Unit 4
Stream Memory

Most of the algorithms assume that we are mining a database. That is, all our data is available when and if we want it. Another assumption is that: 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.

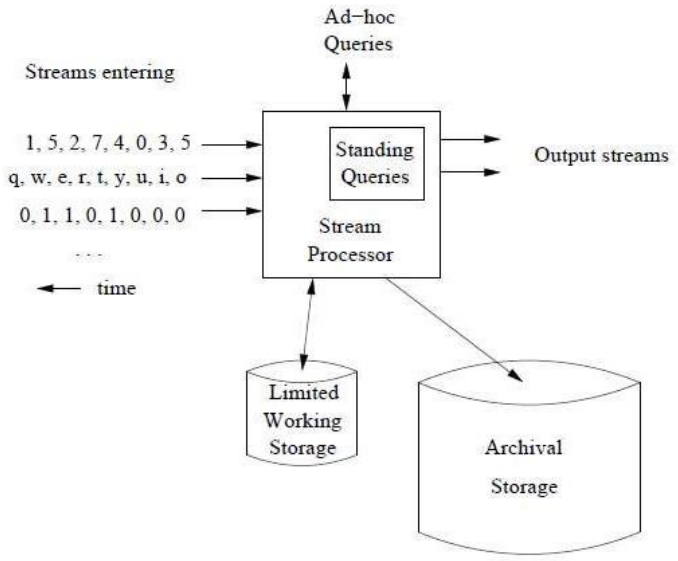


Figure 4.1: A data-stream-management system

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 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 randomnumber 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, . . . ,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) ≤ 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 h1, h2, . . . , hk. 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 hi(K) for some hash function hi and some key value K in S.

To test a key K that arrives in the stream, check that all of h1(K), h2(K), . . . , hk(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 = 109 darts. As there are eight billion bits, there are x = 8 × 109 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 falsepositive 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 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 2R 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 2r 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 2R 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 2R 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 log2m.

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 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 2k?

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 mi 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 (mi)k.

Example 4.6 : The 0th moment is the sum of 1 for each mi 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.

The 1st moment is the sum of the mi'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 mi'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 $102 + 10 \times 92 = 910$. 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 $902 + 10 \times 12 = 8110$. ✷

As in Section 4.4, 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.

4.5.2 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 4.7 : 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 52+42+32+32 = 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 4.8 : 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. ✷

4.5.3 Why the Alon-Matias-Szegedy Algorithm Works

We can prove that the expected value of any variable constructed as in Section 4.5.2 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 4.9 : Consider the stream of Example 4.7. 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 a that appears ma 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, En(2X.value − 1)_ = Pa(ma)2, which is the definition of the second moment.

4.5.4 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. In Section 4.5.2 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. Then, in Section 4.5.3 we saw why this formula works: the terms 2v − 1, for v = 1, 2, . . . ,m sum to m2, where m is the number of times a appears in the stream.

Notice that 2v − 1 is the difference between v2 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 v3−(v−1)3 = 3v2−3v+1. Then Pm v=1 3v2−3v+1 = m3, so we can use as our estimate of the third moment the formula n(3v2−3v+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 nvk − (v − 1)k_.

4.5.5 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

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

Thus, we have shown by induction on the stream length n that all positions have equal probability s/n of being chosen as the position of a variable.

4.5.6 Exercises for Section 4.5

Exercise 4.5.1 : Compute the surprise number (second moment) for the stream 3, 1, 4, 1, 3, 4, 2, 1, 2. What is the third moment of this stream?

Exercise 4.5.2 : If a stream has n elements, of which m are distinct, what are the minimum and maximum possible surprise number, as a function of m and n?

Exercise 4.5.3 : Suppose we are given the stream of Exercise 4.5.1, to which we apply the Alon-Matias-Szegedy Algorithm to estimate the surprise number. For each possible value of i, if Xi is a variable starting position i, what is the value of Xi.value?

Exercise 4.5.4 : Repeat Exercise 4.5.3 if the intent of the variables is to compute third moments. What is the value of each variable at the end? What estimate of the third moment do you get from each variable? How does the average of these estimates compare with the true value of the third moment?

Exercise 4.5.5 : Prove by induction on m that 1+3+5+· · ·+(2m−1) = m2.

Exercise 4.5.6 : If we wanted to compute fourth moments, how would we convert X.value to an estimate of the fourth moment?

4.6 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.

4.6.1 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 4.10 : 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.

4.6.2 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 ǫ > 0, and still uses only O(log2 N) bits (although with a constant factor that grows as ǫ 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,5 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(logN) 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).

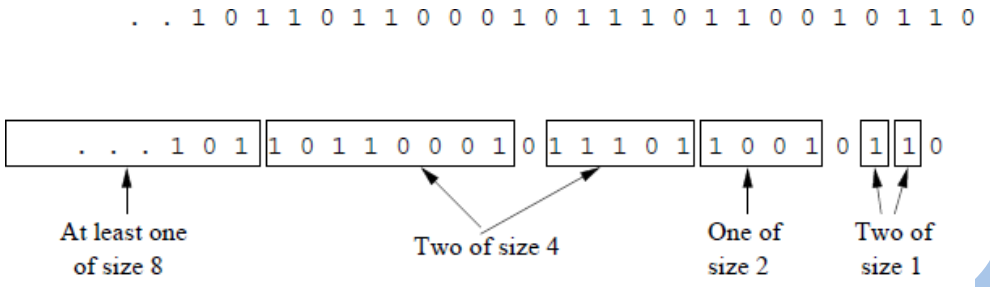


Figure 4.2: A bit-stream divided into buckets following the DGIM rules

Example 4.11 : 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.

4.6.3 Storage Requirements for the DGIM Algorithm

We observed that each bucket can be represented by O(logN) bits. If the window has length N, then there are no more than N 1's, surely. Suppose the largest bucket is of size 2j. Then j cannot exceed log2 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 log2 N down to 1, and no buckets of larger sizes.

We conclude that there are O(logN) buckets. Since each bucket can be represented in O(logN) bits, the total space required for all the buckets representing a window of size N is O(log2 N).

4.6.4 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 4.12 : Suppose the stream is that of Fig. 4.2, 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 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.

4.6.5 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(logN) time.

Example 4.13 : 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.

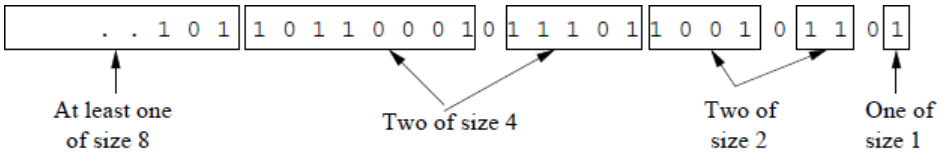


Figure 4.3: Modified buckets after a new 1 arrives in the stream

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 is as shown in Fig. 4.3.

4.6.6 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 $1 + (r - 1)(2^{j-1} + 2^{j-2} + \cdots + 1) = 1 + (r - 1)(2^j - 1)$. The overestimate is $2^{j-1} - 1$. 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 ϱ > 0.

4.6.7 Extensions to the Counting of Ones

It is natural to ask whether we can extend the technique of this section to handle aggregations more general 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 4.6.6 to estimate each ci with fractional error at most ǫ, then the estimate of the true sum has error at most ǫ. The worst case occurs when all the ci's are overestimated or all are underestimated by the same fraction.

4.6.8 Exercises for Section 4.6

Exercise 4.6.1 : Suppose the window is as shown in Fig. 4.2. Estimate the number of 1's the the last k positions, for k = (a) 5 (b) 15. In each case, how far off the correct value is your estimate?

! Exercise 4.6.2 : There are several ways that the bit-stream 1001011011101 could be partitioned into buckets. Find all of them.

Exercise 4.6.3 : Describe what happens to the buckets if three more 1's enter the window represented by Fig. 4.3. You may assume none of the 1's shown leave the window.

4.7 Decaying Windows

We have assumed that a sliding window held a certain tail of the stream, either the most recent N elements for fixed N, or all the elements that arrived after some time in the past. Sometimes we do not want to make a sharp distinction between recent elements and those in the distant past, but want to weight the recent elements more heavily. In this section, we consider "exponentially decaying windows," and an application where they are quite useful: finding the most common "recent" elements.

4.7.1 The Problem of Most-Common Elements

Suppose we have a stream whose elements are the movie tickets purchased all over the world, with the name of the movie as part of the element. We want to keep a summary of the stream that is the most popular movies "currently."

While the notion of "currently" is imprecise, intuitively, we want to discount the popularity of a movie like Star Wars–Episode 4, which sold many tickets, but most of these were sold decades ago. On the other hand, a movie that sold n tickets in each of the last 10 weeks is probably more popular than a movie that sold 2n tickets last week but nothing in previous weeks.

One solution would be to imagine a bit stream for each movie. The ith bit has value 1 if the ith ticket is for that movie, and 0 otherwise. Pick a window size N, which is the number of most

recent tickets that would be considered in evaluating popularity. Then, use the method of Section 4.6 to estimate the number of tickets for each movie, and rank movies by their estimated counts. This technique might work for movies, because there are only thousands of movies, but it would fail if we were instead recording the popularity of items sold at Amazon, or the rate at which different Twitter-users tweet, because there are too many Amazon products and too many tweeters. Further, it only offers approximate answers.

4.7.2 Definition of the Decaying Window

An alternative approach is to redefine the question so that we are not asking for a count of 1's in a window. Rather, let us compute a smooth aggregation of all the 1's ever seen in the stream, with decaying weights, so the further back in the stream, the less weight is given. Formally, let a stream currently consist of the elements a1, a2, . . . , at, where a1 is the first element to arrive and at is the current element. Let c be a small constant, such as 10−6 or 10−9. Define the exponentially decaying window for this stream to be the sum

$$\sum_{i=0}^{t-1} a_{t-i}(1-c)^i$$

The effect of this definition is to spread out the weights of the stream elements as far back in time as the stream goes. In contrast, a fixed window with the same sum of the weights, 1/c, would put equal weight 1 on each of the most recent 1/c elements to arrive and weight 0 on all previous elements. The distinction is suggested by Fig. 4.4.

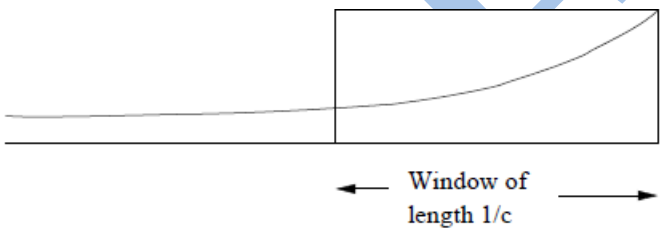


Figure 4.4: A decaying window and a fixed-length window of equal weight

It is much easier to adjust the sum in an exponentially decaying window than in a sliding window of fixed length. In the sliding window, we have to worry about the element that falls out of the window each time a new element arrives. That forces us to keep the exact elements along with the sum, or to use an approximation scheme such as DGIM. However, when a new element at+1 arrives at the stream input, all we need to do is:

1. Multiply the current sum by 1 − c.
2. Add at+1.

The reason this method works is that each of the previous elements has now moved one position further from the current element, so its weight is multiplied by 1 − c. Further, the weight on the current element is (1 − c)0 = 1, so adding at+1 is the correct way to include the new element's contribution.

4.7.3 Finding the Most Popular Elements

Let us return to the problem of finding the most popular movies in a stream of ticket sales.6 We shall use an exponentially decaying window with a constant c, which you might think of as 10−9. That is, we approximate a sliding window holding the last one billion ticket sales. For each

movie, we imagine a separate stream with a 1 each time a ticket for that movie appears in the stream, and a 0 each time a ticket for some other movie arrives. The decaying sum of the 1's measures the current popularity of the movie.

We imagine that the number of possible movies in the stream is huge, so we do not want to record values for the unpopular movies. Therefore, we establish a threshold, say 1/2, so that if the popularity score for a movie goes below this number, its score is dropped from the counting. For reasons that will become obvious, the threshold must be less than 1, although it can be any number less than 1. When a new ticket arrives on the stream, do the following:

1. For each movie whose score we are currently maintaining, multiply its score by (1 − c).
2. Suppose the new ticket is for movie M. If there is currently a score for M, add 1 to that score. If there is no score for M, create one and initialize it to 1.
3. If any score is below the threshold 1/2, drop that score. It may not be obvious that the number of movies whose scores are maintained at any time is limited. However, note that the sum of all scores is 1/c.

There cannot be more than 2/c movies with score of 1/2 or more, or else the sum of the scores would exceed 1/c. Thus, 2/c is a limit on the number of movies being counted at any time. Of course in practice, the ticket sales would be concentrated on only a small number of movies at any time, so the number of actively counted movies would be much less than 2/c.

10. USING GRAPH ANALYTICS FOR BIG DATA

While we have discussed how the various techniques and technologies for big data appliances can increase performance (in terms of scalability for computation speed or for data volumes), we have also seen that the ways that certain applications are mapped to a typical big data stack might have limitations in scalability due to memory access latency or network bandwidth. Yet the promise of big data must go beyond increased scalability for known problems.

Big data analytics systems should enable a platform that can support different analytics techniques that can be adapted in ways that help solve a variety of challenging problems. This suggests that these systems are high performance, elastic distributed data environments that enable the use of creative algorithms to exploit variant modes of data management in ways that differ from the traditional batchoriented approach of traditional approaches to data warehousing.

In this chapter, we look at graph analytics, which is an analytics alternative that uses an abstraction called a graph model. The simplicity of this model allows for rapidly absorbing and connecting large volumes of data from many sources in ways that finesse limitations of the source structures (or lack thereof, of course). Graph analytics is an alternative to the traditional data warehouse model as a framework for absorbing both structured and unstructured data from various sources to enable analysts to probe the data in an undirected manner.

As we will see, the graph model allows you to tightly couple the meaning of entity relationships as part of the representation of the relationship. This effectively embeds the semantics of relationships among different entities within the structure, providing an ability to both invoke traditional-style queries (to answer typical ―search‖ queries modeled after known patterns) and enable more sophisticated undirected analyses. These undirected ―discovery‖ analyses, include inferencing, identification of interesting patterns, and application of deduction, all using an iterative approach that analysts can use to discover actionable knowledge that was previously unknown. This allows the analysts to rapidly seek out emerging patterns and enable real-time knowledge-driven decision making in the context of how these newly discovered patterns impact the corporate business value drivers.

10.1 WHAT IS GRAPH ANALYTICS?

It is worth delving somewhat into the graph model and the methods used for managing and manipulating graphs:

- What constitutes graph analytics?
- Types of problems that are suited to graph analytics.
- Types of questions that are addressed using graph analytics.
- Types of graphs that are commonly encountered.
- The degree of prevalence within big data analytics problems.

This motivates an understanding of its utility and flexibility for discovery-style analysis in relation to specific types of business problems, how common those types of problems are, and why they are nicely abstracted to the graph model. In addition, we will discuss the challenges of attempting to execute graph analytics on conventional hardware and consider aspects of specialty platforms that can help in achieving the right level of scalability and performance.

10.2 THE SIMPLICITY OF THE GRAPH MODEL

Graph analytics is based on a model of representing individual entities and numerous kinds of relationships that connect those entities. More precisely, it employs the graph abstraction for representing connectivity, consisting of a collection of vertices (which are also referred to as nodes or points) that represent the modeled entities, connected by edges (which are also referred to as links, connections, or relationships) that capture the way that two entities are related.

Some simple examples are shown in Figure 10.1.

The flexibility of the model is based on its simplicity. A simple unlabeled undirected graph, in which the edges between vertices neither reflect the nature of the relationship nor indicate their direction, has limited utility.

But, as we see in the examples in Figure 10.1, adding context by labeling the vertices and edges enhances the meanings embedded within the graph, and by extension, the entire representation of a network.

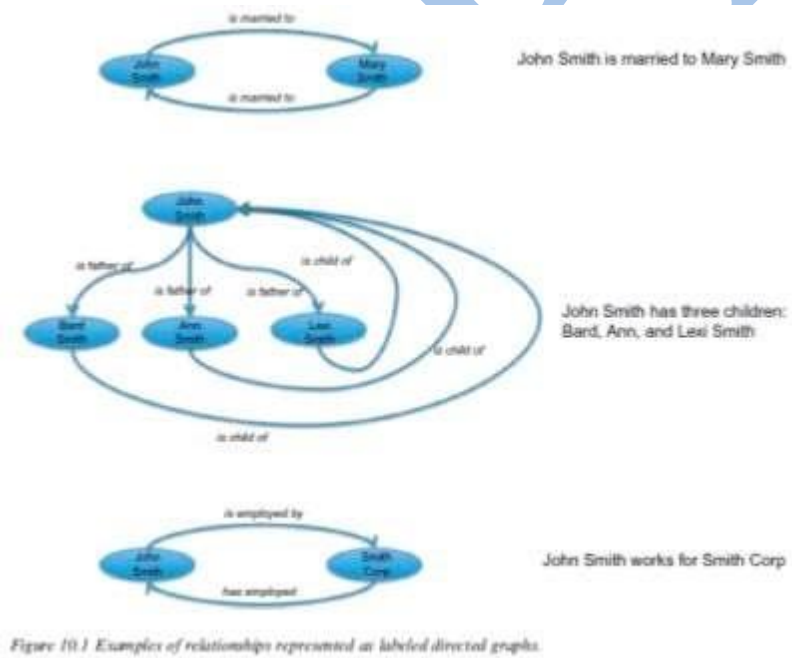


Figure 10.1 Examples of relationships represented as labeled directed graphs.

Among other enhancements, these can enrich the meaning of the nodes and edges represented in the graph model:

• Vertices can be labeled to indicate the types of entities that are related.
• Edges can be labeled with the nature of the relationship.
• Edges can be directed to indicate the ‐flow‖ of the relationship.
• Weights can be added to the relationships represented by the edges.
• Additional properties can be attributed to both edges and vertices.
• Multiple edges can reflect multiple relationships between pairs of vertices.

10.3 REPRESENTATION AS TRIPLES

In essence, these enhancements help in building a semantic graph—a directed graph that can be represented using a triple format consisting of a subject (the source point of the relationship), an object (the target), and a predicate (that models the type of the relationship).

The same examples in Figure 10.1 could be modeled as a set of triples, as shown in Table 10.1.

A collection of these triples is called a semantic database, and this kind of database can capture additional properties of each triple relationship as attributes of the triple. Almost any type of entity and relationship can be represented in a graph model, which means two key things: the process of adding new entities and relationships is not impeded when new datasets are to be included, with new types of entities and connections to be incorporated, and the model is particularly suited to types of discovery analytics that seek out new patterns embedded within the graph that are of critical business interest.

10.4 GRAPHS AND NETWORK ORGANIZATION

The concept of the social network has been around for many years, and in recent times has been materialized as communities in which individual entities create online personas and connect and interact with others within the community. Yet the idea of the social network extends beyond specific online implementations, and encompasses a wide variety of example environments that directly map to the graph model. That being said, one of the benefits of the graph model is the ability to detect patterns or organization that are inherent within the represented network, such as:

• Embedded micronetworks: Looking for small collections of entities that form embedded ‐microcommunities.‖ Some examples include determining the originating sources for a hot new purchasing trend,

Subject	Predicate	Object
John Smith	Is married to	Mary Smith
Mary Smith	Is married to	John Smith
John Smith	Is father of	Brad Smith
John Smith	Is father of	Ann Smith
John Smith	Is father of	Lexi Smith
Brad Smith	Is child of	John Smith
Ann Smith	Is child of	John Smith
Lexi Smith	Is child of	John Smith
John Smith	Is employed by	Smith Corp
Smith Corp	Has employed	John Smith

Triples derived identifying a terrorist cell based on patterns of communication across a broad swath of call detail records, or sets of individuals within a particular tiny geographic region with similar political views.

• Communication models: Modeling communication across a community triggered by a specific event, such as monitoring the ‐buzz‖ across a social media channel associated with the rumored

release of a new product, evaluating best methods for communicating news releases, or correlation between travel delays and increased mobile telephony activity.

• Collaborative communities: Isolating groups of individuals that share similar interests, such as groups of health care professionals working in the same area of specialty, purchasers with similar product tastes, or individuals with a precise set of employment skills.

• Influence modeling: Looking for entities holding influential positions within a network for intermittent periods of time, such as computer nodes that have been hijacked and put to work as proxies for distributed denial of service attacks or for emerging cybersecurity threats, or individuals that are recognized as authorities within a particular area.

• Distance modeling: Analyzing the distances between sets of entities, such as looking for strong correlations between occurrences of sets of statistically improbable phrases among large sets of search engines queries, or the amount of effort necessary to propagate a message among a set of different communities.

Each of these example applications is a discovery analysis that looks for patterns that are not known in advance. As a result, these are less suited to pattern searches form relational database systems, such as a data warehouse or data mart, and are better suited to a more dynamic representation like the graph model.

10.5 CHOOSING GRAPH ANALYTICS

Deciding the appropriateness of an analytics application to a graph analytics solution instead of the other big data alternatives can be based on these characteristics and factors of business problems:

• Connectivity: The solution to the business problem requires the analysis of relationships and connectivity between a variety of different types of entities.

Undirected discovery: Solving the business problem involves iterative undirected analysis to seek out as-of-yet unidentified patterns.

• Absence of structure: Multiple datasets to be subjected to the analysis are provided without any inherent imposed structure.

• Flexible semantics: The business problem exhibits dependence on contextual semantics that can be attributed to the connections and corresponding relationships.

• Extensibility: Because additional data can add to the knowledge embedded within the graph, there is a need for the ability to quickly add in new data sources or streaming data as needed for further interactive analysis.

• Knowledge is embedded in the network: Solving the business problem involves the ability to exploit critical features of the embedded relationships that can be inferred from the provided data.

• Ad hoc nature of the analysis: There is a need to run ad hoc queries to follow lines of reasoning.

• Predictable interactive performance: The ad hoc nature of the analysis creates a need for high performance because discovery in big data is a collaborative man/machine undertaking, and predictability is critical when the results are used for operational decision making.

10.6 GRAPH ANALYTICS USE CASES

Having reviewed the business problems that are suited to using a solution based on a graph analytics model, we can examine some use cases for graph analytics. In these scenarios, we look at the business problem, why the traditional approach may not be optimal, and then discuss how the graph analytics approach is best suited to solving the business challenges.

Some example business applications that are aligned with these criteria include:

• Health care quality analytics, in which patient health encounter histories, diagnoses, procedures, treatment approaches, and results of clinical trials contribute to the ability to analyze the comparative effectiveness of different health care options. The process requires the absorption of many different collections of medical histories, clinical records, laboratory results, and prescription records from many different sources and systems. In turn, the application must guide health care providers in a timely and efficient manner by enabling the rapid assessment of therapies used for other patients with similar characteristics (such as age, clinical history, and associated risk factors) that have the most positive outcomes. This analysis creates a graph representing the corresponding relationships and may also combine health histories with additional data sources that can be easily integrated into the existing graph model.
• Concept-based correlations that seek to organize large bodies of knowledge. Some examples include looking for contextual relationships between scientific health care research and particular types of pharmaceuticals, investigative journalists seeking connections among individuals referred to in a variety of news sources, fraud analysts evaluating financial irregularities across multiple-related organizations, or even correlations between corporate behavior and increased health risks. Each of these examples involve the absorption of many content artifacts from multiple varied data sources, the need to combine isolated pieces of information extracted from among the corpus of documents, and a discovery activity looking for correlations that are currently hidden or unknown.
• Cybersecurity: The numbers of cyberattacks are increasing, and their sophistication is expanding way beyond distributed denial of service (DDoS), which is more frequently being used as a premise and distraction while more insidious attacks seek to traverse corporate firewalls, extract critical business information, or incrementally drain individual financial accounts while operating completely under the radar. Monitoring for cybersecurity events is a process in which a wide variety of massive streaming datasets (such as network logs, NetFlow, DNS, and IDS data) need to be rapidly captured and integrated into a model that allows for both identification of known attack patterns as well as the discovery of new patterns that emerge as the attacks become more elaborate. A graph analytics approach can address the challenge by capturing and loading the entire volume and breadth of the available datasets coupled with an evolving model that can quickly log connections and relationships that is used to identify new patterns of attack. This graph-based approach allows analysts to rapidly get results from many ad hoc queries requested from a model managing massive amounts of data representing thousands of interconnected entities. This approach enables the analysts to quickly identify potential cyber-threats in minutes, or even seconds so that defensive actions can be taken quickly.

10.7 GRAPH ANALYTICS ALGORITHMS AND SOLUTION APPROACHES

The graph model is inherently suited to enable a broad range of analyses that are generally unavailable to users of a standard data warehouse framework. As suggested by these examples, instead of just providing reports or enabling online analytical processing (OLAP) systems, graph analytics applications employ algorithms that traverse or analyze graphs to detect and potentially identify interesting patterns that are sentinels for business opportunities for increasing revenue, identifying security risks, detecting fraud, waste, or abuse, financial trading signals, or even looking for optimal individualized health care treatments. Some of the types of analytics algorithmic approaches include:
• Community and network analysis, in which the graph structures are traversed in search of groups of entities connected in particularly ―close‖ ways. One example is a collection of entities

that are completely connected (i.e., each member of the set is connected to all other members of the set).
• Path analysis, which analyze the shapes and distances of the different paths that connect entities within the graph.
• Clustering, which examines the properties of the vertices and edges to identify characteristics of entities that can be used to group them together.
• Pattern detection and pattern analysis, or methods for identifying anomalous or unexpected patterns requiring further investigation.
• Probabilistic graphical models such as Bayesian networks or Markov networks for various application such as medical diagnosis, protein structure prediction, speech recognition, or assessment of default risk for credit applications.
• Graph metrics that are applied to measurements associated with the network itself, including the degree of the vertices (i.e., the number of edges in and out of the vertex), or centrality and distance (including the degree to which particular vertices are ―centrally located‖ in the graph, or how close vertices are to each other based on the length of the paths between them).
These graph analytic algorithms can yield interesting patterns that might go undetected in a data warehouse model, and these patterns themselves can become the templates or models for new searches. In other words, the graph analytics approach can satisfy both the discovery and the use of patterns typically used for analysis and reporting.

10.8 TECHNICAL COMPLEXITY OF ANALYZING GRAPHS

As more use cases similar to the ones shared in this chapter emerge that are expected to consume and analyze massive datasets, the question of performance gradually stands out as one of the potential barriers to success. To what extent do big data platforms effectively satisfy the demand for high performance? Answering that question requires a better understanding of some of the complexities of graph analytics problems and topologies.
There are some characteristics of graphs that inhibit the ability of typical computing platforms to provide the rapid responses or satisfy the need for scalability, especially as data volumes continue to explode. Some of the factors that might introduce performance penalties that are not easily addressed on standard hardware architectures include:
• Unpredictability of graph memory accesses: The types of discovery analyses in graph analytics often require the simultaneous traversal of multiple paths within a network to find the interesting patterns for further review or optimal solutions to a problem. The graph model is represented using data structures that incorporate the links between the represented entities, and this differs substantially from a traditional database model. In a parallel environment, many traversals can be triggered at the same time, but each graph traversal is inherently dependent on the ability to follow the links from the subject to the target in the representation. Unlike queries to structured databases, the memory access patterns are not predictable, limiting the ability to reduce data access latency by efficiently streaming prefetched data through the different levels of the memory hierarchy, as briefly discussed in Chapter 6. Much execution time is spent accessing memory, while the processors wait for the memory accesses to complete.
• Graph growth models: As more information is introduced into an environment, real-world networks grow in an interesting way. The larger the graph becomes, the more it exhibits what is called ―preferential connectivity‖—newly introduced entities are more likely to connect to already existing ones, and existing nodes with a high degree of connectivity are more likely to continue to be ―popular‖ in that they will continue to attract new connections. This means

that graphs continue to grow, but apparently that growth does not scale equally across the data structure.

• Dynamic interactions with graphs: As with any big data application, graphs to be analyzed are populated from a wide variety of data sources of large or massive data volumes, streamed at varying rates.

But while the graph must constantly absorb many rapidly changing data streams and process the incorporation of connections and relationships into the persistent representation of the graph, the environment must also satisfy the need to rapidly respond to many simultaneous discovery analyses. Therefore, a high-performance graph analytics solution must accommodate the dynamic nature without allowing for any substantial degradation in analytics performance.

• Complexity of graph partitioning: Graphs tend to cluster among centers of high connectivity, as many networks naturally have ―hubs‖ consisting of a small group of nodes with many connections. The benefit of having hubs is that in general it shortens the distances among collections of nodes in the graph. The hubs often showcase entities that may be of particular interest because of their perceived centrality. But while one of the benefits of big data platforms is the expectation of distribution of data and computation, the existence of hubs make it difficult to partition the graph among different processing units because of the multiplicity of connections. Arbitrarily distributing the data structures in ways that span or cross multiple partitions will lead to increased cross-partition network traffic, which effectively eliminates the perceived performance benefit of data distribution.

10.9 FEATURES OF A GRAPH ANALYTICS PLATFORM

If your big data challenge is suited to a graph analytics solution, it is worth enumerating some key features to look for, both from the software and from the hardware platform perspectives. These can effectively grouped into three sets of features: ease of development and implementation, interoperability with complementary reporting and analysis technologies, and system execution performance.

The ease of development features of the platform is enabled through the adoption of industry standards as well as supporting general requirements for big data applications, such as:

• Seamless data intake: Providing a seamless capability to easily absorb and fuse data from a variety of different sources.

• Data integration: A semantics-based approach is necessary for graph analytics to integrate different sets of data that do not have predetermined structure. Similar to other NoSQL approaches, the schema-less approach of the semantic model must provide the flexibility not provided by relational database models.

• Inferencing: The application platform should provide methods for inferencing and deduction of new information and insights derived from the embedded relationships and connectivity.

• Standards-based representation: Any graph analytics platform must employ the resource description framework standard to use triples for representing the graph.

Representing the graph using RDF allows for the use of the SPARQL query language standard for executing queries against a triple-based data management environment.

Interoperability is critical, and here are some considerations for the operational aspects of the platform:

• Workflow integration: Providing a graph analytics platform that is segregated from the existing reporting and analytics environments will have limited value when there are gaps in incorporating results from across the different environments. Make sure that the graph analytics

platform is aligned with the analysis and decision-making workflows for the associated business processes.

• Visualization: Presenting discoveries using visualization tools is critical to highlight their value. Look for platforms that have tight integration with visualization services.

• ‑Complementariness‖: A graph analytics platform augments an organization's analytics capability and is not intended to be a replacement. Any graph analytics capabilities must complement existing data warehouses, data marts, OLAP engines, and Hadoop analytic environments.

Finally, although all of the following are reasonable expectations for any big data platform, these criteria are particularly important for graph analytics due to the nature of the representation and the types of discovery analyses performed:

• High-speed I/O: Meeting this expectation for real-time integration requires a scalable infrastructure, particularly with respect to highspeed I/O channels, which will speed the intake of multiple data streams and thereby support graphs that are rapidly changing as new data is absorbed.

• High-bandwidth network: As a significant burden of data access may cross node barriers, employing a high-speed/high-bandwidth network interconnect will also help reduce data latency delays, especially for chasing pointers across the graph.

• Multithreading: Fine-grained multithreading allows the simultaneous exploration of different paths and efficiently creating, managing, and allocating threads to available nodes on a massively parallel processing architecture can alleviate the challenge of predictability.

• Large memory: A very large memory shared across multiple processors reduces the need to partition the graph across independent environments. This helps to reduce the performance impacts associated with graph partitioning. Large memories are useful in keeping a large part, if not all of the graph resident in-memory. By migrating allocated tasks to where the data is resident in memory reduces the impact of data access latency delays.

10.10 CONSIDERATIONS: DEDICATED APPLIANCES FOR GRAPH ANALYTICS

There are different emerging methods of incorporating graph analytics into the enterprise. One class is purely a software approach, providing an ability to create, populate, and query graphs. This approach enables the necessary functionality and may provide the ease-ofimplementation and deployment. Most, if not all, software implementations will use industry standards, such as RDF and SPARQL, and may even leverage complementary tools for inferencing and deduction. However, the performance of a software-only implementation is limited by its use of the available hardware, and even using commodity servers cannot necessarily enable it to natively exploit performance and optimization.

Another class is the use of a dedicated appliance for graph analytics.

From an algorithmic perspective, this approach is equally capable as one that solely relies on software. However, from a performance perspective, there is no doubt that a dedicated platform will take advantage of high-performance I/O, high-bandwidth networking, in-memory computation, and native multithreading to provide the optimal performance for creating, growing, and analyzing graphs that are built from multiple high-volume data streams.

Software approaches may be satisfactory for smaller graph analytics problems, but as data volumes and network complexity grow, the most effective means for return on investment may necessitate the transition to a dedicated graph analytics appliance.