool for AQP, in the same sense as a sample, or histogram or wavelet representation.

The earliest non-trivial streaming algorithms can be traced back to the late 1970s and early 1980s, when "pass efficient" algorithms for finding the median of a sequence and for finding the most frequently occurring items in a sequence were proposed [68, 65]. However, the growth in interest in streaming as a mechanism for coping with large quantities of data was stimulated by some influential papers in the late 1990s [2, 53], resulting in an explosion of work on stream processing in the first decade of the 21st Century.

We restrict our focus in this chapter to a certain class of streaming summaries known as *sketches*. This term has a variety of connotations, but in this presentation we use it to refer to a summary where each update is handled in the same way, irrespective of the history of updates. This notion is still rather imprecise, so we distinguish an important subset of *linear sketches*. These are data structures which can be represented as a *linear transform* of the input. That is, if we model a relation as defining a vector or matrix (think of the vector of discrete frequencies summarized by a histogram), then the sketch of this is found by multiplying the data by a (fixed) matrix. This is illustrated in Figure 1.1: a fixed sketch matrix multiplies the data (represented as a column vector) to generate the sketch (vector). Such a summary is therefore very flex-

ible: a single update to the underlying data (an insertion or deletion of a row) has the effect of modifying a single entry in the data vector. In turn, the sketch is modified by adding to the sketch the result of applying the matrix to this change alone. This meets our requirement that an update has the same impact irrespective of any previous updates. Another property of linear sketches is that the sketch of the union of two tables can be found as the (vector) sum of their corresponding sketches.

Any given sketch is defined for a particular set of queries. Queries are answered by applying some (technique specific) procedure to a given sketch. In what follows we will see a variety of different sketches. For some sketches, there are several different query procedures that can be used to address different query types, or give different guarantees for the same query type.

We comment that the idea of sketches, and in particular the linear transform view is not so very different from the summaries we have seen so far. Many histogram representations with fixed bucket boundaries can be thought of as linear transforms of the input. The Haar Wavelet Transform is also a linear transform[1]. However, for compactness and efficiency of computation, it is not common to explicitly materialize the (potentially very large) matrix which represents the sketch transform. Instead, all useful sketch algorithms perform a transform which is defined *implicitly* by a much smaller amount of information, often via appropriate randomly chosen hash functions. This is analogous to the way that a histogram transform is defined implicitly by its bucket boundaries, and the HWT is defined implicitly by the process of averaging and differencing.

1.2   Notation and Terminology

As in the preceding sections, we primarily focus on discrete data. We think of the data as defining a multiset D over a domain $\mathscr{U} = \{1, 2, \ldots, M\}$ so that $f(i)$ denotes the number of points in D having a value $i \in \mathscr{U}$. These $f(i)$ values therefore represent a set of frequencies, and can also be thought of as defining a vector f of dimension $M = |\mathscr{U}|$. In fact, many of the sketches we will describe here also apply to the more general case where each $f(i)$

---

[1] A key conceptual difference between the use of HWT and sketches is that the HWT is lossless, and so requires additional processing to produce a more compact summary via thresholding, whereas the sketching process typically provides data reduction directly.

can take on arbitrary real values, and even negative values. We say that f is "strict" when it can only take on non-negative values, and talk of the "general case" when this restriction is dropped.

The sketches we consider, which were primarily proposed in the context of streams of data, can be created from a stream of updates: think of the contents of D being presented to the algorithm in some order. Following Muthukrishnan [69], a stream updates is referred to as a "time-series" if the updates arrive in sorted order of i; "cash-register" if they arrive in some arbitrary order; and "turnstile" if items which have previously been observed can subsequently been removed. "Cash-register" is intended to conjure the image of a collection of unsorted items being rung up by a cashier in a supermarket, whereas "turnstile" hints at a venue where people may enter or leave. Streams of updates in the time-series or cash-register models necessarily generate data in the strict case, whereas the turnstile model can provide strict or general frequency vectors, depending on the exact situation being modeled.

These models are related to the datacube and relational models discussed already: the cash-register and turnstile models, and whether they generate strict or general distributions, can all be thought of as special cases of the relational model. Meanwhile, the time-series model is similar to the datacube model. Most of the emphasis in the design of streaming algorithms is on the cash-register and turnstile models. For more details on models of streaming computations, and on algorithms for streaming data generally, see some of the surveys on the topic [69, 3, 43].

Modeling a relation being updated with insert or delete operations, the number of rows with particular attribute values gives a strict turnstile model. But if the goal is to summarize the distribution of the sum of a particular attribute, grouped by a second attribute, then the general turnstile model may be generated. In both cases, sketch algorithms are designed to correctly reflect the impact of each update on the summary.

1.2.1   Simple Examples: Count, Sum, Average, Variance, Min and Max

Within this framework, perhaps the simplest example of a linear sketch computes the cardinality of a multiset D: this value N is simply tracked exactly, and incremented or decremented with each insertion into D or deletion from D respectively. The sum of all values within a numeric attribute can also be

sketched trivially by maintaining the exact sum and updating it accordingly. These fit our definition of being a (trivial) linear transformation of the input data. The average is found by dividing the sum by the count. Here, in addition to the maintenance of the sketch, it was also necessary to define an operation to extract the desired query answer from the sketch (the division operation). The sample variance of a frequency vector can also be computed in a sketching fashion, by tracking the appropriate sums and sums of squared values.

Considering the case of tracking the maximum value over a stream of values highlights the restriction that linear sketches must obey. There is a trivial sketch algorithm to find the maximum value of a sequence—just remember the largest one seen so far. This is a sketch, in the sense that every value is treated the same way, and the sketch maintenance process keeps the greatest of these. However, it is clearly *not* a linear sketch. Note that any linear sketch algorithm implicitly works in the turnstile model. But there can be no efficient streaming algorithm to find the maximum in a turnstile stream (where there are insertions and deletions to the dataset D): the best thing to do is to retain f in its entirety, and report the greatest i for which $f(i) > 0$.

1.2.2   Fingerprinting as sketching

As a more involved example, we describe an method to *fingerprint* a data set D using the language of sketching. A fingerprint is a compact summary of a multiset so that if two multisets are equal, then their fingerprints are also equal; and if two fingerprints are equal then the corresponding multisets are also equal with high probability (where the probability is over the random choices made in defining the fingerprint function). Given a frequency vector f, one fingerprint scheme computes a fingerprint as

$$h(f) = \sum_{i=1}^{M} f(i)\alpha^i \quad \mod p$$

where p is a prime number sufficiently bigger than M, and α is a value chosen randomly at the start. We observe that $h(f)$ is a linear sketch, since it is a linear function of f. It can easily be computed in the cash register model, since each update to $f(i)$ requires adding an appropriate value to $h(f)$ based on computing α^i and multiplying this by the change in $f(i)$ modulo p.

The analysis of this procedure relies on the fact that a polynomial of degree d can have at most d roots (where it evaluates to zero). Testing whether two multisets D and D' are equal, based on the fingerprints of their corresponding frequency vectors, $h(f)$ and $h(f')$, is equivalent to testing the identity $h(f) - h(f') = 0$. Based on the definition of h, if the two multisets are identical then the fingerprints will be identical. But if they are different and the test still passes, the fingerprint will give the wrong answer. Treating $h()$ as a polynomial in α, $h(f) - h(f')$ has degree no more than M: so there can only be M values of α for which $h(f) - h(f') = 0$. Therefore, if p is chosen to be at least M/δ, the probability (based on choosing a random α) of making a mistake is at most δ, for a parameter δ. This requires the arithmetic operations to be done using $O(\log M + \log 1/\delta)$ bits of precision, which is feasible for most reasonable values of M and δ.

Such fingerprints have been used in streaming for a variety of purposes. For example, Yi *et al.* [80] employ fingerprints within a system to verify outsourced computations over streams.

1.2.3   Comparing Sketching with Sampling

These simple examples seem straightforward, but they serve to highlight the difference between the models of data access assumed by the sketching process. We have already seen that a small sample of the data can only estimate the average value in a data set, whereas this "sketch" can find it exactly. But this is due in part to a fundamental difference in assumptions about how the data is observed: the sample "sees" only those items which were selected to be in the sample whereas the sketch "sees" the entire input, but is restricted to retain only a small summary of it. Therefore, to build a sketch, we must either be able to perform a single linear scan of the input data (in no particular order), or to "snoop" on the entire stream of transactions which collectively build up the input. Note that many sketches were originally designed for computations in situations where the input is never collected together in one place (as in the financial data example), but exists only implicitly as defined by the stream of transactions.

Another way to understand the difference in power between the models of sampling and streaming is to observe that it is possible to design algorithms to draw a sample from a stream (see Section ??), but that there are queries

that can be approximated well by sketches that are provably impossible to compute from a sample. In particular, we will see a number of sketches to approximate the number of distinct items in a relation (Section 1.4), whereas no sampling scheme can give such a guarantee (Section **??**). Similarly, we saw that fingerprinting can accurately determine whether two relations are identical, whereas unless every entry of two relations is sampled, it is possible that the two differ in the unsampled locations. Since the streaming model can simulate sampling, but sampling cannot simulate streaming, the streaming model is strictly more powerful in the context of taking a single pass through the data.

1.2.4 Properties of Sketches

Having seen these simple examples, we now formalize the main properties of a sketching algorithm.

- **Queries Supported.** Each sketch is defined to support a certain set of queries. Unlike samples, we cannot simply execute the query on the sketch. Instead, we need to perform a (possibly query specific) procedure on the sketch to obtain the (approximate) answer to a particular query.
- **Sketch Size.** In the above examples, the sketch is constant size. However, in the examples below, the sketch has one or more parameters which determine the size of the sketch. A common case is where parameters ε and δ are chosen by the user to determine the accuracy (approximation error) and probability of exceeding the accuracy bounds, respectively.
- **Update Speed.** When the sketch transform is very dense (i.e. the implicit matrix which multiplies the input has very few zero entries), each update affects all entries in the sketch, and so takes time linear in the sketch size. But typically the sketch transform can be made very sparse, and consequently the time per update may be much less than updating every entry in the sketch.
- **Query Time.** As noted, each sketch algorithm has its own procedure for using the sketch to approximately answer queries. The time to do this also varies from sketch to sketch: in some cases it
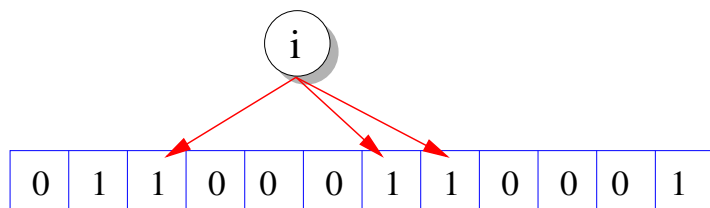
Fig. 1.2 Bloom Filter with $k = 3$, $m = 12$

is linear (or even superlinear) in the size of the sketch, whereas in other cases it can be much less.

- **Sketch Initialization.** By requiring the sketch to be a linear transformation of the input, sketch initialization is typically trivial: the sketch is initialized to the all-zeros vector, since the empty input is (implicitly) also a zero vector. However, if the sketch transform is defined in terms of hash functions, it may be necessary to initialize these hash functions by drawing them from an appropriate family.

1.2.5   Sketching Sets with Bloom Filters

As a more complex example, we briefly discuss the popular Bloom Filter as an example of a sketch. A Bloom filter, named for its inventor [8], is a compact way to represent a subset S of a domain $\mathcal{U}$. It consists of a binary string B of length $m < M$ initialized to all zeros, and k hash functions $h_1 \ldots h_k$, which each independently map elements of $\mathcal{U}$ to $\{1, 2, \ldots m\}$. For each element i in the set S, the sketch sets $B[h_j(i)] = 1$ for all $1 \le j \le k$. This is shown in Figure 1.2: an item i is mapped by $k = 3$ hash functions to a filter of size $m = 12$, and these entries are set to 1. Hence each update takes $O(k)$ time to process.

After processing the input, it is possible to test whether any given i is present in the set: if there is some j for which $B[h_j(i)] = 0$, then the item is not present, otherwise it is concluded that i is in S. From this description, it can be seen that the data structure guarantees no false negatives, but may report false positives.

Analysis of the Bloom Filter.   The false positive rate can be analyzed as a function of $|S| = n$, m and k: given bounds on n and m, optimal values of k can be set. We follow the outline of Broder and Mitzenmacher [10] to derive the relationship between these values. For the analysis, the hash functions are assumed to be fully random. That is, the location that an item is mapped to by any hash function is viewed as being uniformly random over the range of possibilities, and fully independent of the other hash functions. Consequently, the probability that any entry of B is zero after n distinct items have been seen is given by

$$p' = \left(1 - \frac{1}{m}\right)^{kn}$$

since each of the kn applications of a hash function has a $(1 - \frac{1}{m})$ probability of leaving the entry zero.

A false positive occurs when some item not in S hashes to locations in B which are all set to 1 by other items. This happens with probability $(1 - \rho)^k$, where ρ denotes the fraction of bits in B that are set to 0. In expectation, ρ is equal to p', and it can be shown that ρ is very close to p' with high probability. Given fixed values of m and n, it is possible to optimize k, the number of hash functions. Small values of k keep the number of 1s lower, but make it easier to have a collision; larger values of k increase the density of 1s. The false positive rate is approximated well by

$$f = (1 - e^{-kn/m})^k = \exp(k \ln(1 - e^{kn/m}))$$

for all practical purposes. The smallest value of f as a function of k is given by minimizing the exponent. This in turn can be written as $-\frac{m}{n} \ln(p) \ln(1 - p)$, for $p = e^{-kn/m}$, and so by symmetry, the smallest value occurs for $p = \frac{1}{2}$. Rearranging gives $k = (m/n) \ln 2$.

This has the effect of setting the occupancy of the filter to be 0.5, that is, half the bits are expected to be 0, and half 1. This causes the false positive rate to be $f = (1/2)^k = (0.6185)^{m/n}$. To make this probability at most a small constant, it is necessary to make $m > n$. Indeed, setting $m = cn$ gives the false positive probability at 0.6185^c: choosing $c = 9.6$, for example, makes this probability less than 1%.

Bloom Filter viewed as a sketch.   In this form, we consider the Bloom filter to be a sketch, but it does not meet our stricter conditions for being

considered as a linear sketch. In particular, the data structure is not a linear transform of the input: setting a bit to 1 is not a linear operation. We can modify the data structure to make it linear, at the expense of increasing the space required. Instead of a bitmap, the Bloom filter is now represented by an array of counters. When adding an item, we increase the corresponding counters by 1, i.e. $B[h_j(i)] \leftarrow B[h_j(i)] + 1$. Now the transform is linear, and so it can process arbitrary streams of update transactions (including removals of items). The number of entries needed in the array remains the same, but now the entries are counters rather than bits.

One limitation when trying to use the Bloom Filter to describe truly large data sets is that the space needed is proportional to n, the number of items in the set S being represented. Within many approximate query processing scenarios, this much space may not be practical, so instead we look for more compact sketches. These smaller sketches will naturally be less powerful than the Bloom filter: when using a datastructure that is sublinear in size (i.e. $o(n)$) we should not expect to be accurately answer all set-membership queries, even allowing for false positives and false negatives.

1.3 Frequency Based Sketches

In this Section, we present a selection of sketches which solve a variety of problems related to estimating functions of the frequencies, $f(i)$. Our presentation deliberately does not follow the chronological development of these sketches. Instead, we provide the historical context later in the section. We first define each sketch and the basic properties. In later sections, we study them in greater detail for approximate query answering.

1.3.1 Count-Min Sketch

The *Count-Min* sketch is so-called because of the two main operations used: counting of groups of items, and taking the minimum of various counts to produce an estimate [25]. It is most easily understood as keeping a compact array C of $d \times w$ counters, arranged as d rows of length w. For each row a *hash function* h_j maps the input domain $\mathscr{U} = \{1, 2, \ldots, M\}$ uniformly onto
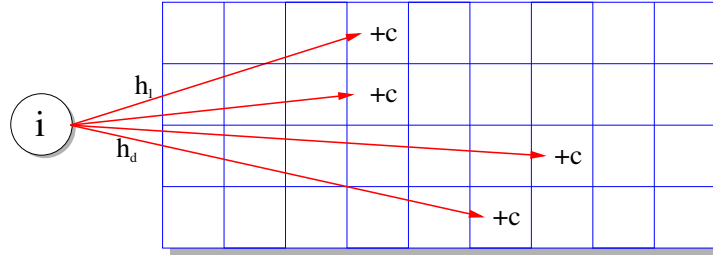
Fig. 1.3 Count-Min sketch data structure with $w = 9$ and $d = 4$

the range $\{1, 2, \ldots, w\}$. The sketch C is then formed as

$$C[j,k] = \sum_{1 \leq i \leq M : h_j(i) = k} f(i)$$

That is, the kth entry in the jth row is the sum of frequencies of all items i which are mapped by the jth hash function to value k. This leads to an efficient update algorithm: for each update to item i, for each $1 \leq j \leq d$, $h_j(i)$ is computed, and the update is added to entry $C[j, h_j(i)]$ in the sketch array. Processing each update therefore takes time $O(d)$, since each hash function evaluation takes constant time. Figure 1.3 shows this process: an item i is mapped to one entry in each row j by the hash function h_j, and the update of c is added to each entry.

The sketch can be used to estimate a variety of functions of the frequency vector. The primary function is to recover an estimate of $f(i)$, for any i. Observe that for it to be worth keeping a sketch in place of simply storing f exactly, it must be that wd is much smaller than M, and so the sketch will necessarily only approximate any $f(i)$. The estimation can be understood as follows: in the first row, it is the case that $C[1, h_1(i)]$ includes the current value of $f(i)$. However, since $w \ll M$, there will be many collisions under the hash function h_1, so that $C[1, h_1(i)]$ also contains the sum of all $f(i')$ for i' that collides with i under h_1. Still, if the sum of such $f(i')$s is not too large, then this will not be so far from $f(i)$.

In the strict case, all these $f(i')$s are non-negative, and so $C[1, h_1(i)]$ will be an overestimate for $f(i)$. The same is true for all the other rows: for each j, $C[j, h_j(i)]$ gives an overestimate of $f(i)$, based on a different set of colliding items. Now, if the hash functions are chosen at random, the items will be dis-

tributed uniformly over the row. So the expected amount of "noise" colliding with i in any given row is just $\sum_{1 \leq i' \leq M, i' \neq i} f(i')/w$, a $1/w$ fraction of the total count. Moreover, by the Markov inequality [67, 66], there is at least a 50% chance that the noise is less than twice this much. Here, the probabilities arise due to the random choice of the hash functions. If each row's estimate of $f(i)$ is an overestimate, then the smallest of these will be the closest to $f(i)$. By the independence of the hash functions, it is now very unlikely that this estimate has error more than $2\sum_{1 \leq i' \leq M} f(i')/w$: this only happens if *every* row estimate is "bad", which happens with probability at most 2^{-d}.

Rewriting this, if we pick $w = 2/\varepsilon$ and $d = \log 1/\delta$, then our estimate of $f(i)$ has error at most εN with probability at least $1 - \delta$. Here, we write $N = \sum_{1 \leq i' \leq M} f(i')$ as the sum of all frequencies—equivalently, the number of rows in the defining relation if we are tracking the cardinality of attribute values. The estimate is simply $\hat{f}(i) = \min_{j=1}^{d} C[j, h_j(i)]$. Producing the estimate is quite similar to the update procedure: the sketch is probed in one entry in each row (as in Figure 1.3). So the query time is the same as the update time, $O(d)$.

1.3.1.1   Perspectives on the Count-Min sketch

At its core, the Count-Min sketch is quite simple: just arrange the input items into groups, and compute the net frequency of the group. As such, we can think of it as a histogram with a twist: first, randomly permute the domain, then create an equi-width histogram with w buckets on this new domain. This is repeated for d random permutations. Query answering to estimate a single $f(i)$ is to find all the histogram buckets the item i is present in, and take the smallest of these. Viewed from another angle, the sketch can also be viewed as a small space, counting version of a Bloom filter [10, 16].

In this presentation, we omit detailed discussion of some of the technical issues surrounding the summary. For example, for the analysis, the hash functions are required to be drawn from a family of pairwise independent functions. However this turns out to be quite a weak condition: such functions are very simple to construct, and can be evaluated very quickly indeed [12, 75]. The estimator described is technically *biased*, in the statistical sense: it never underestimates but may overestimate, and so is not correct in expectation. However, it is straightforward to modify the estimator to be unbiased, by

subtracting an appropriate quantity from the estimate. Heuristically, we can estimate the count of some "dummy" items such as $f(M+1)$ whose "true count" should be zero to estimate the error in the estimation [58]. We discuss the variant estimators in more detail in 1.3.5.3.

Lastly, the same sketch can also be used when the stream is general, and so can contain some items with negative frequencies. In this case, the sketch can be built in the same way, but now it is not correct to take the smallest row estimate as the overall estimate: this could be far from the true value if, for example, all the $f(i)$ values are negative. Instead, one can take the *median* of the row estimates, and apply the following general "Chernoff bounds argument".

Chernoff Bounds Argument. Suppose there are multiple independent copies of an estimator, each of which is a "good" estimate of a desired quantity with at least a constant probability (although it's not possible to tell whether or not an estimate is good just by looking at it). The goal is to combine these to make an estimate which is "good" with high probability. A standard technique is to take the *median* of enough estimates to reduce the error. Although it is not possible to determine which estimates are good or bad, sorting the estimates by value will place all the "good" estimates together in the middle, with "bad" estimates above and below (too low or too high). Then the only way that the median estimate can be bad is if more than half of the estimates are bad, which is unlikely. In fact, the probability of returning a bad estimate is now exponentially small in the number of estimates.

The proof makes use of a Chernoff bound. Assume that each estimate is good with probability at least $7/8$. The outcome of each estimate is an independent random event, so in expectation only $1/8$ of the estimates are bad. So the final result is only bad if the number of bad events exceeds it expectation by a factor of 4. Set the number of estimates to be $4\ln 1/\delta$ for some desired small probability δ. The Chernoff bound (slightly simplified) in this situation states that if X is the sum of independent Poisson trials, then for $0 < \rho \leq 4$,

$$\Pr[X > (1+\rho)\mathsf{E}[X]] < \exp(-\mathsf{E}[X]\rho^2/4).$$

See [67] for a derivation of this bound. Since each estimate is indeed an independent Poisson trial, then this setting is modeled with $\rho = 3$ and $\mathsf{E}[X] =$
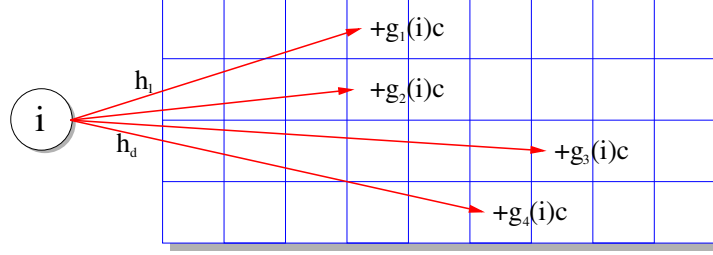
Fig. 1.4 Count Sketch data structure with $w = 9$ and $d = 4$

$\frac{1}{2}\ln 1/\delta$. Hence,

$$\Pr[X > 2\log 1/\delta] < \exp(-9/8\ln 1/\delta) < \delta$$

This implies that the taken the median of $O(\log 1/\delta)$ estimates reduces the probability of finding a bad final estimate to δ.

1.3.2 Count Sketch

The Count-Sketch [14] is similar to the Count-Min sketch, in that it can provide an estimate for the value of any individual frequency. The main difference is the nature of the accuracy guarantee provided for the estimate. Indeed, we can present the Count Sketch by using exactly the same sketch building procedure as the Count-Min sketch, so that the *only* difference is in the estimation procedure.

Now, given the Count-Min sketch data structure built on the stream, the row estimate of $f(i)$ for row j is computed based on two buckets: $h_j(i)$, the bucket which contains $f(i)$, and also

$$h'_j(i) = \begin{cases} h_j(i) - 1 & \text{if } h_j(i) \mod 2 = 0 \\ h_j(i) + 1 & \text{if } h_j(i) \mod 2 = 1 \end{cases}$$

which is an adjacent bucket [2]. So if $h_j(i) = 3$ then $h'_j(i) = 4$, while if $h_j(i) = 6$ then $h'_j(i) = 5$. The row estimate is then $C[j, h_j(i)] - C[j, h'_j(i)]$. Here, we assume that w, the length of each row in the sketch, is even.

The intuition for this estimate comes from considering all the "noise" items which collide with i: the distribution of such items in the $h_j(i)$th entry

---

[2]Equivalently, this can also be written as $[h'_j(i) = h_j(i) + (h_j(i) \mod 2) - (h_j(i) + 1 \mod 2)$

of row j should look about the same as the distribution in the $h'_j(i)$th entry, and so in expectation, these will cancel out, leaving only $f(i)$. More strongly, one can formally prove that this estimator is unbiased. Of course, the estimator still has variance, and so does not guarantee the correct answer. But this variance can be analyzed, and written in terms of the sum of squares of the items, $F_2 = \sum_{i=1}^{M} f(i)^2$. It can be shown that the variance of the estimator is bounded by $O(F_2/w)$. As a result, there is constant probability that each row estimate is within $\sqrt{F_2/w}$ of $f(i)$. Now by taking the median of the d row estimates, the probability of the final estimate being outside these bounds shrinks to $2^{-O(d)}$. Rewriting, if the parameters are picked as $d = O(\log 1/\delta)$ and $w = O(1/\varepsilon^2)$, the sketch guarantees to find an estimate of $f(i)$ so that the error is at most $\varepsilon\sqrt{F_2}$ with probability at least $1 - \delta$.

In fact, this sketch can be compacted further. Observe that whenever a row estimate is produced, it is found as either $C[j, 2k-1] - C[j, 2k]$ or $C[j, 2k] - C[j, 2k-1]$ for some (integer) k. So rather than maintaining $C[j, 2k-1]$ and $C[j, 2k]$ separately, it suffices to keep this difference in a single counter. This can be seen by maintaining separate hash functions g_j which maps all items in $\mathcal{U} = \{1, 2, \ldots, M\}$ uniformly onto $\{-1, +1\}$. Now the sketch is defined via

$$C[j, k] = \sum_{1 \leq i \leq M : h_j(i) = k} g_j(i) f(i).$$

This meets the requirements for a linear sketch, since it is a linear function of the f vector. It can also be computed in time $O(d)$: for each update to item i, for each $1 \leq j \leq d$, $h_j(i)$ is computed, and the update multiplied by $g_j(i)$ is added to entry $C[j, h_j(i)]$ in the sketch array. The row estimate for $f(i)$ is now $g_j(i) * C[j, h_j(i)]$. It can be seen that this version of the sketch is essentially equivalent to the version described above, and indeed all the properties of the sketch are the same, except that w can be half as small as before to obtain the same accuracy bounds. This is the version that was originally proposed in the 2002 paper [14]. An example is shown in Figure 1.4: an update is mapped into one entry in each row by the relevant hash function, and multiplied by a second hash function g. The figure serves to emphasize the similarities between the Count Sketch and Count-Min sketch: the main difference arises in the use of the g_j functions.

1.3.2.1   Refining Count-Sketch and Count-Min Sketch guarantees

We have seen that the Count-Sketch and Count-Min sketch both allow $f(i)$ to be approximated via somewhat similar data structures. They differ in providing distinct space/accuracy trade-offs: the Count sketch gives $\varepsilon\sqrt{F_2}$ error with $O(1/\varepsilon^2)$ space, whereas the Count-Min sketch gives εN error with $O(1/\varepsilon)$ space. In general, these bounds cannot be compared: there exist some frequency vectors where (given the same overall space budget) one guarantee is preferable, and others where the other dominates. Indeed, various experimental studies have shown that over real data it is not always clear which is preferable [22].

However, a common observation from empirical studies is that these sketches give better performance than their worst case guarantees would suggest. This can be explained in part by a more rigorous analysis. Most frequency distributions seen in practice are *skewed*: there are a few items with high frequencies, while most have low frequencies. This phenomenon is known by many names, such as Pareto, Zipfian, and long tailed distributions. For both Count Sketch and Count-Min sketch, a skewed distribution *helps* estimation: when estimating a frequency $f(i)$, it is somewhat unlikely that any of the few high frequency items will collide with i under h_j —certainly it is very unlikely that any will collide with i in a majority of rows. So it is possible to separate out some number k of the most frequent items, and separately analyze the probability that i collides with them. The probability that these items affect the estimation of $f(i)$ can then be bounded by δ. This leaves only a "tail" of lower frequency items, which still collide with i with uniform probability, but the net effect of this is lower since there is less "mass" in the tail. Formally, let f_i denote the ith largest frequency in f, and let

$$F_1^{\mathrm{res}(k)} = \sum_{i=k+1}^{M} f_i \qquad \text{and} \qquad F_2^{\mathrm{res}(k)} = \sum_{i=k+1}^{M} f_i^2$$

denote the sum and the sum of squares of all but the k largest frequencies. As a result, we can bound the error in the sketch estimates in terms of $F_1^{\mathrm{res}(k)}/w$ and $\sqrt{F_2^{\mathrm{res}(k)}/w}$ for the Count-Min and Count sketch respectively, where $k = O(w)$ [14, 26].

1.3.3   The AMS Sketch

The AMS sketch was first presented in the work of Alon, Matias and Szegedy [2]. It was proposed to solve a different problem, to estimate the value of F_2 of the frequency vector, the sum of the squares of the frequencies. Although this is straightforward if each frequency is presented in turn, it becomes more complex when the frequencies are presented implicitly, such as when the frequency of an item is the number of times it occurs within a long, unordered, stream of items. Estimating F_2 may seem like a somewhat obscure goal in the context of approximate query processing. However, it has a surprising number of applications. Most directly, F_2 equates to the self-join size of the relation whose frequency distribution on the join attribute is f (for an equi-join). The AMS sketch turns out to be highly flexible, and is at the heart of estimation techniques for a variety of other problems which are all of direct relevance to AQP.

1.3.3.1   AMS Sketch for Estimating F_2

We again revert to the sketch data structure of the Count-Min sketch as the basis of the AMS sketch, to emphasize the relatedness of all these sketch techniques. Now each row is used in its entirety to make a row estimate of F_2 as

$$\sum_{k=1}^{w/2} (C[j, 2k-1] - C[j, 2k])^2.$$

Expanding out this expression in terms of $f(i)$s, it is clear that the resulting expression includes $\sum_{i=1}^{M} f(i)^2 = F_2$. However, there are also a lot of cross terms of the form $\pm 2f(i)f(i')$ for $i \neq i'$ such that either $h_j(i) = h_j(i')$ or $|h_j(i) - h_j(i')| = 1$. That is, we have errors due to cross terms of frequencies of items placed in the same location or adjacent locations by h_j. Perhaps surprisingly, the expected contribution of these cross terms is zero. There are three cases to consider for each i and i' pair: (a) they are placed in the same entry of the sketch, in which case they contribute $2f(i)f(i')$ to the estimate; (b) one is placed in the $2k$th entry and the other in the $2k-1$th, in which case they contribute $-2f(i)f(i')$ to the estimate; or (c) they are not placed in adjacent entries and so contribute 0 to the estimate. Due to the uniformity of h_j, cases (a) and (b) are equally likely, so in expectation (over the choice of

h_j) the *expected* contribution to the error is zero.

Of course, in any given instance there are some i, i' pairs which contribute to the error in the estimate. However, this can be bounded by studying the variance of the estimator. This is largely an exercise in algebra and applying some inequalities (see [76, 2] for some details). The result is that the variance of the row estimator can be bounded in terms of $O(F_2^2/w)$. Consequently, using the Chebyshev inequality [67], the error is at most $F_2/\sqrt{w}$ with constant probability. Taking the median of the d rows reduces the probability of giving a bad estimate to $2^{-O(d)}$, by the Chernoff bounds argument outlined above.

As in the Count-Sketch case, this sketch can be "compacted" by observing that it is possible to directly maintain $C[j, 2k-1] - C[j, 2k]$ in a single entry, by introducing a second hash function g_j which maps $\mathcal{U}$ uniformly onto $\{-1, +1\}$. Technically, a slightly stronger guarantee is needed on g_j: because the analysis studies the variance of the row estimator, which is based on the expectation of the square of the estimate, the analysis involves looking at products of the frequencies of four items and their corresponding g_j values. To bound this requires g_j to appear independent when considering sets of four items together: this adds the requirement that g_j be *four-wise* independent. This condition is slightly more stringent than the pairwise independence needed of h_j.

Practical Considerations for Hashing.   Although the terminology of pairwise and four-wise independent hash functions may be unfamiliar, they should not be thought of as exotic or expensive. A family of pairwise independent hash functions is given by the functions $h(x) = ax + b \mod p$ for constants a and b chosen uniformly between 0 and $p-1$, where p is a prime. Over the random choice of a and b, the probability that two items collide under the hash function is $1/p$. Similarly, a family of four-wise independent hash functions is given by $h(x) = ax^3 + bx^2 + cx + d \mod p$ for a, b, c, d chosen uniformly from $[p]$ with p prime. As such, these hash functions can be computed very quickly, faster even than more familiar (cryptographic) hash functions such as MD5 or SHA-1. For scenarios which require very high throughput, Thorup has studied how to make very efficient implementations of such hash functions, based on optimizations for particular values of p, and partial precomputations [75, 76].

Consequently, this sketch can be very fast to compute: each update requires only d entries in the sketch to be visited, and a constant amount of hashing work done to apply the update to each visited entry. The depth d is set as $O(\log 1/\delta)$, and in practice this is of the order of 10-30, although d can be set as low as 3 or 4 without obvious problem [22].

AMS sketch with Averaging versus Hashing.   In fact, the original description of the AMS sketch gave an algorithm the was considerably slower: the original AMS sketch was essentially equivalent to the sketch we have described with $w = 1$ and $d = O(\varepsilon^{-2}\log 1/\delta)$. Then the mean of $O(\varepsilon^{-2})$ entries of the sketch was taken, to reduce the variance, and the final estimator found as the median of $O(\log 1/\delta)$ such independent estimates. We refer to this as the "averaging version" of the AMS sketch. This estimator has the same space cost as the version we present here, which was the main objective of [2]. The faster version, based on the "hashing trick" is sometimes referred to as "fast AMS" to distinguish it from the original sketch, since each update is dramatically faster.

Historical Notes.   Historically, the AMS Sketch [2] was the first to be proposed as such in the literature, in 1996. The "Random Subset Sums" technique can be shown to be equivalent to the AMS sketch for estimating single frequencies [49]. The Count-Sketch idea was presented first in 2002. Crucially, this seems to be the first work where it was shown that hashing items to w buckets could be used instead of taking the mean of w repetitions of a single estimator, and that this obtains the same accuracy. Drawing on this, the Count-Min sketch was the first to obtain a guarantee in $O(1/\varepsilon)$ space, albeit for an F_1 instead of F_2 guarantee [25]. Applying the "hashing trick" to the AMS sketch make it very fast to update seems to have been discovered in parallel by several people. Thorup and Zhang were among the first to publish this idea [76], and an extension to inner-products appeared in [20].

1.3.4   Approximate Query Processing with Frequency Based Sketches

Now that we have seen the definitions and basic properties of the Count-Min Sketch, Count-Sketch and AMS Sketch, we go on to see how they can be applied to approximate the results of various aggregation queries.

1.3.4.1   Point Queries and Heavy Hitters

Via Count-Min and Count Sketch, we have two mechanisms to approximate the frequency of any given item i. One has accuracy proportional to εN, the other proportional to $\varepsilon \sqrt{F_2}$. To apply either of these, we need to have in mind some particular item i which is of interest. A more common situation arises when we have no detailed *a priori* knowledge of the frequency distribution, and we wish to find which are the most significant items. Typically, those most significant items are those which have high frequencies – the so-called "heavy hitters". More formally, we define the set of heavy hitters as those items whose frequency exceeds a ϕ fraction of the total frequency, for some chosen $0 < phi < 1$.

The naive way to discover the heavy hitters within a relation is to exhaustively query each possible i in turn. The accuracy guarantees indicate that the sketch should correctly recover those items with $f(i) > \varepsilon N$ or $f(i) > \varepsilon \sqrt{F_2}$. But this procedure can be costly, or impractical, when the domain size M is large—consider searching a space indexed by a 32 or 64 bit integer. The number of queries is so high that there may be false positives unless d is chosen to be sufficiently large to ensure that the overall false positive probability is driven low enough.

In the cash-register streaming model, where the frequencies only increase, a simple solution is to combine update with search. Note that an item can only become a heavy hitter in this model following an arrival of that item. So the current set of heavy hitters can be tracked in a data structure separate to the sketch, such as a heap or list sorted by the estimated frequency [14]. When the frequency of an item increases, at the same time the sketch can be queried to obtain the current estimated frequency. If the item exceeds the current threshold for being a heavy hitter, it can be added to the data structure. At any time, the current set of (approximate) heavy hitters can be found by probing this data structure.

We can compare this to results from sampling: standard sampling results argue that to find the heavy hitters with εN accuracy, a sample of size $O(1/\varepsilon^2)$ items is needed. So the benefits of the Count-Min sketch are clear: the space required is quadratically smaller to give the same guarantee ($O(1/\varepsilon)$ compared to $O(1/\varepsilon^2)$). However, the benefits become more clear in the turnstile case when, if there is significant numbers of deletions in the data, causing the
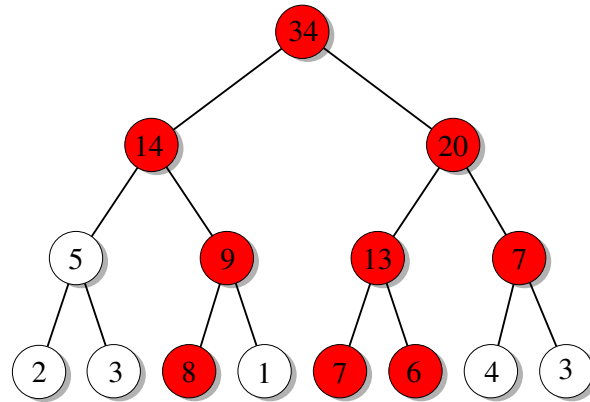
Fig. 1.5 Searching for heavy hitters via binary-tree search

set of Heavy Hitters to change considerably over time. In such cases it is not possible to draw a large sample from the input stream.

Heavy Hitters over Strict Distributions.   When the data arrives in the turnstile model, decreases in the frequency of one item can cause another item to become a heavy-hitter implicitly—because its frequency now exceeds the (reduced) threshold. So the method of keeping track of the current heavy hitters as in the cash-register case will no longer work.

It can be more effective to keep a more complex sketch, based on multiple instances of the original sketch over different views of the data, to allow more efficient retrieval of the heavy hitters. The "divide and conquer" or "hierarchical search" technique conceptually places a fixed tree structure, such as a binary tree structure, over the domain. Each internal node is considered to correspond to the set of items covered by leaves in the induced subtree. Each internal node is treated as a new item, whose frequency is equal to the sum of the frequencies of the items associated with that node. In addition to a sketch of the leaf data, a sketch of each level of the tree (the frequency distribution of each collection of nodes at the same distance from the root) is kept.

Over frequency data in the strict model, it is the case that each ancestor of a heavy hitter leaf must be at least as frequent, and so must appear as a heavy hitter also. This implies a simple divide-and-conquer approach to finding the heavy hitter items: starting at the root, the appropriate sketch is used to es-

timate the frequencies of all children of each current "candidate" node. All nodes whose estimated frequency makes them heavy hitters are added to the list of candidates, and the search continues down the tree. Eventually, the leaf level will be reached, and the heavy hitters should be discovered. Figure 1.5 illustrates the process: given a frequency distribution at the leaf level, a binary tree is imposed, where the frequency of the internal nodes is the sum of the leaf frequencies in the subtree. Nodes which are heavy (in this example, whose frequency exceeds 6) are shaded. All unshaded nodes can be ignored in the search for the heavy hitters.

Using a tree structure with a constant fan-out at each node, there are $O(\log M)$ levels to traverse. Defining a heavy hitter as an item whose count exceeds ϕN for some fraction ϕ, there are at most $1/\phi$ possible (true) heavy hitters at each level. So the amount of work to discover the heavy hitters at the leaves is bounded by $O(\log M/\phi)$ queries, assuming not too many false positives along the way. This analysis works directly for the Count-Min sketch. However, it does not quite work for finding the heavy hitters based on Count-Sketch and an F_2 threshold, since the F_2 of higher levels can be much higher than at the leaf level.

Nevertheless, it seems to be an effective procedure in practice. A detailed comparison of different methods for finding heavy hitters is performed in [22]. There, it is observed that there is no clear "best" sketch for this problem: both approaches have similar accuracy, given the same space. As such, it seems that the Count-Min approach might be slightly preferred, due to its faster update time: the Count-Min sketch processed about 2 million updates per second in speed trials, compared to 1 million updates per second for the Count Sketch. This is primarily since Count-Min requires only one hash function evaluation per row, to the Count Sketch's two [3].

Heavy Hitters over General Distributions.   For general distributions which include negative frequencies, this procedure is not guaranteed to work: consider two items, one with a large positive frequency and the other with a large negative frequency of similar magnitude. If these fall under the same node in the tree, their frequencies effectively cancel each other, and the search

---

[3]To address this, it would be possible to use a single hash function, where the last bit determines g_j and the preceding bits determine h_j

procedure may mistakenly fail to discover them. General distributions can arise for a variety of reasons, for example in searching for items which have significantly different frequencies in two different relations, so it is of interest to overcome this problem. Henzinger posed this question in the context of identifying which terms were experiencing significant change in their popularity within an Internet search engine [52]. Consequently, this problem is sometimes also referred to as the "heavy changers" problem.

In this context, various "group testing" sketches have been proposed. They can be thought of as turning the above idea inside out: instead of using sketches inside a hierarchical search structure, the group testing places the hierarchical search structure inside the sketch. That is, it builds a sketch as usual, but for each entry in the sketch keeps $O(\log M)$ additional information based, for instance, on the binary expansion of the item identifiers. The idea is that the width of the sketch, w, is chosen so that in expectation at most one of the heavy hitters will land in each entry, and the sum of (absolute) frequencies from all other items in the same entry is small in comparison to the frequency of the heavy hitter. Then, the additional information is in the form of a series of "tests" designed to allow the identity of the heavy hitter item to be recovered. For example, one test may keep the sum of frequencies of all items (within the given entry) whose item identifier is odd, and another keeps the sum of all those whose identifier is even. By comparing these two sums it is possible to determine whether the heavy hitter item identifier is odd or even, or if more than one heavy hitter is present, based on whether one or both counts are heavy. By repeating this with a test for each bit position (so $O(\log M)$ in total), it is possible to recover enough information about the item to correctly identify it. The net result is that it is possible to identify heavy hitters over general streams with respect to N or F_2 [24]. Other work in this area has considered trading off more space and greater search times for higher throughput [72]. Recent work has experimented with the hierarchical approach for finding heavy hitters over general distributions, and shown that on realistic data, it is still possible to recover most heavy hitters in this way [11].

1.3.4.2 Join Size Estimation

Given two frequency distributions over the same domain, f and f', their inner product is defined as

$$f \cdot f' = \sum_{i=1}^{M} f(i) * f'(i)$$

This has a natural interpretation, as the size of the equi-join between relations where f denotes the frequency distribution of the join attribute in the first, and f' denote the corresponding distribution in the second. In SQL, this is

```
SELECT COUNT(*) FROM D, D'
WHERE D.id = D'.id
```

The inner product also has a number of other fundamental interpretations that we shall discuss below. For example, it also can be used when each record has an additional "measure" value, and the query is to compute the sum of products of measure values of joining tuples (e.g. finding the total amount of sales given by number of sales of each item multiplied by price of each item). It is also possible to encode the sum of those $f(i)$s where i meets a certain predicate as an inner product where $f'(i) = 1$ if and only if i meets the predicate, and 0 otherwise. This is discussed in more detail below.

Using the AMS Sketch to estimate inner products. Given AMS sketches of f and f', C and C' respectively, that have been constructed with the same parameters (that is, the same choices of w, d, h_j and g_j, the estimate of the inner product is given by

$$\sum_{k=1}^{w} C[j,k] * C'[j,k].$$

That is, the row estimate is the inner product of the rows.

The bounds on the error follow for similar reasons to the F_2 case (indeed, the F_2 case can be thought of as a special case where $f = f'$). Expanding out the sum shows that the estimate gives $f \cdot f'$, with additional cross-terms due to collisions of items under h_j. The expectation of these cross terms in $f(i)f'(i')$ is zero over the choice of the hash functions, as the function g_j is equally likely to add as to subtract any given term. The variance of the

row estimate is bounded via the expectation of the square of the estimate, which depends on $O(F_2(f)F_2(f')/w)$. Thus each row estimate is accurate with constant probability, which is amplified by taking the median of d row estimates.

Comparing this guarantee to that for F_2 estimation, we note that the error is bounded in terms of the product $\sqrt{F_2(f)F_2(f')}$. In general, $\sqrt{F_2(f)F_2(f')}$ can be much larger than $f \cdot f'$, such as when each of f and f' is large but $f \cdot f'$ can still be small or even zero. However, this is unavoidable: lower bounds show that no sketch (more strongly, no small data structure) can guarantee to estimate $f \cdot f'$ with error proportional to $f \cdot f'$ unless the space used is at least M [60].

In fact, we can see this inner product estimation as a generalization of the previous methods. Set $f'(i) = 1$ and $f'(i') = 0$ for all $i' \neq i$. Then $f \cdot f' = f(i)$, so computing the estimate of $f \cdot f'$ should approximate the single frequency $f(i)$ with error proportional to $\sqrt{F_2(f)F_2(f')/w} = \sqrt{F_2(f)/w}$. In retrospect this is not surprising: when we consider building the sketch of the constructed frequency distribution f' and making the estimate, the resulting procedure is identical to the procedure of estimating $f(i)$ via the Count-Sketch approach. Using the AMS sketch to estimate inner-products was first proposed in [1]; the "fast" version was described in [20].

Using the Count-Min sketch to estimate inner products. The Count-Min sketch can be used in a similar way to estimate $f \cdot f'$. In fact, the row estimate is formed the same way as the AMS estimate, as the inner product of sketch rows:

$$\sum_{k=1}^{w} C[j,k] * C'[j,k].$$

Expanding this sum based on the definition of the sketch results in exactly $f \cdot f'$, along with additional error terms of the form $f(i)f'(i')$ from items i and i' which happen to be hashed to the same entry by h_j. The expectation of these terms is not too large, which is proved by a similar argument to that used to analyze the error in making point estimates of $f(i)$. We expect about a $1/w$ fraction of all such terms to occur over the whole summation. So with constant probability, the total error in a row estimate is NN'/w (where $N = \sum_{i=1}^{M} f(i)$ and $N' = \sum_{i=1}^{M} f'(i)$). Repeating and taking the minimum of d

estimates makes the probability of a large error exponentially small in d.

Applying this result to $f = f'$ shows that Count-Min sketch can estimate $f \cdot f = F_2$ with error N^2/w. In general, this will be much larger than the corresponding AMS estimate with the same w, unless the distribution is skewed. For sufficiently skewed distribution, a more careful analysis (separating out the k largest frequencies) gives tighter bounds for the accuracy of F_2 estimation [26]. Setting f' to pick out a single frequency $f(i)$ has the same bounds as the point-estimation case. This is to be expected, since the resulting procedure is identical to the point estimation protocol.

Comparing AMS and Count-Min sketches for join size estimation. The analysis shows the worst case performance of the two sketch methods can be bounded in terms of N or F_2. To get a better understanding of their true performance, Dobra and Rusu performed a detailed study of sketch algorithms [32]. They gave a careful statistical analysis of the properties of sketches, and considered a variety of different methods to extract estimates from sketches. From their empirical evaluation of many sketch variations for the purpose of join-size estimation across a variety of data sets, they arrive at the following conclusions:

- The errors from the hashing and averaging variations of the AMS sketches are comparable for low-skew (near-uniform) data, but are dramatically lower for the hashing version (the main version presented in this chapter) when the skew is high.
- The Count-Min sketch does not perform well when the data has low-skew, due to the impact of collisions with many items on the estimation. But it has the best overall performance when the data is skewed, since the errors in the estimation are relatively much lower.
- The sketches can be implemented to be very efficient: each update to the sketch in their experiments took between 50 and 400 nanoseconds, translating to a processing rate of millions of updates per second.

As a result, the general message seems to be that the (fast) AMS version of the sketches are to be preferred for this kind of estimation, since they exhibit
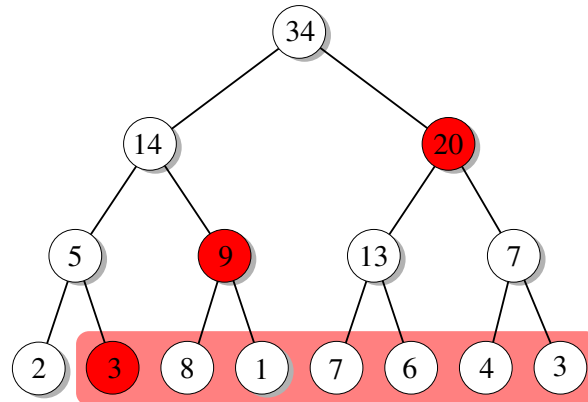
Fig. 1.6 Using dyadic ranges to answer a range query

fast updates and accurate estimations across the broadest range of data types.

1.3.4.3 Range Queries and Quantiles

Another common type of query is a range-count , e.g.

```
SELECT COUNT(*) FROM D
WHERE D.val >= l AND D.val <= h
```

for a range $l \dots h$. Note that these are the exactly the same as the range-count and range-sum queries considered over histogram representations (defined in Section **??**).

A direct solution to this query is to pose an appropriate inner-product query. Given a range, it is possible to construct a frequency distribution f' so that $f'(i) = 1$ if $l \leq i \leq h$, and 0 otherwise. Then $f \cdot f'$ gives the desired range-count. However, when applying the AMS approach to this, the error scales proportional to $\sqrt{F_2(f)F_2(f')}$. So here the error grows proportional to the square root of the length of the range. Using the Count-Min sketch approach, the error is proportional to $N(h - l + 1)$, i.e. it grows proportional to the length of the range, clearly a problem for even moderately sized ranges. Indeed, this is the same error behavior that would result from estimating each frequency in the range in turn, and summing the estimates.

Range Queries via Dyadic Ranges.   The hierarchical approach, outlined in Section 1.3.4.1, can be applied here. A standard technique which appears in many places within streaming algorithms is to represent any range canonically as a logarithmic number of so-called *dyadic* ranges. A dyadic range is a range whose length power of two, and which begins at a multiple of its own length (the same concept is used in the Haar Wavelet Transform, see Section **??**). That is, it can be written as $[j2^a + 1 \ldots (j+1)2^a]$. Examples of dyadic ranges include $[1 \ldots 8]$, $[13 \ldots 16]$, $[5 \ldots 6]$ and $[27 \ldots 27]$. Any arbitrary range can be canonically partitioned into dyadic ranges with a simple procedure: greedily find the longest possible dyadic range from the start of the range, and repeat on what remains. So for example, the range $[18 \ldots 38]$ can be broken into the dyadic ranges

$$[18 \ldots 18], [19 \ldots 20], [21 \ldots 24], [25 \ldots 32], [33 \ldots 36], [37 \ldots 38]$$

Note that there are at most two dyadic ranges of any given length in the canonical decomposition.

Therefore, a range query can be broken up into $O(\log M)$ pieces, and each of these can be posed to an appropriate sketch over the hierarchy of dyadic ranges. A simple example is shown in Figure 1.6. To estimate the range sum of $[2 \ldots 8]$, it is decomposed into the ranges $[2 \ldots 2], [3 \ldots 4], [5 \ldots 8]$, and the sum of the corresponding nodes in the binary tree is found as the estimate. So the range sum is correctly found as 32 (here, we use exact values). When using the Count-Min sketch to approximate counts, the result is immediate: the accuracy of the answer is proportional to $(N \log M)/w$: a clear advantage over the previous accuracy of $N(h-l)/w$. For large enough ranges, this is an exponential improvement in the error.

In the AMS/Count-sketch case, the benefit is less precise: each dyadic range is estimated with error proportional to the square root of the sum of the frequencies in the range. For large ranges, this error can be quite large. However, there are relatively few really large dyadic ranges, so a natural solution is to maintain the sums of frequencies in these ranges exactly [25, 32]. With this modification, it has been shown that the empirical behavior of this technique is quite accurate [32].

Quantiles via Range Queries.   The quantiles of a frequency distribution on an ordered domain divide the total "mass" of the distribution into equal

parts. More formally, the ϕ quantile q of a distribution is that point such that $\sum_{i=1}^{q} f(i) = \phi N$. The most commonly used quantile is the median, which corresponds to the $\phi = 0.5$ point. Other quantiles describe the shape of the distribution, and the likelihood that items will be seen in the "tails" of the distribution. They are also commonly used within database systems for approximate query answering, and within simple equidepth histograms (Section **??**).

There has been great interest in finding the quantiles of distributions defined by streams – see the work of Greenwald and Khanna, and references therein [50]. Gilbert *et al.* were the first to use sketches to track the quantiles of streams in the turnstile model [49]. Their solution is to observe that finding a quantile is the dual problem to a range query: we are searching for a point q so that the range query on $[1 \ldots q]$ gives ϕN. Since the result of this range query is monotone in q, we can perform a binary search for q. Each queried range can be answered using the above techniques for approximately answering range queries via sketches, and in total $O(\log M)$ queries will be needed to find a q which is (approximately) the ϕ-quantile.

Note here that the error guarantee means that we will guarantee to find a "near" quantile. That is, the q which is found is not necessarily an item which was present in the input—recall, in Section 1.2.5 we observed that whenever we store a data structure that is smaller than the size of the input, there is not room to recall which items were or were not present in the original data. Instead, we guarantee that the range query $[1 \ldots q]$ is approximately ϕN, where the quality of the approximation will depend on the size of the sketch used. Typically, the accuracy is proportional to εN, so when $\varepsilon \ll \phi$, the returned point will close to the desired quantile. It also means that extreme quantiles (like the 0.001 or the 0.999 quantile) will not be found very accurately. The most extreme quantiles are the maximum and minimum values in the data set, and we have already noted that such values are not well suited for linear sketches to find.

1.3.4.4   Sketches for Measuring Differences

A *difference query* is used to measure the difference between two frequency distributions. The *Euclidean difference* treats the frequency distributions as vectors, and measures the Euclidean distance between them. That is, given

two frequency distributions, it computes

$$\sqrt{F_2(f - f')} = \sqrt{\sum_{i=1}^{M}(f(i) - f'(i))^2}$$

Note that this is quite similar in form to an F_2 calculation, except that this is being applied to the *difference* of two frequency distributions. However, we can think of this as applying to an *implicit* frequency distribution, where the frequency of i is given by $(f(i) - f'(i))$. This can be negative, if $f'(i) > f(i)$. Here, the flexibility of sketches which can process general streams comes to the fore: it does not matter that there are negative frequencies. Further, it is not necessary to directly generate the difference distribution. Instead, given a sketch of f as C and a sketch of f' as C', it is possible to generate the sketch of $(f - f')$ as the array subtraction $(C - C')$. This is correct, due to the linearity properties of sketches. Therefore, from the two sketches an approximation of $\sqrt{F_2(f - f')}$ can be immediately computed. The accuracy of this approximation varies with $\sqrt{F_2(f - f')}/\sqrt{w}$. This is a powerful guarantee: even if $F_2(f - f')$ is very small compared to $F_2(f)$ and $F_2(f')$, the sketch approach will give a very accurate approximation of the difference.

More generally, arbitrary arithmetic over sketches is possible: the F_2 of sums and differences of frequency distributions can be found by performing the corresponding operations on the sketch representations. Given a large collection of frequency distributions, the Euclidean distance between any pair can be approximated using only the sketches, allowing them to be clustered or otherwise compared. The mathematically inclined can view this as an efficient realization of the Johnson-Lindenstrauss Lemma [59].

1.3.5 Advanced Uses of Sketches

In this section we discuss how the sketches already seen can be applied to higher dimensional data, more complex types of join size estimation, and alternate estimation techniques.

1.3.5.1 Higher Dimensional Data

Sketches can naturally summarize higher dimensional data. Given a multidimensional frequency distribution such as $f(i_1, i_2, i_3)$, it is straightforward for
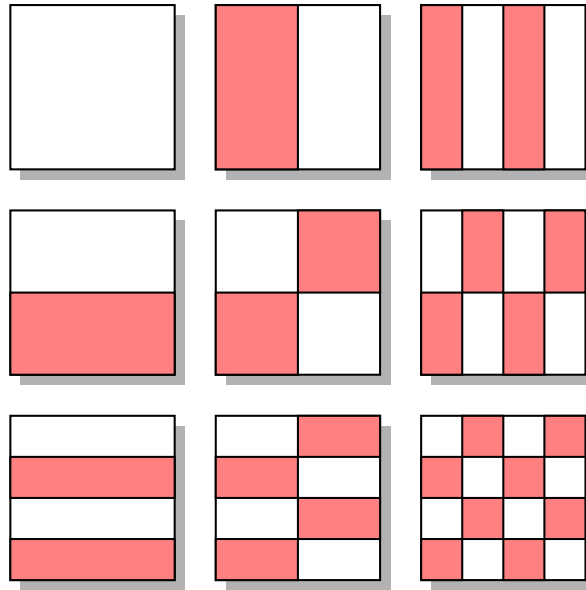
Fig. 1.7 Dyadic decomposition of a 4 by 4 rectangle

most of the sketches to summarize this: we just hash on the index (i_1, i_2, i_3) where needed. This lets us compute, for example, point queries, join size queries and distinct counts over the distributions. But these queries are not really fundamentally different for multi-dimensional data compared to single dimensional data. Indeed, all these results can be seen by considering applying some linearization to injectively map the multidimensional indices to a single dimension, and solving the one-dimensional problem on the resulting data.

Things are more challenging when we move to range queries. Now the query is specified by a product of ranges in each dimension, which specifies a hyper-rectangle. With care, the dyadic range decomposition technique can be lifted to multiple dimensions. In a single dimension, we argued that any range could be decomposed into $O(\log M)$ dyadic ranges. Analogously, any ℓ dimensional range over $\{1, \ldots, M\}^{\ell}$ can be decomposed into $O(\log^{\ell} M)$ dyadic hyper-rectangles: rectangles formed as the product of dyadic ranges. Figure 1.7 shows the decompositions of a small two dimensional rectangle of dimensions 4 by 4. Each of the nine subfigures shows a different combination

of dyadic ranges on the *x* and *y* axes.

Applying this approach, we quickly run into a "curse of dimensionality" problem: there are $\log^{\ell} M$ different types of hyper-rectangle to keep track of, so the space required (along with the time to process each update and each query) increases by at least this factor. So for range queries using Count-Min sketches, for example, the space cost to offer εN accuracy grows in proportion to $(\log M)^{2\ell}/\varepsilon$.[4] For ℓ more than 2 or 3, this cost can be sufficiently large to be impractical. Meanwhile, a sample of $O(1/\varepsilon^2)$ items from the distribution is sufficient to estimate the selectivity of the range with accuracy ε, and hence the size of the range sum with accuracy εN, irrespective of the dimensionality. Therefore, if it is possible to maintain a sample, this will often be preferable for higher dimensional data.

Alternatively, we can make independence assumptions, as discussed in the histogram case (Section **??**): if we believe that there are no correlations between the dimensions, we can keep a sketch of each dimension independently, and estimate the selectivity over the range as the product of the estimated selectivities. However, this does not seem representative of real data, where we expect to see many correlated dimensions. A compromise is to decompose the dimensions into pairs which are believed to be most correlated, and sketch the pairwise distribution of such pairs. Then the product of selectivities on each correlated pair can estimate the overall selectivity, and hence the range sum.

The work of Thaper *et al.* [73] uses multidimensional sketches to derive approximate histogram representations. Given a proposed bucketing (set of hyper-rectangles and weights), it computes the error by measuring the difference between a sketch of the data and a sketch of the histogram. The algorithm can then search over bucketings to find the best (according to the approximations from sketching). Various methods are applied to speed up the search. Considering rectangles in a particular order means that sketches can be computed incrementally; [73] also suggests the heuristic of only considering buckets that are dyadic hyper rectangles. Empirical study shows that the method finds reasonable histograms, but the time cost of the search increases dramatically as the domain size increases, even on two-dimensional data.

---

[4]This exponent is 2ℓ, because we need to store $(\log M/)^{\ell}$ sketches, and each sketch needs to be created with parameter *w* proportional to $(\log M)^{\ell}/\varepsilon$ so that the overall accuracy of the range query is εN.

1.3.5.2 More complex join size estimation

Multi-way join size estimation. Dobra *et al.* propose a technique to extend the join-size estimation results to multiple join conditions [31]. The method applies to joins of r relations, where the join condition is of the form

```
WHERE R1.A1 = R2.A1 AND R2.A2 = R3.A2 AND ...
```

The main idea is to take the averaging-based version of the AMS sketch (where all items are placed into every sketch entry). A sketch is built for each relation, where each update is multiplied by r independent hash functions that map onto $\{-1, +1\}$. Each hash function corresponds to a join condition, and has the property that if a pair of tuples join under that condition then they both hash to the same value (this is a generalization of the technique used in the sketches for approximating the size of a single join). Otherwise, there is no correlation between their hash values. A single estimate is computed by taking the product of the same entry in each sketch. Putting this all together, it follows that sets of r tuples which match on all the join conditions get their frequencies multiplied by 1, whereas all other sets of tuples contribute zero in expectation. Provided the join graph is acyclic, the variance of the estimation grows as $2^a \prod_{j=1}^{r} F_2(R_j)$, where a is the number of join attributes, and $F_2(R_j)$ denotes the F_2 (self-join size) of relation R_j. This gives good results for small numbers of joins.

A disadvantage of the technique is that is uses the slower averaging version of the AMS sketch: each update affects each entry of each sketch. To apply this using the hashing trick, we need to ensure that every pair of matching tuples get hashed into the same entry. This seems difficult for general join conditions, but is achievable in a multi-way join over the same attribute, i.e. a condition of the form

```
WHERE R1.A = R2.A AND R2.A = R3.A AND R3.A = R4.A
```

Now the sketch is formed by hashing each tuple into a sketch based on its A value, and multiplying by up to two other hash functions that map to $\{-1, +1\}$ to detect when the tuple from R_j has joining neighbors from R_{j-1} and R_{j+1}.

Note that such sketches are valuable for estimating join sizes with additional selection conditions on the join attribute: we have a join between R_1

and R_2 on A, and R_3 encodes an additional predicate on A which can be specified at query time. This in turn captures a variety of other settings: estimating the inner product between a pair of ranges, for example.

Non-equi joins.   Most uses of sketches have focused on equi-joins and their direct variations. There has been surprisingly little study of using sketch-like techniques for non-equi joins. Certain natural variations can be transformed into equi-joins with some care. For example, a join condition of the form

```
WHERE R1.A <= R2.B
```

can be incorporated by modifying the data which is sketched. Here, the frequency distribution of R_1 is sketched as usual, but for R_2, we set $f(i)$ to count the total number of rows where attribute B is greater than or equal to i. The inner product of the two frequency distributions is now equal to the size of join. This approach has the disadvantage that it is slow to process the input data: each update to R_2 requires significant effort to propagate the change to the sketch. An alternate approach may be to use dyadic ranges to speed up the updates: the join can be broken into joins of attribute values whose difference is a power of two.

There has been more effort in using sketches for *spatial joins*. This refers to cases where the data is considered to represent points in a d dimensional space. A variety of queries exist here, such as (a) the spatial join of two sets of (hyper) rectangles, where two (hyper)rectangles are considered to join if they overlap; and (b) the distance join of two sets of points, where two points join if they are within distance r of each other. Das *et al.* [28] use sketches to answer these kinds of queries. They assume a discretized domain, where the underlying d dimensional frequency distribution encodes the number of points (if any) at each coordinate location. In one dimension, it is possible to count how many intervals from one set intersect intervals from the second set. The key insight is to view an interval intersection as the endpoint of one interval being present within the other interval (and vice-versa). This can then be captured using an equi-join between the endpoint distribution of one set of intervals and the range of points covered by intervals from the other set. Therefore, sketches can be applied to approximate the size of the spatial join. A little care is needed to avoid double counting, and to handle some boundary

cases, such as when two intervals share a common endpoint.

This one dimensional case extends to higher dimensions in a fairly natural way. The main challenge is handling the growth in the number of boundary cases to consider. This can be reduced by either assuming that each coordinate of a (hyper)rectangle is unique, or by forcing this to be the case by tripling the range of each coordinate to encode whether an object starts, ends, or continues through that coordinate value. This approach directly allows spatial joins to be solved. Distance joins of point sets can be addressed by observing that replacing each point in one of the sets with an appropriate object of radius *r* and then computing the spatial join yields the desired result.

1.3.5.3 Alternate Estimation Methods and Sketches.

There has been much research into getting the best possible accuracy from sketches, based on variations on how they are updated and how the estimates are extracted from the sketch data structure.

Domain Partitioning.   Dobra *et al.* propose reducing the variance of sketch estimators for join size by partitioning the domain into *p* pieces, and keeping (averaging) AMS sketches over each partition [31]. The resulting variance of the estimator is the sum of the products of the self-join sizes of the partitioned relations, which can be smaller than the product of the self-join sizes of the full relations divided by *p*. With *a priori* knowledge of the frequency distributions, optimal partitions can be chosen. However, it seems that gains of equal or greater accuracy arise from using the fast AMS sketch (based on the hashing trick) [32]. The hashing approach can be seen as a random sketch partitioning, where the partition is defined implicitly by a hash function. Since no prior knowledge of the frequency distribution is needed here, it seems generally preferable.

Skimmed Sketches.   The *skimmed sketch* technique [41] observes that much of the error in join size estimation using sketches arises from collisions with high frequencies. Instead, Ganguly *et al.* propose "skimming" off the high frequency items from each relation by extracting the (approximate) heavy hitters, so each relation is broken into a "low" and a "high" relation. The join can now be broken into four pieces, each of which can be

estimated from either the estimated heavy hitters, or from sketches after the contributions of the sketches have been subtracted off. These four pieces can be thought of as (a) high-high (product of frequencies of items which are heavy hitters in both relations) (b) low-low (inner product estimation from the skimmed sketches) and (c) high-low and low-high (product of heavy hitter items with corresponding items from the other sketch). This is shown to be an improvement over the original scheme based on averaging multiple estimates together (Section 1.3.3.1). However, it is unclear whether there is a significant gain over the hashing version of AMS sketches where the hashing randomly separates the heavy hitters with high probability.

Conservative Update.   The *conservative update* method can be applied on Count-Min sketches (and also on Bloom Filters with counters) when the data is presented in the cash-register model. It tries to minimize overestimation by increasing the counters by the smallest amount possible given the information available. However, in doing so it breaks the property that the summary is a linear transform of the input. Consider an update to item i in a Count-Min sketch. The update function maps i to a set of entries in the sketch. The current estimate $\hat{f}(i)$ is given by the least of these: this has to increase by at least the amount of the update u to maintain the accuracy guarantee. But if other entries are larger than $\hat{f}(i) + u$, then they do not need to be increased to ensure that the estimate is correct. So the conservative update rule is to set

$$C[j, h_j(i)] \leftarrow \max(\hat{f}(i) + u, C[j, h_j(i)])$$

for each row j. The same technique can be applied to Bloom Filters that use counters [16], and was first proposed by Estan and Varghese [35].

Least Squares Estimation.   The approach of taking the minimum value as the estimate from Count-Min sketch is appealing for its simplicity. But it is also open to criticism: it does not take full account of all the information available to the estimator. Lee *et al.* studied using a least-squares method to recover estimated frequencies of a subset of items from a Count-Min sketch [62]. That is, using the fact that the sketch is a linear transform of the input, write the sketch as a multiplication between a version of the sketch matrix and a vector of the frequencies of the items of interest. To avoid generating too large a problem to solve, all items that are not of interest are modeled as a

small number of extra variables which add "noise" to the sketch entries. This linear system can be solved by applying the matrix (pseudo)inverse of the sketch matrix, and the result minimizes the difference between the sketch of the reconstructed data and the sketch of the original data. This should be no worse than the simple min-estimator, and could be much better. Experiments in [62] indicate that this approach is reduces error, but as more items are recovered, the time cost to solve the equations grows rapidly.

Several other methods have been considered for squeezing more accuracy out of simple sketch data structures. Lu *et al* use Message Passing, which also tries to find a distribution of counts which is consistent with the values recorded in the sketch of the observed data [63]. Jin *et al* empirically measure the accuracy of an instance of a Count-Min sketch [58]. They estimate the frequency of some items which are known to have zero count, say $M+1, M+2\ldots$ etc. The average of these estimates is used as τ, the expected error, and all estimated counts are reduced by τ. Likewise, Deng and Rafiei propose changing the row estimate of $f(i)$ to the value of the entry containing i, less the average value of the other entries in the same row, and analyze the variance of the resulting estimate [30]. A similar notion was used by Thorup and Zhang within their "new" estimator for F_2, which is shown to give guaranteed accuracy [76].

Skipping and Sampling. Over truly massive data, and extremely high update speeds, even the "fast" sketches Count Min, Count Sketch and (fast) AMS can fail to scale. A natural idea is that if there is so much data, it surely can't be necessary to observe it all to capture the main shape of the frequency distribution. Instead, we can "skip over" some fraction of the input. Bhattacharyya *et al.* [6] study the idea of skipping over items for heavy hitter and self-join size queries. To determine when to sketch and when to skip, they keep track of the volume of data that has been skipped, and only skip when the net effect of the skipped data (whatever the value happens to be) on the estimation cannot be too large.

Rusu and Dobra [71] study sketching over a Bernoulli sample of the data. They analyze the accuracy, and show how much of the variance arises from the sketching, how much from the sampling, and how much from the interaction of the two. As the sampling rate decreases, the sampling has a pro-

portionately greater impact on the overall accuracy. Their experimental study shows that while sampling does decrease the overall accuracy, estimates of join and self-join size can still be quite good when only sketching a tenth or a hundredth of the input data. Note that none of these methods will extend to distinct value queries: as discussed in Section **??**, it is known that no sampling method can guarantee to give good results for all possible inputs. Hence, applying sketching on top of sampling can be no better than the underlying sampling method.

Hardware Implementations.   In addition to the various software implementations discussed so far [32, 22], there has been work on building hardware implementations of sketch methods to further increase their scalability. These can take advantage of the fact that, due to linearity, sketches can be easily parallelized, and even within a single update, there is significant parallelism across the d repetitions. Several teams have studied effective parallel implementations of the Count-Min sketch. Lai and Byrd [61] describe a performance on a SIMD architecture which can achieve high throughput with low energy usage. Thomas *et al.* [74] implement the Count-Min sketch on the Cell processor (multi-core) architecture, and analyze choices in the scheduling and load-balancing issues that arise.

Lower Bounds.   All of the variations try different methods to improve the accuracy or speed of the sketching, with varied results. It is natural to ask, can any procedure asymptotically improve the accuracy, for a given amount of space? In general, the answer is no: for many of the queries studied, there are lower bounds proved which show that the space used by the sketches are essentially optimal in their dependence on ε or M. However, typically these lower bounds are proved by considering various "worst case" frequency distributions. Often the frequency distributions seen in reality are far from worst-case, and often can be well modeled by standard statistical distributions (such as Zipfian or Pareto distributions). Here, it is possible to see better space/accuracy trade-offs. Several prior works have analyzed sketches under distributional assumptions and quantified these trade-offs [14, 26].

1.4 Sketches for Distinct Value Queries

Problem relating to estimating the number of distinct items present in a sequence have been heavily studied in the last two decades. Equivalently, this problem is to find the cardinality of an attribute in a relation. The simple SQL fragment

```
SELECT COUNT (DISTINCT A)
FROM R
```

is sufficiently difficult to approximate in small space that dozens if not hundreds of research papers have tackled the problems and variations. More generally, we are interested in approximating the results of set valued queries: queries which perform a variety of set operations (intersection, union, difference) and then ask for the cardinality of the resulting set. We will see that the key to answering such queries is to first answer the simpler COUNT DISTINCT queries.

1.4.1 Linear Space Solutions

We first present solutions which use space linear in the size of the attribute cardinality. For cash-register streams, a natural solution is to use a compact set representation such as a Bloom filter. For each item in the stream, the procedure then tests whether it is already present in the Bloom filter. If the item is not present in the filter, then it is inserted into the filter, and the current count of distinct items is increased. By the one-sided error nature of the Bloom filter, the resulting count never overestimates the true count, but may underestimate due to collisions. To ensure a small constant rate of under-counting, it is necessary to set the size of the Bloom filter to be proportional to the cardinality being estimated. Due to the compactness of the Bloom filter bit vector, this requires less space than storing the full representation of the set, but only by constant factors.

The linear counting method due to Whang *et al.* [78] takes a related approach. The method can be understood as keeping a Bloom filter with a single hash function ($k = 1$). The number of distinct items is estimated based on the fraction of bits in the filter which remain as 0. If this fraction is z, then the number of distinct items is estimated as $m \ln 1/z$ (where m is the number of

bits in the filter). Again, for this to yield an accurate estimation, the m is required to be proportional to (an upper bound on) the number of distinct items. Based on some numerical analysis, this constant of proportionality is shown to be relatively low: to get low error, it is sufficient to have m be a factor of (roughly) 10 times smaller than the true cardinality of the relation.

Both linear counting and Bloom filters can be modified to allow deletions, by using the trick of replacing bits with counters: each deletion removes the effect of a prior insertion, and the estimators are modified accordingly. However, this extension potentially blows up the space further, since single bits are replaced with, say, 32 bit integers.

Both these approaches have the limitation that some *a priori* knowledge of the cardinality being estimated is needed. That is, to use them practically, it is necessary to know how large to make their filters. If the filter size is underestimated, then the filter will saturate (be almost entirely full of 1s), and the estimation will be useless. On the other hand, if the filter is mostly empty then the estimate will be very accurate, but the unused space will be wasted. Subsequent methods do not require any prior knowledge of the cardinality, and adjust to widely varying cardinalities.

1.4.2   Flajolet-Martin Sketches

The Flajolet-Martin sketch is probably the earliest, and perhaps the best known method for approximating the distinct count in small space [37]. It is also based on a bitmap B, but items are mapped non-uniformly to entries. The size of the bitmap is chosen to be logarithmic in the largest possible cardinality being estimated, so a much weaker upper bound is needed, and typically 32 or 64 bits will suffice. A hash function h is associated with the bitmap, so that half the items are mapped to 1, a quarter to 2, and so on. That is,

$$\Pr[h(i) = j] = 2^{-j}$$

where the probability is taken over the random choice of the hash function. Such a hash function is easy to generate from a function that maps uniformly onto a range: given a uniform hash function h', we set $h(i)$ based on the number of trailing zeros in the binary representation of $h'(i)$. So if $h'(i) = 3$, we set $h(i) = 1$, while if $h'(i) = 24$, we set $h(i) = 4$.

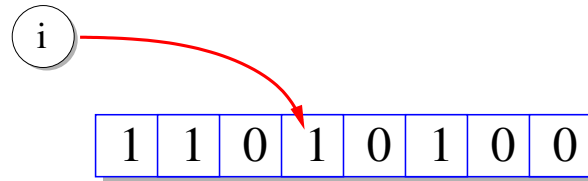The sketch is updated in the same way as a Bloom filter: each update is

Fig. 1.8 FM Sketch Data Structure

hashed by h, and we set $B[h(i)]$ to 1. A simple example is shown in Figure 1.8: an item i is hashed to location 4. There is already a 1 in this location, so the sketch does not change. After seeing n distinct items, the low entries in the bitmap are expected to be set to 1, but it is unlikely that any very high entries will be set to one. More precisely, the expected number of items which are hashed to jth entry is (approximately) $n/2^j$. Locations which expect to receive more than 1 item are very likely to be set to 1, while locations which expect to receive a number of items that is less than 1 are more likely to remain as 0. The transition occurs around where $n/2^j = 1$, i.e. around the $log_2 n$'th entry. So it is unlikely that entries above $\log n$ are set, while most entries below $\log n$ should be set to 1. Around $\log n$ it is expected that there will be a mixture of zeros and ones. A variety of different estimators are possible, such as the position of the leftmost zero in the array, or the position of the rightmost one (indexing the array from the first entry as the leftmost).

Flajolet and Martin advocated using the position of the leftmost zero [37]. Intuitively this is more robust, since it represents the compound event that all n items were not hashed there, whereas a single item can affect the position of the rightmost one. To build an estimator, they take k repetitions of the process with different hash functions, and find the mean position of the leftmost zero across these repetitions as r. The estimated value is given by $\hat{n} = 1.2928 \cdot 2^r$: here, 1.2928 is a scaling constant derived from the analysis assuming that the hash functions are perfectly random. The variance of this estimator grows with $1/\sqrt{k}$, so by taking $O(1/\varepsilon^2 \log 1/\delta)$ repetitions, the resulting estimation $\hat{n}$ is within εn of the true value n with probability at least $1 - \delta$.

Alon *et al.* analyze the effect of using the two raised to the power of the position of the rightmost one as the estimator, when using hash functions with only pairwise independence [2]. Under this restricted setting, they show that the probability of overestimating by a factor of $c > 2$ or underestimating

by a factor of $c' < 1/2$ is at most $1/c + 1/c'$. In other words, the method gives a constant factor approximation with constant probability using only logarithmic space. In fact, the space needed is only $O(\log \log n)$, since we only need to record the index of the rightmost one, rather than the full bitmap. Durand and Flajolet refer to this method as "log-log counting", and analyze it further assuming fully independent hash functions. They provide an unbiased estimator based on an appropriate scaling constant [34]. Taking k repetitions has approximately twice the variance of k instances of the original Flajolet-Martin estimator, but each repetition requires much less space to store.

The downside of these approaches as described is that they are slow to process: $O(k)$ hash functions have to be evaluated for each update. Recognizing this, Flajolet and Martin proposed using "stochastic averaging", where now the items are first hashed to one of k FM sketches, which is then updated in the usual way. Here, each update requires only a constant amount of hashing, and so is much faster to update. The stochastic averaging can be viewed as an analogue of the "hashing trick" in Section 1.3. It is can also be seen as a generalization of the linear hashing described in Section 1.4.1: an FM sketch is kept in each entry of a Bloom filter instead of just a single bit.

1.4.2.1   Linear version of FM Sketch

The methods based on the Flajolet Martin sketch and its variants assume a cash-register model of the stream. But over transaction streams in the cash register model, it is necessary to also process deletions of items. The natural solution is to replace the bits in the sketch with counters which are incremented for each inserted item that touches the entry, and decremented for each deleted item [37]. At any instant, we can extract a corresponding bitmap by setting each non-zero counter to 1, which is exactly the bitmap that would have been obtained by processing just those items which have non-zero counts. It therefore follows immediately that this linear sketch correctly processes deletions.

In the general case, there may be items with negative frequencies. It is less obvious how to interpret a COUNT DISTINCT query over such frequency distributions. However, these distributions can arise implicitly: it has been argued that it is useful to compute the number of items in two different distributions which have different frequencies. By subtracting the two frequency

distributions as $f - f'$, the number of items with different frequencies corresponds to the number of items in $f - f'$ that have non-zero frequency. This measure has been dubbed "the Hamming norm" or L_0, as a limiting case of L_p^p norms [19]. Approximating this quantity requires a different solution: the sum of frequencies of all items which hash to the same entry may be zero, even though not all of the frequencies are zero. Instead of a counter then, we can use a fingerprint of the items mapping to the entry (Section 1.2.2): with high probability, this will identify whether or not the frequency vector of items mapping to an entry is the zero vector.

1.4.3 Distinct Sampling

The idea of distinct sampling (also known as adaptive sampling) is to combine the decreasing probabilities from FM sketches with a sampling technique. Flajolet [36] attributes the invention of the technique to Wegman in the mid 1980s. A similar technique was subsequently proposed by Gibbons [44], and analyzed in the presence of limited independence hash functions with Tirthapura [45].

The method is quite simple: the algorithm maintains a set of at most k items from the input (and possibly some additional information, such as their multiplicity). During the execution of the algorithm, a integer variable l records the current "level" of the sampling. Each item in the input is hashed using a function h which obeys

$$\Pr[h(i) = j] = 2^{-j}$$

i.e. the same conditions as the FM sampling variants. The item is included in the sample if the hash value is at least the current level, so that $h(i) \geq l$ (hence we may say that the level of some item is l, or talk about the items that are at some particular level). Initially, $l = 1$ and so all distinct items are sampled.

When the sample is full (i.e., it contains more than k distinct items), the level is increased by one. The sample is then pruned: all items in the sample whose hash value is less than the current value of l are rejected. Note that when l increases by one, the effective sampling rate halves, and so we expect the sample to decrease in size to approximately $k/2$. At any moment, the current number of distinct items in the whole sequence so far can be estimated as $s2^l$, where s denotes the current number of items in the sample. In the

Level 4: 14

Level 3: 3, 10, 14

Level 2: 3, 8, 10, 14, 20

Level 1: 3, 6, 7, 8, 10, 14, 18, 19, 20

Fig. 1.9 Distinct Sampling with $k = 3$

extreme case when $k = 1$, we can see this as being similar to a single instance of log-log counting method. However, because typically $k > 1$, the accuracy should be better since it is less sensitive to a single item with a very high hash value.

Figure 1.9 shows a small example of distinct sampling for $k = 3$. Level 1 indicates the full set of distinct items that are present in the data, but this exceeds the capacity of the sample. At level 2, five of the nine items in the original input hash to a level greater than one. There are exactly three items that hash to level 3 or above, so the algorithm would currently be running at level $l = 3$. However, as soon as a new item arrives with $h(i) \geq 3$, the capacity of the sample would be exceeded, and the algorithm would advance to $l = 4$. Based on the information in the figure, items "3" and "10" would be dropped when this happens.

Analysis shows that the process has similar variance behavior to the preceding methods. Assuming perfect hash functions, the variance grows with $1/\sqrt{k}$ [36]. With weaker assumptions on the strength of the hash functions, Gibbons and Tirthapura prove a similar results: that setting $k = O(1/\varepsilon^2)$ is sufficient to estimate the true cardinality with relative error ε with constant probability. Taking $O(\log 1/\delta)$ parallel repetitions with different hash functions, and taking the median estimate in the usual way will reduce the failure probability to δ. The proof requires showing that the process stops at the right level, and that the number of items seen at the final level is close to the expected number.
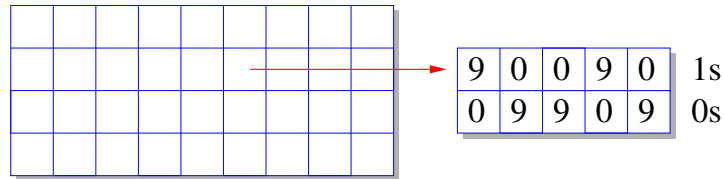
Fig. 1.10 Two-level hash sketch for distinct sampling with deletions

1.4.3.1   Distinct Sampling With Deletions.

Extending these ideas to handle deletions in the stream is less straightforward than for the FM sketch methods. It is certainly possible to apply deletions to the distinct sampling sketch: assuming that cardinality information is kept about the items that are currently being sampled, when the cardinality of a sampled item goes to zero, it is considered to have been deleted from the sample. This procedure is clearly correct, in that the resulting approximations correctly reflect the current support set of the underlying data. However, in extreme cases, when many items are deleted, the number of items in the sample may become very small or even zero. In these cases, the accuracy of the approximations becomes very poor: essentially, the current sampling rate is too low for the current data size. Ideally, the algorithm should revert to a lower level with a smaller value l. But this is not possible without a rescan of the data, since simply changing the current value of l will not recover information items that were previously rejected from the sample.

Ganguly *et al.* proposed the "two-level hash" sketch to guarantee recovering a distinct sample in the presence of many deletions, by merging several of the ideas we have seen already in other sketches [40]. Each item is hashed by a function h with the same probability distribution as for FM sketches. A second hash function g maps uniformly onto the range $[k]$. The sketch tracks information about every combination of g and h values: there are $k \log n$ of these entries. For each such entry $C[h(i), g(i)]$, there are two vectors of $\log M$ counters: corresponding to each of the $\log M$ bit positions in the binary expansion of the item identifiers. The jth counter in the first vector counts the number of updates that map to the entry $C[h(i), g(i)]$ and have a 1 in the jth position of the binary representation of i. The second vector does the same for 0s in the jth position. Since this is a linear sketch, it can be updated over

streams with deletions.

To recover a sample of items from the data structure, the vector of counters in each entry is analyzed. If there is only one item i with non-zero frequency that has been mapped to a particular $C[a,b]$ entry of the sketch, then the two vectors of counters allow it to be recovered: the non-zero counts in the two vectors encode exactly the binary encoding of i. If, on the other hand, more than one item is mapped to $C[a,b]$ then there will be some index j so that both vectors record a non-zero count, and so that entry is abandoned. This method is conceptually similar to the method described in Section 1.3.4.1 for recovering heavy hitters over strict streams. Several variations on this idea for drawing a sample from the set of distinct items have also been proposed [38, 27].

To extract a distinct sample from this distinct structure, each entry $C[a,b]$ of the sketch is examined, with the aim of recovering a single item that was mapped there. For small values of a, many items are mapped to that row, so few items will be recovered. But for a sufficiently large value of a, many items will be recovered from the row. For a given value of a, all recovered items can be returned as the sample from the sketch. Observe that a here plays a similar role to that of the level l in the distinct sampling method. Based on a, and the number of items recovered, the number of distinct items can be estimated. A partial example is shown in Figure 1.10: the figure shows the two vectors for one entry in the sketch. These vectors encode that there is a unique item that has been hashed to that entry, with multiplicity 9. It has binary representation 10010, i.e. it corresponds to the item with identifier "18".

Ganguly subsequently reduced the space required by observing that, for strict distributions, the items can be recovered from the entries more efficiently [39]. Now each entry $C[a,b]$ maintains just three counters:

$$T[a,b] = \sum_{1 \leq i \leq M,\ h(i)=a,\ g(i)=b} f(i)$$

$$U[a,b] = \sum_{1 \leq i \leq M,\ h(i)=a,\ g(i)=b} i \cdot f(i)$$

$$V[a,b] = \sum_{1 \leq i \leq M,\ h(i)=a,\ g(i)=b} i^2 \cdot f(i)$$

Clearly, if $T[a,b] = 0$, then there are no items with non-zero counts mapped to $C[a,b]$. If there is only one unique item mapped to $C[a,b]$, then $U^2[a,b] =$

(a) After 5 distinct items have been observed

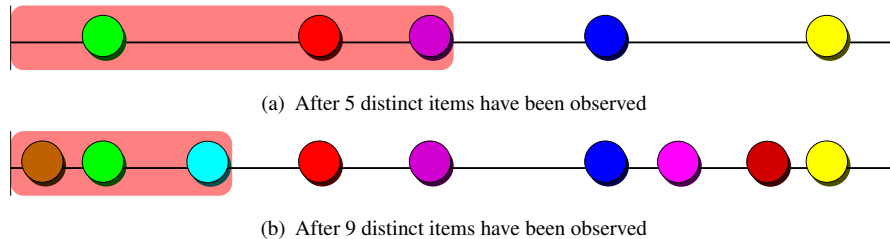(b) After 9 distinct items have been observed

Fig. 1.11 Illustration of the *k* Minimum Values procedure

$T[a,b]V[a,b] = i^2 f^2(i)$. The converse is also true: if $U^2[a,b] = T[a,b]V[a,b]$, then only one distinct item is mapped to $C[a,b]$. Further, that item is $U[a,b]/T[a,b]$ and it has frequency $T[a,b]$. For the example in Figure 1.10, the statistics computed are $T = 9, U = 162$ and $V = 2916$. From these, we can check that $TV = U^2 = 26244$, and that the item encoded is "18" with frequency 9.

1.4.4 *k* Minimum Values

The *k* minimum values technique for cash-register streams was proposed by Bar-Yossef *et al.* [4]. The technique is quite simple to state: a hash function maps items to the range $\{1 \dots M^3\}$ (so there are unlikely to be any hash collisions), and the algorithm tracks the *k* smallest distinct hash values of items seen in the stream. Let h_k denote the *k*th smallest hash value. The number of distinct items is estimated as $k(M^3/h_k)$. Figure 1.11 illustrates the process, for $k = 3$. Figure 1.11(a) shows five distinct items mapped by the hash function onto the range $\{1 \dots M^3\}$ (shown schematically as a number line). Of these, information about the $k = 3$ items which have the smallest hash values is stored, identified with a shaded background. As new items arrive, the $k = 3$ smallest changes: Figure 1.11(b) shows that two new items have entered the *k* smallest, and only information about the current *k* smallest values is stored.

The intuition for the KMV estimator is straightforward: the smallest hash value is expected to be around M^3/n, but the estimate based on this has high variance. Taking *n* random values in the range 1 to M^3, we expect about *k* of them to be less than $(k/n)M^3$. So if the hash values are close to uniform random values, then the result of the estimator should be close to *n*, the number of distinct items. In our example, the $k = 3$ smallest hash value in Figure

1.11(b) is approximately 1/4 of the way along the range, leading us to estimate that n is (roughly) 12 (in the figure, $n = 9$).

This can be made precise: the probability that the estimate is far from n corresponds to having too many or two few hash values falling in the expected range, and the probability of this event can be made small for k large enough. This is proved for pairwise independent hash functions in [4]. Making stronger assumptions about the hash functions, Beyer *et al.* [5] show that an unbiased estimator for n is $(k-1)(M^3/h_k)$, and that this has variance proportional to n^2/k. This is gives the same asymptotic guarantees as the previous analysis, but tightens the constant factors involved considerably. Applying this unbiased estimator to Figure 1.11(b), the estimate of n is now 8.

As with distinct sampling, it is possible to process deletions: simply track the cardinality of each item selected as one of the k smallest, and remove any with zero remaining occurrences. It has the same problems: too many deletions reduce the effective k value, and in extreme cases cause the sample to become empty. Methods based on sampling via linear sketches, as in the two-level hash approach, seem to be the only method to deal with inputs which exhibit such pronounced variation in their support set.

Comparing KMV and Distinct Sampling.   The KMV technique can be connected to distinct sampling: both progressively sample the data with decreasing probabilities. KMV smoothly decreases the sampling probability so that the sample always has size k, whereas distinct sampling forces the sampling rate to always be a power of two. With care, we can set up a precise correspondence. By appropriate choice of the hash functions, it is possible to arrange that the distinct sample is always a subset of the items sampled by KMV, further indicating the conceptual similarities of the methods.

The concept of hashing items and tracking information about the smallest hash values has appeared many times and has been given many different names. The idea of *min-wise hashing* (also known as *min-wise independent permutations*) is essentially the same [9]. There, focus has been on designing compact families of hash functions which have the desired properties without needing to make strong independence assumptions. Estimators based on min-wise hashing also typically keep k independent repetitions and take the item with least hash value in each repetition, which takes more time to pro-

cess each update. Cohen and Kaplan's work on *bottom-k* sketches generalizes KMV ideas to cases when items have weights which are combined with the hash values to determine which items should be retained in the sketch [15]. Beyer *et al.* [5] have also identified connections between estimators for KMV and for priority sampling (which does not consider duplicate elimination) [33].

1.4.5 Approximate Query Processing on Set Valued Queries

As remarked above, the data structures which provide us with estimates for count distinct queries can also be extended to a variety of other "distinct" queries.

1.4.5.1 Union and Sum Queries

The FM sketch variants are mainly limited to estimating the simple count-distinct queries initially discussed. However, they (in common with all the methods discussed for count distinct estimation) naturally allow the size of unions to be approximated with the same accuracy. That is, given some arbitrary collection of sets, each of which is sketched using the same hash function(s), it is possible to accurately approximate the cardinality of the union of certain sets. For FM sketches, it suffices to simply build a new FM sketch where each entry is the bitwise-or of all the corresponding entries in the sketches of the sets in the union, and apply the estimation procedure to this new sketch. This follows because the resulting sketch is exactly that which would have been obtained had the union of the sets been sketched directly.

Similar results apply for KMV and distinct sampling methods: in those cases, the procedure just takes the items sampled in the union of the sketches as the input to a new sketch (with the same hash function), and extracts the estimate from the resulting sketch. The correctness of the resulting sketch follows immediately by observing that no other items from the input could be present in the sketch of the union.

It is also possible to use these sketches to approximate various kinds of "distinct sum" queries. Here, the stream may contain multiple values of $f(i)$, and the aim is to compute the sum of the max of each value for a given i. This can be accomplished by replacing each $f(i)$ in the stream with new items

$(i,1),(i,2),\ldots(i,f(i))$. The number of distinct items in the new stream gives exactly the distinct sum. However, when the $f(i)$s can be very large, it is quite time consuming to generate and sketch so many new items. Instead, various methods have been suggested to more rapidly compute the results, either via simulating the effect of adding $f(i)$ items quickly [18], or by designing "range efficient" algorithms for count distinct [70].

1.4.5.2   Distinct Prefix queries

FM sketches and Distinct samples also allow a limited kind of COUNT DIS-TINCT with selection queries to be answered accurately over cash-register streams. We refer to these as Distinct Prefix queries: the query is to approximate the number of distinct items seen whose identifier is less than a certain value. For FM sketches, instead of keeping a single bit in the jth sketch entry, the sketch instead records the smallest identifier of any item that has been hashed to $B[j]$. Then, given a query value q, the estimation procedure extracts a bitmap from the sketch, by setting the jth bit to 1 if $B[j] < q$, and 0 otherwise. The result is exactly the FM sketch that would have been obtained if only those items less than q had been inserted.

Likewise, for distinct sampling, rather than advancing the level whenever the sample becomes full, the sketch instead keeps samples for all levels l. At each level l, it retains the k smallest identifiers which hash to that level or above. Then, given the query q, the estimate is formed by finding the first level l where there are some items greater than q, and estimating the answer as $r2^l$, where r is the number of items at level l that are less than q. Again, it is possible to argue that we recover exactly the information that would have been seen had only the items less than q arrived. Figure 1.9 shows what would happen when applying this for a sample of size 3: the items in the shaded region at each level would be retained.

These variations are quite natural, and were originally proposed to address estimating the number of distinct items seen within a recent time window [29, 46], which can be interpreted exactly as a Distinct Prefix Query. Note that the results here are quite strong: the approximate answers are within relative error of the true answer, since it is possible to argue that there is enough information to build the synopsis of the selected data.

1.4.5.3 Predicates on Items

The Distinct Prefix can be thought of as applying a predicate to items and asking for the number of distinct items satisfying the predicate. More generally, we might want to know how many distinct items pass any given predicate at query time. Since KMV and Distinct Sampling both provide a sample of the distinct items, it is natural to apply the predicate to the sample, and build an estimate: take the fraction of items in the sample which pass the predicate, $\hat{\rho}$, and multiply this by the estimate of the total number of distinct items $\hat{n}$. It can be argued that this a good estimator for the true answer [44].

However, the error in this estimator is proportional not to the true answer, but rather to the accuracy with which $\hat{n}$ is estimated. This should not be surprising: the same behavior follows for approximating the selectivity of a predicate via sampling without the DISTINCT keyword (see Section **??**). For example, if a predicate is highly selective, then it is unlikely that many items passing the predicate happened to be placed in the sample, and so we do not have a good handle on the exact selectivity. More precisely, if the sample size is k, then the variance of the estimator of ρ, the true fraction of items, behaves like ρ/k. So to get a selectivity estimate which is within relative error of ε, the size of the samples k needs to be $O(\varepsilon^{-2}\rho^{-1})$. This technique for estimating the selectivity of predicates has been applied in a variety of situations: Frahling *et al.* use it to estimate quantities over geometric streams such as the weight of the minimum spanning tree [38].

1.4.5.4 Set Operations

Many queries are based on estimating the cardinality of the results of performing operations on a collection of sets. We have already seen that set unions can be easily answered by sketch data structures for count distinct. We now show how methods which draw a sample, such as KMV and distinct sampling, can use the samples from different sets to approximate more complex set-based queries.

Consider first estimating the size of the intersection between two sets. Given sketches that draw k samples uniformly over each relation R_A and R_B, then these can be combined to find a sample from the union. However, it is not correct to just take the union of the samples: this would not be a uniform

sample if one relation were much larger than the other. Instead, the correct thing to do is to take the samples corresponding to the k distinct smallest hash values. This is a uniform sample over the union $R_A \cup R_B$.

We can now view estimating the intersection size as a special case of estimating the selectivity of a predicate: the predicate selects those items which are present in both R_A and R_B. Although there is not enough information to evaluate this predicate over arbitrary sets, there is enough to evaluate it over the sample that has been built: just count how many items in the union sample are present in the samples of both relations. The fraction of matching items, ρ is an unbiased estimator for the true fraction, i.e.

$$\mathsf{E}[\rho] = \frac{|R_A \cap R_B|}{|R_A \cup R_B|}.$$

So multiplying the estimate for $|R_A \cup R_B|$, the size of the union, gives a good estimator for the intersection size.

The accuracy of this estimator for intersection depends primarily on the size of the union. This is because, if the intersection is very small, it is unlikely that the procedure would sample items from the intersection unless the samples of both relations are large. It is more likely that it would sample many items from the union of the relations outside the intersection. The variance of the selectivity estimation can be shown to scale with $|R_A \cap R_B|/(|R_A \cup R_B|\sqrt{k})$.

The same concept extends to arbitrary set expressions over relations $R_A, R_B, R_C \ldots$. Again, the estimation procedure takes items with the k smallest hash values from the union of all the sketches of the relations, and then evaluates the selectivity of the set expression on this sample as ρ. This can also be viewed as a more complex predicate which can be evaluated exactly on the sampled items. With this view, the variance of the estimator can be analyzed in a similar way to before. This technique is implicit in the work of Ganguly *et al.* [40], and is generalized and made explicit in the work of Beyer *et al.* [5].

The same idea works for estimating the cardinality of multiset operations such as multiset difference. Here, it is necessary to include the counts of items in the samples. Operations on the samples are applied in the natural way: union operations take the sum of counts of sampled items, while multiset differences make appropriate subtractions of counts. The fraction of matching

items in the samples is found by counting those that have non-zero counts after applying the appropriate computations on the sample [5].

Certain set operations can also be evaluated using Flajolet-Martin sketch variants. This follows by using certain identities. For example, for sets A and B,

$$|A \cap B| = |A| + |B| - |A \cup B|.$$

Therefore, since Flajolet-Martin sketches can be combined to approximate the size of the union of sets, the estimates can also yield approximations of the intersection size. However, here the (absolute) error here is proportional to $|A \cup B|/\sqrt{k}$, which is greater than the error of the estimators derived based on KMV and distinct sampling.

1.4.5.5 Comparison between sketches for Distinct Value Estimation

Analytically, all the estimators proposed for distinct value estimation have broadly the same performance: as a function of a size parameter k, their variance is proportional to $1/\sqrt{k}$. Consequently, they can all provide ε relative error using space $O(1/\varepsilon^2 \log 1/\delta)$. The exact space required is a function of the exact constants in the variance (which are understood very well for most methods), and on exactly what information needs to be retained by the sketch (bitmaps, hash values, or sampled items).

Still, to fully understand which methods are preferable for particular tasks, empirical evaluation is necessary. A recent study by Metwally *et al.* performed a thorough comparison of algorithms for the core problem of distinct count estimation [64]. They gave each method the same amount of space for a sketch, and compared the accuracy and time cost on a networking dataset. Their experiments shows that methods using bitmaps could be the most accurate—perhaps unsurprising, since it is possible to squeeze in more information into bitmaps, compared to retaining item identifiers. Indeed, the provocative conclusion of their study is that one of the earliest, methods, Linear Counting (Section 1.4.1), is preferable to the more complex methods that have come since. This is due in part to its speed and accuracy on the data in question, for which approximate cardinalities are known in advance. However, in the wider context of Approximate Query Processing, it could be argued that the extra flexibility that arises from having a sample of (hashed) items to answer broader classes of queries is more desirable.

Distinct sampling and KMV methods show an order of magnitude worse accuracy than bitmap based methods in the experiments in [64]: this is perhaps to be expected, since the (hashed) item identifiers are also at least tens of bits in size. The processing times reported for all methods are comparable, thought: approximately a million updates per second.

However, the conclusion is not the same for queries more complex than simple distinct counting. Beyer *et al.* performed experiments comparing the KMV approach to a version of log-log counting for set operation estimation [5]. They allocated space equivalent to $k = 8192$ for the KMV estimator. Across a variety of queries, the KMV approach obtained accuracy of around 2-3% average relative error, whereas the log-log method incurred around two to three time as much error in the same space. Beyer *et al.* also observe that the choice of hash functions can empirically affect the quality of estimators, which may explain their differing behavior seen across experimental studies.

1.4.6 Lower Bounds

Ultimately, all the methods for count-distinct estimation take $\tilde{O}(1/\varepsilon^2)$ space to give ε relative accuracy (where $\tilde{O}$ notation hides logarithmic dependency on other factors like *m*). In fact, this dependency has been shown to be tight, in that no method can have a lower dependency on ε. Indyk and Woodruff demonstrated this by analyzing the complexity of an abstract problem called "Gap-Hamming" [55]. They showed that approximating Count-Distinct with sufficient accuracy can solve Gap-Happing, and that Gap-Hamming has a high complexity, implying the lower bound. Subsequently, Jayram *et al.* considerably simplified the hardness proof for Gap-Hamming [57].

1.5 Other topics in sketching

In this section, we briefly survey some perspectives on sketch-based methods for Approximate Query Processing.

1.5.1 Sketches for Building other summaries

One demonstration of the flexibility of sketch-based methods is the fact that there has been considerable work on using sketches to build different types of summary. In fact, for all the other major summaries in approximate query

processing—samples, histograms, and wavelets—there are methods to extract such a summary from sketch information. This can be useful when users are familiar with the semantics of such summaries, or have well-established methods for visualizing and processing such summaries.

Sketches for Sampling. The methods discussed in Sections 1.4.3 and 1.4.4 produce samples from the set of distinct items in a relation in a sketch-like manner. Further, the two-level hash structure described in Section 1.4.3.1 gives a linear sketch structure to achieve this. However, there are limited results for using sketches to sample from the raw frequency distribution, i.e. to include item i in the sample with probability $f(i)/N$. In particular, there are no results on building linear sketches to do this sampling, which would guarantee always drawing a sample of size $O(k)$ over turnstile streams.

Sketches for Histograms. Gilbert *et al.* [47] used sketches to build histogram representations of data. The sketch needed is similar to other sketch constructions, but augmented to allow range sums to be computed rapidly. This is needed when trying to compute the impact of picking a particular bucket to be part of the histogram. Based on a dynamic-programming method for constructing the histogram, the final result guarantees an approximation to the optimal histogram under a given error metric (L_1 or L_2 error). Thaper *et al.* used sketches to find histograms of multi-dimensional data, based on a greedy search [73].

Sketches for Wavelets. There has been much interest in building sketches to recover wavelet representations. Gilbert *et al.* [48] introduce the problem. They observe that the problem can be solved exactly when the data is presented in the timeseries model (i.e. sorted and aggregated), since there are only $O(\log N)$ coefficients that "straddle" any index in the data, which can be tracked to find the largest coefficients. They also show that, in the cash-register model, any wavelet coefficient can be found via appropriate range sum computations, and proposed using sketches to estimate the largest coefficients. It was subsequently suggested that it may be more efficient to compute the wavelet transform "on the fly": since the HWT is a linear transform, it is possible to build a sketch of the coefficients, without materializing the coeffi-

cient vector explicitly [21]. The problem is then reduced to finding the heavy hitters under the F_2 measure, i.e. to find all items whose frequency exceeds ϕF_2 for some fraction ϕ. This is solved by a variant of the AMS sketch with multiple levels of hashing and grouping.

1.5.2 Other Sketch Methods

The ideas within sketches have been used to build sketches for different queries. Many common ideas are present: use of limited-independence hash functions, hierarchical decompositions of ranges and so on. Some additional ideas are also commonly used: picking random values from appropriate distributions, combining multiple sketch methods together, and use of pseudo-random number generators to minimize the space needed. We briefly survey some of these, and see how they are connected to the methods we have discussed already.

1.5.2.1 Sketches for L_p norms and Entropy.

The F_2 estimation problem is a special case of a more general class of functions over sketches, the L_p norms. For a general distribution of frequencies, the L_p norm is defined as

$$L_p = \left(\sum_{i=1}^{M} |f(i)|^p \right)^{1/p}$$

The AMS sketch therefore accurately approximates the L_2 norm. Within the streaming community, there is considerable interest in being able to provide accurate approximations of other L_p norms. As $p < 1$ approaches 0, the norm approaches the "Hamming norm", i.e. the number of non-zero entries in the vector. Computations based on such norms (e.g. clustering) vary whether the emphasis is on the magnitude or the number of differences between vectors. Estimating L_p norms for $p = 1 \pm \delta$ for small values of δ has also been instrumental in some methods for estimating entropy [51].

A general technique for $0 < p \leq 2$ is based on sketches that use *stable distributions*. Each entry in the sketch is formed as the inner product of the frequency distribution with a vector of entries each drawn randomly from a stable distribution. A stable distribution is a statistical distribution that has

a stability parameter α. They have the property that sums of multiples of stable distributions are also distributed as a stable distribution—this can be viewed as a generalization of the central limit theorem, and indeed the normal distribution is stable with parameter $\alpha = 2$ [81].

By fixing the stability parameter of all distributions to be $\alpha = p$, the resulting sketch entry is distributed as a stable distribution scaled by L_p. Taking the median of all the sketch entries was shown by Indyk to be an accurate approximation of L_p [54]. To make this a small space approximation, the entries of the sketch matrix are not stored. Instead, they are generated on demand using a pseudo-random number generator so that the entry is the same every time it is accessed. Such sketches have been used in a variety of settings, such as for approximately clustering large data under L_p distance [23].

There has also been a lot of work on approximating L_p for $p > 2$. Here, strong lower bounds indicate that the size of the sketch needed is at least $\Omega(M^{1-2/p})$ [79]. Various solutions have been proposed which achieve this space cost (up to logarithmic factors) [56, 7]. The HSS (for Hierarchical Sampling from Sketches) is based on randomly selecting subsets of items via hash functions, and then extracting the heavy hitters from these subsets of items using Count Sketches. The information about the approximate frequencies of the recovered items is then combined to form an estimate of the overall L_p norm of the full frequency distribution. Owing to the lower bounds, the resulting data structure can be very large.

The ideas within the HSS sketch have also been applied to approximate the *empirical entropy* of the frequency distribution. That is, to estimate

$$H = \sum_{i=1}^{M} \frac{f(i)}{N} \log(\frac{N}{f(i)})$$

However, tighter error bounds result from using methods based on sampling and tracking information about the sampled elements [13], or by via interpolating estimates from sketches for L_p norms [51].

1.5.2.2 Combinations of Sketches

We have previously discussed several examples of combining different types of sketches: take, for example, the use of fingerprints within Flajolet-Martin sketches to track count-distinct over general frequency distributions (Section

1.4.2.1). For linear sketches in particular, it is often possible to combine multiple sketches via nesting to answer more complex queries.

A particular example arises when we wish to make sketches "duplicate resilient". The "distinct heavy hitters" problem arises when the input consists of a sequence of tuples (i, j), and the frequency of i in the multiset input $\mathscr{D}$ is now defined as as $d(i) = |\{j : (i, j) \in \mathscr{D}\}|$, the number of distinct js that i occurs with. This models a variety of problems, particularly ones which occur in networking settings [77]. One solution combines the Count-Min Sketch with Flajolet-Martin sketches: instead of keeping a counter of the items mapped there, each entry of the Count-Min sketch can keep the Flajolet-Martin sketch summary of the items [17]. This sketch allows the estimation of $d(i)$ for a given i by applying the Count-Min estimation procedure to the estimates from each Flajolet-Martin sketch in the data structure. A careful analysis is needed to study the space-accuracy tradeoff of the resulting "CM-FM sketch".

Several other combinations of sketches have been studied. In general, it is not possible to form arbitrary combinations of sketches: instead, a more cautious approach is needed, with the new estimators requiring new analysis on a case-by-case basis.

1.5.2.3 Deterministic Sketches

Most of the sketches we have seen so far involve randomization. Apart from trivial sketches (for sum, count, average etc.), the accuracy guarantees hold only with some probability over the random choice of hash functions. It is natural to ask whether the more complex queries truly require randomization to approximate. It turns out that this is not always the case: there exist sketches for certain queries which do not require randomness. However, they require substantially more space and time to create than their randomized equivalents, so they may be primarily of academic interest.

The *CR-Precis* data structure as analyzed by Ganguly and Majumder [42] (first suggested by Gasieniec and Muthukrishnan [69]) is similar to the Count-Min sketch, but determines which items to count together in each entry based on residues modulo prime numbers. The accuracy guarantees for point queries are shown based on the Chinese Remainder theorem, hence the CR in the name. The sketch keeps $d = O(1/\varepsilon \log M)$ rows, but the rows are of different lengths. The jth row corresponds to the jth prime number that

is greater than k (k is a parameter of the data structure), p_j. The ℓth entry in the jth row contains the sum of all frequencies of items whose identifier is ℓ mod p_j. That is, the entries of the sketch C are given by

$$C[j,\ell] = \sum_{1 \leq i \leq M, \quad i \mod p_j = \ell} f(i).$$

Due to the choice of parameters of the sketch, no pair of items i and i' can collide in every row (by the Chinese Remainder theorem). In fact, any pair can collide in at most $\log_k M$ rows. Point queries can be answered as in the Count-Min case, by extracting the count of every entry where i is placed, and taking the minimum of these. Using the bound on collisions, the error in the estimate is at most $N(\log_k M)/d$. Choosing d to be large enough makes this error as small as desired. To guarantee εN error, the total space needed grows proportional to $1/\varepsilon^2$ — in comparison to the $1/\varepsilon$ growth of the (randomized) Count-Min sketch. Given this deterministic bound for point queries, deterministic results for heavy hitters, range-queries and quantiles all follow using the same reductions as in Section 1.3.4. However, for other queries, no deterministic sketch is possible: it is known that randomness is required to estimate the number of distinct items and to estimate F_2 in small space [2, 53].

References

[1] N. Alon, P. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. In *ACM Principles of Database Systems*, 1999.

[2] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *ACM Symposium on Theory of Computing*, 1996.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *ACM Principles of Database Systems*, 2002.

[4] Z. Bar-Yossef, T. Jayram, R. Kumar, D. Sivakumar, and L. Trevisian. Counting distinct elements in a data stream. In *Proceedings of RANDOM 2002*, 2002.

[5] K. S. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. In *ACM SIGMOD International Conference on Management of Data*, 2007.

[6] S. Bhattacharrya, A. Madeira, S. Muthukrishnan, and T. Ye. How to scalably skip past streams. In *Scalable Stream Processing Systems (SSPS) Workshop with ICDE 2007*, 2007.

[7] L. Bhuvanagiri, S. Ganguly, D. Kesh, and C. Saha. Simpler algorithm for estimating frequency moments of data streams. In *ACM-SIAM Symposium on Discrete Algorithms*, 2006.

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

[9] A. Broder, M. Charikar, A. Frieze, and M. Mitzenmacher. Min-wise independent permutations. In *ACM Symposium on Theory of Computing*, 1998.

[10] A. Z. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4), 2003.

[11] T. Bu, J. Cao, A. Chen, and P. P. C. Lee. A fast and compact method for unveiling significant patterns in high speed networks. In *IEEE INFOCOMM*, 2007.

[12] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[13] A. Chakrabarti, G. Cormode, and A. McGregor. A near-optimal algorithm for computing the entropy of a stream. In *ACM-SIAM Symposium on Discrete Algorithms*, 2007.

[14] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 2002.

[15] E. Cohen and H. Kaplan. Summarizing data using bottom-k sketches. In *ACM Conference on Principles of Distributed Computing (PODC)*, 2007.

[16] S. Cohen and Y. Matias. Spectral bloom filters. In *ACM SIGMOD International Conference on Management of Data*, 2003.

[17] J. Considine, M. Hadjieleftheriou, F. Li, J. W. Byers, and G. Kollios. Robust approximate aggregation in sensor data management systems. *ACM Transactions on Database Systems*, 34(1), Apr. 2009.

[18] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *IEEE International Conference on Data Engineering*, 2004.

[19] G. Cormode, M. Datar, P. Indyk, and S. Muthukrishnan. Comparing data streams using Hamming norms. In *International Conference on Very Large Data Bases*, 2002.

[20] G. Cormode and M. Garofalakis. Sketching streams through the net: Distributed approximate query tracking. In *International Conference on Very Large Data Bases*, 2005.

[21] G. Cormode, M. Garofalakis, and D. Sacharidis. Fast approximate wavelet tracking on streams. In *International Conference on Extending Database Technology*, 2006.

[22] G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. In *International Conference on Very Large Data Bases*, 2008.

[23] G. Cormode, P. Indyk, N. Koudas, and S. Muthukrishnan. Fast mining of tabular data via approximate distance computations. In *IEEE International Conference on Data Engineering*, 2002.

[24] G. Cormode and S. Muthukrishnan. What's new: Finding significant differences in network data streams. In *Proceedings of IEEE Infocom*, 2004.

[25] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[26] G. Cormode and S. Muthukrishnan. Summarizing and mining skewed data streams. In *SIAM Conference on Data Mining*, 2005.

[27] G. Cormode, S. Muthukrishnan, and I. Rozenbaum. Summarizing and mining inverse distributions on data streams via dynamic inverse sampling. In *International Conference on Very Large Data Bases*, 2005.

[28] A. Das, J. Gehrke, and M. Riedewald. Approximation techniques for spatial data. In *ACM SIGMOD International Conference on Management of Data*, 2004.

[29] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *ACM-SIAM Symposium on Discrete Algorithms*, 2002.

[30] F. Deng and D. Rafiei. New estimation algorithms for streaming data: Count-min can do more. http://www.cs.ualberta.ca/~fandeng/paper/cmm.pdf, 2007.

[31] A. Dobra, M. Garofalakis, J. E. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *ACM SIGMOD International Conference on Management of Data*, 2002.

[32] A. Dobra and F. Rusu. Statistical analysis of sketch estimators. *ACM Transactions on Database Systems*, 33(3), 2008.

[33] N. Duffield, C. Lund, and M. Thorup. Estimating flow distributions from sampled flow statistics. In *ACM SIGCOMM*, 2003.

[34] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In *European Symposium on Algorithms (ESA)*, 2003.

[35] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *ACM SIGCOMM*, 2002.

[36] P. Flajolet. On adaptive sampling. *Computing*, 43(4), 1990.

[37] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for database applications. *Journal of Computer and System Sciences*, 31:182–209, 1985.

[38] G. Frahling, P. Indyk, and C. Sohler. Sampling in dynamic data streams and applications. In *Symposium on Computational Geometry*, June 2005.

[39] S. Ganguly. Counting distinct items over update streams. *Theoretical Computer Science*, 378(3):211–222, 2007.

[40] S. Ganguly, M. Garofalakis, and R. Rastogi. Processing set expressions over continuous update streams. In *ACM SIGMOD International Conference on Management of Data*, 2003.

[41] S. Ganguly, M. Garofalakis, and R. Rastogi. Processing data-stream join aggregates using skimmed sketches. In *International Conference on Extending Database Technology*, 2004.

[42] S. Ganguly and A. Majumder. CR-precis: A deterministic summary structure for update data streams. In *ESCAPE*, 2007.

[43] M. Garofalakis, J. Gehrke, and R. Rastogi. Querying and mining data streams: You only get one look. In *ACM SIGMOD International Conference on Management of Data*, 2002.

[44] P. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *International Conference on Very Large Data Bases*, 2001.

[45] P. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2001.

[46] P. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2002.

[47] A. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *ACM Symposium on Theory of Computing*, 2002.

[48] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *International Conference on Very Large Data Bases*, 2001.

[49] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *International Conference on Very Large Data Bases*, 2002.

[50] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *ACM SIGMOD International Conference on Management of Data*, 2001.

[51] N. J. A. Harvey, J. Nelson, and K. Onak. Sketching and streaming entropy via approximation theory. In *FOCS*, 2008.

[52] M. Henzinger. Algorithmic challenges in search engines. *Internet Mathematics*, 1(1):115–126, 2003.

[53] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. Technical Report SRC 1998-011, DEC Systems Research Centre, 1998.

[54] P. Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. In *IEEE Conference on Foundations of Computer Science*, 2000.

[55] P. Indyk and D. Woodruff. Tight lower bounds for the distinct elements problem. In *IEEE Conference on Foundations of Computer Science*, 2003.

[56] P. Indyk and D. P. Woodruff. Optimal approximations of the frequency moments of data streams. In *ACM Symposium on Theory of Computing*, 2005.

[57] T. S. Jayram, R. Kumar, and D. Sivakumar. The one-way communication complexity of gap hamming distance. `http://www.madalgo.au.dk/img/SumSchoo2007_Lecture_20slides/Bibliography/p14_Jayram_07_Manusc_ghd.pdf`, 2007.

[58] C. Jin, W. Qian, C. Sha, J. X. Yu, and A. Zhou. Dynamically maintaining frequent items over a data stream. In *CIKM*, 2003.

[59] W. Johnson and J. Lindenstrauss. Extensions of Lipshitz mapping into Hilbert space. *Contemporary Mathematics*, 26:189–206, 1984.

[60] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997.

[61] Y.-K. Lai and G. T. Byrd. High-throughput sketch update on a low-power stream processor. In *Proceedings of the ACM/IEEE symposium on Architecture for networking and communications systems*, 2006.

[62] G. M. Lee, H. Liu, Y. Yoon, and Y. Zhang. Improving sketch reconstruction accuracy using linear least squares method. In *Internet Measurement Conference (IMC)*, 2005.

[63] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani. Counter braids: a novel counter architecture for per-flow measurement. In *SIGMETRICS*, 2008.

[64] A. Metwally, D. Agrawal, and A. E. Abbadi. Why go logarithmic if we can go linear?: Towards effective distinct counting of search traffic. In *International Conference on Extending Database Technology*, 2008.

[65] J. Misra and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2:143–152, 1982.

[66] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. CUP, 2005.

[67] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[68] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12:315–323, 1980.

[69] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Now Publishers, 2005.

[70] A. Pavan and S. Tirthapura. Range-efficient counting of distinct elements in a massive data stream. *SIAM Journal on Computing*, 37(2):359–379, 2007.

[71] F. Rusu and A. Dobra. Sketches for size of join estimation. *ACM Transactions on Database Systems*, 33(3), 2008.

[72] R. T. Schweller, Z. Li, Y. Chen, Y. Gao, A. Gupta, Y. Zhang, P. A. Dinda, M.-Y. Kao, and G. Memik. Reversible sketches: enabling monitoring and analysis over high-speed data streams. *IEEE Transactions on Networks*, 15(5), 2007.

[73] N. Thaper, P. Indyk, S. Guha, and N. Koudas. Dynamic multidimensional histograms. In *ACM SIGMOD International Conference on Management of Data*, 2002.

[74] D. Thomas, R. Bordawekar, C. C. Aggarwal, and P. S. Yu. On efficient query process-ing of stream counts on the cell processor. In *IEEE International Conference on Data Engineering*, 2009.

[75] M. Thorup. Even strongly universal hashing is pretty fast. In *ACM-SIAM Symposium on Discrete Algorithms*, 2000.

[76] M. Thorup and Y. Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. In *ACM-SIAM Symposium on Discrete Algorithms*, 2004.

[77] S. Venkataraman, D. X. Song, P. B. Gibbons, and A. Blum. New streaming algorithms for fast detection of superspreaders. In *Network and Distributed System Security Sym-posium NDSS*, 2005.

[78] K. Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems*, 15(2):208, 1990.

[79] D. Woodruff. Optimal space lower bounds for all frequency moments. In *ACM-SIAM Symposium on Discrete Algorithms*, 2004.

[80] K. Yi, F. Li, M. Hadjieleftheriou, G. Kollios, and D. Srivastava. Randomized synopses for query assurance on data streams. In *IEEE International Conference on Data Engi-neering*, 2008.

[81] V. M. Zolotarev. *One Dimensional Stable Distributions*, volume 65 of *Translations of Mathematical Monographs*. American Mathematical Society, 1983.