UNIT III MINING DATA STREAMS 9

Introduction To Streams Concepts – Stream Data Model and Architecture - Stream Computing - Sampling Data in a Stream – Filtering Streams – Counting Distinct Elements in a Stream – Estimating Moments – Counting Oneness in a Window – Decaying Window - Real time Analytics Platform(RTAP) Applications - Case Studies - Real Time Sentiment Analysis, Stock Market Predictions

---

COUNTING DISTINCT ELEMENTS IN A STREAM

Sampling and filtering is tricky to do i.e it needs a reasonable amount of main memory, so we use a variety of hashing and a randomized algorithm to get approximately little space needed per stream.

The Count-Distinct Problem

Suppose stream elements are chosen from some universal set. We would like to know how many different elements have appeared in the stream, counting either from the beginning of the stream or from some known time in the past.

Example: Consider a Web site gathering statistics on how many unique users it has seen in each given month. The universal set is the set of logins for that site, and a stream element is generated each time someone logs in. This measure is appropriate for a site like Amazon, where the typical user logs in with their unique login name. A similar problem is a Web site like Google that does not require login to issue a search query, and may be able to identify users only by the IP address from which they send the query. There are about 4 billion IP addresses,2 sequences of four 8-bit bytes will serve as the universal set in this case.

The obvious way to solve the problem is to keep in main memory a list of all the elements seen so far in the stream. Keep them in an efficient search structure such as a hash table or search tree, so one can quickly add new elements and check whether or not the element that just arrived on the stream was already seen. As long as the number of distinct elements is not too great, this structure can fit in main memory and there is little problem obtaining an exact answer to the question how many distinct elements appear in the stream. However, if the number of distinct elements is too great, or if there are too many streams that need to be processed at once (e.g., Yahoo! wants to count the number of unique users viewing each of its pages in a month), then we cannot store the needed data in main memory.

There are several options.

- We could use more machines, each machine handling only one or several of the streams.

- We could store most of the data structure in secondary memory and batch stream elements so whenever we bought a disk block to main memory there would be many tests and updates to be performed on the data in that block.
- We could use the strategy where we only estimate the number of distinct elements but use much less memory than the number of distinct elements.

The Flajolet-Martin Algorithm

It is possible to estimate the number of distinct elements by hashing the elements of the universal set to a bit-string that is sufficiently long. The length of the bit-string must be sufficient that there are more possible results of the hash function than there are elements of the universal set. For example, 64 bits is sufficient to hash URLs. We shall pick many different hash functions and hash each element of the stream using these hash functions. The important property of a hash function is that when applied to the same element, it always produces the same result.

The idea behind the Flajolet-Martin Algorithm is that the more different elements we see in the stream, the more different hash-values we shall see. As we see more different hash-values, it becomes more likely that one of these values will be “unusual.” The particular unusual property we shall exploit is that the value ends in many 0’s, although many other options exist. Whenever we apply a hash function h to a stream element a, the bit string h(a) will end in some number of 0’s, possibly none. Call this number the tail length for a and h. Let R be the maximum tail length of the stream. Then 2^R is calculated to estimate the number of distinct elements seen in the stream.

This estimate makes intuitive sense. The probability that a given stream element a, has h(a) ending in at least r 0’s is $\mathbf{2^{-r}}$.

Suppose there are m distinct elements in the stream. Then the probability that none of them has tail length at least r is $\mathbf{(1 - 2^{-r})^m}$. We can rewrite it as $\mathbf{((1 - 2^{-r})^{2r})^{m2-r}}$. Assuming r is reasonably large, the inner expression is of the form $\mathbf{(1 - \epsilon)^{1/\epsilon}}$, which is approximately $\mathbf{1/\epsilon}$.

Thus, the probability of not finding a stream element with as many as r 0’s at the end of its hash value is e^{-m2-r}.

We can conclude:

1. If m is much larger than 2^r, then the probability that we shall find a tail of length at least r approaches 1.
2. If m is much less than 2^r, then the probability of finding a tail length at least r approaches 0.

We conclude from these two points that the proposed estimate of m, which is 2 R is unlikely to be either much too high or much too low.

Combining Estimates

Unfortunately, there is a trap regarding the strategy for combining the estimates of m, the number of distinct elements, that we obtain by using many different hash functions. Our first assumption would be that if we take the average of the values 2^R that we get from each hash function, we shall get a value that approaches the true m, the more hash functions we use. However, that is not the case, and the reason has to do with the influence an overestimate has on the average. Consider a value of r such that 2^r is much larger than m. There is some probability p that we shall discover r to be the largest number of 0's at the end of the hash value for any of the m stream elements. Then the probability of finding r + 1 to be the largest number of 0's instead is at least p/2. However, if we do increase by 1 the number of 0's at the end of a hash value, the value of 2 R doubles.

Consequently, the contribution from each possible large R to the expected value of 2R grows as R grows, and the expected value of 2R is actually infinite. Another way to combine estimates is to take the median of all estimates. The median is not affected by the occasional outsized value of 2R, so the worry described above for the average should not carry over to the median. Unfortunately, the median suffers from another defect: it is always a power of 2. Thus, no matter how many hash functions we use, should the correct value of m be between two powers of 2, say 400, then it will be impossible to obtain a close estimate. There is a solution to the problem, however. We can combine the two methods. First, group the hash functions into small groups, and take their average. Then, take the median of the averages. It is true that an occasional outsized 2^R will bias some of the groups and make them too large. However, taking the median of group averages will reduce the influence of this effect almost to nothing. Moreover, if the groups themselves are large enough, then the averages can be essentially any number, which enables us to approach the true value m as long as we use enough hash functions. In order to guarantee that any possible average can be obtained, groups should be of size at least a small multiple of log2 m.

Space Requirements

Observe that as we read the stream it is not necessary to store the elements seen. The only thing we need to keep in main memory is one integer per hash function; this integer records the largest tail length seen so far for that hash function and any stream element. If we are processing only one stream, we could use millions of hash functions, which is far more than we need to get a close estimate. Only if we are trying to process many streams at the same time would main memory constrain the number of hash functions we could associate with any one stream. In practice, the time it takes to compute hash values for each stream element would be the more significant limitation on the number of hash functions we use.

ESTIMATING MOMENTS

Consider a generalization of the problem of counting distinct elements in a stream. The problem, called computing "moments," involves the distribution of frequencies of different elements in the stream. We shall define moments of all orders and concentrate on computing second moments, from which the general algorithm for all moments is a simple extension.

Definition of Moments

Suppose a stream consists of elements chosen from a universal set. Assume the universal set is ordered so we can speak of the ith element for any i. Let m_i be the number of occurrences of the ith element for any i. Then the kth-order moment (or just kth moment) of the stream is the sum over all i of $(m_i)^k$

Example:

The 0th moment is the sum of 1 for each m_i that is greater than 0.4 That is, the 0th moment is a count of the number of distinct elements in the stream. The 1st moment is the sum of the m_i 's, which must be the length of the stream. Thus, first moments are especially easy to compute; just count the length of the stream seen so far. The second moment is the sum of the squares of the m_i 's. It is sometimes called the surprise number, since it measures how uneven the distribution of elements in the stream is. To see the distinction, suppose we have a stream of length 100, in which eleven different elements appear. The most even distribution of these eleven elements would have one appearing 10 times and the other ten appearing 9 times each. In this case, the surprise number is $10^2 + 10 \times 9^2 = 910$. At the other extreme, one of the eleven elements could appear 90 times and the other ten appear 1 time each. Then, the surprise number would be $90^2 + 10 \times 1^2 = 8110$.

There is no problem computing moments of any order if we can afford to keep in main memory a count for each element that appears in the stream. However, also as in that section, if we cannot afford to use that much memory, then we need to estimate the kth moment by keeping a limited number of values in main memory and computing an estimate from these values. For the case of distinct elements, each of these values were counts of the longest tail produced by a single hash function. We shall see another form of value that is useful for second and higher moments.

The Alon-Matias-Szegedy Algorithm for Second Moments

For now, let us assume that a stream has a particular length n. We shall show how to deal with growing streams in the next section. Suppose we do not have enough space to count all the mi's for all the elements of the stream. We can still estimate the second moment of the stream using a limited amount of space; the more space we use, the more

accurate the estimate will be. We compute some number of variables. For each variable X, we store:

1. A particular element of the universal set, which we refer to as X.element, and
2. An integer X.value, which is the value of the variable. To determine the value of a variable X, we choose a position in the stream between 1 and n, uniformly and at random. Set X.element to be the element found there, and initialize X.value to 1. As we read the stream, add 1 to X.value each time we encounter another occurrence of X.element.

Example: Suppose the stream is a, b, c, b, d, a, c, d, a, b, d, c, a, a, b. The length of the stream is n = 15. Since a appears 5 times, b appears 4 times, and c and d appear three times each, the second moment for the stream is $5^2 + 4^2 + 3^2 + 3^2 = 59$. Suppose we keep three variables, X1, X2, and X3. Also, assume that at "random" we pick the 3rd, 8th, and 13th positions to define these three variables. When we reach position 3, we find element c, so we set X1.element = c and X1.value = 1. Position 4 holds b, so we do not change X1. Likewise, nothing happens at positions 5 or 6. At position 7, we see c again, so we set X1.value = 2. At position 8 we find d, and so set X2.element = d and X2.value = 1. Positions 9 and 10 hold a and b, so they do not affect X1 or X2. Position 11 holds d so we set X2.value = 2, and position 12 holds c so we set X1.value = 3. At position 13, we find element a, and so set X3.element = a and X3.value = 1. Then, at position 14 we see another a and so set X3.value = 2. Position 15, with element b does not affect any of the variables, so we are done, with final values X1.value = 3 and X2.value = X3.value = 2. We can derive an estimate of the second moment from any variable X. This estimate is n(2X.value − 1).

Example: Consider the three variables from Example 4.7. From X1 we derive the estimate n(2X1.value − 1) = 15 × (2 × 3 − 1) = 75. The other two variables, X2 and X3, each have value 2 at the end, so their estimates are 15 × (2 × 2 − 1) = 45. Recall that the true value of the second moment for this stream is 59. On the other hand, the average of the three estimates is 55, a fairly close approximation.

Why the Alon-Matias-Szegedy Algorithm Works

We can prove that the expected value of any variable constructed is the second moment of the stream from which it is constructed. Some notation will make the argument easier to follow. Let e(i) be the stream element that appears at position i in the stream, and let c(i) be the number of times element e(i) appears in the stream among positions i, i + 1, . . . , n.

Example: Consider the stream of Example. e(6) = a, since the 6th position holds a. Also, c(6) = 4, since a appears at positions 9, 13, and 14, as well as at position 6. Note that a also appears at position 1, but that fact does not contribute to c(6).

The expected value of n(2X.value − 1) is the average over all positions i between 1 and n of n(2c(i) − 1), that is

$$E\big(n(2X.value - 1)\big) = \frac{1}{n}\sum_{i=1}^{n} n(2c(i) - 1)$$

We can simplify the above by canceling factors 1/n and n, to get

$$E\big(n(2X.value - 1)\big) = \sum_{i=1}^{n}\big(2c(i) - 1\big)$$

However, to make sense of the formula, we need to change the order of summation by grouping all those positions that have the same element. For instance, concentrate on some element that appears m_a times in the stream. The term for the last position in which a appears must be 2 × 1 − 1 = 1. The term for the next-to-last position in which a appears is 2 × 2 − 1 = 3. The positions with a before that yield terms 5, 7, and so on, up to 2ma − 1, which is the term for the first position in which a appears. That is, the formula for the expected value of 2X.value − 1 can be written:

$$E\big(n(2X.value - 1)\big) = \sum_{a} 1 + 3 + 5 + \cdots + (2m_a - 1)$$

Note that 1+ 3+ 5+· · ·+ (2ma−1) = (ma)2. The proof is an easy induction on the number of terms in the sum. Thus, E(n(2X.value − 1)=$E_a(m_a)^2$, which is the definition of the second moment.

Higher-Order Moments

We estimate kth moments, for k > 2, in essentially the same way as we estimate second moments. The only thing that changes is the way we derive an estimate from a variable. We used the formula n(2v − 1) to turn a value v, the count of the number of occurrences of some particular stream element a, into an estimate of the second moment. Notice that 2v − 1 is the difference between v^2 and $(v-1)^2$. Suppose we wanted the third moment rather than the second. Then all we have to do is replace 2v−1 by $v^3-(v-1)^3$ = $3v^2-3v+1$. Then $\sum_{v=1}^{m}$ 3v2−3v+1 = m^3, so we can use as our estimate of the third moment the formula $n(3v^2 - 3v + 1)$, where v = X.value is the value associated with some variable X. More generally, we can estimate kth moments for any k ≥ 2 by turning value v = X.value into $n(v^k - (v-1)^k)$

Dealing With Infinite Streams

Technically, the estimate we used for second and higher moments assumes that n, the stream length, is a constant. In practice, n grows with time. That fact, by itself, doesn’t cause problems, since we store only the values of variables and multiply some

function of that value by n when it is time to estimate the moment. If we count the number of stream elements seen and store this value, which only requires log n bits, then we have n available whenever we need it. A more serious problem is that we must be careful how we select the positions for the variables. If we do this selection once and for all, then as the stream gets longer, we are biased in favor of early positions, and the estimate of the moment will be too large. On the other hand, if we wait too long to pick positions, then early in the stream we do not have many variables and so will get an unreliable estimate.

The proper technique is to maintain as many variables as we can store at all times, and to throw some out as the stream grows. The discarded variables are replaced by new ones, in such a way that at all times, the probability of picking any one position for a variable is the same as that of picking any other position. Suppose we have space to store s variables. Then the first s positions of the stream are each picked as the position of one of the s variables. Inductively, suppose we have seen n stream elements, and the probability of any particular position being the position of a variable is uniform, that is s/n. When the (n+1)st element arrives, pick that position with probability s/(n+1). If not picked, then the s variables keep their same positions. However, if the (n + 1)st position is picked, then throw out one of the current s variables, with equal probability. Replace the one discarded by a new variable whose element is the one at position n + 1 and whose value is 1.

Surely, the probability that position n + 1 is selected for a variable is what it should be: s/(n + 1). However, the probability of every other position also is s/(n + 1), as we can prove by induction on n. By the inductive hypothesis, before the arrival of the (n + 1)st stream element, this probability was s/n. With probability 1 − s/(n + 1) the (n + 1)st position will not be selected, and the probability of each of the first n positions remains s/n. However, with probability s/(n + 1), the (n + 1)st position is picked, and the probability for each of the first n positions is reduced by factor (s − 1)/s. Considering the two cases, the probability of selecting each of the first n positions is

$$(1 - \frac{s}{n+1})(\frac{s}{n}) + (\frac{s}{n+1})(\frac{s-1}{s})(\frac{s}{n})$$

This expression simplifies to

$$(1 - \frac{s}{n+1})(\frac{s}{n}) + (\frac{s-1}{n+1})(\frac{s}{n})$$

and then to

$$\left((1 - \frac{s}{n+1}) + (\frac{s-1}{n+1})\right)(\frac{s}{n})$$

which in turn simplifies to

$$(\frac{n}{n+1})(\frac{s}{n}) = \frac{s}{n+1}$$

COUNTING ONES IN A WINDOW

We now turn our attention to counting problems for streams. Suppose we have a window of length N on a binary stream. We want at all times to be able to answer queries of the form "how many 1's are there in the last k bits?" for any k ≤ N. As in previous sections, we focus on the situation where we cannot afford to store the entire window. After showing an approximate algorithm for the binary case, we discuss how this idea can be extended to summing numbers.

The Cost of Exact Counts

To begin, suppose we want to be able to count exactly the number of 1's in the last k bits for any k ≤ N. Then we claim it is necessary to store all N bits of the window, as any representation that used fewer than N bits could not work. In proof, suppose we have a representation that uses fewer than N bits to represent the N bits in the window. Since there are 2N sequences of N bits, but fewer than 2N representations, there must be two different bit strings w and x that have the same representation. Since w 6= x, they must differ in at least one bit. Let the last k − 1 bits of w and x agree, but let them differ on the kth bit from the right end.

Example: If w = 0101 and x = 1010, then k = 1, since scanning from the right, they first disagree at position 1. If w = 1001 and x = 0101, then k = 3, because they first disagree at the third position from the right. Suppose the data representing the contents of the window is whatever sequence of bits represents both w and x. Ask the query "how many 1's are in the last k bits?" The query-answering algorithm will produce the same answer, whether the window contains w or x, because the algorithm can only see their representation. But the correct answers are surely different for these two bit-strings. Thus, we have proved that we must use at least N bits to answer queries about the last k bits for any k. In fact, we need N bits, even if the only query we can ask is "how many 1's are in the entire window of length N?" The argument is similar to that used above. Suppose we use fewer than N bits to represent the window, and therefore we can find w, x, and k as above. It might be that w and x have the same number of 1's, as they did in both cases of Example 4.10. However, if we follow the current window by any N − k bits, we will have a situation where the true window contents resulting from w and x are identical except for

the leftmost bit, and therefore, their counts of 1's are unequal. However, since the representations of w and x are the same, the representation of the window must still be the same if we feed the same bit sequence to these representations. Thus, we can force the answer to the query "how many 1's in the window?" to be incorrect for one of the two possible window contents.

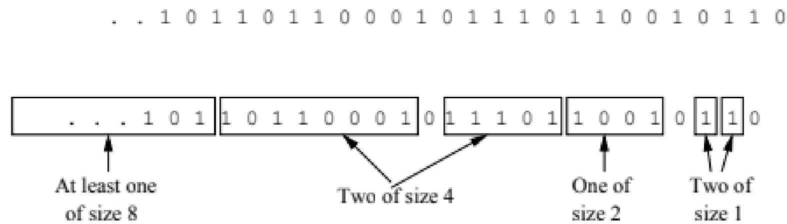
The Datar-Gionis-Indyk-Motwani Algorithm

We shall present the simplest case of an algorithm called DGIM. This version of the algorithm uses O(log2 N) bits to represent a window of N bits, and allows us to estimate the number of 1's in the window with an error of no more than 50%. Later, we shall discuss an improvement of the method that limits the error to any fraction ϙ > 0, and still uses only O(log2 N) bits (although with a constant factor that grows as ϙ shrinks). To begin, each bit of the stream has a timestamp, the position in which it arrives. The first bit has timestamp 1, the second has timestamp 2, and so on. Since we only need to distinguish positions within the window of length N, we shall represent timestamps modulo N, so they can be represented by log2 N bits. If we also store the total number of bits ever seen in the stream (i.e., the most recent timestamp) modulo N, then we can determine from a timestamp modulo N where in the current window the bit with that timestamp is. We divide the window into buckets, consisting of:

1. The timestamp of its right (most recent) end.
2. The number of 1's in the bucket. This number must be a power of 2, and we refer to the number of 1's as the size of the bucket.

To represent a bucket, we need log2 N bits to represent the timestamp (modulo N) of its right end. To represent the number of 1's we only need log2 log2 N bits. The reason is that we know this number i is a power of 2, say 2j, so we can represent i by coding j in binary. Since j is at most log2 N, it requires log2 log2 N bits. Thus, O(log N) bits suffice to represent a bucket. There are six rules that must be followed when representing a stream by buckets.

- The right end of a bucket is always a position with a 1.
- Every position with a 1 is in some bucket.
- No position is in more than one bucket.
- There are one or two buckets of any given size, up to some maximum size.
- All sizes must be a power of 2.
- Buckets cannot decrease in size as we move to the left (back in time).



Example: Figure 4.2 shows a bit stream divided into buckets in a way that satisfies the DGIM rules. At the right (most recent) end we see two buckets of size 1. To its left we see one bucket of size 2. Note that this bucket covers four positions, but only two of them are 1. Proceeding left, we see two buckets of size 4, and we suggest that a bucket of size 8 exists further left. Notice that it is OK for some 0's to lie between buckets. Also, observe from Fig. 4.2 that the buckets do not overlap; there are one or two of each size up to the largest size, and sizes only increase moving left. In the next sections, we shall explain the following about the DGIM algorithm:

1. Why the number of buckets representing a window must be small.
2. How to estimate the number of 1's in the last k bits for any k, with an error no greater than 50%.
3. How to maintain the DGIM conditions as new bits enter the stream.

Storage Requirements for the DGIM Algorithm

We observed that each bucket can be represented by O(log N) bits. If the window has length N, then there are no more than N 1's, surely. Suppose the largest bucket is of size 2^j. Then j cannot exceed $\log^2 N$, or else there are more 1's in this bucket than there are 1's in the entire window. Thus, there are at most two buckets of all sizes from $\log^2 N$ down to 1, and no buckets of larger sizes.

We conclude that there are O(log N) buckets. Since each bucket can be represented in O(log N) bits, the total space required for all the buckets representing a window of size N is $O(\log^2 N)$.

Query Answering in the DGIM Algorithm

Suppose we are asked how many 1's there are in the last k bits of the window, for some $1 \leq k \leq N$. Find the bucket b with the earliest timestamp that includes at least some of the k most recent bits. Estimate the number of 1's to be the sum of the sizes of all the buckets to the right (more recent) than bucket b, plus half the size of b itself.

Example : Suppose the stream is that of Figure, and k = 10. Then the query asks for the number of 1's in the ten rightmost bits, which happen to be 0110010110. Let the current timestamp (time of the rightmost bit) be t. Then the two buckets with one 1, having timestamps $t - 1$ and $t - 2$ are completely included in the answer. The bucket of size 2, with timestamp $t - 4$, is also completely included. However, the rightmost bucket of size

4, with timestamp t − 8 is only partly included. We know it is the last bucket to contribute to the answer, because the next bucket to its left has a timestamp less than t − 9 and thus is completely out of the window. On the other hand, we know the buckets to its right are completely inside the range of the query because of the existence of a bucket to their left with timestamp t − 9 or greater. Our estimate of the number of 1's in the last ten positions is thus 6. This number is the two buckets of size 1, the bucket of size 2, and half the bucket of size 4 that is partially within range. Of course the correct answer is 5. Suppose the above estimate of the answer to a query involves a bucket b of size 2j that is partially within the range of the query. Let us consider how far from the correct answer c our estimate could be. There are two cases: the estimate could be larger or smaller than c.

Case 1: The estimate is less than c. In the worst case, all the 1's of b are actually within the range of the query, so the estimate misses half bucket b, or 2j−1 1's. But in this case, c is at least 2j; in fact it is at least 2j+1 − 1, since there is at least one bucket of each of the sizes 2j−1, 2j−2, . . . , 1. We conclude that our estimate is at least 50% of c.

Case 2: The estimate is greater than c. In the worst case, only the rightmost bit of bucket b is within range, and there is only one bucket of each of the sizes smaller than b. Then c = 1 + 2j−1 + 2j−2 + · · · + 1 = 2j and the estimate we give is 2j−1 + 2j−1 + 2j−2 + · · · + 1 = 2j + 2j−1 − 1. We see that the estimate is no more than 50% greater than c.

Maintaining the DGIM Conditions

Suppose we have a window of length N properly represented by buckets that satisfy the DGIM conditions. When a new bit comes in, we may need to modify the buckets, so they continue to represent the window and continue to satisfy the DGIM conditions.

First, whenever a new bit enters:

- Check the leftmost (earliest) bucket. If its timestamp has now reached the current timestamp minus N, then this bucket no longer has any of its 1's in the window.

Therefore, drop it from the list of buckets. Now, we must consider whether the new bit is 0 or 1. If it is 0, then no further change to the buckets is needed. If the new bit is a 1, however, we may need to make several changes. First:

- Create a new bucket with the current timestamp and size 1.

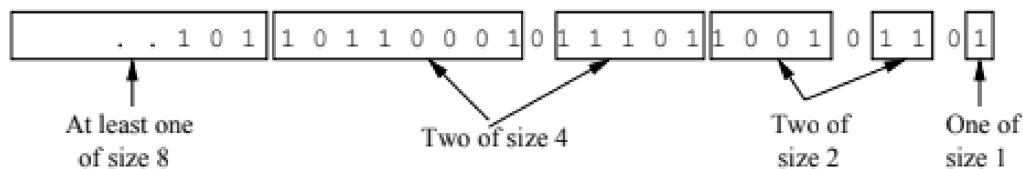
If there was only one bucket of size 1, then nothing more needs to be done. However, if there are now three buckets of size 1, that is one too many. We fix this problem by combining the leftmost (earliest) two buckets of size 1.

- To combine any two adjacent buckets of the same size, replace them by one bucket of twice the size. The timestamp of the new bucket is the timestamp of the rightmost (later in time) of the two buckets.

Combining two buckets of size 1 may create a third bucket of size 2. If so, we combine the leftmost two buckets of size 2 into a bucket of size 4. That, in turn, may

create a third bucket of size 4, and if so we combine the leftmost two into a bucket of size 8. This process may ripple through the bucket sizes, but there are at most log2 N different sizes, and the combination of two adjacent buckets of the same size only requires constant time. As a result, any new bit can be processed in O(log N) time.

Example : Suppose we start with the buckets of Fig. 4.2 and a 1 enters. First, the leftmost bucket evidently has not fallen out of the window, so we do not drop any buckets. We create a new bucket of size 1 with the current timestamp, say t. There are now three buckets of size 1, so we combine the leftmost two. They are replaced with a single bucket of size 2. Its timestamp is t − 2, the timestamp of the bucket on the right (i.e., the rightmost bucket that actually appears in Fig. 4.2.



There are now two buckets of size 2, but that is allowed by the DGIM rules. Thus, the final sequence of buckets after the addition of the 1

Reducing the Error

Instead of allowing either one or two of each size bucket, suppose we allow either r − 1 or r of each of the exponentially growing sizes 1, 2, 4, . . ., for some integer r > 2. In order to represent any possible number of 1's, we must relax this condition for the buckets of size 1 and buckets of the largest size present; there may be any number, from 1 to r, of buckets of these sizes. The rule for combining buckets is essentially the same as in Section 4.6.5. If we get r + 1 buckets of size 2j, combine the leftmost two into a bucket of size 2j+1. That may, in turn, cause there to be r + 1 buckets of size 2j+1, and if so we continue combining buckets of larger sizes.

The argument used in Section 4.6.4 can also be used here. However, because there are more buckets of smaller sizes, we can get a stronger bound on the error. We saw there that the largest relative error occurs when only one 1 from the leftmost bucket b is within the query range, and we therefore overestimate the true count. Suppose bucket b is of size 2j. Then the true count is at least 1 + (r − 1)(2j−1 + 2j−2 + · · · + 1) = 1 + (r − 1)(2j − 1). The overestimate is 2j−1 − 1. Thus, the fractional error is

$$\frac{2^{j-1} - 1}{1 + (r - 1)(2^j - 1)}$$

No matter what j is, this fraction is upper bounded by 1/(r − 1). Thus, by picking r sufficiently large, we can limit the error to any desired ϵ > 0.

Extensions to the Counting of Ones

It is natural to ask whether we can extend the technique of this section to handle aggregations more generally than counting 1's in a binary stream. An obvious direction to look is to consider streams of integers and ask if we can estimate the sum of the last k integers for any $1 \leq k \leq N$, where N, as usual, is the window size. It is unlikely that we can use the DGIM approach to streams containing both positive and negative integers. We could have a stream containing both very large positive integers and very large negative integers, but with a sum in the window that is very close to 0. Any imprecision in estimating the values of these large integers would have a huge effect on the estimate of the sum, and so the fractional error could be unbounded.

For example, suppose we broke the stream into buckets as we have done, but represented the bucket by the sum of the integers therein, rather than the count of 1's. If b is the bucket that is partially within the query range, it could be that b has, in its first half, very large negative integers and in its second half, equally large positive integers, with a sum of 0. If we estimate the contribution of b by half its sum, that contribution is essentially 0. But the actual contribution of that part of bucket b that is in the query range could be anything from 0 to the sum of all the positive integers. This difference could be far greater than the actual query answer, and so the estimate would be meaningless.

On the other hand, some other extensions involving integers do work. Suppose that the stream consists of only positive integers in the range 1 to 2m for some m. We can treat each of the m bits of each integer as if it were a separate stream. We then use the DGIM method to count the 1's in each bit. Suppose the count of the ith bit (assuming bits count from the low-order end, starting at 0) is ci. Then the sum of the integers is

$$\sum_{i=0}^{m-1} c_i 2^i$$

If we use the technique of Section to estimate each ci with fractional error at most ϵ, then the estimate of the true sum has error at most ϵ. The worst case occurs when all the ci's are overestimated or all are underestimated by the same fraction.