Small Summaries for Big Data

Graham Cormode and Ke Yi

Acknowledgments

We thank the many colleagues and friends who have read drafts of this volume, and provided feedback and suggestions.

These include Edith Cohen, Gil Einziger, Floris Geerts, Nikolai Karpov, Edo Liberty, Lee Rhodes, Justin Thaler, Andrew Twigg, Pavel Veselý, and Qin Zhang.

Contents

1

Introduction

"Space," it says, "is big. Really big. You just won't believe how vastly, hugely, mindbogglingly big it is. I mean, you may think it's a long way down the road to the chemist's, but that's just peanuts to space, listen..."

Douglas Adams, The Hitchhiker's Guide to the Galaxy

1.1 Small Summaries for Big Data

Data, to paraphrase Douglas Adams, is big. Really big. Moreover, it is getting bigger, due to increased abilities to measure and capture more information. Sources of big data are becoming increasingly common, while the resources to deal with big data (chiefly, processor power, fast memory and slower disk) are growing at a slower pace. The consequence of this trend is that we need more effort in order to capture and process data in applications. Careful planning and scalable architectures are needed to fulfill the requirements of analysis and information extraction on big data. While the 'big' in big data can be interpreted more broadly, to refer to the big potential that data has to offer, or the wide variety of data, the focus of this volume is primarily on the scale of data.

Some examples of applications that generate large volumes of data include:

Physical Data. The growing development of sensors and sensor deployments have led to settings where measurements of the physical world are available at very high dimensionality and at a great rate. Scientific measurements are the cutting edge of this trend. Astronomy data gathered from modern telescopes can easily generate terabytes of data

in a single night. Aggregating large quantities of astronomical data provides a substantial big data challenge to support the study and discovery of new phenomena. Big data from particle physics experiments is also enormous: each experiment can generate many terabytes of readings, which can dwarf what is economically feasible to store for later comparison and investigation.

Medical Data. It is increasingly feasible to sequence entire genomes. A single genome is not so large—it can be represented in under a gigabyte—but considering the entire genetic data of a large population represents a big data challenge. This may be accompanied by increasing growth in other forms of medical data, based on monitoring multiple vital signs for many patients at fine granularity. Collectively, this leads to the area of data-driven medicine, seeking better understanding of disease, and leading to new treatments and interventions, personalized for each individual patient.

Activity Data. Human activity data is increasingly being captured and stored. Online social networks record not just friendship relations but interactions, messages, photos and interests. Location data is also more available, due to mobile devices which can obtain GPS information. Other electronic activities, such as patterns of website visits, email messages and phone calls can be collected and analyzed. Collectively, this provides ever-larger collections of activity information. Service providers who can collect this data seek to make sense of it in order to identify patterns of behavior or signals of behavioral change, and opportunities for advertising and marketing.

Business Data. Businesses are increasingly able to capture more and complex data about their customers. Online stores can track millions of customers as they explore their site, and seek patterns in purchasing and interest, with the aim of providing better service, and anticipating future needs. The detail level of data is getting finer and finer. Previously, data would be limited to just the items purchased, but now extends to more detailed shopping and comparison activity, tracking the whole path to purchase.

Across all of these disparate settings, certain common themes emerge. The data in question is large, and growing. The applications seek to extract patterns, trends or descriptions of the data. Scalability and timeliness of response are vital in many of these applications.

In response to these needs, new computational paradigms are being adopted to deal with the challenge of big data. Large scale distributed computation is a central piece: the scope of the computation can exceed what is feasible on a single machine, and so clusters of machines work together in parallel. On top of these architectures, parallel algorithms are designed that can take the complex task and break it into independent pieces suitable for distribution over multiple machines.

A central challenge within any such system is how to compute and represent complex features of big data in a way that can be processed by many single machines in parallel. One answer is to be able to build and manipulate a compact summary of a large amount of data, modeled as a mathematical object. This notion of a small summary is the subject of study of this work. The idea of a summary is a natural and familiar one. It should represent something large and complex in a compact fashion. Inevitably, a summary must dispense with some of the detail and nuance of the object which it is summarizing. However, it should also preserve some key features of the object in an accurate fashion.

There is no single summary which accurately captures all properties of a data set, even approximately. Thus, at the heart of the study of small summaries are the questions of *what should be preserved?* and *how accurately can it be preserved?*. The answer to the first question determines which of many different possible summary types may be appropriate, or indeed whether any compact summary even exists. The answer to the second question can determine the size and processing cost of working with the summary in question.

Another important question about summaries for big data is how they can be constructed and maintained as new data arrives. Given that it is typically not feasible to load all the data into memory on one machine, we need summaries which can be constructed incrementally. That is, we seek summaries that can be built by observing each individual data item in turn, and updating the partial summary. Or, more strongly, we seek summaries such that summaries of different subsets of data built on different machines can be combined together to obtain a single summary that accurately represents the full data set.

Note that the notion of summarization is distinct from that of *compression*. In general, lossless compression is concerned with identifying regularity and redundancy in datasets to provide a more compact exact representation of the data. This is done for the purpose of compactly storing the data, or reducing the data transmission time. However, in general, there is no guarantee of significant size reduction from

compression. The compressed form is also typically difficult to analyze, and decompression is required in order to work with the data. In contrast, summarization is intended to provide a very significant reduction in the size of the data (sometimes several orders of magnitude), but does not promise to reconstruct the original data, only to capture certain key properties. Lossy compression methods fall in between, as they can provide guaranteed size reductions. They also aim to allow an approximate reconstruction of the original data with some limited loss of fidelity: typically, based on the human perception of multimedia data, such as audio or video. Summarization aims to provide only small loss of fidelity, but measured along other dimensions; summaries do not necessarily provide a way to reconstruct even an approximation of the original input.

As a first example of summarization, consider a data set consisting of a large collection of temperature readings over time. A suitable summary might be to keep the sum of all the temperatures seen, and the count. From this summary given by two numbers, we can extract the average temperature. This summary is easy to update incrementally, and can also be combined with a corresponding summary of different data by computing the overall sum and count. A different summary retains only the maximum and minimum temperature observed so far. From this, we can extract the range of temperatures observed. This too is straightforward to maintain under updates, and to merge across multiple subsets. However, neither summary is good at retrieving the median temperature, or some other properties of the statistical distribution of temperatures. Instead, more complex summaries and maintenance procedures are required.

This work aims to describe and explain the summaries that have been developed to deal with big data, and to compare summaries for similar goals in terms of the forms of data that they accept, and their flexibility of use. It follows a fairly technical approach, describing each summary in turn. It lists the type of data that can be summarized, and what operations can be performed on the summary to include more data in it, and to extract information about the summarized data. We assume some familiarity with mathematical and computer science concepts, but provide some necessary background in subsequent sections.

1.2 Preliminaries

This section lays down some of the basics of working with summaries: the kinds of data that they can take as inputs; the operations that may be performed on the summaries during their use; and the types of guarantees they provide over their output.

1.2.1 Data Models

In this volume, we focus on data sets that arise from the aggregation of many small pieces of data. That is, the challenge arises from the scale of billions or trillions of simple observations. This matches the motivating applications described above: high frequency sensor readings, social network activities, transactions and so on all have a moderate number of different types, but potentially huge quantities of each type. The summaries we describe will operate on a large number of "tuples" of a common type, which collectively describe a complex whole.

The types of data we consider are therefore each quite simple, and it is their scale that presents the challenge for summarization. We describe the types of data in somewhat abstract terms, with the understanding that these can be mapped onto the specific applications when needed.

Set Data. The simplest form of data we consider is a set of items. That is, the input forms a set A, as a subset of some universe of possible items U. For example, U could be the set of 64-bit integers (denoting, perhaps, serial numbers of items), and each item x in the data is then some particular 64-bit integer.

A very basic summary over set data is a random sample. A random sample is a quite general purpose summary in the sense that it is useful for answering many possible questions about the underlying set A, although the accuracy may not be satisfactory. For example, a basic query that we may wish to pose on a set A is whether a particular item x is present in A, i.e., a *membership* query; or, for two sets A and B, how similar (the notion will be made more precise later) they are. Random samples can be used in place of the full data sets for answering these queries, but clearly will frequently make errors. The majority of the work on data summarization is thus devoted to constructing summaries targeted at certain specific queries, usually with (much) better accuracies than random samples.

Problems on sets often get more challenging if the same item may be

fed into the summary multiple times, while A is still considered as a set, i.e., duplicates should be removed. In this case, even counting the cardinality of A becomes nontrivial, if we do not want the summary to store every distinct input item.

Multiset Data. With set data, we typically assume the semantics that an item is either present or absent from the set. Under the multiset semantics, each item has a multiplicity. That is, we count the number of occurrences of each item. Again, the input is supported over a set U. Now, queries of interest relate to the multiplicity of items: how many occurrences of x are there in the data? which items x occur most frequently?

It is sometimes convenient to think of multiset data as defining a vector of values, v. Then v_x denotes the multiplicity of item x in the input. Natural queries over vectors include asking for the (Euclidean) norm of the vector, the distance between a pair of vectors, or the inner-product between two vectors. The accuracy of such estimators is often expressed in terms of the ℓ_p norm of the vector, $\|v\|_p$, where

$$\|v\|_p = \left(\sum_{i \in U} |v_i|^p \right)^{1/p}$$

Important special cases include the Euclidean norm, $\|v\|_2$, and the Manhattan norm, $\|v\|_1$ (the sum of absolute values). We may also abuse notation and make reference to the ℓ_0 norm, sometimes called the Hamming norm, which is defined as $\|v\|_0 = |\{i : v_i \neq 0\}|$. This counts the number of non-zero entries in the vector v, i.e., the number of *distinct* items in the multiset. When dealing with skewed data, that is, where a few items have much larger count than others, we sometimes give bounds in terms of the residual ℓ_p norm. This is denoted as $\|v\|_p^{\mathrm{res}(k)}$, where, if we re-index v so that v_i is the ith largest (absolute) value, then

$$\|v\|_p^{\mathrm{res}(k)} = \left(\sum_{i=k+1}^{|U|} |v_i|^p \right)^{1/p}.$$

That is, the ℓ_p norm after removing the k largest entries of v.

Weighted Multiset Data. More generally, input describing a multiset may arrive with corresponding weights. This can represent, for example, a customer buying several instances of the same item in a single

transaction. The multiplicity of the item across the whole input is the sum of all weights associated with it. The vector representation of the multiset naturally models the weighted case well, where v_i is the sum of weights of item i processed by the summary. The above queries all make sense over this style of input—to find the total weight for a given item, or the items with the largest total weights. Guarantees for summaries may be expressed in terms of vector norms such as $\|v\|_2$ or $\|v\|_1$. Different summaries can cope with different constraints on the weights: whether the weights should be integral, or can be arbitrary.

Of some concern is whether a summary allows *negative* weights. A negative weight corresponds to the removal of some copies of an item. Some summaries only tolerate non-negative weights (the positive weights case), while others allow arbitrary positive and negative weights (which we call the general weights case). Lastly, a few summaries work in the "strict" case, where positive and negative weights are permitted, provided that the final weight of every item is non-negative when the summary is interrogated. By contrast, in the general case, we allow the multiplicity of an item to be negative. For the positive weights and strict cases, guarantees may be given in terms of $W = \|v\|_1$, the sum of the weights. Some summaries have guarantees in terms of $W^{\mathrm{res}(k)} = \|v\|_1^{\mathrm{res}(k)}$, the weight of the input (in the positive weight or strict case) after removing the k heaviest weights.

Matrices. Going beyond vectors, we may have data that can be thought of as many different vectors. These can be naturally collected together as large matrices. We are typically interested in $n \times d$ matrices M where both n and d are considerably large. In some cases, one or other of n and d is not so large, in which case we have a "short fat matrix" or a "tall skinny matrix" respectively.

As with the vector case, the constraints on the data can affect what is possible. Are the entries in the matrix integer or real valued? Is each entry in the matrix seen once only, or subject to multiple additive updates? Are entries seen in any particular order (say, a row at time), or without any order? Guarantees may be given in terms of a variety of matrix norms, including entrywise norms, such as the Frobenius norm,

$$\|M\|_F = \sqrt{\sum_{i,j} M_{i,j}^2}$$

or the p-norm, taken over unit norm vectors x,

$$||M||_p = \sup_{||x||_p=1} ||Mx||_p$$

Ordered Data. When U has a total order—namely, given any two items, we can compare them and determine which is the greater and which is the lesser under the order—we can formulate additional queries. For example, *how many occurrences of items in a given range are there (range queries)?*; *what is the median of the input?* and more generally, *what does the data distribution look like on U?*

Some summaries manipulate items only by comparison, that is, given two items, checking whether one is greater or less than the other, or the two are equal. These summaries are said to be *comparison based*. They thus do not need to assume a fixed universe U beforehand, which is useful when dealing with, e.g., variable-length strings or user-defined data types.

Geometric Data. Multidimensional geometric data naturally arise in big data analytics. Any point on earth is characterized by latitude and longitude; a point in space has three coordinates. More importantly, many types of multidimensional data can be interpreted and analyzed geometrically, although they are not inherently geometric by nature. For example, we may see readings which include temperature, pressure, and humidity. In data mining, various features can be extracted from an object, which map it to a high-dimensional point.

Over such data, the summary may support range queries, which could generalize one-dimensional ranges in different ways such as axis-parallel rectangles, halfspaces, or simplexes. Moreover, one could ask for many interesting geometric properties to be preserved by the summary, for example, the diameter, the convex hull, the minimum enclosing ball, pairwise distances, and various clusterings.

Graph Data. Graph data is a different kind of multidimensional data, where each input item describes an edge in a graph. Typically, the set of possible nodes V is known upfront, and each edge is a member of $V \times V$. However, in some cases V is defined implicitly from the set of edges that arrive. Over graphs, typical queries supported by summaries may be to approximate the distance between a pair of nodes, determine the number of connected components in the graph, or count the number of a particular subgraph, such as counting the number of triangles.

1.2.2 Operations on Summaries

For uniformity of presentation, each summary we describe typically supports the same set of basic operations, although these have different meanings for each summary. These basic operations are INITIALIZE, UPDATE, MERGE and QUERY. Some summaries additionally have methods to CONSTRUCT and COMPRESS them.

INITIALIZE. The INITIALIZE operation for a summary is to initialize a new instance of the summary. Typically, this is quite simple, just creating empty data structures for the summary to use. For summaries that use randomization, this can also involve drawing the random values that will be used throughout the operation of the summary.

UPDATE. The UPDATE operation takes a new data item, and updates the summary to reflect this. The time to do this UPDATE should be quite fast, since we want to process a large input formed of many data items. Ideally, this should be faster than reading the whole summary. Since UPDATE takes a single item at a time, the summary can process a stream of items one at a time, and only retain the current state of the summary at each step.

Many summaries described in this book support not only adding a new item to the summary, but also deleting a previously inserted item. To maintain uniformity, we treat a deletion as an UPDATE operation with a negative multiplicity. Examples include the Count-Min Sketch (Section 3.4), Count Sketch (Section 3.5), and the AMS Sketch (Section 3.6). This usually follows from the fact the summary is a linear transformation of the multiplicity vector representing the input, and such summaries are often called *linear sketches*. This concept is discussed in more detail towards the end of the book (Section 9.3.4).

MERGE. When faced with a large amount of data to summarize, we would like to distribute the computation over multiple machines. Performing a sequence of UPDATE operations does not guarantee that we can parallelize the action of the summary, so we also need the ability to MERGE together a pair of summaries to obtain a summary of the union of their inputs. This is possible in the majority of cases, although a few summaries only provide an UPDATE operation and not a MERGE. MERGE is often a generalization of UPDATE: applying MERGE when one of the input summaries consists of just a single item usually reduces

to the UPDATE operation. In general a MERGE operation is slower than UPDATE, since it requires reading through both summaries in full.

QUERY. At various points we want to use the summary to learn something about the data that is summarized. We abstract this as QUERY, with the understanding that the meaning of QUERY depends on the summary: different summaries capture different properties of the data. In some cases, QUERY takes parameters, while for other summaries, there is a single QUERY operation. Some summaries can be used to answer several different types of query. In this presentation, we typically pick one primary question to answer with the QUERY operation, and then discuss the other ways in which the summary can be used.

CONSTRUCT. We can always construct a summary by adding items one by one into the summary using the UPDATE and MERGE operations. However, for a few summaries, UPDATE is expensive, complicated, or even impossible. In these cases, we will describe how to CONSTRUCT the summary from the given input in an offline setting.

COMPRESS. Some summaries also provide an additional operation which seeks to COMPRESS the data structure. This is the case when the effect of UPDATE and MERGE operations allow the size of the summary to grow. In this case, COMPRESS will aim to reduce the size of the summary as much as possible, while retaining an accurate representation. However, since the time cost for this operation may be higher than UPDATE, it is not performed with every UPDATE operation, but on a slower schedule, say after some number of UPDATE operations have been performed.

A Simple Example: Counts, Sums, Means, Variances. We give an illustration of how the operation above apply to the simple case of keeping counts. These give a first example of a summary allowing us to track the number of events that have been observed. Counters also easily allow us to track the sum of a sequence of weights; find their mean; and compute the observed variance/standard deviation.

We will illustrate the use of a counter c, and a sum of weights w, as well as a sum of squared weights s. The INITIALIZE operation sets all of these to zero. Given an update of an item i, with a possible weight w_i, we can UPDATE c by incrementing it: $c \leftarrow c + 1$. The sum of weights is updated as $w \leftarrow w + w_i$, and the sum of squared weights as $s \leftarrow s + w_i^2$. To MERGE together two counter summaries, we can simply sum the

corresponding values: the merge of c_1 and c_2 is $c_1 + c_2$, the merge of w_1 and w_2 is $w_1 + w_2$, and the merge of s_1 and s_2 is $s_1 + s_2$. We can apply different QUERY operations to obtain different aggregates: the total count of all the updates and the total sum of all the weights are simply the final values of c and w, respectively. The mean weight is given by w/c, and the variance of the weights is $s/w - (w/c)^2$.

1.2.3 Models of Computation

Traditionally, Computer Science has focused on the random access machine (RAM) model of computation to study algorithms and data structures. This abstraction is a good match for single-threaded computation on a single machine, but other models are required to fit computation on large volumes of data. The summaries that we describe are flexible, and can be implemented in a variety of different settings.

The Streaming Model. The streaming model of computation considers data which arrives as a massive sequence of discrete observations, which collectively describe the data. For example, we might think of the data as describing a vector, by giving a list of increments to entries in the vector (initially zero) in some arbitrary order. Since we require our summaries to support an UPDATE operation, we can usually make each piece of information about the data the subject of an UPDATE operation to build a summary of the whole data in the streaming model. This assumes that there is a single (centralized) observer; variants involving multiple, distributed observers can also be accommodated, as described below.

Parallel Processing. When data is observed in parallel, we can have each parallel thread perform UPDATE operations to build their own summaries of part of the data. Data can be assigned to each thread in some fashion: round-robin scheduling, or hash partitioning, for example. To collect all the observations, we can then MERGE together the summaries. Some extra effort may be needed to handle synchronization and coordination issues, which we assume would be taken care of within the parallel system.

Distributed Processing. Summaries can likewise be used in systems that handle data that is distributed over multiple machines. Multiple UPDATE operations can build a local summary, and these summaries

can then be combined with MERGE operations by a central entity to allow QUERY operations on the global summary. This can easily be implemented within various distributed frameworks, such as the MapReduce model within the Apache Hadoop and Spark systems.

1.2.4 Implementations of Summaries

In many cases, the summaries that we describe are relatively simple to implement. The pseudocode to outline each of the operations is often only a few lines long. Consequently, they can be implemented with relative ease from scratch. However, there are some subtleties, such as the use of suitable random hash functions, or the reliance on lower level data structures with efficient maintenance operations. It is therefore preferable to rely on pre-existing implementations for some or all of the summary functions.

Fortunately, there are many implementations and libraries freely available online, particularly for the most well-known summaries (such as BloomFilter). Inevitably, these are of varying quality and reliability, and adopt a number of different languages and coding styles. Throughout the main section of the book, we will make reference to the Apache DataSketches library as a main reference for implementations. This is a well-established project to provide flexible implementations of the most important summaries in Java. It includes several carefully engineered features, such as internal memory management to avoid overheads from the default heap management and garbage collection routines. The project was initiated within Yahoo!, then open sourced, and most recently transitioned to the Apache Software Foundation. The home for this project is `https://datasketches.github.io/`. After DataSketches, the stream-lib library (also in Java) also has many Java implementations of summaries, with multiple contributors (`https://github.com/addthis/stream-lib`).

1.2.5 Output Guarantees: Approximation and Randomization

Necessarily, any summary must lose fidelity in its description of the data. In many cases, we cannot expect a summary to answer every QUERY with perfect accuracy (unless the summary only supports a few fixed simple queries like sum and variance as just discussed). If this were the case, it may be possible to carefully choose a battery of queries so that we would be able to recover almost every detail of the original

input. This intuition can be formalized to prove strong lower bounds on the size of any summary which hopes to provide such strong guarantees. More detail on reasoning about the size of a summary to answer certain queries is given in Chapter 10.4.

Therefore, in order to provide a summary which is more compact than the original data, we must tolerate some loss of accuracy. There are two natural ways that this is formalized: approximation, and randomization. Most summaries we describe will include one or both of these.

Approximation. Often the answer to a QUERY is numerical. Rather than the exact answer, it is often sufficient to provide some approximation of the answer. A *relative error approximation* gives an answer that is guaranteed to be within a fixed fraction of the true answer. For example, a 2-approximation is guaranteed to be at most twice the true answer. An *additive approximation* provides an answer that is guaranteed to be within some fixed amount of the true answer (this amount may depend on other properties of the input, such as the total size of the input). For example, a summary might guarantee to approximate the fraction of N input items satisfying a particular condition, up to additive error $0.01 \cdot N$.

Often, the quality of the approximation is a tunable parameter, which affects the size of the summary, and the time to perform operations on it. In this case, we may express the quality of approximation in terms of a parameter ε. This may lead to a $(1 + \varepsilon)$ relative error approximation, or an ε additive error, where the size of the summary is then expressed as a function of ε.

Randomization. There are many cases where guaranteeing a correct answer requires a very large summary, but allowing a small probability of error means that we create a much smaller summary. This typically works by making some random choices during the operation of the summary, and providing some probabilistic analysis to show that the summary provides a correct answer sufficiently often. Typically, the quality of a randomized summary is expressed in terms of a parameter δ, with the understanding that the probability of the summary failing to provide a correct answer is δ. The space used by the summary, and the time to perform operations upon it, is then expressed in terms of δ.

For most summaries, it is possible to set δ to be very small, without significantly increasing the size of the summary. Then this guarantee

holds except with a vanishingly small probability, say 10^{-20}, comparable to the probability that there is a CPU error sometime during the processing of the data. Note that the probability analysis will depend only on the random choices made by the algorithm—there are no assumptions that the input data is "random" in any way.

Approximation and Randomization. In many cases, the summaries described adopt both randomization *and* approximation, based on parameters ε and δ. The interpretation of this guarantee is that "the summary provides a $(1 + \varepsilon)$ approximation, with probability at least $1 - \delta$". With probability δ, this approximation guarantee does not hold.

1.3 Summaries in Applications

In this section, we outline a few examples of data processing, and describe how summaries with certain properties might be able to help overcome the resource challenges. We refer to various different types of summaries that are discussed in detail in later chapters.

1.3.1 Data Center Monitoring

Consider a large data center, supporting millions of users who cause the execution of billions of processes. Each process consumes a variety of resources: CPU, bandwidth, memory, disk usage, etc. These processes are distributed over tens of thousands of machines, where each machine has many processors, and each processor has multiple cores. Each process may be placed over multiple cores throughout the center. The data center operators would like to build a 'dashboard' application which provides information on the overall behavior of the center. It should provide information on the processes that are consuming a large fraction of resources.

To exactly track the amount of resources used is itself a potentially costly operation. The total amount of information involved is non-trivial: for each thread on each core, we will keep at least tens of bytes, enough to identify the process and to record its resource consumption in multiple dimensions. Multiplied by billions of processes, this is tens to hundreds of gigabytes of state information. Storing this amount of information is no great challenge. However, communicating this level of data

potentially incurs an overhead. Suppose we wish to gather statistics every second. Then a simplistic approach could communicate a hundred gigabytes of data a second to a monitoring node. This requires substantial network bandwidth to support: approaching a terabit, if this data is to pass over a single link. This speed even taxes memory access times, which can comfortably cope with up to only ten gigabytes per second. Thus, implementing the simple exact tracking solution will require some amount of effort to parallelize and distribute the monitoring.

An alternative is to adopt a lightweight approximate approach. Here, we allow a little imprecision in the results in order to reduce the amount of information needed to be shipped around. This imprecision can easily be made comparable to the measurement error in tracking the results. For example, we can adopt a summary such as the Count-Min Sketch or the SpaceSaving structure to track resource usage accurate up to 0.01%. The summary can be bounded in size to around 100KB. We can build a summary of the activity on a single machine, and ship it up to an intermediate node in the network. This node can collect summaries from a large number of machines, and MERGE these together to obtain a single summary which combines the results from all the inputs. These merged summaries can be passed on to the monitor, which can further MERGE all received summaries to obtain a single 100KB summary of the whole network. From this compact summary, the processes with high resource usage can be easily extracted.

The communication costs of this approach are much reduced: if we have 10,000 machines in the data center, with 100 intermediate nodes, the maximum bandwidth required is just 10MB/s: each node receives 100 summaries of 100KB each, and combines these to a single summary; the monitor receives these 100 summaries. Other cost regimes can be achieved by organizing the data transfer in others ways, or adjusting other parameters of the data collection. The cost reductions are achieved due to the use of summaries, which prune away insignificant information, and because we are able to perform *in-network aggregation*: rather than wait to the end to reduce the information, we can use the MERGE property of summaries to combine them at intermediate stages.

Another advantage of using summaries in this setting is that it is easy to reason about their properties, and accuracy: we have the assurance that the size of the summary remains fixed no matter how many MERGE operations we perform, and that the accuracy guarantees remain cor-

respondingly fixed. While it would be possible to design implementations that apply pruning to the collection of exact counts, it would require some effort and analysis to understand the tradeoffs between amount of pruning and the resulting accuracy. By adopting summary techniques, this tradeoff is already well-understood.

1.3.2 Network Scanning Detection

Consider the operator of a large data network, over which a large amount of Internet traffic passes. Within this network, it is important to identify unusual or suspicious behavior, as this can be indicative of an attack or the spread of unwanted software (viruses, worms etc.). There has been much study of the signals that can be mined in the networking context to identify such activity. Here, we focus on a relatively simple case, of detecting port scan activity.

A port scan is when a single host tries to connect to a large number of different machines on different ports, in the hope of finding an open port, which can potentially be used to attack the machine. Although such scans may represent a large number of distinct connections, in terms of the total bandwidth or number of packets, they can represent a very small fraction, and so can be easily lost among the overall traffic. Simple techniques, such as sampling, may be unable to detect the presence of port scan activity. Keeping logs of the whole traffic for offline analysis is rarely practical: the information is huge, and arrives at a very high rate (terabytes per hour).

A first approach is to track for each active host on the network the number of distinct (IP address, port) combinations that it tries to connect to. When this becomes large, it is indicative that the host is performing a port scan. One way to do this is to make use of the BloomFilter data structure. We can keep one BloomFilter for each host, along with a counter initialized to zero. This allows to compactly store the set of (IP address, port) combinations that the host has connected to. For every connection that is seen, we can test whether it is already stored in the set: if not, then we add it to the set, and increase the counter. If we want to detect accurately when a host has made more than 1000 distinct connections, say, then a BloomFilter of size approximately 1KB will suffice. For cases where we see a moderate number of distinct hosts – say, a few million – then this approach will suffice, consuming only a few gigabytes of fast memory in total.

However, in cases where we have more limited resources to devote

to the monitoring process, and where there are a greater number of hosts active in the network, a more compact solution may be required. A more advanced approach is to combine types of summaries to obtain accurate identification of port scan activity. We would like to adopt a summary such as Count-Min Sketch or SpaceSaving, as in the previous example. However, these are good at identifying those items that have large absolute weights associated with them: this would find those hosts which use a large amount of bandwidth, which is distinct from port scanning. Rather, we would like these summaries to allow us to find those hosts that have a large amount of distinct connections. This can be accomplished by modifying the summaries: replacing the counters in the summaries with distinct counters.

Understanding the impact of combining two summaries is somewhat complex. However, this approach has been applied successfully in a number of cases [221, 154, 73], allowing efficient identification of all hosts that are responsible for more than a given fraction of distinct connections with a summary totalling only megabytes in size. Further discussion is given in Section 9.4.3.

1.3.3 Service Quality Management

Consider an organization that hosts a large number of services for many different customers. Each customer has a set of service level agreements (SLAs) that determine the level of service they are guaranteed by contract. The hosting organization needs to monitor the behavior of all services, to ensure that all the SLAs are met. Such agreements are typically of the form "95% of responses are made within 100ms". The organization would therefore like to track the adherence to these SLAs across its different customers. While exact tracking may be required in some cases, it is also helpful to allow lightweight approximate tracking to identify when there is a danger of not meeting these agreements, and to respond accordingly by adjusting parameters or deploying more resources.

For SLAs of this form, we need to be able to track the quantiles of the monitored quantity. That is, for the example above, given the series of response times, we need to identify what is the 95th percentile of this series, and how it compares to 100ms. We can maintain a list of all response times in sorted order, and periodically probe this list to check this quantile. However, this requires a lot of storage, and can be slow to update as the list grows.

Instead, we can make use of summaries which support quantile queries. Example summaries include the GK and Q-Digest summaries. These have differing properties. Q-Digest works when the number of possible values is bounded. So, if we have response times measured to microsecond accuracy, then it is suitable to use Q-Digest. On the other hand, if we have a very large range of possible values, GK can be used. The space of Q-Digest remains bounded, no matter how many items are summarized by the summary, or how many times we perform MERGE operations to combine different summaries. Meanwhile, the GK summary may grow (logarithmically) with the size of its input in the worst case.

In other situations, we might have additional requirements for the monitoring. For example, we might want to track not just the full history of response times, but rather a moving window of response times: what is the 95th percentile of the responses in the last hour. A crude approach is to start a fresh summary periodically – say, every five minutes – and to maintain multiple summaries in parallel. This imposes a greater overhead on the monitoring process. A more refined approach is to partition time into buckets – say, five minute intervals – and track a summary for each bucket separately. Then we can MERGE the summaries for multiple such buckets to get the summary for a recent window. However, this still only approximates the desired window size. More complex solutions can be used to generate a summary for any window size that gives a stronger guarantee of accuracy – see Section 9.2.2.

1.3.4 Query Optimization

In database management systems, data is organized into relations, where each relation has a number of fields. Queries perform operations on relations, such as selecting records with particular field values, joining relations based on matching values, and applying aggregates such as count and sum. For most non-trivial queries, there are multiple ways to perform a query, and the system wants to pick the one with the lowest (estimated) cost. Here, the 'cost' of a query execution plan may be the time taken to perform it, the amount of disk access, or other measure.

Summary techniques have been adopted by many different database systems to allow approximation of the cost of different plans, and hence the selection of one that is believed to be cheap. Most commonly, basic summaries, such as a count and sum of numeric values, are main-

tained. For low cardinality attributes (ones taking on a small number of different values), it is natural to keep a count of the frequency of each value. For attributes with more possible values, a RandomSample of items is kept for each field in a relation, to allow simple statistics to be estimated. A common basic question is to estimate the *selectivity* of a particular predicate – that is, to estimate how many records in a relation satisfy a particular property, such as being equal to a particular constant, being less than some value, or falling in some range. A RandomSample is a natural way to accurately estimate these values; more sophisticated schemes take account of weights associated with items, such as WeightedRandomSample.

More recently, database systems have adopted more complex summary methods. To summarize numeric attributes, a histogram describing how it is distributed can be useful, and the most common type of histogram is an *equi-depth histogram*, where the bucket boundaries are quantiles of the item distribution. Some systems may simply recompute the quantiles of the field periodically (either exactly, or by making a summary of the current values); others may maintain a summary as records are added and deleted to the relation. Other statistics on the values in the relation, such as the number of distinct values, and estimations of the join size between two relations, may be maintained using appropriate summaries.

1.3.5 Ad Impression Monitoring and Audience Analysis

Online advertising has made it possible for advertisers to obtain more detailed information about who has seen their adverts, in comparison to traditional broadcast media. Each day, billions of adverts are shown to users of the web and apps by ad publishers, and tracking these "impressions" presents a substantial data management challenge. The same ad may be shown to the same user multiple times, but which should only be counted once (or the count capped). Advertisers also want to know which demographic have seen their ad – females aged 18-35 working in white collar jobs with a university level education, say. Current advertising networks have reasonably accurate profiles of web users based on information gathered and inferred about them. But allowing questions about different ads and different demographics in real time stretches the ability of large scale data management systems.

There has been considerable success in using summaries to answer these kind of queries. This is a situation where an approximate answer

is acceptable – advertisers want to know with reasonable precision how many have seen their advert, but the numbers are large enough that error of 1% or so can be tolerated. For the basic question of tracking the number of different people who have seen an advert, methods such as the KMV and HLL summaries answer this directly and effectively, at the cost of a few kilobytes per advert. Very small space for this tracking is important when millions of ads may be in rotation from a single publisher.

A simple way to deal with queries that ask for different subpopulations is to keep a summary for each combination of possible demographic group and ad. This quickly becomes unscalable as the number of demographic groups grows exponentially with the number of features stored on each user. The question to be answered is, for each ad, to look at the different demographics of viewers (male/female, age group, education level and so on) and to find the cardinality of the intersection of the desired sets – the female, 18-35 year old, university educated viewers of the above example. It turns out that it is possible to estimate the size of these intersections from the aforementioned KMV and HLL summaries. That is, given a summary for the number of distinct female viewers and another for the number of distinct university educated viewers, we can combine these to obtain an estimate for the number of female, university educated viewers. The accuracy of these intersection estimates can degrade quickly, particularly when the size of the intersection is small compared to the total number of distinct views of the ad. Nevertheless, the results can be effective, and have formed the technical underpinning of a number of businesses formed around this question. The benefits are that arbitrary questions can be answered almost instantaneously from combining a number of small summaries. This is dramatically faster than any database system that has to rescan the raw view data, even if indexes are available and preprocessing is done on the data.

1.4 Computational and Mathematical tools

Throughout, we assume familiarity with basic computational and mathematical tools, such as the big-Oh ($O(\cdot)$) notation to express the asymptotic growth behavior of time and space bounds. We also assume familiarity with standard data structures, such as heaps, queues and lists. The description of properties of summaries makes use of standard prob-

abilistic analysis and tail bounds. For convenience, we list the forms of the tail bounds that we make use of (for more details, see the standard randomized algorithms texts such as [184, 178]).

Fact 1.1 (Markov Inequality) *Given a non-negative random variable X with expectation $\mathsf{E}[X]$, we have*

$$\Pr[X > k\mathsf{E}[X]] \leq \frac{1}{k}$$

Further Discussion. Throughout this book, we avoid requiring proofs to be read in order to understand the ideas introduced. For those wanting to understand more details, we provide optional material, marked out from the rest of the text. We begin this optional material with a simple proof of the Markov inequality.

The Markov inequality can be proved using some basic rules of probability. Consider the event that $X > c$ for some constant $c > 0$. For any value of x, we have $cI(x \geq c) < x$, where $I(b)$ is 1 if b evaluates to true, and 0 otherwise. This can be checked by considering the two cases $x \geq c$ and $x < c$. We can apply this to the variable X to get $cI(X \geq c) < X$, and take the expectation of both sides:

$$c\mathsf{E}[I(X \geq c)] = c\Pr[X \geq c] < \mathsf{E}[X].$$

Rearranging, and substituting $c = k\mathsf{E}[X]$, we obtain the inequality in the quoted form.

The variance of X is

$$\mathsf{Var}[X] = \mathsf{E}[(X - \mathsf{E}[X])^2] = \mathsf{E}[X^2] - (\mathsf{E}[X])^2,$$

while the covariance of two random variables X and Y is

$$\mathsf{Cov}[X, Y] = \mathsf{E}[XY] - \mathsf{E}[X]\mathsf{E}[Y]$$

The variance satisfies several properties that we make use of:

Fact 1.2 (Properties of variance) *Given a random variable X and constants a, b, we have*

$$\mathsf{Var}[aX + b] = a^2\mathsf{Var}[X]$$

Given n random variables X_i, we have

$$\mathsf{Var}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathsf{Var}[X_i] + \sum_{1 \le i < j \le n} \mathsf{Cov}[X_i, X_j]$$

When the n random variables X_i are uncorrelated *(have zero covariance), we have*

$$\mathsf{Var}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathsf{Var}[X_i]$$

Applying the Markov inequality to the variance of X, we obtain the Chebyshev inequality:

Fact 1.3 (Chebyshev inequality) *Given any random variable X, we have*

$$\Pr[|X - \mathsf{E}[X]| > k] \le \frac{\mathsf{Var}[X]}{k^2}$$

Chernoff bounds arise from applying the Markov inequality to exponential functions of variables. We use two forms of Chernoff bounds, the (additive) Chernoff-Hoeffding bound, and the relative Chernoff bound.

Fact 1.4 (Additive Chernoff-Hoeffding bound) *Given n independent random variables $X_1 \ldots X_n$ such that there are bounds $a_i \le X_i \le b_i$ for each X_i, we write $X = \sum_{i=1}^{n} X_i$. Then*

$$\Pr[|X - \mathsf{E}[X]| > k] \le 2 \exp\left(\frac{-2k^2}{\sum_{i=1}^{n}(b_i - a_i)^2}\right)$$

A Bernoulli random variable X with a single parameter p is such that $\Pr[X = 1] = p, \Pr[X = 0] = (1 - p)$. Hence, $\mathsf{E}[X] = p$.

Fact 1.5 (Multiplicative Chernoff bound) *Given independent Bernoulli random variables $X_1 \ldots X_n$, such that $X = \sum_{i=1}^{n} X_i$ and $\mathsf{E}[X] = \sum_{i=1}^{n} \mathsf{E}[X_i] = \mu$, then, for $0 < \beta \le 1$, and $0 \le \rho \le 4$,*

$$\Pr[X \le (1 - \beta)\mu] \le \exp\left(\frac{-\beta^2 \mu}{2}\right)$$

$$\Pr[X \ge (1 + \rho)\mu] \le \exp\left(\frac{-\rho^2 \mu}{4}\right)$$

Further Discussion. We do not provide detailed proofs of all Chernoff bounds, but for a flavour, we describe one case to show how

the proof builds on basic ideas such as the Markov inequality above. Let $\Pr[X_i = 1] = p_i$ so that $\mathsf{E}[X] = \sum_{i=1}^{n} p_i = \mu$. We seek to bound $\Pr[X > (1+\rho)\mu]$. We introduce a (positive) parameter t, and apply an exponential function to both sides of the inequality. This does not change the probability, so

$$\Pr[X > (1+\rho)\mu] = \Pr[\exp(tX) > \exp(t(1+\rho)\mu)]$$

By the Markov inequality, we have

$$\Pr[\exp(tX) > \exp(t(1+\rho)\mu)] \leq \mathsf{E}[\exp(tX)]/\exp(t(1+\rho)\mu) \qquad (1.1)$$

The rest of the proof aims to simplify the form of this expression. Observe that, from the definition of X and by the independence of the X_is,

$$\mathsf{E}[\exp(tX)] = \mathsf{E}[\exp(t\sum_{i=1}^{n} X_i)] = \prod_{i=1}^{n} \mathsf{E}[\exp(tX_i)]$$

The expectation of $\exp(tX_i)$ is a summation of two cases: X_i is zero with probability $1 - p_i$, giving a contribution of $\exp(0) = 1$; or X_i is one with probability p_i, giving a contribution of $\exp(t)$. Thus,

$$\prod_{i=1}^{n} \mathsf{E}[\exp(tX_i)] = \prod_{i=1}^{n}((1-p_i) + p_i e^t)$$

Using the usual expansion of the exponential function and the fact that $t > 0$,

$$\exp(p_i(e^t - 1)) = 1 + (p_i(e^t - 1)) + \ldots > 1 - p_i + p_i e^t$$

so we can write

$$\prod_{i=1}^{n}(1-p_i+p_i e^t) \leq \prod_{i=1}^{n}\exp(p_i(e^t-1)) = \exp(\sum_{i=1}^{n} p_i(e^t-1)) = \exp(\mu(e^t-1)$$

Substituting this back into (1.1), we obtain

$$\Pr[X > (1+\rho)\mu] \leq \exp(\mu(e^t-1)-\mu t(1+\rho)) \leq \exp(\mu(-\rho t+t^2/2+t^3/6+\ldots))$$

At this point, we can choose the value of t to give the final form of the bound. In this case, we can pick $t = \frac{2}{5}\rho$. One can verify that for this choice of t in the range $0 \leq \rho < 4$, we have $\exp(\mu(-\rho t + t^2/2 + t^3/6 + \ldots)) < \exp(\rho^2\mu/4)$.

Last, we sometimes make use of a simple bound, which allows us to reason about the probability of any one of multiple events.

Fact 1.6 (Union bound)

$$\Pr[A \cup B] \leq \Pr[A] + \Pr[B]$$

Note that we do not require the events A and B to be independent. This fact follows immediately from the fact that $\Pr[A \cup B] = \Pr[A] + \Pr[B] - \Pr[A \cap B]$. We often use this fact to argue about the probability of success of an algorithm that relies on many events being true. That is, if there are n "bad events", $B_1 \ldots B_n$, but each one is very unlikely, we can argue that the probability of *any* bad event happening is at most $\Pr[\cup_{i=1}^{n} B_i] \leq \sum_{i=1}^{n} \Pr[B_i]$, by repeated application of Fact 1.6. If each $\Pr[B_i]$ is sufficiently small – say, $\Pr[B_i] = 1/n^2$, then the probability of any of them happening is still very small, in this case, at most $\sum_{i=1}^{n} 1/n^2 = 1/n$.

Another idea often used in arguing for the correctness of a randomized algorithm is the *principle of deferred decisions*. A simple application of this principle, as used in Section 2.2, is to sample an item uniformly at random from two sets S_1 and S_2, of sizes n_1 and n_2, respectively. The direct way of doing is to simply sample an item from all the $n_1 + n_2$ items uniformly at random. However, we could also do it in two steps: We first decide which one of the two sets to choose the sample from: S_1 should be picked with probability $\frac{n_1}{n_1+n_2}$ while S_2 picked with probability $\frac{n_2}{n_1+n_2}$. In the second step, which the algorithm may choose to do at a later time, is to pick an item from the chose set uniformly at random. The correctness of this principle follows easily from the fact that, for any two events A and B,

$$\Pr[A \cap B] = \Pr[A] \Pr[B \mid A].$$

1.4.1 A Chernoff bounds argument

A standard application of the Chernoff bound is to take multiple estimates of a quantity, and combine them to pick an estimate which is good with high probability. Specifically, we have estimates, each of which is a "good" estimate of a desired quantity with at least a constant probability. However, it's not possible to tell whether or not an estimate is good just by looking at it. The goal is to combine all these to make an estimate which is "good" with high probability. The approach is to take the *median* of enough estimates to reduce the error. Although it is not possible to determine which estimates are good or bad, sorting the estimates by value will place all the "good" estimates together in the middle of the sorted order, with "bad" estimates above and below

(too low or too high). Then the only way that the median estimate can be bad is if more than half of the estimates are bad, which is unlikely. In fact, the probability of returning a bad estimate is now exponentially small in the number of estimates.

The proof makes use of the Chernoff bound from Fact 1.5. Assume that each estimate is good with probability at least 7/8. The outcome of each estimate is an independent random event, so in expectation only 1/8 of the estimates are bad. Thus the final result is only bad if the number of bad events exceeds its expectation by a factor of 4. Set the number of estimates to be $4\ln(1/\delta)$ for some desired small probability δ. Since whether each estimate is 'good' or 'bad' can be modeled by a Bernoulli random variable with expectation 1/8, then this setting is modeled with $\rho = 3$ and $\mathsf{E}[X] = \frac{1}{2}\ln(1/\delta)$. Hence,

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

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

1.4.2 Hash Functions

Many summaries make use of *hash functions* , which are functions picked at random from a family $\mathcal{F}$ containing many possible hash functions, where each maps onto some range R. In order to provide the guarantees on the summary, we typically require that the family of functions satisfies some properties. The most common property is that the family is t-wise independent. This means that (over the random choice of the hash function), the probability of seeing any t values appears uniform: given any t distinct items $x_1, \ldots x_t$, and any t values in the output of the function $y_1, \ldots y_t \in R^t$, we have

$$\Pr_{f \in \mathcal{F}}[f(x_1) = y_1, f(x_2) = y_2, f(x_t) = y_t] = \frac{1}{|R|^t}$$

A simple family of functions that meets this requirement[1] is the family of polynomials,

[1] Strictly speaking, the above probability for this family can be very slightly larger than $\frac{1}{|R|^t}$, but this does not affect the analysis or practical use in any significant way.

$$\mathcal{F} = \left\{ \sum_{i=1}^{t} c_i x^{i-1} \mod p \mod |R| \right\}$$

where p is a (fixed) prime number and c_i range over all (non-zero) values modulo p. Thus, to draw a function from this family, we simply pick the t coefficients c_1 to c_t uniformly from the range $1 \ldots p-1$. Further discussion and code for efficient implementations of these functions is given by Thorup and Zhang [212, 214].

The reason for this analysis of the degree of independence of hash functions is to quantify the "strength" of the functions that is required. In many cases, it is shown that summaries require only 2-wise (pairwise) or 4-wise independent hash functions. In other cases, the analysis makes the assumption that the hash functions are "fully independent" for convenience of analysis, even though this assumption is technically unrealistic: hash functions which guarantee to act independently over very many items require a lot of space to represent. In practice, functions without this strict guarantee are used, without any reported problem.

Where fully independent hash functions are needed, some widely adopted hash function (without full independence) is typically used. *Cryptographic* hash functions are ones that are designed to provide hash values that are very difficulty to invert: given the hash value, it should be impossible to find what input it was applied to, short of trying all possibilities. Well-known examples include MD5, SHA-1 and SHA-3. However, such functions tend to be much slower than pairwise independent hash functions, as they apply multiple rounds of expensive operations to their inputs in order to mask the contents. This can be a bottleneck in high-performance systems. Instead, *non-cryptographic* hash functions may be most suitable. These do not aim to provide the level of non-invertibility of the cryptographic counterparts; rather they seek to ensure that the output appears random given the input. They are constructed based on combining parts of their input through fast operations (such as exclusive-or and bit shifts) with carefully chosen constants. A popular example is the *murmurhash* function[2] which can be implemented using a small number of fast low-level bit-manipulation operations, and has been found to be very effective for applications such as those discussed here.

[2] `https://github.com/aappleby/smhasher`

As a rough guide, the simplest pairwise independent hash functions are the fastest, and can be evaluated many hundreds of millions of times per second on a single processor core. Four-wise independence is about half the speed of pairwise. Murmurhash is of comparable speed, while SHA-1 and MD5 are about four times slower. Some example numbers from experiments on a commodity CPU indicate that pairwise hash functions can easily process in excess of 500 million 32-bit keys in 1 second, while fourwise hash functions can manage 250 million, murmurhash 200 million, and SHA-1/MD5 around 50 million. This indicates that cryptographic hash functions can be 10 times slower than simple pairwise hashing.

1.5 Organization of the Book

This book is primarily intended as an introduction and reference for both researchers and practitioners whenever the need of some kind of data summarization arises. It is broken into two main parts. The first part of the book introduces the principle examples of the algorithms and data structures which form the summaries of interest. For each summary described in detail in the main text, we provide description under several standard headings. We first provide a **Brief Summary** of the summary to describe its general properties and operation. The bulk of the description is often in describing the details of the **Operations on the summary** (such as INITIALIZE, UPDATE and QUERY). These are illustrated with an **Example** and described in pseudocode where appropriate, to give a more formal outline of the key operations. Sometimes **Implementation Issues** and **Available Implementations** are also discussed. More detailed analysis, including technical explanations of summary properties, is given under **Further Discussion**, with the intention that the reader can achieve a basic understanding and be able to use the summary without reading this part. Finally, some **History and Background** is outlined with links to further reading and alternative approaches.

The second part of the book looks at more complex summary techniques for specific kinds of data (such as geometric, graph or matrix data), which combine or build upon the fundamental summaries. It also describes how to modify summary techniques to accommodate weights and time-decay. Here, we relax the constraints on the presentation, and provide more general high-level descriptions of the sum-

maries, and their applications. We also discuss lower bounds, which provide limitations on what it is possible to summarize within a certain amount of memory space.