Chapter 4

Mining Data Streams

Most of the algorithms described in this book assume that we are mining a database. That is, all our data is available when and if we want it. In this chapter, we shall make another assumption: data arrives in a stream or streams, and if it is not processed immediately or stored, then it is lost forever. Moreover, we shall assume that the data arrives so rapidly that it is not feasible to store it all in active storage (i.e., in a conventional database), and then interact with it at the time of our choosing.

The algorithms for processing streams each involve summarization of the stream in some way. We shall start by considering how to make a useful sample of a stream and how to filter a stream to eliminate most of the "undesirable" elements. We then show how to estimate the number of different elements in a stream using much less storage than would be required if we listed all the elements we have seen.

Another approach to summarizing a stream is to look at only a fixed-length "window" consisting of the last n elements for some (typically large) n. We then query the window as if it were a relation in a database. If there are many streams and/or n is large, we may not be able to store the entire window for every stream, so we need to summarize even the windows. We address the fundamental problem of maintaining an approximate count on the number of 1's in the window of a bit stream, while using much less space than would be needed to store the entire window itself. This technique generalizes to approximating various kinds of sums.

4.1 The Stream Data Model

Let us begin by discussing the elements of streams and stream processing. We explain the difference between streams and databases and the special problems that arise when dealing with streams. Some typical applications where the stream model applies will be examined.

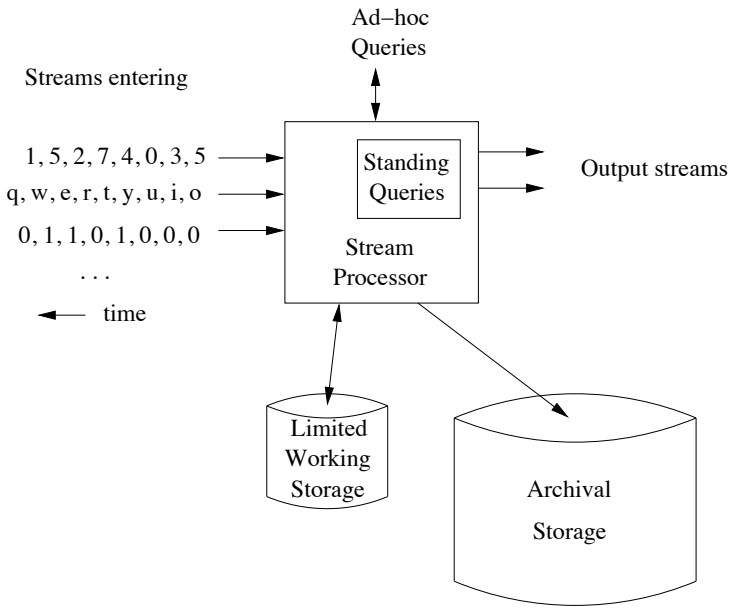

Figure 4.1: A data-stream-management system

4.1.1 A Data-Stream-Management System

In analogy to a database-management system, we can view a stream processor as a kind of data-management system, the high-level organization of which is suggested in Fig. 4.1. Any number of streams can enter the system. Each stream can provide elements at its own schedule; they need not have the same data rates or data types, and the time between elements of one stream need not be uniform. The fact that the rate of arrival of stream elements is not under the control of the system distinguishes stream processing from the processing of data that goes on within a database-management system. The latter system controls the rate at which data is read from the disk, and therefore never has to worry about data getting lost as it attempts to execute queries.

Streams may be archived in a large *archival store*, but we assume it is not possible to answer queries from the archival store. It could be examined only under special circumstances using time-consuming retrieval processes. There is also a *working store*, into which summaries or parts of streams may be placed, and which can be used for answering queries. The working store might be disk, or it might be main memory, depending on how fast we need to process queries. But either way, it is of sufficiently limited capacity that it cannot store all the data from all the streams.

4.1.2 Examples of Stream Sources

Before proceeding, let us consider some of the ways in which stream data arises naturally.

Sensor Data

Imagine a temperature sensor bobbing about in the ocean, sending back to a base station a reading of the surface temperature each hour. The data produced by this sensor is a stream of real numbers. It is not a very interesting stream, since the data rate is so low. It would not stress modern technology, and the entire stream could be kept in main memory, essentially forever.

Now, give the sensor a GPS unit, and let it report surface height instead of temperature. The surface height varies quite rapidly compared with temperature, so we might have the sensor send back a reading every tenth of a second. If it sends a 4-byte real number each time, then it produces 3.5 megabytes per day. It will still take some time to fill up main memory, let alone a single disk.

But one sensor might not be that interesting. To learn something about ocean behavior, we might want to deploy a million sensors, each sending back a stream, at the rate of ten per second. A million sensors isn't very many; there would be one for every 150 square miles of ocean. Now we have 3.5 terabytes arriving every day, and we definitely need to think about what can be kept in working storage and what can only be archived.

Image Data

Satellites often send down to earth streams consisting of many terabytes of images per day. Surveillance cameras produce images with lower resolution than satellites, but there can be many of them, each producing a stream of images at intervals like one second. London is said to have six million such cameras, each producing a stream.

Internet and Web Traffic

A switching node in the middle of the Internet receives streams of IP packets from many inputs and routes them to its outputs. Normally, the job of the switch is to transmit data and not to retain it or query it. But there is a tendency to put more capability into the switch, e.g., the ability to detect denial-of-service attacks or the ability to reroute packets based on information about congestion in the network.

Web sites receive streams of various types. For example, Google receives several hundred million search queries per day. Yahoo! accepts billions of "clicks" per day on its various sites. Many interesting things can be learned from these streams. For example, an increase in queries like "sore throat" enables us to track the spread of viruses. A sudden increase in the click rate for a link could

indicate some news connected to that page, or it could mean that the link is broken and needs to be repaired.

4.1.3 Stream Queries

There are two ways that queries get asked about streams. We show in Fig. 4.1 a place within the processor where *standing queries* are stored. These queries are, in a sense, permanently executing, and produce outputs at appropriate times.

Example 4.1: The stream produced by the ocean-surface-temperature sensor mentioned at the beginning of Section 4.1.2 might have a standing query to output an alert whenever the temperature exceeds 25 degrees centigrade. This query is easily answered, since it depends only on the most recent stream element.

Alternatively, we might have a standing query that, each time a new reading arrives, produces the average of the 24 most recent readings. That query also can be answered easily, if we store the 24 most recent stream elements. When a new stream element arrives, we can drop from the working store the 25th most recent element, since it will never again be needed (unless there is some other standing query that requires it).

Another query we might ask is the maximum temperature ever recorded by that sensor. We can answer this query by retaining a simple summary: the maximum of all stream elements ever seen. It is not necessary to record the entire stream. When a new stream element arrives, we compare it with the stored maximum, and set the maximum to whichever is larger. We can then answer the query by producing the current value of the maximum. Similarly, if we want the average temperature over all time, we have only to record two values: the number of readings ever sent in the stream and the sum of those readings. We can adjust these values easily each time a new reading arrives, and we can produce their quotient as the answer to the query. □

The other form of query is *ad-hoc*, a question asked once about the current state of a stream or streams. If we do not store all streams in their entirety, as normally we can not, then we cannot expect to answer arbitrary queries about streams. If we have some idea what kind of queries will be asked through the ad-hoc query interface, then we can prepare for them by storing appropriate parts or summaries of streams as in Example 4.1.

If we want the facility to ask a wide variety of ad-hoc queries, a common approach is to store a *sliding window* of each stream in the working store. A sliding window can be the most recent n elements of a stream, for some n, or it can be all the elements that arrived within the last t time units, e.g., one day. If we regard each stream element as a tuple, we can treat the window as a relation and query it with any SQL query. Of course the stream-management system must keep the window fresh, deleting the oldest elements as new ones come in.

Example 4.2: Web sites often like to report the number of unique users over the past month. If we think of each login as a stream element, we can maintain a window that is all logins in the most recent month. We must associate the arrival time with each login, so we know when it no longer belongs to the window. If we think of the window as a relation `Logins(name, time)`, then it is simple to get the number of unique users over the past month. The SQL query is:

```
SELECT COUNT(DISTINCT(name))
FROM Logins
WHERE time >= t;
```

Here, t is a constant that represents the time one month before the current time.

Note that we must be able to maintain the entire stream of logins for the past month in working storage. However, for even the largest sites, that data is not more than a few terabytes, and so surely can be stored on disk. □

4.1.4 Issues in Stream Processing

Before proceeding to discuss algorithms, let us consider the constraints under which we work when dealing with streams. First, streams often deliver elements very rapidly. We must process elements in real time, or we lose the opportunity to process them at all, without accessing the archival storage. Thus, it often is important that the stream-processing algorithm is executed in main memory, without access to secondary storage or with only rare accesses to secondary storage. Moreover, even when streams are "slow," as in the sensor-data example of Section 4.1.2, there may be many such streams. Even if each stream by itself can be processed using a small amount of main memory, the requirements of all the streams together can easily exceed the amount of available main memory.

Thus, many problems about streaming data would be easy to solve if we had enough memory, but become rather hard and require the invention of new techniques in order to execute them at a realistic rate on a machine of realistic size. Here are two generalizations about stream algorithms worth bearing in mind as you read through this chapter:

- Often, it is much more efficient to get an approximate answer to our problem than an exact solution.

- As in Chapter 3, a variety of techniques related to hashing turn out to be useful. Generally, these techniques introduce useful randomness into the algorithm's behavior, in order to produce an approximate answer that is very close to the true result.

4.2 Sampling Data in a Stream

As our first example of managing streaming data, we shall look at extracting reliable samples from a stream. As with many stream algorithms, the "trick" involves using hashing in a somewhat unusual way.

4.2.1 A Motivating Example

The general problem we shall address is selecting a subset of a stream so that we can ask queries about the selected subset and have the answers be statistically representative of the stream as a whole. If we know what queries are to be asked, then there are a number of methods that might work, but we are looking for a technique that will allow ad-hoc queries on the sample. We shall look at a particular problem, from which the general idea will emerge.

Our running example is the following. A search engine receives a stream of queries, and it would like to study the behavior of typical users.[1] We assume the stream consists of tuples (user, query, time). Suppose that we want to answer queries such as "What fraction of the typical user's queries were repeated over the past month?" Assume also that we wish to store only 1/10th of the stream elements.

The obvious approach would be to generate a random number, say an integer from 0 to 9, in response to each search query. Store the tuple if and only if the random number is 0. If we do so, each user has, on average, 1/10th of their queries stored. Statistical fluctuations will introduce some noise into the data, but if users issue many queries, the law of large numbers will assure us that most users will have a fraction quite close to 1/10th of their queries stored.

However, this scheme gives us the wrong answer to the query asking for the average number of duplicate queries for a user. Suppose a user has issued s search queries one time in the past month, d search queries twice, and no search queries more than twice. If we have a 1/10th sample, of queries, we shall see in the sample for that user an expected $s/10$ of the search queries issued once. Of the d search queries issued twice, only $d/100$ will appear twice in the sample; that fraction is d times the probability that both occurrences of the query will be in the 1/10th sample. Of the queries that appear twice in the full stream, $18d/100$ will appear exactly once. To see why, note that 18/100 is the probability that one of the two occurrences will be in the 1/10th of the stream that is selected, while the other is in the 9/10th that is not selected.

The correct answer to the query about the fraction of repeated searches is $d/(s+d)$. However, the answer we shall obtain from the sample is $d/(10s+19d)$. To derive the latter formula, note that $d/100$ appear twice, while $s/10+18d/100$ appear once. Thus, the fraction appearing twice in the sample is $d/100$ divided

[1]While we shall refer to "users," the search engine really receives IP addresses from which the search query was issued. We shall assume that these IP addresses identify unique users, which is approximately true, but not exactly true.

by $d/100 + s/10 + 18d/100$. This ratio is $d/(10s + 19d)$. For no positive values of s and d is $d/(s + d) = d/(10s + 19d)$.

4.2.2 Obtaining a Representative Sample

The query of Section 4.2.1, like many queries about the statistics of typical users, cannot be answered by taking a sample of each user's search queries. Thus, we must strive to pick 1/10th of the users, and take all their searches for the sample, while taking none of the searches from other users. If we can store a list of all users, and whether or not they are in the sample, then we could do the following. Each time a search query arrives in the stream, we look up the user to see whether or not they are in the sample. If so, we add this search query to the sample, and if not, then not. However, if we have no record of ever having seen this user before, then we generate a random integer between 0 and 9. If the number is 0, we add this user to our list with value "in," and if the number is other than 0, we add the user with the value "out."

That method works as long as we can afford to keep the list of all users and their in/out decision in main memory, because there isn't time to go to disk for every search that arrives. By using a hash function, one can avoid keeping the list of users. That is, we hash each user name to one of ten buckets, 0 through 9. If the user hashes to bucket 0, then accept this search query for the sample, and if not, then not.

Note we do not actually store the user in the bucket; in fact, there is no data in the buckets at all. Effectively, we use the hash function as a random-number generator, with the important property that, when applied to the same user several times, we always get the same "random" number. That is, without storing the in/out decision for any user, we can reconstruct that decision any time a search query by that user arrives.

More generally, we can obtain a sample consisting of any rational fraction a/b of the users by hashing user names to b buckets, 0 through $b - 1$. Add the search query to the sample if the hash value is less than a.

4.2.3 The General Sampling Problem

The running example is typical of the following general problem. Our stream consists of tuples with n components. A subset of the components are the *key* components, on which the selection of the sample will be based. In our running example, there are three components – user, query, and time – of which only *user* is in the key. However, we could also take a sample of queries by making *query* be the key, or even take a sample of user-query pairs by making both those components form the key.

To take a sample of size a/b, we hash the key value for each tuple to b buckets, and accept the tuple for the sample if the hash value is less than a. If the key consists of more than one component, the hash function needs to combine the values for those components to make a single hash-value. The

result will be a sample consisting of all tuples with certain key values. The selected key values will be approximately a/b of all the key values appearing in the stream.

4.2.4 Varying the Sample Size

Often, the sample will grow as more of the stream enters the system. In our running example, we retain all the search queries of the selected 1/10th of the users, forever. As time goes on, more searches for the same users will be accumulated, and new users that are selected for the sample will appear in the stream.

If we have a budget for how many tuples from the stream can be stored as the sample, then the fraction of key values must vary, lowering as time goes on. In order to assure that at all times, the sample consists of all tuples from a subset of the key values, we choose a hash function h from key values to a very large number of values $0, 1, \ldots, B-1$. We maintain a *threshold* t, which initially can be the largest bucket number, $B-1$. At all times, the sample consists of those tuples whose key K satisfies $h(K) \leq t$. New tuples from the stream are added to the sample if and only if they satisfy the same condition.

If the number of stored tuples of the sample exceeds the allotted space, we lower t to $t-1$ and remove from the sample all those tuples whose key K hashes to t. For efficiency, we can lower t by more than 1, and remove the tuples with several of the highest hash values, whenever we need to throw some key values out of the sample. Further efficiency is obtained by maintaining an index on the hash value, so we can find all those tuples whose keys hash to a particular value quickly.

4.2.5 Exercises for Section 4.2

Exercise 4.2.1: Suppose we have a stream of tuples with the schema

Grades(university, courseID, studentID, grade)

Assume universities are unique, but a courseID is unique only within a university (i.e., different universities may have different courses with the same ID, e.g., "CS101") and likewise, studentID's are unique only within a university (different universities may assign the same ID to different students). Suppose we want to answer certain queries approximately from a 1/20th sample of the data. For each of the queries below, indicate how you would construct the sample. That is, tell what the key attributes should be.

(a) For each university, estimate the average number of students in a course.

(b) Estimate the fraction of students who have a GPA of 3.5 or more.

(c) Estimate the fraction of courses where at least half the students got "A."

4.3 Filtering Streams

Another common process on streams is selection, or filtering. We want to accept those tuples in the stream that meet a criterion. Accepted tuples are passed to another process as a stream, while other tuples are dropped. If the selection criterion is a property of the tuple that can be calculated (e.g., the first component is less than 10), then the selection is easy to do. The problem becomes harder when the criterion involves lookup for membership in a set. It is especially hard, when that set is too large to store in main memory. In this section, we shall discuss the technique known as "Bloom filtering" as a way to eliminate most of the tuples that do not meet the criterion.

4.3.1 A Motivating Example

Again let us start with a running example that illustrates the problem and what we can do about it. Suppose we have a set S of one billion allowed email addresses – those that we will allow through because we believe them not to be spam. The stream consists of pairs: an email address and the email itself. Since the typical email address is 20 bytes or more, it is not reasonable to store S in main memory. Thus, we can either use disk accesses to determine whether or not to let through any given stream element, or we can devise a method that requires no more main memory than we have available, and yet will filter most of the undesired stream elements.

Suppose for argument's sake that we have one gigabyte of available main memory. In the technique known as *Bloom filtering*, we use that main memory as a bit array. In this case, we have room for eight billion bits, since one byte equals eight bits. Devise a hash function h from email addresses to eight billion buckets. Hash each member of S to a bit, and set that bit to 1. All other bits of the array remain 0.

Since there are one billion members of S, approximately 1/8th of the bits will be 1. The exact fraction of bits set to 1 will be slightly less than 1/8th, because it is possible that two members of S hash to the same bit. We shall discuss the exact fraction of 1's in Section 4.3.3. When a stream element arrives, we hash its email address. If the bit to which that email address hashes is 1, then we let the email through. But if the email address hashes to a 0, we are certain that the address is not in S, so we can drop this stream element.

Unfortunately, some spam email will get through. Approximately 1/8th of the stream elements whose email address is not in S will happen to hash to a bit whose value is 1 and will be let through. Nevertheless, since the majority of emails are spam (about 80% according to some reports), eliminating 7/8th of the spam is a significant benefit. Moreover, if we want to eliminate every spam, we need only check for membership in S those good and bad emails that get through the filter. Those checks will require the use of secondary memory to access S itself. There are also other options, as we shall see when we study the general Bloom-filtering technique. As a simple example, we could use a cascade

of filters, each of which would eliminate 7/8th of the remaining spam.

4.3.2 The Bloom Filter

A *Bloom filter* consists of:

1. An array of n bits, initially all 0's.
2. A collection of hash functions $h_1, h_2, \ldots, h_k$. Each hash function maps "key" values to n buckets, corresponding to the n bits of the bit-array.
3. A set S of m key values.

The purpose of the Bloom filter is to allow through all stream elements whose keys are in S, while rejecting most of the stream elements whose keys are not in S.

To initialize the bit array, begin with all bits 0. Take each key value in S and hash it using each of the k hash functions. Set to 1 each bit that is $h_i(K)$ for some hash function h_i and some key value K in S.

To test a key K that arrives in the stream, check that all of

$$h_1(K), h_2(K), \ldots, h_k(K)$$

are 1's in the bit-array. If all are 1's, then let the stream element through. If one or more of these bits are 0, then K could not be in S, so reject the stream element.

4.3.3 Analysis of Bloom Filtering

If a key value is in S, then the element will surely pass through the Bloom filter. However, if the key value is not in S, it might still pass. We need to understand how to calculate the probability of a *false positive*, as a function of n, the bit-array length, m the number of members of S, and k, the number of hash functions.

The model to use is throwing darts at targets. Suppose we have x targets and y darts. Any dart is equally likely to hit any target. After throwing the darts, how many targets can we expect to be hit at least once? The analysis is similar to the analysis in Section 3.4.2, and goes as follows:

- The probability that a given dart will not hit a given target is $(x-1)/x$.
- The probability that none of the y darts will hit a given target is $\left(\frac{x-1}{x}\right)^y$. We can write this expression as $(1-\frac{1}{x})^{x(\frac{y}{x})}$.
- Using the approximation $(1-\epsilon)^{1/\epsilon} = 1/e$ for small ϵ (recall Section 1.3.5), we conclude that the probability that none of the y darts hit a given target is $e^{-y/x}$.

Example 4.3: Consider the running example of Section 4.3.1. We can use the above calculation to get the true expected number of 1's in the bit array. Think of each bit as a target, and each member of S as a dart. Then the probability that a given bit will be 1 is the probability that the corresponding target will be hit by one or more darts. Since there are one billion members of S, we have $y = 10^9$ darts. As there are eight billion bits, there are $x = 8 \times 10^9$ targets. Thus, the probability that a given target is not hit is $e^{-y/x} = e^{-1/8}$ and the probability that it *is* hit is $1 - e^{-1/8}$. That quantity is about 0.1175. In Section 4.3.1 we suggested that $1/8 = 0.125$ is a good approximation, which it is, but now we have the exact calculation. □

We can apply the rule to the more general situation, where set S has m members, the array has n bits, and there are k hash functions. The number of targets is $x = n$, and the number of darts is $y = km$. Thus, the probability that a bit remains 0 is $e^{-km/n}$. We want the fraction of 0 bits to be fairly large, or else the probability that a nonmember of S will hash at least once to a 0 becomes too small, and there are too many false positives. For example, we might choose k, the number of hash functions to be n/m or less. Then the probability of a 0 is at least e^{-1} or 37%. In general, the probability of a false positive is the probability of a 1 bit, which is $1 - e^{-km/n}$, raised to the kth power, i.e., $(1 - e^{-km/n})^k$.

Example 4.4: In Example 4.3 we found that the fraction of 1's in the array of our running example is 0.1175, and this fraction is also the probability of a false positive. That is, a nonmember of S will pass through the filter if it hashes to a 1, and the probability of it doing so is 0.1175.

Suppose we used the same S and the same array, but used two different hash functions. This situation corresponds to throwing two billion darts at eight billion targets, and the probability that a bit remains 0 is $e^{-1/4}$. In order to be a false positive, a nonmember of S must hash twice to bits that are 1, and this probability is $(1 - e^{-1/4})^2$, or approximately 0.0493. Thus, adding a second hash function for our running example is an improvement, reducing the false-positive rate from 0.1175 to 0.0493. □

4.3.4 Exercises for Section 4.3

Exercise 4.3.1: For the situation of our running example (8 billion bits, 1 billion members of the set S), calculate the false-positive rate if we use three hash functions? What if we use four hash functions?

! Exercise 4.3.2: Suppose we have n bits of memory available, and our set S has m members. Instead of using k hash functions, we could divide the n bits into k arrays, and hash once to each array. As a function of n, m, and k, what is the probability of a false positive? How does it compare with using k hash functions into a single array?

!! **Exercise 4.3.3:** As a function of n, the number of bits and m the number of members in the set S, what number of hash functions minimizes the false-positive rate?

4.4 Counting Distinct Elements in a Stream

In this section we look at a third simple kind of processing we might want to do on a stream. As with the previous examples – sampling and filtering – it is somewhat tricky to do what we want in a reasonable amount of main memory, so we use a variety of hashing and a randomized algorithm to get approximately what we want with little space needed per stream.

4.4.1 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 4.5: As a useful example of this problem, 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 brought a disk block to main memory there would be many tests and updates to be performed on the data in that block. Or we could use the strategy to be discussed in this section, where we

[2] At least that will be the case until IPv6 becomes the norm.

only estimate the number of distinct elements but use much less memory than the number of distinct elements.

4.4.2 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 URL's. 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. Notice that this property was also essential for the sampling technique of Section 4.2.

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 any a seen so far in the stream. Then we shall use estimate 2^R for 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 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 $(1 - 2^{-r})^m$. This sort of expression should be familiar by now. We can rewrite it as $\left((1 - 2^{-r})^{2^r}\right)^{m2^{-r}}$. Assuming r is reasonably large, the inner expression is of the form $(1 - \epsilon)^{1/\epsilon}$, which is approximately $1/e$. 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 (recall R is the largest tail length for any stream element) is unlikely to be either much too high or much too low.

4.4.3 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 2^R grows as R grows, and the expected value of 2^R is actually infinite.[3]

Another way to combine estimates is to take the median of all estimates. The median is not affected by the occasional outsized value of 2^R, 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 $\log_2 m$.

4.4.4 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

[3]Technically, since the hash value is a bit-string of finite length, there is no contribution to 2^R for R's that are larger than the length of the hash value. However, this effect is not enough to avoid the conclusion that the expected value of 2^R is much too large.

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.

4.4.5 Exercises for Section 4.4

Exercise 4.4.1: Suppose our stream consists of the integers 3, 1, 4, 1, 5, 9, 2, 6, 5. Our hash functions will all be of the form $h(x) = ax + b \mod 32$ for some a and b. You should treat the result as a 5-bit binary integer. Determine the tail length for each stream element and the resulting estimate of the number of distinct elements if the hash function is:

(a) $h(x) = 2x + 1 \mod 32$.

(b) $h(x) = 3x + 7 \mod 32$.

(c) $h(x) = 4x \mod 32$.

! **Exercise 4.4.2:** Do you see any problems with the choice of hash functions in Exercise 4.4.1? What advice could you give someone who was going to use a hash function of the form $h(x) = ax + b \mod 2^k$?

4.5 Estimating Moments

In this section we 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.

4.5.1 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 4.6: 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. We can use the method of Section 4.4 to estimate the 0th moment of a stream.

[4]Technically, since m_i could be 0 for some elements in the universal set, we need to make explicit in the definition of "moment" that 0^0 is taken to be 0. For moments 1 and above, the contribution of m_i's that are 0 is surely 0.