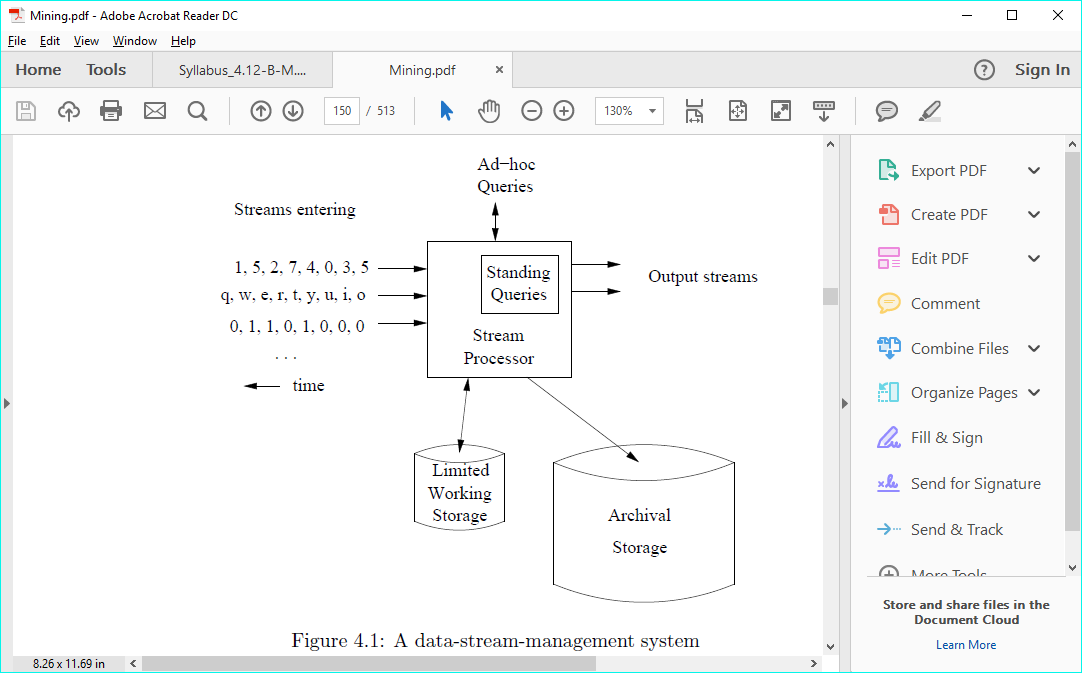
**The Stream Data Model**

Some typical applications where the stream model applies will be examined.



**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 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 entirely, 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.

**4.1.4 Issues in Stream Processing**

* 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:

1. Often, it is much more efficient to get an approximate answer to our problem than an exact solution.
2. 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.
* 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. 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 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 random number generator, with the important property that, when applied to the same user several times, we always get the same “random” number.
* That is, without storing the in/out decision for any user, we can reconstruct that decision any time a search query by that user arrives.
* More generally, we can obtain a sample consisting of any rational fraction a/b of the users by hashing user names to b buckets, 0 through b − 1. Add the search query to the sample if the hash value is less than a.

4.2.3 The General Sampling Problem

* The running example is typical of the following general problem. Our stream consists of tuples with n components.
* A subset of the components are the key components, on which the selection of the sample will be based. In our running
* example, there are three components – user, query, and time – of which only user is in the key.
* However, we could also take a sample of queries by making query be the key, or even take a sample of user-query pairs by making both those components form the key.
* To take a sample of size a/b, we hash the key value for each tuple to b buckets, and accept the tuple for the sample if the hash value is less than a.
* If the key consists of more than one component, the hash function needs to combine the values for those components to make a single hash-value.
* The result will be a sample consisting of all tuples with certain key values 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.

4.3 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.

**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. 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**

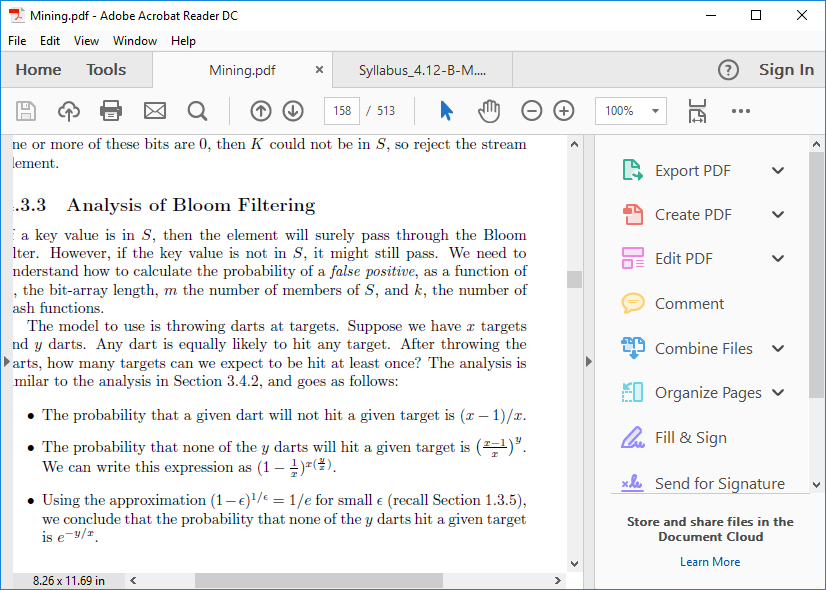
* A Bloom filter consists of:

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:



**4.4 Counting Distinct Elements in a Stream**

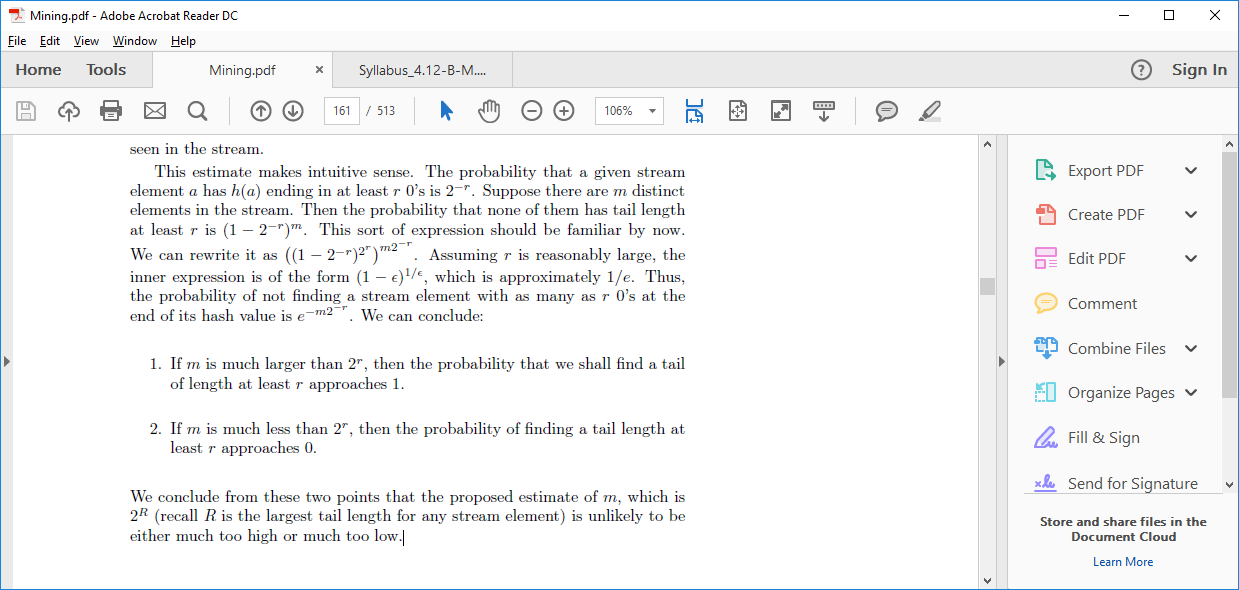
* In this section we look at a third simple kind of processing we might want to do on a stream. As with the previous examples – sampling and filtering – it is somewhat tricky to do what we want in a reasonable amount of main memory,so we use a variety of hashing and a randomized algorithm to get approximately 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 2R for the number of distinct elements seen in the stream.



**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.

**4.5 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.

**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 × 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 × 12 = 8110.

**4.5.2 The Alon-Matias-Szegedy Algorithm for Second Moments**

* For now, let us assume that a stream has a particular length n.
* 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.

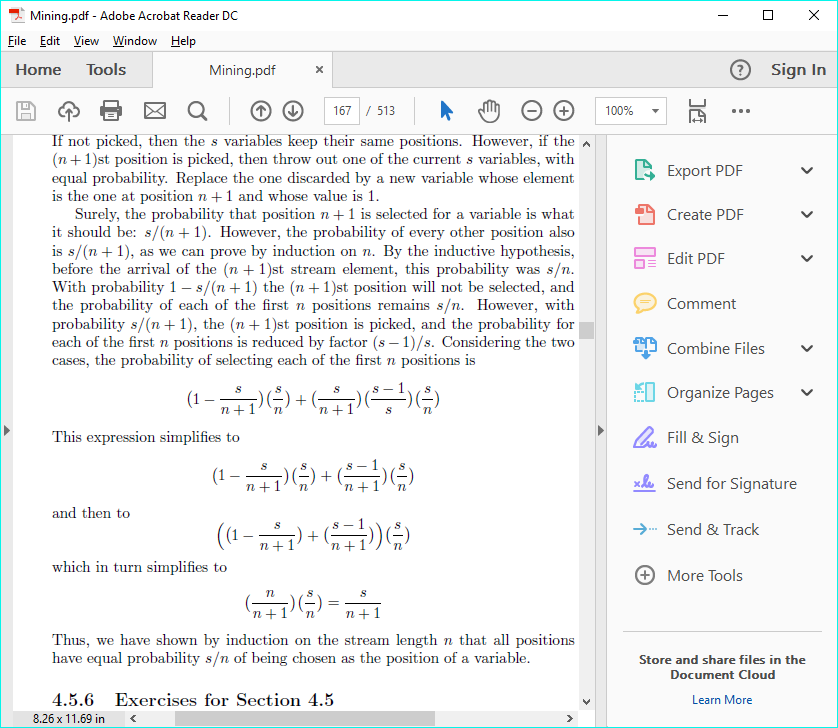
* To determine the value of a variable X, we choose a position in the stream between 1 and n, uniformly and at random. Set X.element to be the element found there,and initialize X.value to 1. As we read the stream, add 1 to X.value each time we encounter another occurrence of X.element.
* **Example 4.7 :** 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.

**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 n(vk − (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



**4.6 Counting Ones in a Window**

* 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.

**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.

**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%.
* 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.

1. 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.
2. Thus, O(logN) bits suffice to represent a bucket.
3. 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.

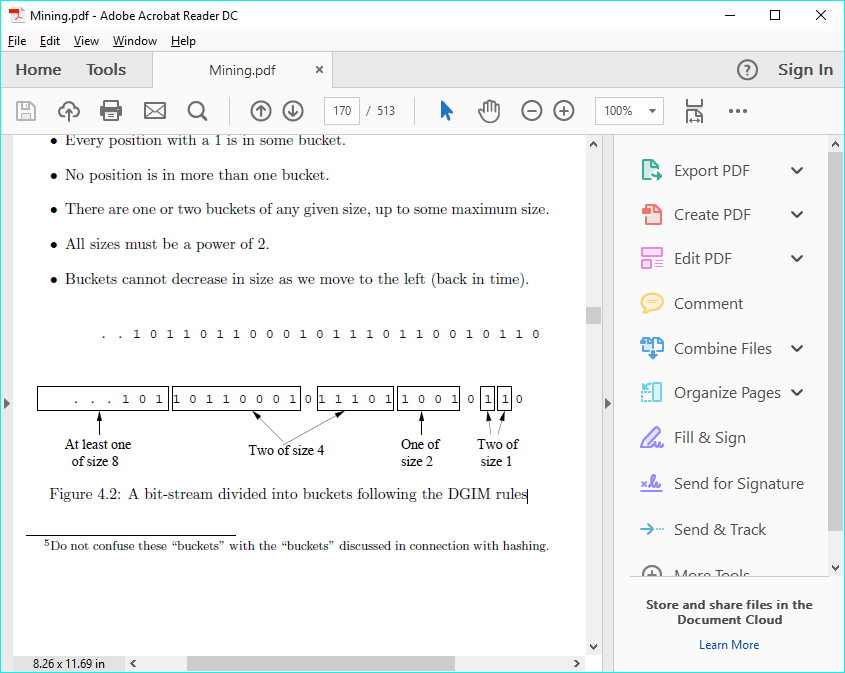
• Every position with a 1 is in some bucket.

• 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).



**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 ≤ k ≤ 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.

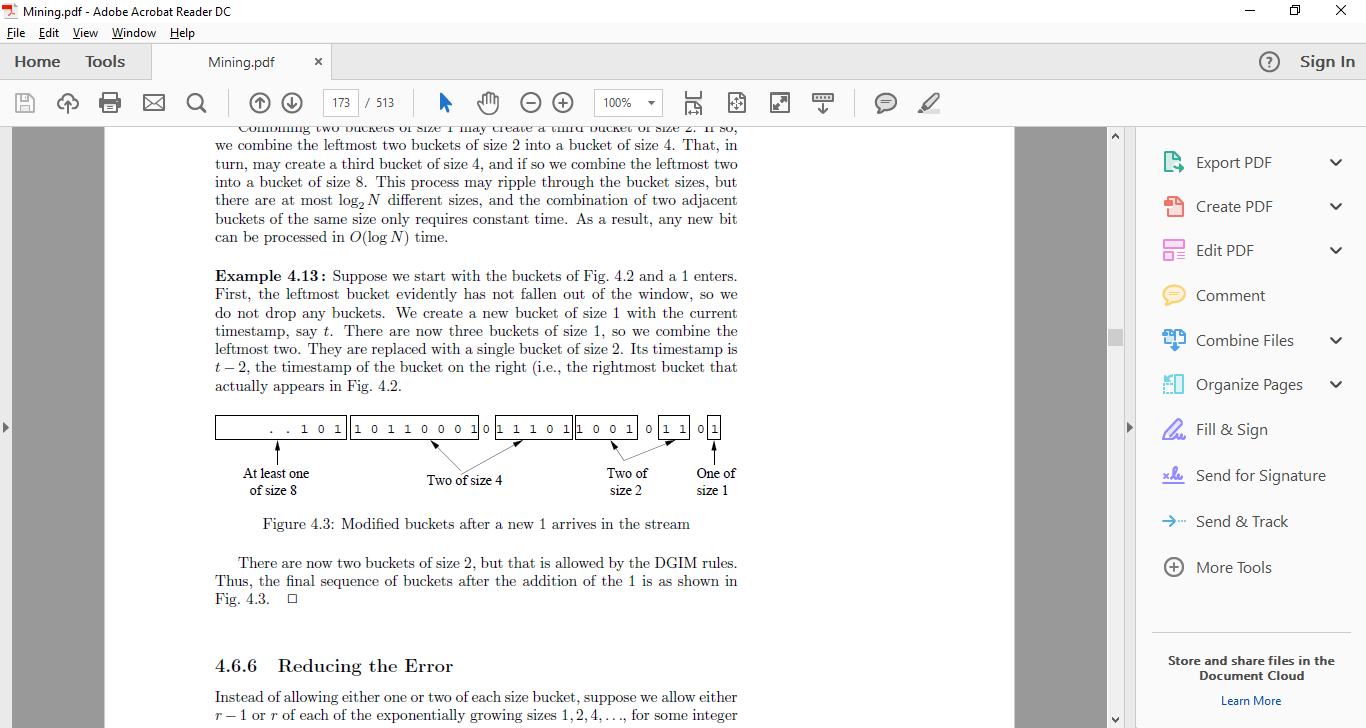
1. Case 1: The estimate is less than c. In the worst case, all the 1’s of b are actually within the range of the query, so the estimate misses half bucket b, or 2j−1 1’s. But in this case, c is at least 2j; in fact it is at least 2j+1 − 1, since there is at least one bucket of each of the sizes 2j−1,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

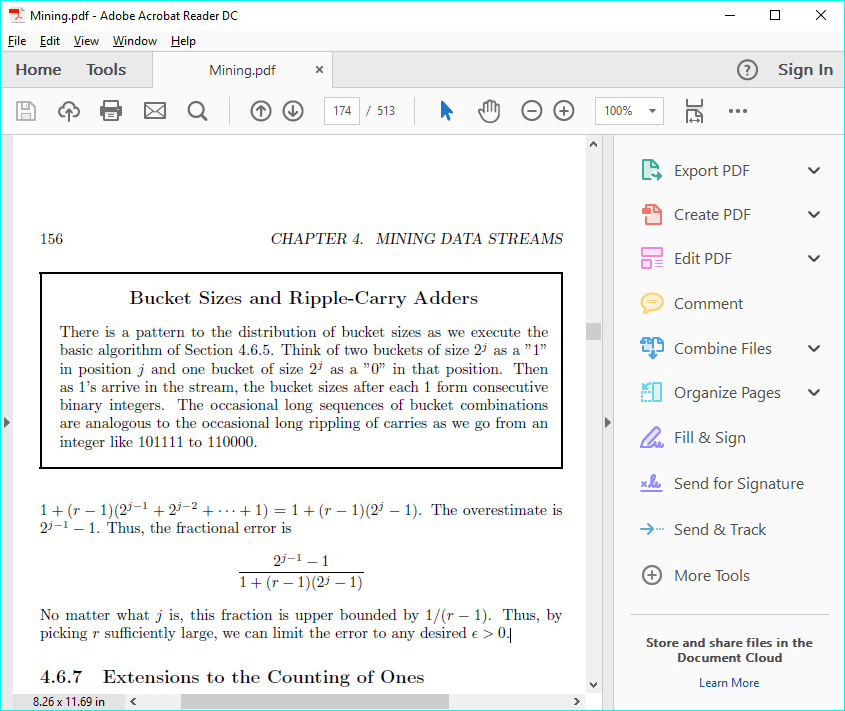
* Suppose we have a window of length N properly represented by buckets that satisfy the DGIM conditions. When a new bit comes in, we may need to modify the buckets, so they continue to represent the window and continue to satisfy the DGIM conditions. First, whenever a new bit enters:

1. 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 mayneed 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.



**4.6.6 Reducing the Error**

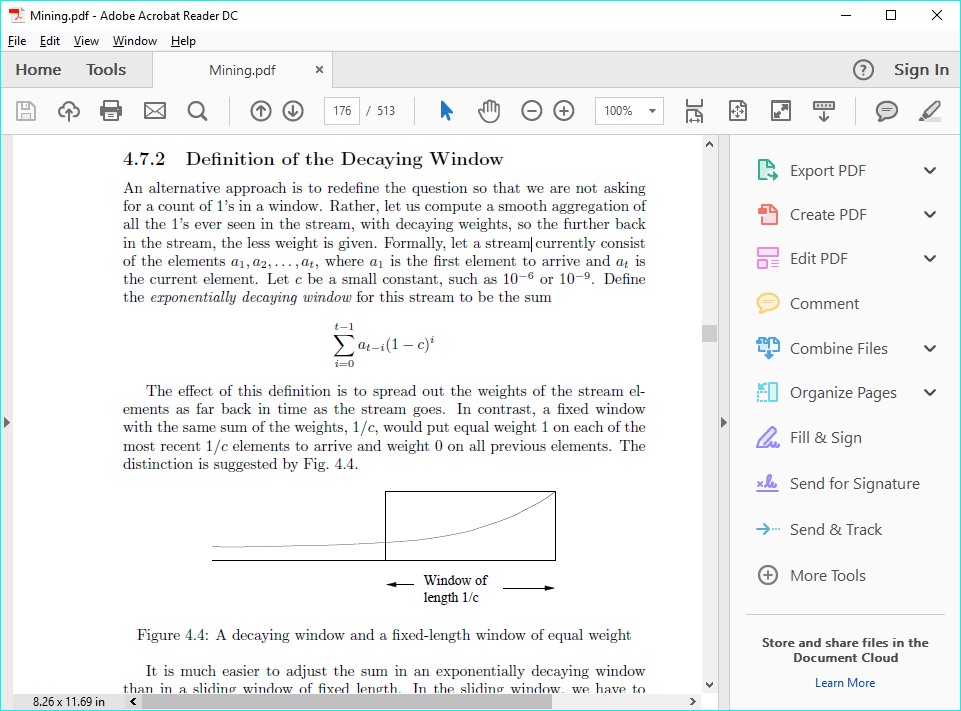
* Instead of allowing either one or two of each size bucket, suppose we allow either r −1 or r of each of the exponentially growing sizes 1, 2, 4, . . ., for some integer r > 2. In order to represent any possible number of 1’s, we must relax this condition for the buckets of size 1 and buckets of the 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 

**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.



* 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.