

Hash-Based Data Structures for Extreme Conditions

A dissertation presented

by

Adam Lavitt Kirsch

to

The School of Engineering and Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

May 2008

©2008 - Adam Lavitt Kirsch

All rights reserved.

Hash-Based Data Structures for Extreme Conditions

Abstract

This thesis is about the design and analysis of Bloom filter and multiple choice hash table variants for application settings with extreme resource requirements. We employ a very flexible methodology, combining theoretical, numerical, and empirical techniques to obtain constructions that are both analyzable and practical.

First, we show that a wide class of Bloom filter variants can be effectively implemented using very easily computable combinations of only two fully random hash functions. From a theoretical perspective, these results show that Bloom filters and related data structures can often be substantially derandomized with essentially no loss in performance. From a practical perspective, this derandomization allows for a significant speedup in certain query intensive applications.

The rest of this work focuses on designing space-efficient, open-addressed, multiple choice hash tables for implementation in high-performance router hardware. Using multiple hash functions conserves space, but requires every hash table operation to consider multiple hash buckets, forcing a tradeoff between the slow speed of examining these buckets serially and the hardware complications of parallel examinations. Improving on previous constructions, we show that a small Bloom filter-based data structure in fast memory can essentially allow us to use multiple hash functions while only examining a single bucket during a hash table operation.

For scenarios where we can afford the parallelization above, the space utilization of standard multiple choice hash table constructions can be improved by allowing items to be moved within the hash table after they are initially inserted. While there are a number of known hash table constructions with this property, the worst case insertion times are too large for the applications we consider. To address this problem, we introduce and analyze a wide variety of hash table constructions that move at most one item in the during the insertion of a new item. Using differential equation approximations and numerical methods, we are able to quantify the performance of our schemes tightly and show that they are superior to standard constructions that do not allow moves.

Contents

Title Page	i
Abstract	iii
Table of Contents	iv
Citations to Previously Published Work	vi
Acknowledgments	vii
Dedication	viii
1 Introduction	1
2 Background	4
2.1 Hash-Based Data Structures	4
2.1.1 Hash Functions	4
2.1.2 Bloom Filters	6
2.1.3 Multiple Choice Hash Tables	9
2.2 Hash Tables and Routers	10
2.2.1 Hardware Design Issues	11
2.2.2 Applications of Hash Tables	12
3 Faster Bloom Filters Through Double Hashing	15
3.1 Introduction	15
3.2 A Simple Construction Using Two Hash Functions	17
3.3 A General Framework	19
3.4 Some Specific Hashing Schemes	25
3.4.1 Partition Schemes	25
3.4.2 (Enhanced) Double Hashing Schemes	26
3.5 Rate of Convergence	28
3.6 Multiple Queries	33
3.7 Experiments	39
3.8 A Modified Count-Min Sketch	42
3.8.1 Count-Min Sketch Review	42
3.8.2 Using Fewer Hash Functions	42
3.9 Conclusion	45

4	Simple Summaries for Hashing with Choices	47
4.1	Introduction	47
4.2	Related Work	49
4.3	The Scheme of Song <i>et al.</i> [56]	49
4.4	Separating Hash Tables and Their Summaries	50
4.5	The Multilevel Hash Table (MHT)	51
4.5.1	Approximate and Exact Calculations for MHTs	52
4.5.2	The MHT Skew Property	53
4.6	An Interpolation Search Summary	54
4.7	Bloom Filter-Based MHT Summaries	56
4.7.1	A Natural First Attempt	56
4.7.2	On Skew, and an Improved Construction	58
4.8	Numerical Evaluation (Insertions Only)	59
4.9	Experimental Validation (Insertions Only)	62
4.10	Deletions	62
4.10.1	Lazy Deletions	62
4.10.2	Counter-Based Deletion Schemes	66
4.11	Additional Technical Details	67
4.11.1	An Asymptotic Bound on the Crisis Probability	68
4.11.2	Asymptotics of the Bloom Filter Summaries	70
4.11.3	Calculating Various Quantities of Interest	71
4.12	Conclusions and Further Work	73
5	The Power of One Move: Hashing Schemes for Hardware	75
5.1	Introduction	75
5.2	The Standard Multilevel Hash Table (MHT), Revisited	76
5.3	Hashing Schemes and Differential Equations	78
5.4	Allowing One Move: A Conservative Scheme	79
5.5	The Second Chance Scheme	80
5.6	The Extreme Second Chance Scheme	82
5.7	Multiple Items Per Bucket	84
5.8	Evaluation (Insertions Only)	86
5.9	Deletions	92
5.10	Evaluation (Insertions and Deletions)	97
5.11	Conclusion	101
	Bibliography	103

Citations to Previously Published Work

All of the work in this thesis is in the process of being published in traditional academic formats and venues. Specifically, the material in Chapter 3 either has been or is scheduled to appear in the following:

A. Kirsch and M. Mitzenmacher. Less Hashing, Same Performance: Building a Better Bloom Filter. *Random Structures and Algorithms*, to appear.

A. Kirsch and M. Mitzenmacher. Less Hashing, Same Performance: Building a Better Bloom Filter. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA)*, pp. 456-467, 2006.

A. Kirsch and M. Mitzenmacher. Building a Better Bloom Filter. Harvard University Computer Science Technical Report TR-02-05, 2005.

Similarly, the material in Chapter 4 appears in:

A. Kirsch and M. Mitzenmacher. Simple Summaries for Hashing with Choices. *IEEE/ACM Transactions on Networking*, 16(1):218-231.

A. Kirsch and M. Mitzenmacher. Simple Summaries for Hashing with Multiple Choices. In *Proceedings of the Forty-Third Annual Allerton Conference on Communication, Control, and Computing*, 2005.

Finally, the material in Chapter 5 appears in:

A. Kirsch and M. Mitzenmacher. The Power of One Move: Hashing Schemes for Hardware. In *Proceedings of the 27th IEEE International Conference on Computer Communications (INFOCOM)*, 2008.

Acknowledgments

I could not have completed this thesis without the substantial personal and professional support I received from my mentors, colleagues, friends, and family. First of all, I thank my advisor, Michael Mitzenmacher, for teaching me through his vision, his words of encouragement, his consideration of every one of the countless emails I continue to send him (no matter how misguided they are), and the candor, pragmatism, and soundness of his advice. I am also very grateful to Eli Upfal, who has been a mentor to me since my undergraduate days, for introducing me to research in general and theoretical computer science in particular, for recommending me to Michael for my graduate studies, and for seeing in me a potential that took longer for me to see in myself. I also thank Andrei Broder and Ravi Kumar for their supervision during my summer internship at Yahoo! and the resulting collaborations that they fostered. On a more scholarly note, I thank Peter C. Dillinger and Panagiotis Manolios for introducing Michael and me to the subject of Chapter 3, as well as Salil Vadhan and Matt Welsh for serving on my dissertation committee, cheerfully accommodating the bizarre scheduling problems surrounding my defense, and giving me valuable comments on my work. I also give my gratitude to Susan Wiczorek for her almost magical ability and willingness to, with a smile and a laugh, make even the most arcane of Harvard's bureaucratic oddities seem totally transparent.

More personally, I doubt I could have survived graduate school without my absolutely awesome colleagues and friends. In particular, over the years I have been fortunate to share my office with Yan-Cheng Chang, Johnny Chen, Kai-Min Chung, Eleni Drinea, Vitaly Feldman, Alex Healy, Minh Huyen-Nguyen, Shaili Jain, Zhenming Liu, Loizos Michael, Shien Jin Ong, and Emanuele Viola. I give special thanks to Eleni (as promised a long time ago) for tolerating my chattiness during an otherwise lonely summer of 2005 in MD138, when I probably should have just let her write her dissertation in peace. Outside Harvard, I thank Daniel Byerly, Katharine Clark, Christopher Esposito, Idan and Tiferet Gazit, James Hollyer, Jack Menzel, and Tyler Wellman for their friendships and, let's face it, helping to keep me sane.

I also thank my parents, George and Susan, for (literally) making all of my achievements possible. Much more than that, I am eternally grateful to them for their unconditional love, support, and faith in me. I am continually amazed and honored by how much of themselves they give to help me, despite the hardships they face in their own lives. They are my role models, and for that I dedicate this thesis to them.

Finally, I thank my greatest friend of all, my fiancée Ana DiRago. Her love for me has and continues to sustain me through some very trying times, and I simply cannot thank her enough.

For my parents.

Chapter 1

Introduction

This thesis demonstrates that it is possible to design suitable hash-based data structures (specifically, variants of Bloom filters and multiple choice hash tables) for a variety of practical applications with extreme resource requirements. Our notion of *extreme* is a general one, incorporating settings where there is at least one critical resource that must be conserved in such a way as to make general purpose data structures unacceptable. In particular, the long term goal of this (and continuing) work is to directly influence the implementation of real-world systems through the judicious use of theoretical, numerical, and empirical methods. Thus, not only do we strive to make technical contributions for the specific problems that we analyze, but more general methodological contributions as well. Indeed, there is a large amount of room for different approaches in analyzing applications of hashing, which is not surprising given its widespread use by both theoreticians and practitioners.

For instance, the most ubiquitous hash-based data structure for set membership is almost certainly a standard hash table with chaining and one hash function. This sort of implementation is now essentially standard in most data structure libraries (e.g., the STL for C++ and the `java.util` package for Java). Furthermore, it is telling that this is probably the first form of hashing ever invented [31]. For our purposes, even a cursory look at this example serves as an extraordinary inspiration. Indeed, under some simple, albeit naive, modeling assumptions on the hash function (e.g., that it is fully random), one can easily make a compelling theoretical argument for why such a data structure should perform well. With some significant additional insight, both theoretical and empirical, into how to design good hash functions, it is possible to reliably achieve these results in a wide range of real world settings. The result is a data structure that is so widely used that it is often taken for granted.

Nevertheless, there are many situations where the above and other standard approaches to hashing leave much to be desired, even to the point of being impractical. For example, the standard approaches to resolving collisions in a hash table (such as using chaining or linear probing) can result in occasional long lookup times that may be unsuitable for an application where amortized guarantees alone are not sufficient. Such issues arise frequently in the setting of high-performance routers, which is the application on which we concentrate in most of this work. We focus on these sorts of extreme but increasingly

important scenarios, where very efficient use of all available computational (and possibly even energy) resources is paramount.

In these cases, the simplicity of a data structure naturally becomes critical. Paradoxically, even as we strive to simplify our data structures, their evaluation becomes increasingly complicated. Indeed, theoretical results that obscure constant factors are not very compelling, since these constants can, and often do, dominate the comparison of various methods. To compound the problem further, strong empirical validation becomes even harder to obtain. After all, in the absence of a precise theorem or numerical technique, there is often no way to verify whether an undesirable, improbable event is truly negligible without a direct simulation, which is likely impractical. For a simple hash-based data structure, the entire functionality may essentially reduce to the occurrence of a particular set of hash collisions, and so to compare methods in practice, it becomes necessary to directly compare the probabilities of the corresponding intuitively negligible events. To emphasize the difficulty of this issue, we note that this is strikingly similar to the underlying problem for the entire field of large deviations theory: accurately estimating the probability of extremely rare events.

All is not lost, however. While the challenges outlined above are pervasive in this and related work, they can often be dealt with through a clever choice of the proposed data structure and the corresponding methodology for evaluation. In particular, we are very flexible in the sort of performance guarantees that we seek. Where exact, informative theorems seem possible, we certainly look for them. But if that approach does not seem viable, then we actively search for other approaches, often combining a mixture of heuristic approximations and numerical methods. Overall, we look for *any* approach with the potential to accurately and precisely predict and explain the behavior of a real-world system, and we motivate all of our methods by focusing on this very pragmatic goal.

More specifically, the outline of the thesis is as follows. In Chapter 2, we provide a review of the theoretical aspects of hash-based data structures and the practical consequences for routers, which is important for properly motivating and understanding the significance of our technical results. In Chapter 3, we show that a wide class of variants of the classical Bloom filter data structure can be very efficiently derandomized to use only two fully random hash functions. (For readers unfamiliar with the classical Bloom filter, it is a simple, hash-based data structure for set membership; we will review it in detail in Chapter 2.) From a theoretical perspective, these results show that Bloom filters and related data structures can often be substantially derandomized with essentially no loss in performance. From a practical perspective, this derandomization allows for a tremendous speedup in query intensive applications where the processor time required to perform a Bloom filter operation is at a premium. This speedup was first observed empirically by Dillinger and Manolios in [19, 20], and was our original motivation for studying this problem.

In Chapters 4 and 5, we study two main issues that arise in the design of efficient, open-addressed hash tables for implementation in high-performance router hardware. (We describe the relevance of hash tables to routers in Chapter 2.) Here, the most important measure of efficiency is, unsurprisingly, the size of the table. This leads one to consider multiple choice hash tables (e.g., [2, 60]) where items can be placed according to any one of a number of hash functions, as these tables require less space than tables with a single

hash function. This technique requires every hash table operation to consider multiple locations in the hash table, which is potentially expensive. Indeed, one must trade off between examining cells in the hash table serially and complicating the hardware design to perform these examinations in parallel. Specifically, performing these examinations in parallel requires additional pins to the off-chip memory that is holding the hash table, so that multiple cells can be read at once. Following a general framework proposed by Song *et al.* in [56], we show in Chapter 4 that a small, simple Bloom filter-based data structure in fast memory can essentially allow us to use a multiple choice hash table while only examining a single cell of the hash table during an insertion, lookup, or deletion operation.

For scenarios where we can afford the parallelization above, the space utilization of standard multiple choice hash table constructions can be improved by allowing items to be moved within the hash table after they are initially inserted. There are a number of interesting hash table constructions with this property in the theory literature, particularly cuckoo hashing [26, 48], which appears to be well suited to this setting. Unfortunately, while these schemes typically exhibit very good amortized behavior, the worst case insertion times are often too large for the applications considered here. In particular, while the average time required to insert an item is very small with high probability, it is also true that, with non-negligible probability, there is some item whose insertion requires a substantial amount of time. Such delays are unacceptable in high-performance router hardware.

We address this issue in Chapter 5. Specifically, we introduce and analyze a wide variety of hash table constructions that allow at most one item in the table to be moved during the insertion of a new item. These schemes allow us to harness some of the benefits of the already-known hashing schemes that allow moves, while placing an upper bound on the worst case time for an insertion operation. Using differential equation approximations and numerical methods, we are able to tightly quantify the performance of our schemes and show that they are superior to standard constructions that do not allow moves. Our ability to numerically approximate the exact behavior of these new schemes is especially interesting because the exact behavior of existing schemes that allow moves, such as cuckoo hashing, is not well understood.

From a methodological perspective, the technical chapters of this thesis are arranged roughly from primarily theoretical to primarily numerical and experimental. Indeed, the role of theorems and proofs in our analyses decreases over the course of this work, but there is always a strong theoretical foundation motivating our approaches. Similarly, the role of numerical and experimental methods increases between chapters, but our techniques are always motivated by a strong desire to achieve good performance for a real application. Overall, each chapter shows how a different blend of theoretical, numerical, and experimental approaches can be formulated to address a practical problem. Taken as a whole, this thesis demonstrates the power of a flexible methodology for designing real-world systems that incorporates theoretical foundations for intuition and formal reasoning, numerical evaluations for precise results, and experiments for verification.

Chapter 2

Background

This chapter gives some brief background useful for motivating and understanding the significance of the contributions of this work. In particular, we start by surveying some of the key theoretical and practical issues in designing the sorts of hash-based data structures that we use in subsequent chapters. Then we explore the use of hash tables in router hardware, providing important background for the results of Chapters 4 and 5.

2.1 Hash-Based Data Structures

This section gives a brief review of the history, constructions, and issues in the design of hash-based data structures that are relevant to this work. In particular, we describe the way that we think about hash functions, review the standard Bloom filter data structure, and discuss multiple choice hash tables.

2.1.1 Hash Functions

Intuitively, a hash function $H : U \rightarrow V$ is a function that maps every element $x \in U$ to a hash value $H(x) \in V$ in a fashion that is somehow random. The most natural mathematical model for a hash function is that it is *fully random*; that is, it is a random element of V^U , the set of functions with domain U and codomain V . Under this assumption, the hash values $\{H(x) : x \in U\}$ corresponding to the elements in U are independent random variables that are each uniformly distributed over V . Clearly, this is a very appealing scenario for probabilistic analysis.

Unfortunately, it is almost always impractical to construct fully random hash functions, as the space required to store such a function is essentially the same as that required to encode an arbitrary function in V^U as a lookup table. This sort of thinking quickly leads to compromises between the randomness properties that we desire in a hash function and the computational resources needed to store and evaluate such a function. The seminal work along these lines is the introduction of *universal hash families* by Carter and Wegman [12, 66], which introduces the idea of choosing a hash function H randomly (and efficiently) from a set \mathcal{H} of potential hash functions, chosen so that the joint distribution of the random variables $\{H(x) : x \in U\}$ satisfies limited but intuitively powerful

properties. In particular, these works introduce *strongly 2-universal* hash families, where the random variables $\{H(x) : x \in U\}$ are pairwise independent and uniformly distributed over V , as well as *weakly 2-universal* hash families, where for any $x, y \in U$, it holds that $\Pr(H(x) = H(y)) \leq 1/|V|$. (A set S of random variables is *pairwise independent* if any two random variables in S are independent.) As a matter of terminology, strongly and weakly 2-universal hash families are often called strongly and weakly *pairwise independent* hash families, respectively, especially outside of the hashing literature. Furthermore, the qualifiers *strongly* and *weakly* are frequently omitted, in which case one usually assumes the stronger assumption, even though the weaker one is often adequate. Also, the term *k-wise independent hash family* for some $k > 2$ refers to the generalization of (strongly) pairwise independent hash families where the random variables $\{H(x) : x \in U\}$ are *k-wise independent* and uniformly distributed over V . (A set S of random variables is *k-wise independent* if any set of k random variables in S is independent.) Finally, the terms *hash functions* and *hash families* are often used interchangeably when the meaning is clear. We make use of these conventions in this thesis.

Since Carter and Wegman’s original work [12], there has been a substantial amount of research on efficient constructions of hash functions that are theoretically suitable for use in data structures and algorithms. Two major developments in this line are due to Siegel [54] and Östlin and Pagh [46]. Siegel’s hash families have $U = \{1, \dots, m\}$ and $V = \{1, \dots, n\}$ and are *k-wise independent* for some $k < m^\epsilon$, for some constant $\epsilon < 1$. Furthermore, a hash function can be sampled from the family in constant time and m^δ space, for some constant $\delta < 1$. In many situations, Siegel’s hash families can be applied more-or-less directly to an analysis of a randomized data structure or algorithm that assumes fully random hash functions to obtain the same probabilistic guarantees, while ensuring that all operations can be performed in the standard RAM model of computation (where a random bit can be obtained in constant time). Östlin and Pagh [46] use Siegel’s hash families to show how to construct, in their words, “A hash function that, on any set of n inputs, behaves like a truly random function with high probability, can be evaluated in constant time on a RAM, and can be stored in $O(n)$ words.” This result allows any analysis of a randomized data structure or algorithm that assumes fully random hash functions to be directly adapted to the standard RAM model. Unfortunately, Siegel’s hash classes are impractical, as the time required to evaluate a hash function, while constant, is very large. Thus, at present, these results do not seem to have much potential to directly impact a real implementation of hash functions.

Fortunately, it seems that in practice simple hash functions perform very well. Indeed, they can be implemented very efficiently. For example, Dietzfelbinger *et al.* [17] exhibit a hash function that can be implemented with a single multiplication and a right shift operation and satisfies a slightly weakened version of weak pairwise independence. For scenarios where multiplications are undesirable, Carter and Wegman’s original work [12] provides a practical strongly pairwise independent hash function that relies on XOR operations. Some practical evaluations of these hash functions and others, for both hardware and software applications (including Bloom filters, discussed in Section 2.1.2 and used heavily in this work), are given in [24, 58, 50, 51, 52]. Overall, these works suggest that it is possible to choose very simple hash functions that work very well in practical applications.

There is also theoretical work that strives to explain why simple hash functions seem to perform well in practice. One common approach is to examine a particular theoretical analysis that uses the assumption of fully random hash functions, and then attempt to modify the analysis to obtain a comparable result for a class of simple hash functions (e.g., pairwise independent hash functions), or a particular family. For instance, it is an easy exercise to show that partitioned Bloom filters (described in Section 2.1.2) can be implemented with any pairwise independent hash functions, with only a small increase in the false positive probability. As an example of this technique that works only for a specific hash family, Woelfel [67] shows that one can implement d -left hashing (described in Section 2.1.3) using a particular type of simple hash function. In a different direction, Mitzenmacher and Vadhan [43] show that for certain applications, if one is willing to assume that the set of items being hashed satisfies certain randomness properties, then any analysis based on the assumption that the hash functions are fully random is also valid with pairwise independent hash functions (up to some small, additional error probability). In particular, this result applies to Bloom filters; we discuss this aspect in more detail in Chapter 3, as the main result of that chapter is critical for the Bloom filter result in [43].

Having reviewed the approaches to hashing most related to this work, we now articulate our perspective on hash functions. Since the ultimate goal of this thesis is to directly impact the implementation of real-world systems, and since it is usually possible to choose a simple, practical hash function for an application that results in performance similar to what we would expect for a fully random hash function, we allow ourselves to assume that our hash functions are fully random. Furthermore, unless the cost of hashing is absolutely critical for the performance of the application under consideration, we generally assume that hashing is an inexpensive operation. The exception to this assumption occurs in Chapter 3, where the application of interest makes use of computationally intensive hash functions (e.g., MD5), and the cost of evaluating these hash functions is essentially the sole determiner of performance. Thus, in that chapter, we are very conservative in our use of hashing. Still, however, when we do hash, we assume that our hash functions are fully random. From a pure, traditional theoretical computer science perspective, it may seem strange to be simultaneously conservative in our use of hashing and liberal in our assumption that our hash functions are fully random. However, our view makes perfect sense in the context of a real application when we think of *modeling* the hash functions for the sake of predicting performance in a statistical sense, as opposed to explicitly constructing the hash functions to satisfy concrete theoretical guarantees. In Chapters 4 and 5, this tension disappears, as the resources most critical to the applications considered there are memory space and memory accesses, and so we use hash functions liberally.

2.1.2 Bloom Filters

A *Bloom filter* [5] is a simple space-efficient randomized data structure for representing a set in order to support membership queries. We begin by reviewing the fundamentals, based on the presentation of the survey [9], which we refer to for further details. A Bloom filter for representing a set $S = \{x_1, x_2, \dots, x_n\}$ of n elements from a large universe U consists of an array of m bits, initially all set to 0. The filter uses k independent (fully random) hash functions h_1, \dots, h_k with range $\{1, \dots, m\}$. For each element $x \in S$, the bits

$h_i(x)$ are set to 1 for $1 \leq i \leq k$. (A location can be set to 1 multiple times.) To check if an item y is in S , we check whether all $h_i(y)$ are set to 1. If not, then clearly y is not a member of S . If all $h_i(y)$ are set to 1, we assume that y is in S , and hence a Bloom filter may yield a *false positive*.

The probability of a false positive for an element not in the set, or the *false positive probability*, can be estimated in a straightforward fashion, given our assumption that the hash functions are fully random. After all the elements of S are hashed into the Bloom filter, the probability that a specific bit is still 0 is

$$p' = (1 - 1/m)^{kn} \approx e^{-kn/m}.$$

In this section, we generally use the approximation $p = e^{-kn/m}$ in place of p' for convenience.

If ρ is the proportion of 0 bits after all the n elements are inserted in the table, then conditioned on ρ the probability of a false positive is

$$(1 - \rho)^k \approx (1 - p')^k \approx (1 - p)^k = \left(1 - e^{-kn/m}\right)^k.$$

These approximations follow since $\mathbf{E}[\rho] = p'$, and ρ can be shown to be highly concentrated around p' using standard techniques. It is easy to show that the expression $(1 - e^{-kn/m})^k$ is minimized when $k = \ln 2 \cdot (m/n)$, giving a false positive probability f of

$$f = \left(1 - e^{-kn/m}\right)^k = (1/2)^k \approx (0.6185)^{m/n}.$$

In practice, k must be an integer, and a smaller, sub-optimal k might be preferred since this reduces the number of hash functions that have to be computed.

This analysis provides us (roughly) with the probability that a single item $z \notin S$ gives a false positive. We would like to make a broader statement, that in fact this gives a false positive *rate*. That is, if we choose a large number of *distinct* elements not in S , the fraction of them that yield false positives is approximately f . This result follows immediately from the fact that ρ is highly concentrated around p' , and for this reason, the false positive probability is sometimes called the *false positive rate*. As we will see, in our variations (particularly in Chapter 3), it is not always as clear that the false positive probability acts like a false positive rate, and so we clearly distinguish between the two concepts. (Indeed, we think this clarification is a contribution of this work.)

Before moving on, we note that sometimes Bloom filters are described slightly differently, with each hash function having a disjoint range of m/k consecutive bit locations instead of having one shared array of m bits. We refer to this variant as a *partitioned* Bloom filter. Repeating the analysis above, we find that in this case the probability that a specific bit is 0 is

$$\left(1 - \frac{k}{m}\right)^n \approx e^{-kn/m},$$

and so, asymptotically, the performance is the same as the original scheme. In practice, however, the partitioned Bloom filter tends to perform slightly worse than the non-partitioned Bloom filter. This is explained by the observation that

$$\left(1 - \frac{1}{m}\right)^{kn} > \left(1 - \frac{k}{m}\right)^n$$

when $k > 1$, so partitioned filters tend to have more 1's than non-partitioned filters, resulting in larger false positive probabilities.

The standard Bloom filter naturally supports insertion operations: to add a new element x to the set represented by the filter, we simply set the corresponding bits of the filter to 1. Unfortunately, the data structure does not support deletions, since changing bits of the filter from 1 to 0 could introduce false negatives. Of course, if we wish to support deletions, we can simply replace each bit of the filter with a counter, initially set to 0. In this case, the filter naturally represents a *multi-set* S of items, rather than a set. To insert an item x into the filter, we now increment its corresponding counters $h_1(x), \dots, h_k(x)$, and to delete an item known to be in the multi-set represented by the filter, we decrement those counters. To test whether an item y is in S , we can simply check whether all the counters $h_1(y), \dots, h_k(y)$ are positive, obtaining a false positive if $y \notin S$ but none of the counters are 0. (More generally, we can test whether an item y occurs in S with multiplicity at least $\ell \geq 1$ by testing whether the counters $h_1(y), \dots, h_k(y)$ are at least ℓ , with some probability of a false positive.)

This Bloom filter variant is called a *counting Bloom filter* [24]. Clearly, all of our prior analysis for standard Bloom filters applies to counting Bloom filters. However, there is a complication in choosing the number of bits to use in representing a counter. Indeed, if a counter overflows at some point, then the filter may yield a false negative in the future. It is easy to see that the number of times a particular counter is incremented has distribution $\text{Binomial}(nk, 1/m) \approx \text{Poisson}(nk/m) = \text{Poisson}(\ln 2)$, by the Poisson approximation to the binomial distribution (assuming $k = (m/n) \ln 2$ as above). By a union bound, the probability that some counter overflows if we use b -bit counters is at most $m \Pr(\text{Poisson}(\ln 2) \geq 2^b)$. As an example, for a sample configuration with $n = 10000$, $m = 80000$, $k = (m/n) \ln 2 = 8 \ln 2$, and $b = 4$, we have $f = (1/2)^k = 2.14\%$ and $m \Pr(\text{Poisson}(\ln 2) \geq 2^b) = 1.78 \times 10^{-11}$, which is negligible. (Of course, in practice k must be an integer, but the point is clear.) This sort of calculation is typical for counting Bloom filters.

We also emphasize that the performance of a Bloom filter does not depend at all on the size of the items in the set that it represents (except for the additional complexity required for the hash functions for larger items). Thus, although Bloom filters allow false positives, the space savings over more traditional data structures for set membership, such as hash tables, often outweigh this drawback. It is therefore not surprising that Bloom filters and their many variations have proven increasingly important for many applications (see, for instance, the survey [9]). As just a partial listing of examples, *compressed Bloom filters* are optimized to minimize space when transmitted [38], *retouched Bloom filters* trade off false positives and false negatives [21], *Bloomier filters* keep function values associated with set elements (thereby offering more than set membership) [13], *count-min sketches* [14] and *multistage filters* [23] track counts associated with items in data streams, and *approximate concurrent state machines* track the dynamically changing state of a changing set of items [6]. Although recently more complex but asymptotically better alternatives have been proposed (e.g., [7, 47]), the Bloom filter's simplicity, ease of use, and excellent performance make it a standard data structure that is and will continue to be of great use in many applications.

2.1.3 Multiple Choice Hash Tables

As mentioned in Chapter 1, the canonical example of a hash table is one that uses a single hash function (which is assumed to be fully random), with chaining to deal with collisions. The standard analysis shows that if the number of buckets in the table is proportional to the number of items inserted, the expected number of items that collide with a particular item is constant. Thus, on average, lookup operations should be fast. However, for applications where such average case guarantees are not sufficient, we also need some sort of probabilistic worst case guarantee. Here, the qualifier that our worst case guarantees be *probabilistic* excludes, for instance, the case where all items in the table are hashed to the same bucket. Such situations, while technically possible, are so ridiculously unlikely that they do not warrant serious consideration. As an example of a probabilistic worst case guarantee, we consider throwing n balls independently and uniformly at random into n bins. In this case, a classical result (e.g., [42, Lemmas 5.1 and 5.12]) shows that the maximum number of balls in a bin is $\Theta((\log n)/\log \log n)$ with high probability. This result translates directly to a probabilistic worst case guarantee for a standard hash table with n items and n buckets: while the expected time to lookup a particular item is constant, with high probability the longest time that *any* lookup can require is $\Theta((\log n)/\log \log n)$.

The previous example illustrates a connection between hashing and *balanced allocations*, where some number of balls is placed into bins according to some probabilistic procedure, with the implicit goal of achieving an allocation where the balls are more-or-less evenly distributed among the bins. In a seminal work, Azar *et al.* [2] strengthened this connection by showing a very powerful balanced allocation result: if n balls are placed sequentially into $m \geq n$ bins for $m = O(n)$, with each ball being placed in one of a constant $d \geq 2$ randomly chosen bins with minimal load at the time of its insertion, then with high probability the maximal load in a bin after all balls are inserted is $(\ln \ln n)/\ln d + O(1)$. In particular, if we modify the standard hash table with chaining from above to use d hash functions, inserting an item into one of its d hash buckets with minimal total load, and performing a lookup for an item by checking all d of its hash buckets, then the expected lookup time is still constant (although larger than before), but the probabilistic worst case lookup time drops exponentially. This scheme, usually called *d-way chaining*, is arguably the simplest instance of a *multiple choice hash table*, where each item is placed according to one of several hash functions.

Unsurprisingly, the impact of [2] on the design of randomized algorithms and data structures, particularly hash tables and their relatives, has been enormous. For details and a more complete list of references, we refer to the survey [41]. Before moving on, however, we mention an important improvement of the main results in [2] due to Vöcking [60]. That work exhibits the *d-left* hashing scheme, which works as follows. There are n items and m buckets. The buckets are partitioned into d groups of approximately equal size, and the groups are laid out from left to right. There is one hash function for each group, mapping the items to a randomly chosen bucket in the group. The items are inserted sequentially into the table, with an item being inserted into the least loaded of its d hash buckets (using chaining), with ties broken to the left. Vöcking [60] shows that if $m = n$ and $d \geq 2$ is constant, then the maximum load of a bucket after all the items are inserted is $(\ln \ln n)/d\phi_d + O(1)$, where ϕ_d is the asymptotic growth rate of the d -th order Fibonacci numbers. In particular, this

improves the factor of $\ln d$ in the denominator of the $(\ln \ln n)/\ln d + O(1)$ result of Azar *et al.* [2]. Furthermore, [60] shows that d -left hashing is optimal up to an additive constant.

Both d -way chaining and d -left hashing are practical schemes, with d -left hashing being generally preferable. In particular, the partitioning of the hash buckets into groups for d -left hashing makes that scheme more amenable to a hardware implementation, since it allows for an item's d hash locations to be examined in parallel. In this work, however, we avoid hashing schemes that resolve collisions with chaining, instead preferring open-addressed hash tables where each bucket can store a fixed constant number of items. More specifically, we usually assume that each bucket can hold at most one item, although our results can be generalized. We proceed in this way partly because it makes our analysis possible, but also because open-addressed hash tables seem much more amenable to hardware implementations than hash tables that use chaining. For the applications to high-performance routers that motivate our work, the suitability of our schemes to hardware implementations is critical.

The multiple choice hash table that we use in this work is the multilevel hash table (MHT) of Broder and Karlin [8]. This is a hash table consisting of d sub-tables T_1, \dots, T_d , with each T_i having one hash function h_i . We view these tables as being laid out from left to right. To insert an item x , we find the minimal i such that $T_i[h_i(x)]$ is unoccupied, and place x there. If $T_1[h_1(x)], \dots, T_d[h_d(x)]$ are all occupied, then we declare a *crisis*. There are multiple things that we can do to handle a crisis. The approach in [8] is to resample the hash functions and rebuild the entire table. That work shows that it is possible to insert n items into a properly designed MHT with $O(n)$ total space and $d = O(\log \log n)$ in $O(n)$ expected time, assuming only 4-wise independent hash functions. In Chapter 4, we use a similar design and analysis to ensure that, with high probability, no rehashings are necessary; in effect, this eliminates the possibility of a crisis. In Chapter 5, we allow a small fraction of the n items to overflow into a content-addressable memory (CAM) (discussed in more detail in Section 2.2.1), which supports insertions, deletions, and lookups in constant time. As before, no rehashings are necessary in practice, as we size the CAM so that it is extremely unlikely to fill up.

We defer the details of the various ways to construct MHTs to Chapters 4 and 5, where MHTs play a critical role. For the moment, however, we simply note that MHTs naturally support deletions, as one can just perform a lookup on an item to find its location in the table, and then mark the corresponding bucket as deleted so that the item is eventually overwritten as it would be if the item were simply removed from its bucket. Also, MHTs appear well-suited to a hardware implementation. In particular, their open-addressed nature seems to make them preferable to approaches that involve chaining, and their use of separate sub-tables allows for the possibility that all of the hash locations for a particular item can be accessed in parallel. Indeed, these considerations are part of the original motivation from [8].

2.2 Hash Tables and Routers

We now give some background on the use of hash tables in routers, which is important for understanding the significance of the results in Chapters 4 and 5. We start

by outlining the high-level hardware design issues that arise in this setting, and then we examine some of the actual applications where hash tables are used.

2.2.1 Hardware Design Issues

While detailed hardware design considerations are outside the scope of this work, we must be careful to keep some major issues in mind in order to appropriately motivate and evaluate the hash-based data structures discussed in Chapters 4 and 5. To this end, we provide the following list of concerns for designing router hardware, paraphrased from a standard text [59]. Note that all numbers listed here are almost certainly out of date, as they are taken directly from [59], which was published in 2004. However, while the specific numbers themselves may be obsolete, the qualitative picture that they paint is still valid.

- **Chip Complexity:** the amount of combinatorial logic and memory available on a chip. There are two major chip technologies: application-specific integrated circuits (ASICs) and field-programmable gate arrays (FPGAs). As their names suggest, ASICs are tailor-made for a specific application, whereas FPGAs are more general purpose, reprogrammable devices. In particular, FPGAs tend to require more power than ASICs and offer much less on-chip memory, but their programmable nature reduces development costs.
- **Chip Speed:** roughly speaking, the rate at which a chip can perform computation. A chip's speed is often represented by its clock speed, but hardware parallelism (e.g., pipelining and the use of wide memory words) can allow multiple operations per clock cycle, making the notion of chip speed difficult to quantify. For the sake of concreteness, it is useful to consider an example where a router must operate on packets of 40 bytes (the minimum packet size for TCP, corresponding to an acknowledgment with no payload) at a wire speed of 40 Gbps (the OC-768 standard data rate for transmission on optical fiber). At this speed, the router has 8 ns to handle a packet. For comparison, the clock cycle corresponding to a 1 Ghz clock speed is 1 ns. While clock speeds of, say, 3 Ghz are common, and hardware parallelism can boost performance, this example illustrates that even simple computational operations are precious and should not be taken for granted.
- **Chip I/O:** the capacity of a chip to handle I/O. Current technology supports at most about 1000 I/O pins per chip. If an application requires more pins, then it must be modified to use more chips, increasing power consumption and consuming board space.
- **Serial I/O:** the speed of the links between chips. Similarly to chip I/O, if these speeds are insufficient for an application, then more pins must be used for communication, possibly requiring more chips. As before, increasing the number of chips increases power requirements and consumes board space.
- **Memory:** the amounts, types, and location (on-chip or off-chip) RAM needed for a specific application. There are two major memory technologies: SRAM and DRAM. SRAM is faster than DRAM, but also more expensive. For instance, sample access

times for the various possibilities are: 1 ns for on-chip SRAM, 2.5 ns for off-chip SRAM, 30 ns for on-chip DRAM, and 60 ns for off-chip DRAM. DRAM costs roughly 4-10 times less than SRAM (per bit). Once again, it is useful to consider the example where a router must operate on packets of 40 bytes at a wire speed of 40 Gbps. The router then has 8 ns to handle a packet. Clearly, this time constraint places severe limitations on both the memory technology that can be used and the number of times that it can be accessed per packet. (In particular, DRAM may be totally unsuitable for such a demanding setting.)

- **Power and Packaging:** the overall power consumption of a router and its physical size. High-speed routers have large power requirements and therefore require sophisticated cooling systems. Like every other component of the router, such a system requires physical space, which is a precious resource to Internet service providers.

We must also point out that routers often make use of content-addressable memories (CAMs), which are fully associative memories, usually based on SRAM technology (so that improvements in SRAM technology tend to translate into improvements in CAM technology), that support a lookup of a data item in a single access (e.g., 10 ns) by performing lookups on all memory locations in parallel. (Note that the example 10 ns CAM access time is much slower than the example SRAM access times above, and is actually too slow for the particular 40 Gbps example above; still, it is certainly possible to design CAMs suitable for high-performance router applications.) There are also ternary CAMs that support wildcard bits, which is an extremely useful feature for prefix-match lookups, a critical router application discussed in Section 2.2.2. This specialty hardware is much faster than any data structure built with commodity SRAM, such as a hash table, could ever be. However, the parallel lookup feature of CAMs causes them to use a lot more power than comparable SRAMs. Furthermore, the smaller market for this sort of technology results in CAMs being much more expensive, per bit, than SRAMs. For both peak power consumption and cost per bit, an order of magnitude difference between a CAM and a comparable SRAM would not be surprising.

In this work, we regard CAMs as being expensive, special-purpose hardware for table lookups that are practical only when storing small sets of items. Thus, we do not think of CAMs as being a replacement for hash tables, but we do advocate their use for parts of our hash-based data structures. In particular, in Chapter 5 we design MHTs from which some small number of items are expected to overflow. We suggest using a CAM to store those items (but not all items, which we view as prohibitively expensive), allowing us to claim excellent worst case lookup times for the entire data structure.

2.2.2 Applications of Hash Tables

Hash tables play an increasingly important role in a growing number of applications implemented in router hardware. We see two major reasons this phenomenon. The first is essentially a classical property of hash tables: they are very efficient and general data structures for lookup problems. Second, as hardware technology improves, it is becoming increasingly practical for network processors to include built-in hashing functionality (e.g.,

[1]), which in turn allows for more sophisticated hashing-based approaches in packet processing algorithms that must operate at wire speeds. Therefore there is enormous potential for improving a large class of applications by designing hash tables that are *specifically optimized* for the particular hardware constraints of high-performance routers.

Unfortunately, few concrete applications of hash tables in routers are discussed in the networking literature, primarily because the actual implementations of these techniques are widely considered to be trade secrets by router manufacturers. Broadly speaking, though, hash tables are considered generally useful for a variety of packet classification, network monitoring, and lookup applications. We review some of these applications here, drawing heavily on the brief survey in [56]. Indeed, that work explicitly argues that hash table lookup performance is an important consideration for networking applications, and that appropriately customized designs are important for improving the performance of a wide variety of packet processing techniques.

Perhaps the simplest class of applications where hash tables are useful in routers is for IP route lookup, which is the problem of determining the next hop for a packet based on its destination IP address. The algorithmic problem corresponding to this application is the *longest prefix match* problem: designing a data structure for a set S of strings (implicitly containing the empty string) that finds, for a query string s , a string in S that is a prefix of s of maximal length. Basically, each string in S represents a rule for forwarding packets, with longer strings corresponding to rules of higher priority. The goal of the IP route lookup problem is to route a packet according to a rule of highest priority. For more details, see, for example, [59].

The most prominent example in the networking literature of a hash-based approach to a router application is probably the longest prefix match data structure of Waldvogel *et al.* [64]. Here (in essence), for each possible prefix length ℓ , there is a hash table containing all strings in S of length at least ℓ . The longest match for a query string is then found by a binary search on the length of a prefix. Of course, this is just a high-level description of a naive version of the scheme. For improvements, see, for example [10, 57, 64].

Hash tables are also important for the implementation of packet classification algorithms. In particular, many of these techniques operate by first performing a lookup for a single header field and then using the results to narrow the search space. Examples include [3, 25, 32, 37, 35]. Thus, all of these methods can potentially benefit from improved hash table constructions that are customized for the high-performance router setting. Furthermore, going forward, we expect advances in these sorts of constructions to naturally impact the development of new applications.

Before continuing, we note that while it is common for a router application to be amenable (at least in principle) to a hash-based approach, there is something of a historical bias against these techniques. The issue is that hash tables and related data structures can only offer probabilistic guarantees, as opposed to the more commonly accepted deterministic worst case performance bounds offered by classical algorithmic approaches. For instance, for the longest prefix match problem described above, a hardware designer may prefer a trie-based approach to a variant of the scheme of Waldvogel *et al.* [64], whose effectiveness ultimately relies on the efficiency of hash table lookups, which can be difficult to quantify. While this attitude may be changing, it is important to remember that the sorts of

probabilistic guarantees offered by hash-based solutions always depend strongly on certain randomness assumptions (i.e., the assumption that hash functions are fully random), and so convincingly evaluating the performance of such a scheme requires extensive analysis and experimentation. Such analysis is a major theme of this work.

Chapter 3

Faster Bloom Filters Through Double Hashing

3.1 Introduction

In this chapter, we show that applying a standard technique from the hashing literature, often called *double hashing*, can simplify the implementation of Bloom filters significantly. The idea is the following: two hash functions $h_1(x)$ and $h_2(x)$ can simulate more than two hash functions of the form $g_i(x) = h_1(x) + ih_2(x)$. (See, for example, Knuth’s discussion of open addressing with double hashing [31].) In our context i will range from 0 up to some number $k - 1$ to give k hash functions, and the hash values are taken modulo the size of the relevant hash table. We demonstrate that this technique can be usefully applied to Bloom filters and related data structures. Specifically, only two hash functions are necessary to effectively implement a Bloom filter without any increase in the asymptotic false positive probability. This leads to less computation and potentially less need for randomness in practice. Specifically, in query-intensive applications where computationally non-trivial hash functions are used (such as in [19, 20]), hashing can be a potential bottleneck in using Bloom filters, and reducing the number of required hashes can yield an effective speedup. (Indeed, the only nontrivial computations performed by a Bloom filter are the constructions and evaluations of pseudorandom hash functions.) This improvement was found empirically in the work of Dillinger and Manolios [19, 20], who suggested using the hash functions

$$g_i(x) = h_1(x) + ih_2(x) + i^2 \pmod{m},$$

where m is the size of the hash table.

Here we provide a full theoretical analysis that holds for a wide class of variations of this technique, justifies and gives insight into the previous empirical observations, and is interesting in its own right. In particular, our methodology generalizes the standard asymptotic analysis of a Bloom filter, exposing a new convergence result that provides a common unifying intuition for the asymptotic false positive probabilities of the standard Bloom filter and the generalized class of Bloom filter variants that we analyze in this chapter. We obtain this result by a surprisingly simple approach; rather than attempt to directly

analyze the asymptotic false positive probability, we formulate the initialization of the Bloom filter as a balls-and-bins experiment, prove a convergence result for that experiment, and then obtain the asymptotic false positive probability as a corollary.

We start by analyzing a specific, somewhat idealized Bloom filter variation that provides the main insights and intuition for deeper results. We then move to a more general setting that covers several issues that might arise in practice, such as when the size of the hash table is a power of two as opposed to a prime. Finally, we demonstrate the utility of this approach beyond the simple Bloom filter by showing how it can be used to reduce the number of hash functions required for count-min sketches [14], a variation of the Bloom filter idea used for keeping approximate counts of frequent items in data streams.

Before beginning, we note that Luecker and Molodowitch [36] and Schmidt and Siegel [53] have shown that in the setting of open-addressed hash tables, the double hashing technique gives the same performance as uniform hashing. These results are similar in spirit to ours, but the Bloom filter setting is sufficiently different from that of an open-addressed hash table that we do not see a direct connection. We also note that our use of hash functions of the form $g_i(x) = h_1(x) + ih_2(x)$ may appear similar to the use of pairwise independent hash functions, and that one might wonder whether there is any formal connection between the two techniques in the Bloom filter setting. Unfortunately, this is not the case; a straightforward modification of the standard Bloom filter analysis suggests that if pairwise independent hash functions are used instead of fully random hash functions, then the space required to retain the same bound on the false positive probability increases by a constant factor. Mitzenmacher and Vadhan [43] make this idea more precise by exhibiting a pairwise independent hash family and a set for the Bloom filter to represent for which this increase in space is necessary. (In fact, that result holds for any constant-wise independence, not just pairwise independence.) In contrast, we show that using the g_i 's causes *no* increase in the false positive probability, so they can truly be used as a replacement for fully random hash functions.

We also note that there have already been some interesting, unanticipated applications of this work. On the theoretical side, Mitzenmacher and Vadhan [43] show that if the data stream supplying the set S of items to be represented by a Bloom filter and the item being queried has a reasonable amount of (Renyi) entropy, then our two pseudorandom hash functions can be replaced by pairwise independent hash functions without substantially increasing the false positive probability. This result serves an explanation for why pairwise independent hash functions often work well in certain applications, even though the usual worst case theoretical guarantees do not seem to hold. We emphasize that our derandomization result is an important part of the analysis in [43]. Without it, the entropy required from the data stream would scale linearly with the number k of hash functions used by the Bloom filter.

On a more practical side, Wang *et al.* [65] propose using Bloom filters to represent defect maps for nanoscale devices. The main issue here is that nanoscale fabrication techniques are likely to have much higher defect rates than traditional methods, and so the architecture must have a built-in capacity to deal with defective components. Wang *et al.* propose using Bloom filters (implemented with the standard, reliable CMOS technology) with double hashing to represent the set of defective components. Here, the use

of double hashing allows for simplification of the hardware design. Similarly, it would not be surprising if our double hashing techniques could be used to simplify other hardware implementations of Bloom filters.

3.2 A Simple Construction Using Two Hash Functions

As an instructive example case, we consider a specific application of the general technique described at the start of the chapter. We devise a Bloom filter that uses k fully random hash functions on some universe U of items, each with range $\{0, 1, 2, \dots, p-1\}$ for a prime p . Our hash table consists of $m = kp$ bits; each hash function is assigned a disjoint subarray of p bits in the filter, which we treat as numbered $\{0, 1, 2, \dots, p-1\}$. Our k hash functions will be of the form $g_i(x) = h_1(x) + ih_2(x) \bmod p$, where $h_1(x)$ and $h_2(x)$ are two independent, uniform random hash functions on the universe with range $\{0, 1, 2, \dots, p-1\}$, and throughout we assume that i ranges from 0 to $k-1$.

As with a standard partitioned Bloom filter, we fix some set $S \subseteq U$ and initialize the filter with S by first setting all of the bits to 0 and then, for each $x \in S$ and i , setting the $g_i(x)$ -th bit of the i -th subarray to 1. For any $y \in U$, we answer a query of the form “Is $y \in S$?” with “Yes” if and only if the $g_i(y)$ -th bit of the i -th subarray is 1 for every i . Thus, an item $z \notin S$ generates a false positive if and only if each of its hash locations in the array is also a hash location for some $x \in S$.

The advantage of our simplified setting is that for any two elements $x, y \in U$, exactly one of the following three cases occurs:

- $g_i(x) \neq g_i(y)$ for all i , or
- $g_i(x) = g_i(y)$ for exactly one i , or
- $g_i(x) = g_i(y)$ for all i .

That is, because we have partitioned the bit array into disjoint hash tables, each hash function can be considered separately. Moreover, by working modulo p , we have arranged that if $g_i(x) = g_i(y)$ for at least two values of i , then we must have $h_1(x) = h_1(y)$ and $h_2(x) = h_2(y)$, so all hash values are the same. This codifies the intuition behind our result: the most likely way for a false positive to occur is when each element in the Bloom filter set S collides with at most one array bit corresponding to the element generating the false positive; other events that cause an element to generate a false positive occur with vanishing probability. It is this intuition that motivates our analysis; in Section 3.3, we consider more general cases where other non-trivial collisions can occur.

Proceeding formally, we fix a set $S = \{x_1, x_2, \dots, x_n\}$ of n elements from U and another element $z \notin S$, and compute the probability that z yields a false positive. A false positive corresponds to the event \mathcal{F} that for each i there is (at least) one j such that $g_i(z) = g_i(x_j)$. Obviously, one way this can occur is if $h_1(x_j) = h_1(z)$ and $h_2(x_j) = h_2(z)$ for some j . The probability of this event \mathcal{E} is

$$\Pr(\mathcal{E}) = 1 - (1 - 1/p^2)^n = 1 - (1 - k^2/m^2)^n.$$

Notice that when $m/n = c$ is a constant and k is a constant, as is standard for a Bloom filter, we have $\Pr(\mathcal{E}) = o(1)$. Now since

$$\begin{aligned} \Pr(\mathcal{F}) &= \Pr(\mathcal{F} \mid \mathcal{E})\Pr(\mathcal{E}) + \Pr(\mathcal{F} \mid \neg\mathcal{E})\Pr(\neg\mathcal{E}) \\ &= \Pr(\mathcal{E}) + \Pr(\mathcal{F} \mid \neg\mathcal{E})\Pr(\neg\mathcal{E}) \\ &= o(1) + \Pr(\mathcal{F} \mid \neg\mathcal{E})(1 - o(1)), \end{aligned}$$

it suffices to consider $\Pr(\mathcal{F} \mid \neg\mathcal{E})$ to obtain the (constant) asymptotic false positive probability.

Conditioned on $\neg\mathcal{E}$ and $(h_1(z), h_2(z))$, the pair $(h_1(x_j), h_2(x_j))$ is uniformly distributed over the $p^2 - 1$ values in $V = \{0, \dots, p-1\}^2 - \{(h_1(z), h_2(z))\}$. Of these, for each $i^* \in \{0, \dots, k-1\}$, the $p-1$ pairs in

$$V_{i^*} = \{(a, b) \in V : a \equiv i^*(h_2(z) - b) + h_1(z) \pmod{p}, b \not\equiv h_2(z) \pmod{p}\}$$

are the ones such that if $(h_1(x_j), h_2(x_j)) \in V_{i^*}$, then i^* is the unique value of i such that $g_i(x_j) = g_i(z)$. We can therefore view the conditional probability as a variant of a balls-and-bins problem. There are n balls (each corresponding to some $x_j \in S$), and k bins (each corresponding to some $i^* \in \{0, \dots, k-1\}$). With probability $k(p-1)/(p^2-1) = k/(p+1)$ a ball lands in a bin, and with the remaining probability it is discarded; when a ball lands in a bin, the bin it lands in is chosen uniformly at random. What is the probability that all of the bins have at least one ball?

This question is surprisingly easy to answer. By the Poisson approximation and the fact that $p = m/k = cn/k$, the total number of balls that are not discarded has distribution $\text{Binomial}(n, k/(p+1)) \approx \text{Poisson}(k^2/c)$. Since each ball that is not discarded lands in a bin chosen at random, the joint distribution of the number of balls in the bins is asymptotically the same as the joint distribution of k independent $\text{Poisson}(k/c)$ random variables, by a standard property of Poisson random variables. The probability that each bin has a least one ball now clearly converges to

$$\Pr(\text{Poisson}(k/c) > 0)^k = \left(1 - e^{-k/c}\right)^k = \left(1 - e^{-kn/m}\right)^k,$$

which is the asymptotic false positive probability for a standard Bloom filter.

We make the above argument much more general and rigorous in Section 3.3, but for now we emphasize that we have actually characterized much more than just the false positive probability of our Bloom filter variant. In fact, we have characterized the asymptotic joint distribution of the number of items in S hashing to the locations used by some $z \notin S$ as being independent $\text{Poisson}(k/c)$ random variables. Furthermore, from a technical perspective, this approach appears fairly robust. In particular, the above analysis uses only the facts that the probability that some $x \in S$ shares more than one of z 's hash locations is $o(1)$, and that if some $x \in S$ shares exactly one of z 's hash locations, then that hash location is nearly uniformly distributed over z 's hash locations. These observations suggest that the techniques used in this section can be generalized to handle a much wider class of Bloom filter variants, and form the intuitive basis for the arguments in Section 3.3.

Now, as in Section 2.1.2, we must argue that the asymptotic false positive probability also acts like a false positive *rate*. Similar to the case for the standard Bloom filter, this fact boils down to a concentration argument. Once the set S is hashed, there is a set

$$B = \{(b_1, b_2) : h_1(z) = b_1 \text{ and } h_2(z) = b_2 \text{ implies } z \text{ gives a false positive}\}.$$

Conditioned on $|B|$, the probability of a false positive for any element in $U - S$ is $|B|/p^2$, and these events are independent. If we show that $|B|$ is concentrated around its expectation, it follows easily that the fraction of false positives in a set of distinct elements not in S is concentrated around the false positive probability.

A simple Doob martingale argument suffices (e.g., [42, Section 12.5]). Each hashed element of S can change the number of pairs in B by at most kp in either direction. This observation follows immediately from the fact that given any element x , its hash values $h_1(x)$ and $h_2(x)$, and some $i \in \{0, \dots, k-1\}$, there are exactly p solutions $(b_1, b_2) \in \{0, \dots, p-1\}^2$ to the equation

$$h_1(x) + ih_2(x) \equiv b_1 + ib_2 \pmod{p}.$$

By [42, Section 12.5], we now have that for any $\epsilon > 0$,

$$\Pr(|B - \mathbf{E}[B]| \geq \epsilon p^2) \leq 2 \exp[-2\epsilon^2 p^2 / nk^2].$$

It is now easy to derive the desired conclusion. We defer further details until Section 3.6, where we consider a similar but much more general argument.

3.3 A General Framework

In this section, we introduce a general framework for analyzing Bloom filter variants, such as the one examined in Section 3.2. We start with some new notation. For any integer ℓ , we define the set $[\ell] = \{0, 1, \dots, \ell - 1\}$ (note that this definition is slightly non-standard). We denote the support of a random variable X by $\text{Supp}(X)$. For a multi-set M , we use $|M|$ to denote the number of distinct elements of M , and $\|M\|$ to denote the number of elements of M with multiplicity. For two multi-sets M and M' , we define $M \cap M'$ and $M \cup M'$ to be, respectively, the intersection and union of M' as *multi-sets*. Furthermore, in an abuse of standard notation, we define the statement $i, i \in M$ as meaning that i is an element of M of multiplicity at least 2.

We are now ready to define the framework. As before, U denotes the universe of items and $S \subseteq U$ denotes the set of n items for which the Bloom filter will answer membership queries. We define a *hashing scheme* to be a method of assigning hash locations to every element of U . Formally, a hashing scheme is specified by a joint distribution of discrete random variables $\{H(u) : u \in U\}$ (implicitly parameterized by n), where for $u \in U$, $H(u)$ represents the multi-set of hash-locations assigned to u by the hashing scheme. We do not require a hashing scheme to be defined for every value of n , but we do insist that it be defined for infinitely many values of n , so that we may take limits as $n \rightarrow \infty$. For example, for the class of hashing schemes discussed in Section 3.2, we think of the constants k and c as being fixed to give a particular hashing scheme that is defined for those values of n such that $p \triangleq m/k$ is a prime, where $m \triangleq cn$. Since there are infinitely many primes, the

asymptotic behavior of this hashing scheme as $n \rightarrow \infty$ is well-defined and is the same as in Section 3.2, where we let m be a free parameter and analyzed the behavior as $n, m \rightarrow \infty$ subject to m/n and k being fixed constants, and m/k being prime.

Having defined the notion of a hashing scheme, we may now formalize some important concepts with new notation (all of which is implicitly parameterized by n). We define H to be the set of all hash locations that can be assigned by the hashing scheme (formally, H is the set of elements that appear in some multi-set in the support of $H(u)$, for some $u \in U$). For $x \in S$ and $z \in U - S$, define $C(x, z) = H(x) \cap H(z)$ to be the multi-set of hash collisions of x with z . We let $\mathcal{F}(z)$ denote the *false positive event* for $z \in U - S$, which occurs when each of z 's hash locations is also a hash location for some $x \in S$.

In the hashing schemes that we consider, $\{H(u) : u \in U\}$ will always be independent and identically distributed, and we will always have $\|H(u)\| = k$. In this case, $\Pr(\mathcal{F}(z))$ is the same for all $z \in U - S$, as is the joint distribution of $\{C(x, z) : x \in S\}$. Thus, to simplify the notation, we may fix an arbitrary $z \in U - S$ and simply use $\Pr(\mathcal{F})$ instead of $\Pr(\mathcal{F}(z))$ to denote the false positive probability, and we may use $\{C(x) : x \in S\}$ instead of $\{C(x, z) : x \in S\}$ to denote the joint probability distribution of the multi-sets of hash collisions of elements of S with z . Furthermore, since $\|H(u)\| = k$ for every $u \in U$, we can assign every element of $H(u)$ a unique number in $[k]$ (treating multiple instances of the same hash location as distinct elements). More formally, we define an arbitrary bijection f_M from M to $[k]$ for every multi-set $M \subseteq H$ with $\|M\| = k$ (where f_M treats multiple instances of the same hash location in M as distinct elements), and label the elements of $H(u)$ according to $f_{H(u)}$ by defining $H_i(u) = f_{H(u)}^{-1}(i)$ and writing $H(u) = (H_0(u), \dots, H_{k-1}(u))$.

The main technical result of this section is the following key theorem, which is a formalization and generalization of the analysis of the asymptotic false positive probability in Section 3.2.

Theorem 3.3.1. *Fix a hashing scheme. Suppose that there are constants λ and k and functions $\gamma_1(n) = o(1/n)$ and $\gamma_2(n) = o(1)$ such that:*

1. $\{H(u) : u \in U\}$ are independent and identically distributed.

2. For $u \in U$, $\|H(u)\| = k$.

3. For $x \in S$, $\Pr(\|C(x)\| = i) = \begin{cases} 1 - \frac{\lambda}{n} + O(\gamma_1(n)) & i = 0 \\ \frac{\lambda}{n} + O(\gamma_1(n)) & i = 1 \\ O(\gamma_1(n)) & i > 1 \end{cases} .$

4. For $x \in S$, $\max_{i \in H} |\Pr(i \in C(x) \mid \|C(x)\| = 1, i \in H(z)) - \frac{1}{k}| = O(\gamma_2(n))$.

Then $\lim_{n \rightarrow \infty} \Pr(\mathcal{F}) = (1 - e^{-\lambda/k})^k$.

Remark 3.3.1. It is not difficult to verify that the hashing scheme analyzed in Section 3.2 satisfies the conditions of Theorem 3.3.1 with $\lambda = k^2/c$. However, more complicated hashing schemes are not so amenable to a direct application of Theorem 3.3.1. Thus, after proving the theorem, we give a result that identifies another set of conditions that imply the hypotheses of Theorem 3.3.1 and are easier to verify. Indeed, the assumption on the

marginal distributions of the $C(x)$'s is satisfied if, for each $u \in U$, the random variables $H_1(u), \dots, H_k(u)$ are approximately pairwise independent and uniform over H ; we present a formalization of this idea in the next result, Lemma 3.3.1.

Proof. For $i \in [k]$ and $x \in S$, define $X_i(x) = 1$ if $H_i(z) \in C(x)$ and 0 otherwise, and define $X_i = \sum_{x \in S} X_i(x)$. We show that $X^n \triangleq (X_0, \dots, X_{k-1})$ converges in distribution to a vector $P \triangleq (P_0, \dots, P_{k-1})$ of k independent Poisson random variables with parameter λ/k , as $n \rightarrow \infty$. To do this, we make use of moment generating functions. For a random variable R , the moment generating function of R is defined by $M_R(t) \triangleq \mathbf{E}[\exp(tR)]$. We show that for any t_0, \dots, t_{k-1} and t ,

$$\lim_{n \rightarrow \infty} M_{\sum_{i=0}^{k-1} t_i X_i}(t) = M_{\sum_{i=0}^{k-1} t_i P_i}(t),$$

which is sufficient by [4, Theorem 29.4 and p. 390], since

$$\begin{aligned} M_{\sum_{i=0}^{k-1} t_i P_i}(t) &= \mathbf{E} \left[e^{t \sum_{i \in [k]} t_i P_i} \right] \\ &= \prod_{i \in [k]} \mathbf{E} \left[e^{t t_i \text{Poisson}(\lambda/k)} \right] \\ &= \prod_{i \in [k]} \sum_{j=0}^{\infty} e^{-\lambda/k} \frac{\lambda^j}{k^j j!} e^{t t_i j} \\ &= \prod_{i \in [k]} e^{\frac{\lambda}{k} (e^{t t_i} - 1)} \\ &= e^{\frac{\lambda}{k} (\sum_{i \in [k]} e^{t t_i} - 1)} < \infty, \end{aligned}$$

where the first step is just the definition of the moment generating function, the second step follows from independence of the $t_i P_i(\lambda/k)$'s, the third step is just the definition of the Poisson distribution, the fourth step follows from the Taylor series for e^x , and the fifth step is obvious.

Proceeding, we write

$$\begin{aligned} &M_{\sum_{i \in [k]} t_i X_i}(t) \\ &= M_{\sum_{i \in [k]} t_i \sum_{x \in S} X_i(x)}(t) \\ &= M_{\sum_{x \in S} \sum_{i \in [k]} t_i X_i(x)}(t) \\ &= \left(M_{\sum_{i \in [k]} t_i X_i(x)}(t) \right)^n \\ &= \left(\Pr(\|C(x)\| = 0) \right. \\ &\quad \left. + \sum_{j=1}^k \Pr(\|C(x)\| = j) \sum_{T \subseteq [k]: |T|=j} \Pr(C(x) = f_{H(z)}^{-1}(T) \mid \|C(x)\| = j) e^{t \sum_{i \in T} t_i} \right)^n \\ &= \left(1 - \frac{\lambda}{n} + \frac{\lambda}{n} \sum_{i \in [k]} \Pr(H_i(z) \in C(x) \mid \|C(x)\| = 1) e^{t t_i} + o(1/n) \right)^n \end{aligned}$$

$$\begin{aligned}
&= \left(1 - \frac{\lambda}{n} + \frac{\lambda}{n} \sum_{i \in [k]} \left(\frac{1}{k} + o(1) \right) e^{tt_i} + o(1/n) \right)^n \\
&= \left(1 - \frac{\lambda}{n} + \frac{\lambda \sum_{i \in [k]} e^{tt_i}}{kn} + o(1/n) \right)^n \\
&\rightarrow e^{-\lambda + \frac{\lambda}{k} \sum_{i \in [k]} e^{tt_i}} \quad \text{as } n \rightarrow \infty \\
&= e^{\frac{\lambda}{k} (\sum_{i \in [k]} (e^{tt_i} - 1))} \\
&= M_{\sum_{i \in [k]} t_i \text{Poisson}_i(\lambda/k)}(t).
\end{aligned}$$

The first two steps are obvious. The third step follows from the fact that the $H(x)$'s are independent and identically distributed (for $x \in S$) conditioned on $H(z)$, so the $\sum_{i \in [k]} t_i X_i(x)$'s are too, since each is a function of the corresponding $H(x)$. The fourth step follows from the definition of the moment generating function. The fifth and sixth steps follow from the assumptions on the distribution of $C(x)$ (in the sixth step, the conditioning on $i \in H(z)$ is implicit in our convention that associates integers in $[k]$ with the elements of $H(z)$). The seventh, eighth, and ninth steps are obvious, and the tenth step follows from a previous computation.

Now fix some bijection $g : \mathbb{Z}_{\geq 0}^k \rightarrow \mathbb{Z}_{\geq 0}$, and define $h : \mathbb{Z}_{\geq 0} \rightarrow \{0, 1\} : h(x) = 1$ if and only if every coordinate of $g^{-1}(x)$ is greater than 0. Since $\{X^n\}$ converges to P in distribution, $\{g(X^n)\}$ converges to $g(P)$ in distribution, because g is a bijection and X^n and P have discrete distributions. Skorohod's Representation Theorem [4, Theorem 25.6] now implies that there is some probability space where one may define random variables $\{Y_n\}$ and P' , where $Y_n \sim g(X^n)$ and $P' \sim g(P)$, and $\{Y_n\}$ converges to P' almost surely. Of course, since the Y_n 's only take integer values, whenever $\{Y_n\}$ converges to P' , there must be some n_0 such that $Y_{n_0} = Y_{n_1} = P'$ for any $n_1 > n_0$, and so $\{h(Y_n)\}$ trivially converges to $h(P')$. Therefore, $\{h(Y_n)\}$ converges to $h(P')$ almost surely, so

$$\begin{aligned}
\mathbf{Pr}(\mathcal{F}) &= \mathbf{Pr}(\forall i \in [k], X_i > 0) \\
&= \mathbf{E}[h(g(X^n))] \\
&= \mathbf{E}[h(Y_n)] \\
&\rightarrow \mathbf{E}[h(P')] \quad \text{as } n \rightarrow \infty \\
&= \mathbf{Pr}(\text{Poisson}(\lambda/k) > 0)^k \\
&= \left(1 - e^{-\lambda/k} \right)^k,
\end{aligned}$$

where the fourth step is the only nontrivial one, and it follows from [4, Theorem 5.4]. \square

It turns out that the conditions of Theorem 3.3.1 can be verified very easily in many cases.

Lemma 3.3.1. *Fix a hashing scheme. Suppose that there are constants λ and k and a function $\gamma(n) = o(1/n)$ such that:*

1. $\{H(u) : u \in U\}$ are independent and identically distributed.

2. For $u \in U$, $\|H(u)\| = k$.
3. For $u \in U$, $\max_{i \in H} |\Pr(i \in H(u)) - \frac{\lambda}{kn}| = O(\gamma(n))$.
4. For $u \in U$, $\max_{i_1, i_2 \in H} \Pr(i_1, i_2 \in H(u)) = O(\gamma(n))$.
5. The set of all possible hash locations H satisfies $|H| = O(n)$.

Then the conditions of Theorem 3.3.1 hold with $\gamma_1(n) = \gamma(n)$ and $\gamma_2(n) = n\gamma(n)$ (and the same values for λ and k).

Remark 3.3.2. Recall that, under our notation, the statement $i, i \in H(u)$ is true if and only if i is an element of $H(u)$ of multiplicity at least 2.

Proof. The proof is essentially just a number of applications of the first two Boole-Bonferroni inequalities (e.g., [45, Proposition C.2]).

The first two conditions of Theorem 3.3.1 are trivially satisfied. For the third condition, observe that for any $j \in \{2, \dots, k\}$ and $x \in S$,

$$\begin{aligned}
& \Pr(\|C(x)\| = j) \\
& \leq \Pr(\|C(x)\| > 1) \\
& = \Pr(\exists i_1 < i_2 \in [k] : H_{i_1}(z), H_{i_2}(z) \in H(x) \text{ or } \exists i \in H : i \in H(x), i, i \in H(z)) \\
& \leq \Pr(\exists i_1 < i_2 \in [k] : H_{i_1}(z), H_{i_2}(z) \in H(x)) \\
& \quad + \Pr(\exists i \in H : i \in H(x), i, i \in H(z)) \\
& \leq \sum_{i_1 < i_2 \in [k]} \Pr(H_{i_1}(z), H_{i_2}(z) \in H(x)) + \sum_{i \in H} \Pr(i \in H(x)) \Pr(i, i \in H(z)) \\
& \leq \binom{k}{2} O(\gamma(n)) + |H| \left(\frac{\lambda}{kn} + O(\gamma(n)) \right) O(\gamma(n)) \\
& = O(\gamma(n)) + O(n) O(\gamma(n)/n) \\
& = O(\gamma(n)),
\end{aligned}$$

and

$$\begin{aligned}
\Pr(\|C(x)\| = 1) & \leq \Pr(|C(x)| \geq 1) \\
& \leq \sum_{i \in [k]} \Pr(H_i(z) \in H(x)) \leq k \left(\frac{\lambda}{kn} + O(\gamma(n)) \right) = \frac{\lambda}{n} + O(\gamma(n)),
\end{aligned}$$

and

$$\begin{aligned}
& \Pr(\|C(x)\| \geq 1) \\
& = \Pr\left(\bigcup_{i \in [k]} H_i(z) \in H(x)\right) \\
& \geq \sum_{i \in [k]} \Pr(H_i(z) \in H(x)) - \sum_{i_1 < i_2 \in [k]} \Pr(H_{i_1}(z) \in H(x), H_{i_2}(z) \in H(x))
\end{aligned}$$

$$\begin{aligned}
&\geq \sum_{i \in [k]} \Pr(H_i(z) \in H(x)) \\
&\quad - \left(\sum_{i_1 < i_2 \in [k]} \Pr(H_{i_1}(z), H_{i_2}(z) \in H(x)) + \sum_{i \in H} \Pr(i \in H(x)) \Pr(i, i \in H(z)) \right) \\
&\geq k \left(\frac{\lambda}{kn} + O(\gamma(n)) \right) - O(\gamma(n)) \\
&= \frac{\lambda}{n} + O(\gamma(n)),
\end{aligned}$$

so

$$\begin{aligned}
\Pr(\|C(x)\| = 1) &= \Pr(\|C(x)\| \geq 1) - \Pr(\|C(x)\| > 1) \\
&\geq \frac{\lambda}{n} + O(\gamma(n)) - O(\gamma(n)) \\
&= \frac{\lambda}{n} + O(\gamma(n)).
\end{aligned}$$

Therefore,

$$\Pr(\|C(x)\| = 1) = \frac{\lambda}{n} + O(\gamma(n)),$$

and

$$\Pr(\|C(x)\| = 0) = 1 - \sum_{j=1}^k \Pr(\|C(x)\| = j) = 1 - \frac{\lambda}{n} + O(\gamma(n)).$$

We have now shown that the third condition of Theorem 3.3.1 is satisfied.

For the fourth condition, we observe that for any $i \in [k]$ and $x \in S$,

$$\Pr(H_i(z) \in C(x), \|C(x)\| = 1) \leq \Pr(H_i(z) \in H(x)) = \frac{\lambda}{kn} + O(\gamma(n)),$$

and

$$\begin{aligned}
\Pr(H_i(z) \in C(x), \|C(x)\| = 1) &= \Pr(H_i(z) \in H(x)) - \Pr(H_i(z) \in H(x), \|C(x)\| > 1) \\
&\geq \Pr(H_i(z) \in H(x)) - \Pr(\|C(x)\| > 1) \\
&= \frac{\lambda}{kn} + O(\gamma(n)) - O(\gamma(n)),
\end{aligned}$$

so

$$\Pr(H_i(z) \in C(x), \|C(x)\| = 1) = \frac{\lambda}{kn} + O(\gamma(n)),$$

implying that

$$\begin{aligned}
\Pr(H_i(z) \in C(x) \mid \|C(x)\| = 1) &= \frac{\Pr(H_i(z) \in C(x), \|C(x)\| = 1)}{\Pr(\|C(x)\| = 1)} \\
&= \frac{\frac{\lambda}{kn} + O(\gamma(n))}{\frac{\lambda}{n} + O(\gamma(n))} = \frac{1}{k} + O(n\gamma(n)).
\end{aligned}$$

completing the proof. (It is easy to see that the above equation is equivalent to the fourth condition in the proof of Theorem 3.3.1, as the conditioning on $i \in H(z)$ is once again implied by the convention that associates elements of $[k]$ with the hash locations in $H(z)$.) \square

3.4 Some Specific Hashing Schemes

We are now ready to analyze some specific hashing schemes. In particular, we examine a natural generalization of the scheme described in Section 3.2, as well as the double hashing and enhanced double hashing schemes introduced in [19, 20]. In both of these cases, we consider a Bloom filter consisting of an array of $m = cn$ bits and k hash functions, where $c > 0$ and $k \geq 1$ are fixed constants. The nature of the hash functions depends on the particular hashing scheme under consideration.

3.4.1 Partition Schemes

First, we consider the class of *partition schemes*, where the Bloom filter is defined by an array of m bits that is partitioned into k disjoint arrays of $m' = m/k$ bits (we require that m be divisible by k), and an item $u \in U$ is hashed to location

$$h_1(u) + ih_2(u) \bmod m'$$

of array i , for $i \in [k]$, where h_1 and h_2 are independent fully random hash functions with codomain $[m']$. Note that the hashing scheme analyzed in Section 3.2 is a partition scheme where m' is prime (and so is denoted by p in Section 3.2).

Unless otherwise stated, henceforth we do all arithmetic involving h_1 and h_2 modulo m' . We prove the following theorem concerning partition schemes.

Theorem 3.4.1. *For a partition scheme, $\lim_{n \rightarrow \infty} \Pr(\mathcal{F}) = (1 - e^{-k/c})^k$.*

Proof. We show that the $H(u)$'s satisfy the conditions of Lemma 3.3.1 with $\lambda = k^2/c$ and $\gamma(n) = 1/n^2$. For $i \in [k]$ and $u \in U$, define $g_i(u) = (i, h_1(u) + ih_2(u))$ and $H(u) = (g_i(u) : i \in [k])$. That is, $g_i(u)$ is u 's i th hash location, and $H(u)$ is the multi-set of u 's hash locations. This notation is obviously consistent with the definitions required by Lemma 3.3.1.

Since h_1 and h_2 are independent and fully random, the first two conditions are trivial. The last condition is also trivial, since there are $m = cn$ possible hash locations. For the remaining two conditions, fix $u \in U$. Observe that for $(i, r) \in [k] \times [m']$,

$$\Pr((i, r) \in H(u)) = \Pr(h_1(u) = r - ih_2(u)) = 1/m' = (k^2/c)/kn,$$

and that for distinct $(i_1, r_1), (i_2, r_2) \in [k] \times [m']$, we have

$$\begin{aligned}
& \Pr((i_1, r_1), (i_2, r_2) \in H(u)) \\
&= \Pr(i_1 \in H(u)) \Pr(i_2 \in H(u) \mid i_1 \in H(u)) \\
&= \frac{1}{m'} \Pr(h_1(u) = r_2 - i_2 h_2(u) \mid h_1(u) = r_1 - i_1 h_2(u)) \\
&= \frac{1}{m'} \Pr((i_1 - i_2) h_2(u) = r_1 - r_2) \\
&\leq \frac{1}{m'} \cdot \frac{\gcd(|i_2 - i_1|, m')}{m'} \leq \frac{k}{(m')^2} = O(1/n^2),
\end{aligned}$$

where the fourth step is the only nontrivial step, and it follows from the standard fact that for any $r, s \in [m]$, there are at most $\gcd(r, m)$ values $t \in [m]$ such that $rt \equiv s \pmod{m}$ (see, for example, [29, Proposition 3.3.1]). Finally, since it is clear from the definition of the hashing scheme that $|H(u)| = k$ for all $u \in U$, we have that for any $(i, r) \in [k] \times [m']$, $\Pr((i, r), (i, r) \in H(u)) = 0$. \square

3.4.2 (Enhanced) Double Hashing Schemes

Next, we consider the class of double hashing and enhanced double hashing schemes, which are analyzed empirically in [19, 20]. In these hashing schemes, an item $u \in U$ is hashed to location

$$h_1(u) + ih_2(u) + f(i) \pmod{m}$$

of the array of m bits, for $i \in [k]$, where h_1 and h_2 are independent fully random hash functions with codomain $[m]$, and $f : [k] \rightarrow [m]$ is an arbitrary function. When $f(i) \equiv 0$, the hashing scheme is called a *double hashing scheme*. Otherwise, it is called an *enhanced double hashing scheme (with f)*. We show that the asymptotic false positive probability for an (enhanced) double hashing scheme is the same as for a standard Bloom filter.

Theorem 3.4.2. *For any (enhanced) double hashing scheme,*

$$\lim_{n \rightarrow \infty} \Pr(\mathcal{F}) = \left(1 - e^{-k/c}\right)^k.$$

Remark 3.4.1. The result holds for any choice of f . In fact, f can even be drawn from an arbitrary probability distribution over $[m]^{[k]}$, as long as it is drawn independently of the two random hash functions h_1 and h_2 .

Proof. We proceed by showing that this hashing scheme satisfies the conditions of Lemma 3.3.1 with $\lambda = k^2/c$ and $\gamma(n) = 1/n^2$. Since h_1 and h_2 are independent and fully random, the first two conditions trivially hold. The last condition is also trivial, since there are $m = cn$ possible hash locations.

Showing that the third and fourth conditions hold requires more effort. First, we need some notation. For $u \in U$, $i \in [k]$, define

$$\begin{aligned}
g_i(u) &= h_1(u) + ih_2(u) + f(i) \\
H(u) &= (g_i(u) : i \in [k]).
\end{aligned}$$

That is, $g_i(u)$ is u 's i th hash location, and $H(u)$ is the multi-set of u 's hash locations. This notation is obviously consistent with the definitions required by Lemma 3.3.1. Fix $u \in U$. For $r \in [m]$,

$$\Pr(\exists j \in [k] : g_j(u) = r) \leq \sum_{j \in [k]} \Pr(h_1(u) = r - jh_2(u) - f(j)) = \frac{k}{m}.$$

Furthermore, for any $j_1, j_2 \in [k]$ and $r_1, r_2 \in [m]$

$$\begin{aligned} \Pr(g_{j_1}(u) = r_1, g_{j_2}(u) = r_2) &= \Pr(g_{j_1}(u) = r_1) \Pr(g_{j_2}(u) = r_2 \mid g_{j_1}(u) = r_1) \\ &= \frac{1}{m} \Pr(g_{j_2}(u) = r_2 \mid g_{j_1}(u) = r_1) \\ &= \frac{1}{m} \Pr((j_1 - j_2)h_2(u) = r_1 - r_2 + f(j_2) - f(j_1)) \\ &\leq \frac{1}{m} \cdot \frac{\gcd(|j_1 - j_2|, m)}{m} \\ &\leq \frac{1}{m} \cdot \frac{k}{m} \\ &= \frac{k}{m^2} \\ &= O(1/n^2), \end{aligned}$$

where the fourth step is the only nontrivial step, and it follows from the standard fact that for any $r, s \in [m]$, there are at most $\gcd(r, m)$ values $t \in [m]$ such that $rt \equiv s \pmod{m}$ (see, for example, [29, Proposition 3.3.1]). Therefore, for $r \in [m]$,

$$\begin{aligned} \Pr(\exists j \in [k] : g_j(u) = r) &\geq \sum_{j \in [k]} \Pr(g_j(u) = r) - \sum_{j_1 < j_2 \in [k]} \Pr(g_{j_1}(u) = r, g_{j_2}(u) = r) \\ &\geq \frac{k}{m} - k^2 O(1/n^2) \\ &= \frac{k^2/c}{n} + O(1/n^2), \end{aligned}$$

implying that

$$\Pr(r \in H(u)) = \Pr(\exists j \in [k] : g_j(u) = r) = \frac{k^2/c}{n} + O(1/n^2),$$

so the third condition of Lemma 3.3.1 holds. For the fourth condition, fix any $r_1, r_2 \in [m]$. Then

$$\Pr(r_1, r_2 \in H(u)) \leq \sum_{j_1, j_2 \in [k]} \Pr(g_{j_1}(u) = r_1, g_{j_2}(u) = r_2) \leq k^2 O(1/n^2) = O(1/n^2),$$

completing the proof. \square

3.5 Rate of Convergence

In the previous sections, we identified a broad class of non-standard Bloom filter hashing schemes that have the same asymptotic false positive probability as a standard Bloom filter. For many applications, we would also like to know that these asymptotics kick in fairly quickly, for reasonable values of n . With these applications in mind, we provide an analysis of the rate of convergence in Theorem 3.3.1, and then apply that analysis to the specific hashing schemes discussed in Section 3.4. Our results indicate that those hashing schemes yield performance almost identical to that of a standard Bloom filter for a wide range of practical settings. Furthermore, in Section 3.7, we show the results of some simple experiments as further evidence of this fact.

The rate of convergence analysis proceeds along the following lines, where the underlying intuition is drawn from the analysis of the asymptotic false positive probability in Section 3.2, and we assume the hypotheses of Lemma 3.3.1. First, for each $x \in S$, we couple $\|C(x)\|$ with a Bernoulli(λ/n) random variable B_x (where Bernoulli(\cdot) denotes the Bernoulli distribution). (We specify exactly how to do the coupling later.) We then define a Binomial($n, \lambda/n$) random variable $B = \sum_{x \in S} B_x$. In the terminology of Section 3.2, each $x \in S$ represents a ball that is discarded if and only if $B_x = 0$, so B is the total number of balls that are not discarded. Next, conditioned on $T = \{x : B_x = 1\}$, for each $x \in T$, we couple the smallest element C_x of $f_{H(z)}(C(x))$ with a random variable T_x selected uniformly from $[k]$ (recall from the proof of Theorem 3.3.1 that $f_{H(z)}$ defines a correspondence between $H(z)$ and $[k]$). (In the asymptotically insignificant case where $\|C(x)\| = 0$, we simply define $C_x = -1$.) In the terminology of Section 3.2, T is the set of balls that are thrown into the bins, and for each $x \in T$, the random variable T_x represents the bin where it lands.

We can now bound the difference between the false positive probability (for a particular value of n) and the probability that every bin in $[k]$ is hit by at least one ball by the probability that at least one of the random variables just defined is different than the random variable to which it is coupled. Thus, we relate the true false positive probability to the same simple balls and bins experiment as in Section 3.2. Finally, as in Section 3.2, the asymptotics of the balls and bins experiment are easy to analyze; we just couple B with a Poisson(λ) random variable Y and bound $\Pr(B \neq Y)$. (This is because, for both the experiment where B balls are thrown (that is, not discarded) and the experiment where Poisson(λ) balls are thrown, each ball is placed in a bin that is chosen randomly from the k bins, so for each ball that is thrown in both experiments, the random variables indicating the bins where it lands in the two experiments can be trivially coupled.)

We now formalize the above argument. In particular, we obtain rate of convergence results that subsume many of the results in Sections 3.3 and 3.4. However, we have chosen to keep our earlier results because they demonstrate that the underlying Poisson convergence of interest can be cleanly derived using a standard moment generating function approach.

Before proceeding, we define the *total variation distance* between two discrete distributions (or random variables) X_1 and X_2 to be

$$\text{dist}(X_1, X_2) = \sum_{x \in \text{Supp}(X_1) \cup \text{Supp}(X_2)} |\Pr(X_1 = x) - \Pr(X_2 = x)|.$$

Also, if X_1 and X_2 are jointly distributed random variables, then we use the notation

$X_1 \mid X_2$ to refer to the conditional distribution of X_1 given X_2 .

Theorem 3.5.1. *Consider a hashing scheme that satisfies the hypotheses of Lemma 3.3.1. Then*

$$\left| \Pr(\mathcal{F}) - \left(1 - e^{-\lambda/k}\right)^k \right| = O(n\gamma(n) + 1/n).$$

Proof. With the above outline in mind, define the events

$$\begin{aligned} \mathcal{E}_1 &= \{\forall x \in S \ \|C(x)\| = B_x\} \\ \mathcal{E}_2 &= \{\forall x \in T \ C_x = T_x\}. \end{aligned}$$

Let \mathcal{F}' denote the event that in the experiment where Y balls are thrown randomly into k bins, each bin receives at least one ball. Since $Y \sim \text{Poisson}(\lambda)$, a standard fact of Poisson random variables tells us that the joint distribution of number of balls in each bin is the same as the joint distribution of k independent $\text{Poisson}(\lambda/k)$ random variables. Thus, $\Pr(\mathcal{F}') = (1 - \exp[-\lambda/k])^k$. Letting $\mathbf{1}(\cdot)$ denote the indicator function, we write

$$\begin{aligned} & \left| \Pr(\mathcal{F}) - (1 - \exp[-\lambda/k])^k \right| \\ &= \left| \Pr(\mathcal{F}) - \Pr(\mathcal{F}') \right| \\ &= \left| \mathbf{E} [\mathbf{1}(\mathcal{F}) - \mathbf{1}(\mathcal{F}')] \right| \\ &\leq \mathbf{E} [|\mathbf{1}(\mathcal{F}) - \mathbf{1}(\mathcal{F}')|] \\ &= \Pr(\mathbf{1}(\mathcal{F}) \neq \mathbf{1}(\mathcal{F}')) \\ &\leq \Pr(\neg\mathcal{E}_1 \cup \neg\mathcal{E}_2 \cup (B \neq Y)) \\ &\leq \Pr(\neg\mathcal{E}_1 \cup \neg\mathcal{E}_2) + \Pr(B \neq Y) \\ &= \mathbf{E}[\Pr(\neg\mathcal{E}_1 \cup \neg\mathcal{E}_2 \mid H(z))] + \Pr(B \neq Y) \\ &= \mathbf{E}[\Pr(\neg\mathcal{E}_1 \mid H(z)) + \Pr(\mathcal{E}_1 \mid H(z))\Pr(\neg\mathcal{E}_2 \mid \mathcal{E}_1, H(z))] + \Pr(B \neq Y) \end{aligned}$$

where we have used the fact that if $\mathbf{1}(\mathcal{F}) \neq \mathbf{1}(\mathcal{F}')$, then some random variable in the coupling established above is not equal to the random variable to which it is coupled.

Before continuing, we must address the issue of actually constructing the couplings described above. Of course, our goal is to find a coupling so that random variables with similar distributions are likely to be equal in the probability space where they are coupled. To construct the coupling, we fix some ordering x_1, \dots, x_n of the elements in S . We first sample $H(z)$ and then sample independent $\text{Uniform}[0, 1]$ random variables U_1, \dots, U_{2n+1} . For $i = 1, \dots, n$, we use U_i to perform a coupling between B_{x_i} and the conditional distribution of $\|C(x_i)\|$ given $H(z)$ (we specify exactly how we do this later). If $B_{x_i} = 1$, then we use U_{i+n} to perform a coupling between T_{x_i} and the conditional distribution of C_{x_i} given $\|C(x_i)\|$ and $H(z)$. Note that here we are using the fact that all of the $C(x)$'s are conditionally independent given $H(z)$, so these pairwise couplings are consistent with the appropriate joint distribution of the random variables. Finally, we use U_{2n+1} and the already sampled value of B to sample Y ; this gives the coupling between B and Y . As for how we construct the pairwise couplings, it follows from standard facts (see, for example, [28, Exercise 4.12.5]) that we can construct a coupling of any pair of distributions X_1 and

X_2 so that $\Pr(X_1 \neq X_2) = \frac{1}{2}\text{dist}(X_1, X_2)$ by representing X_2 as function of X_1 and a Uniform(0, 1) random variable that is independent of X_1 . We perform all of our pairwise couplings in this way.

Now we define

$$\begin{aligned} Z_1 &= \Pr(\|C(x)\| \neq B_x \mid H(z)) \\ Z_2 &= \Pr(C_x \neq T_x \mid x \in T, \mathcal{E}_1, H(z)) \end{aligned}$$

for some $x \in S$; note that the choice of x does not matter. A union bound now gives $\Pr(\neg\mathcal{E}_1 \mid H(z)) \leq nZ_1$, and another union bound gives

$$\Pr(\exists x \in T : C_x \neq T_x \mid |T|, \mathcal{E}_1, H(z)) \leq |T|Z_2.$$

Therefore,

$$\begin{aligned} \Pr(\neg\mathcal{E}_2 \mid \mathcal{E}_1, H(z)) &= \mathbf{E}[\Pr(\exists x \in T : f_{H(z)}(C_x) \neq T_x \mid |T|, \mathcal{E}_1, H(z)) \mid \mathcal{E}_1, H(z)] \\ &\leq \mathbf{E}[|T| \mid \mathcal{E}_1, H(z)]Z_2 \\ &\leq \frac{\mathbf{E}[|T| \mid H(z)]Z_2}{\Pr(\mathcal{E}_1 \mid H(z))}. \end{aligned}$$

Combining these results gives

$$\begin{aligned} &\mathbf{E}[\Pr(\neg\mathcal{E}_1 \mid H(z)) + \Pr(\mathcal{E}_1 \mid H(z))\Pr(\neg\mathcal{E}_2 \mid \mathcal{E}_1, H(z))] \\ &\leq n\mathbf{E}[Z_1] + \mathbf{E}[|T|Z_2] \\ &= n\mathbf{E}[Z_1] + \mathbf{E}[|T|]\mathbf{E}[Z_2] \\ &= n\mathbf{E}[Z_1] + \lambda\mathbf{E}[Z_2], \end{aligned}$$

where we have used the fact that T and $H(z)$ are independent in our coupling, which implies that T and Z_2 are independent (since Z_2 is a function of $H(z)$).

As for the coupling between the Binomial($n, \lambda/n$) random variable B and the Poisson(λ) random variable Y , we have $\Pr(B \neq Y) = \frac{1}{2}\text{dist}(X, Y)$. A standard result (e.g. [28, Section 4.12]) tells us $\text{dist}(X, Y) \leq 2\lambda^2/n$. Therefore,

$$\left| \Pr(\mathcal{F}) - \left(1 - e^{-\lambda/k}\right)^k \right| = O(n\mathbf{E}[Z_1] + \mathbf{E}[Z_2] + 1/n).$$

It remains to show that $\mathbf{E}[Z_1] = O(\gamma(n))$ and $\mathbf{E}[Z_2] = O(n\gamma(n))$. The proof technique is essentially the same as in Lemma 3.3.1. First, we write

$$\begin{aligned} \mathbf{E}[Z_1] &= \frac{1}{2}\mathbf{E}[\text{dist}(\text{Bernoulli}(\lambda/n), \|C(x)\| \mid H(z))] \\ &= \frac{1}{2}\left(\mathbf{E}[\Pr(\|C(x)\| = 0 \mid H(z)) - 1 + \lambda/n] + \mathbf{E}[\Pr(\|C(x)\| = 1 \mid H(z)) - \lambda/n] \right. \\ &\quad \left. + \Pr(\|C(x)\| \geq 2)\right) \end{aligned}$$

By Lemma 3.3.1, we have $\Pr(\|C(x)\| \geq 2) = O(\gamma(n))$. For the other two terms, we write

$$\begin{aligned} \Pr(\|C(x)\| \geq 1 \mid H(z)) &= \Pr\left(\bigcup_{i \in [k]} H_i(z) \in H(x) \mid H(z)\right) \\ &\leq \sum_{i \in [k]} \Pr(H_i(z) \in H(x) \mid H(z)) \\ &= \frac{\lambda}{n} + O(\gamma(n)) \end{aligned}$$

and

$$\begin{aligned} &\Pr(\|C(x)\| \geq 1 \mid H(z)) \\ &= \Pr\left(\bigcup_{i \in [k]} H_i(z) \in H(x) \mid H(z)\right) \\ &\geq \sum_{i \in [k]} \Pr(H_i(z) \in H(x) \mid H(z)) - \sum_{i_1 < i_2 \in [k]} \Pr(H_{i_1}(z) \in H(x), H_{i_2}(z) \in H(x)) \\ &\geq \sum_{i \in [k]} \Pr(H_i(z) \in H(x) \mid H(z)) \\ &\quad - \left(\sum_{i_1 < i_2 \in [k]} \Pr(H_{i_1}(z), H_{i_2}(z) \in H(x)) + \mathbf{1}(\|H(z)\| < k) \sum_{i \in H(z)} \Pr(i \in H(x)) \right) \\ &\geq \frac{\lambda}{n} + O(\gamma(n)) - \mathbf{1}(\|H(z)\| < k) \frac{\lambda}{n}. \end{aligned}$$

Therefore,

$$\begin{aligned} &\Pr(\|C(x)\| = 1 \mid H(z)) - \frac{\lambda}{n} \\ &\leq \Pr(\|C(x)\| \geq 1 \mid H(z)) - \frac{\lambda}{n} \\ &\leq O(\gamma(n)) \end{aligned}$$

and

$$\begin{aligned} &\Pr(\|C(x)\| = 1 \mid H(z)) - \frac{\lambda}{n} \\ &= \Pr(\|C(x)\| \geq 1 \mid H(z)) - \Pr(\|C(x)\| \geq 2 \mid H(z)) - \frac{\lambda}{n} \\ &\geq \Pr(\|C(x)\| \geq 1 \mid H(z)) + O(\gamma(n)) - \frac{\lambda}{n} \\ &\geq O(\gamma(n)) - \mathbf{1}(\|H(z)\| < k) \frac{\lambda}{n}. \end{aligned}$$

Thus,

$$\begin{aligned} \mathbf{E}[\|\mathbf{Pr}(\|C(x)\| = 1 \mid H(z)) - \lambda/n\|] &\leq O(\gamma(n)) + \frac{\lambda}{n} \mathbf{Pr}(|H(z)| < k) \\ &\leq O(\gamma(n)) + \frac{\lambda}{n} \sum_{i \in H} \mathbf{Pr}(i, i \in H(z)) \\ &= O(\gamma(n)). \end{aligned}$$

Also,

$$\begin{aligned} \mathbf{E}[\|\mathbf{Pr}(\|C(x)\| = 0 \mid H(z)) - 1 + \lambda/n\|] &= \mathbf{E}[\|\lambda/n - \mathbf{Pr}(\|C(x)\| \geq 1 \mid H(z))\|] \\ &\leq \mathbf{E}[O(\gamma(n)) + \mathbf{1}(|H(z)| < k)\lambda/n] \\ &= O(\gamma(n)) + \frac{\lambda}{n} \mathbf{Pr}(|H(z)| < k) \\ &\leq O(\gamma(n)) + \frac{\lambda}{n} \sum_{i \in H} \mathbf{Pr}(i, i \in H(z)) \\ &= O(\gamma(n)). \end{aligned}$$

We have now shown that $\mathbf{E}[Z_1] = O(\gamma(n))$.

To complete the proof, we show that $\mathbf{E}[Z_2] = O(n\gamma(n))$. Now,

$$\begin{aligned} \mathbf{E}[Z_2] &= \frac{1}{2} \mathbf{E}[\text{dist}(\text{Uniform}([k]), C_x \mid \|C(x)\| = 1, H(z))] \\ &= \frac{1}{2} \sum_{i \in [k]} \mathbf{E}[\|\mathbf{Pr}(C_x = i \mid \|C(x)\| = 1, H(z)) - 1/k\|] \\ &= \frac{1}{2} \sum_{i \in [k]} \mathbf{E}[\|\mathbf{Pr}(H_i(z) \in H(x) \mid \|C(x)\| = 1, H(z)) - 1/k\|]. \end{aligned}$$

We show that for $i \in [k]$, we have

$$\mathbf{E}[\|\mathbf{Pr}(H_i(z) \in H(x) \mid \|C(x)\| = 1, H(z)) - 1/k\|] = O(n\gamma(n)).$$

Indeed,

$$\begin{aligned} \mathbf{Pr}(H_i(z) \in H(x), \|C(x)\| = 1 \mid H(z)) &\leq \mathbf{Pr}(H_i(z) \in H(x) \mid H(z)) \\ &\leq \frac{\lambda}{kn} + O(\gamma(n)) \end{aligned}$$

and

$$\begin{aligned} &\mathbf{Pr}(H_i(z) \in H(x), \|C(x)\| = 1 \mid H(z)) \\ &\geq \mathbf{Pr}(H_i(z) \in H(x) \mid H(z)) - \mathbf{Pr}(\|C(x)\| \geq 2 \mid H(z)) \\ &\geq \frac{\lambda}{kn} + O(\gamma(n)). \end{aligned}$$

Furthermore, previous calculations give

$$|\Pr(\|C(x)\| = 1 \mid H(z), |H(z)| = k) - \lambda/n| = O(\gamma(n)),$$

and so

$$\begin{aligned} & \mathbf{E}[|\Pr(H_i(z) \in H(x) \mid \|C(x)\| = 1, H(z)) - 1/k| \mid |H(z)| = k] \\ &= \mathbf{E} \left[\left| \frac{\Pr(H_i(z) \in H(x), \|C(x)\| = 1 \mid H(z))}{\Pr(\|C(x)\| = 1 \mid H(z))} - \frac{1}{k} \right| \mid |H(z)| = k \right] \\ &= \mathbf{E} \left[\left| \frac{\frac{\lambda}{kn} + O(\gamma(n))}{\frac{\lambda}{n} + O(\gamma(n))} - \frac{1}{k} \right| \mid |H(z)| = k \right] \\ &= O(n\gamma(n)). \end{aligned}$$

Thus,

$$\begin{aligned} & \mathbf{E}[|\Pr(H_i(z) \in H(x) \mid \|C(x)\| = 1, H(z)) - 1/k|] \\ &\leq \Pr(|H(z)| < k) + \mathbf{E}[|\Pr(H_i(z) \in H(x) \mid \|C(x)\| = 1, H(z)) - 1/k| \mid |H(z)| = k] \\ &\leq \Pr(|H(z)| < k) + O(n\gamma(n)) \\ &= O(n\gamma(n)) + \sum_{j \in H} \Pr(j, j \in H(z)) \\ &= O(n\gamma(n)), \end{aligned}$$

completing the proof. \square

We now use Theorem 3.5.1 to bound the rate of convergence in Theorems 3.4.1 and 3.4.2.

Theorem 3.5.2. *For any partition or (enhanced) double hashing scheme,*

$$\left| \Pr(\mathcal{F}) - \left(1 - e^{-\lambda/k}\right)^k \right| = O(1/n).$$

Proof. In the proofs of Theorems 3.4.1 and 3.4.2, we show that all of these hashing schemes satisfy the conditions of Lemma 3.3.1 with $\gamma(n) = 1/n^2$. Theorem 3.5.1 now gives the desired result. \square

3.6 Multiple Queries

In the previous sections, we analyzed the behavior of $\Pr(\mathcal{F}(z))$ for some fixed z and moderately sized n . Unfortunately, this quantity is not directly of interest in most applications. Instead, one is usually concerned with certain characteristics of the distribution of the number of, say, $z_1, \dots, z_\ell \in U - S$ for which $\mathcal{F}(z)$ occurs. In other words, rather than being interested in the probability that a particular false positive occurs, we are concerned with, for example, the fraction of distinct queries on elements of $U - S$ posed to the filter

for which it returns false positives. Since $\{\mathcal{F}(z) : z \in U - S\}$ are not independent, the behavior of $\mathbf{Pr}(\mathcal{F})$ alone does not directly imply results of this form. This section is devoted to overcoming this difficulty.

Now, it is easy to see that in the hashing schemes that we analyze here, once the hash locations for every $x \in S$ have been determined, the events $\{\mathcal{F}(z) : z \in U - S\}$ are independent and occur with equal probability. More formally, $\{\mathbf{1}(\mathcal{F}(z)) : z \in U - S\}$ are conditionally independent and identically distributed given $\{H(x) : x \in S\}$. Thus, conditioned on $\{H(x) : x \in S\}$, an enormous number of classical convergence results (e.g., the law of large numbers and the central limit theorem) can be applied to $\{\mathbf{1}(\mathcal{F}(z)) : z \in U - S\}$.

These observations motivate a general technique for deriving the sort of convergence results for $\{\mathbf{1}(\mathcal{F}(z)) : z \in U - S\}$ that one might desire in practice. First, we show that with high probability over the set of hash locations used by elements of S (that is, $\{H(x) : x \in S\}$), the random variables $\{\mathbf{1}(\mathcal{F}(z)) : z \in U - S\}$ are essentially independent Bernoulli(p) random variables, for $p \triangleq \lim_{n \rightarrow \infty} \mathbf{Pr}(\mathcal{F})$. From a technical standpoint, this result is the most important in this section. Next, we show how to use that result to prove counterparts to the classical convergence theorems mentioned above that hold in our setting.

Proceeding formally, we begin with a critical definition.

Definition 3.6.1. Consider any hashing scheme where $\{H(u) : u \in U\}$ are independent and identically distributed. Write $S = \{x_1, \dots, x_n\}$. The *false positive rate* is defined to be the random variable

$$R = \mathbf{Pr}(\mathcal{F} \mid H(x_1), \dots, H(x_n)).$$

The false positive rate gets its name from the fact that, conditioned on R , the random variables $\{\mathbf{1}(\mathcal{F}(z)) : z \in U - S\}$ are independent Bernoulli(R) random variables. Thus, the fraction of a large number of queries on elements of $U - S$ posed to the filter for which it returns false positives is very likely to be close to R . In this sense, R , while a random variable, acts like a rate for $\{\mathbf{1}(\mathcal{F}(z)) : z \in U - S\}$.

It is important to note that in much of literature concerning standard Bloom filters, the false positive rate is not defined as above. Instead the term is often used as a synonym for the false positive probability. Indeed, for a standard Bloom filter, the distinction between the two concepts as we have defined them is unimportant in practice, since, as mentioned in Section 2.1.2, one can easily show that R is very close to $\mathbf{Pr}(\mathcal{F})$ with extremely high probability (see, for example, [38]). It turns out that this result generalizes very naturally to the framework presented in this chapter, and so the practical difference between the two concepts turns out to be largely unimportant even in our very general setting. However, the proof is more complicated than in the case of a standard Bloom filter, and so we will be very careful to use the terms as we have defined them.

Theorem 3.6.1. Consider a hashing scheme where the conditions of Lemma 3.3.1 hold. Furthermore, assume that there is some function g and independent identically distributed random variables $\{V_u : u \in U\}$, each of which is uniformly distributed over some finite set

V , such that for $u \in U$ we have $H(u) = g(V_u)$. Define

$$\begin{aligned} p &= \left(1 - e^{-\lambda/k}\right)^k \\ \Delta &= \max_{i \in H} \Pr(i \in H(u)) - \frac{\lambda}{nk} \quad (= o(1/n)) \\ \xi &= nk\Delta(2\lambda + k\Delta) \quad (= o(1)) \end{aligned}$$

Then for any $\epsilon = \epsilon(n) > 0$ with $\epsilon = \omega(|\Pr(\mathcal{F}) - p|)$, for n sufficiently large so that $\epsilon > |\Pr(\mathcal{F}) - p|$,

$$\Pr(|R - p| > \epsilon) \leq 2 \exp \left[\frac{-2n(\epsilon - |\Pr(\mathcal{F}) - p|)^2}{\lambda^2 + \xi} \right].$$

Furthermore, for any function $h(n) = o(\min(1/|\Pr(\mathcal{F}) - p|, \sqrt{n}))$, we have that $(R - p)h(n)$ converges to 0 in probability as $n \rightarrow \infty$.

Remark 3.6.1. Since $|\Pr(\mathcal{F}) - p| = o(1)$ by Lemma 3.3.1, we may take $h(n) = 1$ in Theorem 3.6.1 to conclude that R converges to p in probability as $n \rightarrow \infty$.

Remark 3.6.2. From the proofs of Theorems 3.4.1 and 3.4.2, it is easy to see that for both the partition and (enhanced) double hashing schemes, $\Delta = 0$, so $\xi = 0$ for both hashing schemes as well.

Remark 3.6.3. We have added a new condition on the distribution of $H(u)$, but it trivially holds in all of the hashing schemes that we discuss in this chapter (since, for independent fully random hash functions h_1 and h_2 , the random variables $\{(h_1(u), h_2(u)) : u \in U\}$ are independent and identically distributed, and $(h_1(u), h_2(u))$ is uniformly distributed over its support, which is finite).

Proof. The proof is essentially a standard application of Azuma's inequality to an appropriately defined Doob martingale. Specifically, we employ the technique discussed in [42, Section 12.5].

For convenience, write $S = \{x_1, \dots, x_n\}$. For $h_1, \dots, h_n \in \text{Supp}(H(u))$, define

$$f(h_1, \dots, h_n) = \Pr(\mathcal{F} \mid H(x_1) = h_1, \dots, H(x_n) = h_n),$$

and note that $R = f(H(x_1), \dots, H(x_n))$. Now consider some d such that for any $h_1, \dots, h_j, h'_j, h_{j+1}, \dots, h_n \in \text{Supp}(H(u))$,

$$|f(h_1, \dots, h_n) - f(h_1, \dots, h_{j-1}, h'_j, h_{j+1}, \dots, h_n)| \leq d.$$

Since the $H(x_i)$'s are independent, we may apply the result of [42, Section 12.5] to obtain

$$\Pr(|R - \mathbf{E}[R]| \geq \delta) \leq 2e^{-2\delta^2/nd^2},$$

for any $\delta > 0$.

To find an appropriate and small choice for d , we write

$$\begin{aligned}
& |f(h_1, \dots, h_n) - f(h_1, \dots, h_{j-1}, h'_j, h_{j+1}, \dots, h_n)| \\
&= |\Pr(\mathcal{F} \mid H(x_1) = h_1, \dots, H(x_n) = h_n) \\
&\quad - \Pr(\mathcal{F} \mid H(x_1) = h_1, \dots, H(x_{j-1}) = h_{j-1}, H(x_j) = h'_j, \\
&\quad\quad H(x_{j+1}) = h_{j+1}, \dots, H(x_n) = h_n)| \\
&= \frac{\left| \left| \{v \in V : g(v) \subseteq \bigcup_{i=1}^n h_i\} \right| - \left| \{v \in V : g(v) \subseteq \bigcup_{i=1}^n \begin{cases} h'_j & i = j \\ h_i & i \neq j \end{cases} \} \right| \right|}{|V|} \\
&\leq \frac{\max_{v' \in V} |\{v \in V : |g(v) \cap g(v')| \geq 1\}|}{|V|} \\
&= \max_{M' \in \text{Supp}(H(u))} \Pr(|H(u) \cap M'| \geq 1),
\end{aligned}$$

where the first step is just the definition of f , the second step follows from the definitions of V_u and g , the third step holds since changing one of the h_i 's to some $M' \in \text{Supp}(H(u))$ cannot change

$$\left| \left\{ v \in V : g(v) \subseteq \bigcup_{i=1}^n h_i \right\} \right|$$

by more than

$$|\{v \in V : |g(v) \cap M'| \geq 1\}|,$$

and the fourth step follows from the definitions of V_u and g .

Now consider any fixed $M' \in \text{Supp}(H(u))$, and let $y_1, \dots, y_{|M'|}$ be the distinct elements of M' . Recall that $\|M'\| = k$, so $|M'| \leq k$. Applying a union bound, we have that

$$\begin{aligned}
\Pr(|H(u) \cap M'| \geq 1) &= \Pr\left(\bigcup_{i=1}^{|M'|} y_i \in H(u)\right) \\
&\leq \sum_{i=1}^{|M'|} \Pr(y_i \in H(u)) \\
&\leq \sum_{i=1}^{|M'|} \frac{\lambda}{kn} + \Delta \\
&\leq \frac{\lambda}{n} + k\Delta.
\end{aligned}$$

Therefore, we may set $d = \frac{\lambda}{n} + k\Delta$ to obtain

$$\Pr(|R - \mathbf{E}[R]| > \delta) \leq 2 \exp\left[\frac{-2n\delta^2}{\lambda^2 + \xi}\right],$$

for any $\delta > 0$. Since $\mathbf{E}[R] = \mathbf{Pr}(\mathcal{F})$, we write (for sufficiently large n so that $\epsilon > |\mathbf{Pr}(\mathcal{F}) - p|$)

$$\begin{aligned} \mathbf{Pr}(|R - p| > \epsilon) &\leq \mathbf{Pr}(|R - \mathbf{Pr}(\mathcal{F})| > \epsilon - |\mathbf{Pr}(\mathcal{F}) - p|) \\ &\leq 2 \exp \left[\frac{-2n(\epsilon - |\mathbf{Pr}(\mathcal{F}) - p|)^2}{\lambda^2 + \xi} \right]. \end{aligned}$$

To complete the proof, we see that for any constant $\delta > 0$,

$$\mathbf{Pr}(|R - p|h(n) > \delta) = \mathbf{Pr}(|R - p| > \delta/h(n)) \rightarrow 0 \quad \text{as } n \rightarrow \infty,$$

where the second step follows from the fact that $|\mathbf{Pr}(\mathcal{F}) - p| = o(1/h(n))$, so for sufficiently large n ,

$$\begin{aligned} \mathbf{Pr}(|R - p| > \delta/h(n)) &\leq 2 \exp \left[\frac{-2n(\delta/h(n) - |\mathbf{Pr}(\mathcal{F}) - p|)^2}{\lambda^2 + \xi} \right] \\ &\leq 2 \exp \left[-\frac{\delta^2}{\lambda^2 + \xi} \cdot \frac{n}{h(n)^2} \right] \\ &\rightarrow 0 \quad \text{as } n \rightarrow \infty, \end{aligned}$$

and the last step follows from the fact that $h(n) = o(\sqrt{n})$. \square

Since, conditioned on R , the events $\{\mathcal{F}(z) : z \in U - S\}$ are independent and each occur with probability R , Theorem 3.6.1 suggests that $\{\mathbf{1}(\mathcal{F}(z)) : z \in U - S\}$ are essentially independent Bernoulli(p) random variables. We formalize this idea in the next result, where we use Theorem 3.6.1 to prove versions of the strong law of large numbers, the weak law of large numbers, Hoeffding's inequality, and the central limit theorem.

Theorem 3.6.2. *Consider a hashing scheme that satisfies the conditions of Theorem 3.6.1. Let $Z \subseteq U - S$ be countably infinite, and write $Z = \{z_1, z_2, \dots\}$. Then we have:*

1.

$$\mathbf{Pr} \left(\lim_{\ell \rightarrow \infty} \frac{1}{\ell} \sum_{i=1}^{\ell} \mathbf{1}(\mathcal{F}_n(z_i)) = R_n \right) = 1.$$

2. For any $\epsilon > 0$, for n sufficiently large so that $\epsilon > |\mathbf{Pr}(\mathcal{F}) - p|$,

$$\mathbf{Pr} \left(\left| \lim_{\ell \rightarrow \infty} \frac{1}{\ell} \sum_{i=1}^{\ell} \mathbf{1}(\mathcal{F}_n(z_i)) - p \right| > \epsilon \right) \leq 2 \exp \left[\frac{-2n(\epsilon - |\mathbf{Pr}(\mathcal{F}) - p|)^2}{\lambda^2 + \xi} \right].$$

In particular, $\lim_{\ell \rightarrow \infty} \frac{1}{\ell} \sum_{i=1}^{\ell} \mathbf{1}(\mathcal{F}_n(z_i))$ converges to p in probability as $n \rightarrow \infty$.

3. For any function $Q(n)$, $\epsilon > 0$, and n sufficiently large so that $\epsilon/2 > |\mathbf{Pr}(\mathcal{F}) - p|$,

$$\begin{aligned} \mathbf{Pr} \left(\left| \frac{1}{Q(n)} \sum_{i=1}^{Q(n)} \mathbf{1}(\mathcal{F}_n(z_i)) - p \right| > \epsilon \right) \\ \leq 2e^{-Q(n)\epsilon^2/2} + 2 \exp \left[\frac{-2n(\epsilon/2 - |\mathbf{Pr}(\mathcal{F}) - p|)^2}{\lambda^2 + \xi} \right]. \end{aligned}$$

4. For any function $Q(n)$ with $\lim_{n \rightarrow \infty} Q(n) = \infty$ and $Q(n) = o(\min(1/|\mathbf{Pr}(\mathcal{F}) - p|^2, n))$,

$$\sum_{i=1}^{Q(n)} \frac{\mathbf{1}(\mathcal{F}_n(z_i)) - p}{\sqrt{Q(n)p(1-p)}} \rightarrow \mathbf{N}(0, 1) \text{ in distribution as } n \rightarrow \infty.$$

Remark 3.6.4. By Theorem 3.5.2, $|\mathbf{Pr}(\mathcal{F}) - p| = O(1/n)$ for both the partition and double hashing schemes introduced in Section 3.4. Thus, for each of the hashing schemes, the condition $Q(n) = o(\min(1/|\mathbf{Pr}(\mathcal{F}) - p|^2, n))$ in the fourth part of Theorem 3.6.2 becomes $Q(n) = o(n)$.

Proof. Since, given R_n , the random variables $\{\mathbf{1}(\mathcal{F}_n(z)) : z \in Z\}$ are conditionally independent Bernoulli(R_n) random variables, a direct application of the strong law of large numbers yields the first item.

For the second item, we note that the first item implies that

$$\lim_{\ell \rightarrow \infty} \frac{1}{\ell} \sum_{i=1}^{\ell} \mathbf{1}(\mathcal{F}_n(z_i)) \sim R_n.$$

A direct application of Theorem 3.6.1 then gives the result.

For the third item, we write

$$\begin{aligned} & \Pr \left(\left| \frac{1}{Q(n)} \sum_{i=1}^{Q(n)} \mathbf{1}(\mathcal{F}_n(z_i)) - p \right| > \epsilon \right) \\ & \leq \Pr \left(\left| \frac{1}{Q(n)} \sum_{i=1}^{Q(n)} \mathbf{1}(\mathcal{F}_n(z_i)) - R_n \right| > \epsilon/2 \mid |R_n - p| \leq \epsilon/2 \right) + \Pr(|R_n - p| > \epsilon/2) \\ & \leq 2e^{-Q(n)\epsilon^2/2} + 2 \exp \left[\frac{-2n(\epsilon/2 - |\mathbf{Pr}(\mathcal{F}) - p|)^2}{\lambda^2 + \xi} \right], \end{aligned}$$

where the last step follows from Hoeffding's inequality and Theorem 3.6.1.

For the fourth item, we write

$$\begin{aligned} & \sum_{i=1}^{Q(n)} \frac{\mathbf{1}(\mathcal{F}_n(z_i)) - p}{\sqrt{Q(n)p(1-p)}} \\ & = \sqrt{\frac{R_n(1-R_n)}{p(1-p)}} \left(\sum_{i=1}^{Q(n)} \frac{\mathbf{1}(\mathcal{F}_n(z_i)) - R_n}{\sqrt{Q(n)R_n(1-R_n)}} + (R_n - p) \sqrt{\frac{Q(n)}{R_n(1-R_n)}} \right). \end{aligned}$$

By Slutsky's theorem, it suffices to show the following three items:

1. $R_n \rightarrow p$ in probability as $n \rightarrow \infty$,
2. $(R_n - p)\sqrt{Q(n)} \rightarrow 0$ in probability as $n \rightarrow \infty$, and

3.

$$\sum_{i=1}^{Q(n)} \frac{\mathbf{1}(\mathcal{F}_n(z_i)) - R_n}{\sqrt{Q(n)R_n(1-R_n)}} \rightarrow N(0,1) \quad \text{in distribution as } n \rightarrow \infty.$$

The first item holds by Remark 3.6.1, and the second item holds by Theorem 3.6.1, since $\sqrt{Q(n)} = o(\min(1/|\mathbf{Pr}(\mathcal{F}) - p|, \sqrt{n}))$. The third item requires a little more work. First, we need a version of the central limit theorem that allows us to bound its rate of convergence.

Lemma 3.6.1. [15, Theorem 24] *Let X_1, X_2, \dots be independent random variables with some common distribution X , where $\mathbf{E}[X] = 0$ and $\mathbf{Var}[X] = 1$. For $n \geq 1$, let $Y_n = \sum_{i=1}^n X_i/\sqrt{n}$. Then there is some constant a such that for any $n \geq 1$ and $x \in \mathbb{R}$,*

$$|\mathbf{Pr}(Y_n \leq x) - \mathbf{Pr}(N(0,1) \leq x)| \leq a\mathbf{E}[|X|^3]/\sqrt{n}.$$

Fix some constant $\epsilon > 0$ so that $I \triangleq [p - \epsilon, p + \epsilon] \subseteq (0, 1)$, and let

$$b = \min_{x \in I} \sqrt{x(1-x)} > 0.$$

With Lemma 3.6.1 in mind, define

$$X_i(n) = \frac{\mathbf{1}(\mathcal{F}_n(z_i)) - R_n}{\sqrt{R_n(1-R_n)}}.$$

Since, given R_n , the random variables are independent Bernoulli(R_n) random variables, Lemma 3.6.1 tells us that for any $x \in \mathbb{R}$,

$$\begin{aligned} & \left| \mathbf{Pr} \left(\sum_{i=1}^{Q(n)} X_i(n)/\sqrt{Q(n)} \right) - \mathbf{Pr}(N(0,1) \leq x) \right| \\ & \leq \mathbf{Pr}(|R_n - p| > \epsilon) + \left| \mathbf{Pr} \left(\sum_{i=1}^{Q(n)} \frac{X_i(n)}{\sqrt{Q(n)}} \leq x \mid |R_n - p| \leq \epsilon \right) - \mathbf{Pr}(N(0,1) \leq x) \right| \\ & \leq \mathbf{Pr}(|R_n - p| > \epsilon) + \frac{a(1/b)^3}{\sqrt{Q(n)}} \\ & \rightarrow 0 \quad \text{as } n \rightarrow \infty, \end{aligned}$$

where the last step follows from Remark 3.6.1. □

3.7 Experiments

In this section, we evaluate the theoretical results of the previous sections empirically for small values of n . We are interested in the following specific hashing schemes: the standard Bloom filter scheme, the partition scheme, the double hashing scheme, and the enhanced double hashing schemes where $f(i) = i^2$ and $f(i) = i^3$.

For $c \in \{4, 8, 12, 16\}$, we do the following. First, compute the value of $k \in \{\lceil c \ln 2 \rceil, \lceil c \ln 2 \rceil\}$ that minimizes $p = (1 - \exp[-k/c])^k$. Next, for each of the hashing

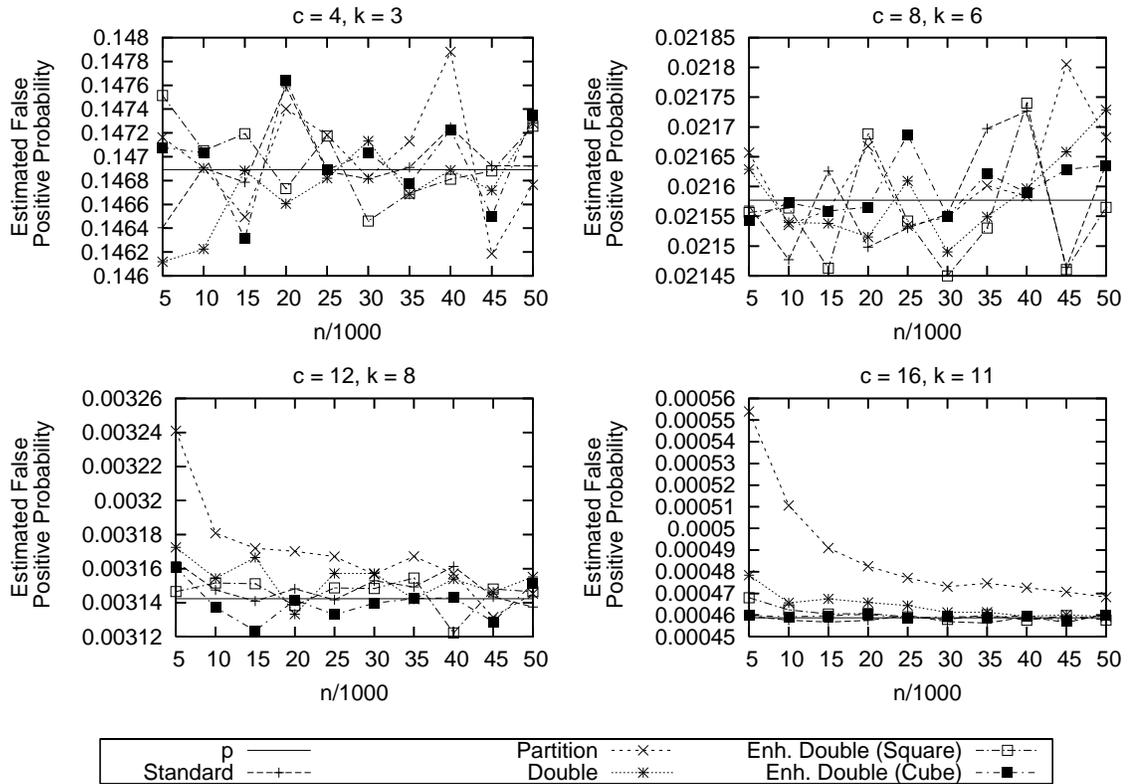


Figure 3.1: Estimates of the false positive probability for various hashing schemes and parameters.

schemes under consideration, repeat the following procedure 10,000 times: instantiate the filter with the specified values of n , c , and k , populate the filter with a set S of n items, and then query $\lceil 10/p \rceil$ elements not in S , recording the number Q of those queries for which the filter returns a false positive. We then approximate the false positive probability of the hashing scheme by averaging the results over all 10,000 trials. Furthermore, we bin the results of the trials by their values for Q in order to examine the other characteristics of Q 's distribution. In all our experiments, we simulate fully random hash functions using the standard Java pseudorandom number generator.

The results are shown in Figures 3.1 and 3.2. In Figure 3.1, we see that for small values of c , the different hashing schemes are essentially indistinguishable from each other, and simultaneously have a false positive probability/rate close to p . This result is particularly significant since the filters that we are experimenting with are fairly small, supporting our claim that these hashing schemes are useful even in settings with very limited space. However, we also see that for the slightly larger values of $c \in \{12, 16\}$, the partition scheme is no longer particularly useful for small values of n , while the other hashing schemes are. This result is not particularly surprising; for large values of c and small values of n , the probability of a false positive can be substantially affected by the asymptotically vanishing probability that one element in the set can yield multiple collisions with an element not in

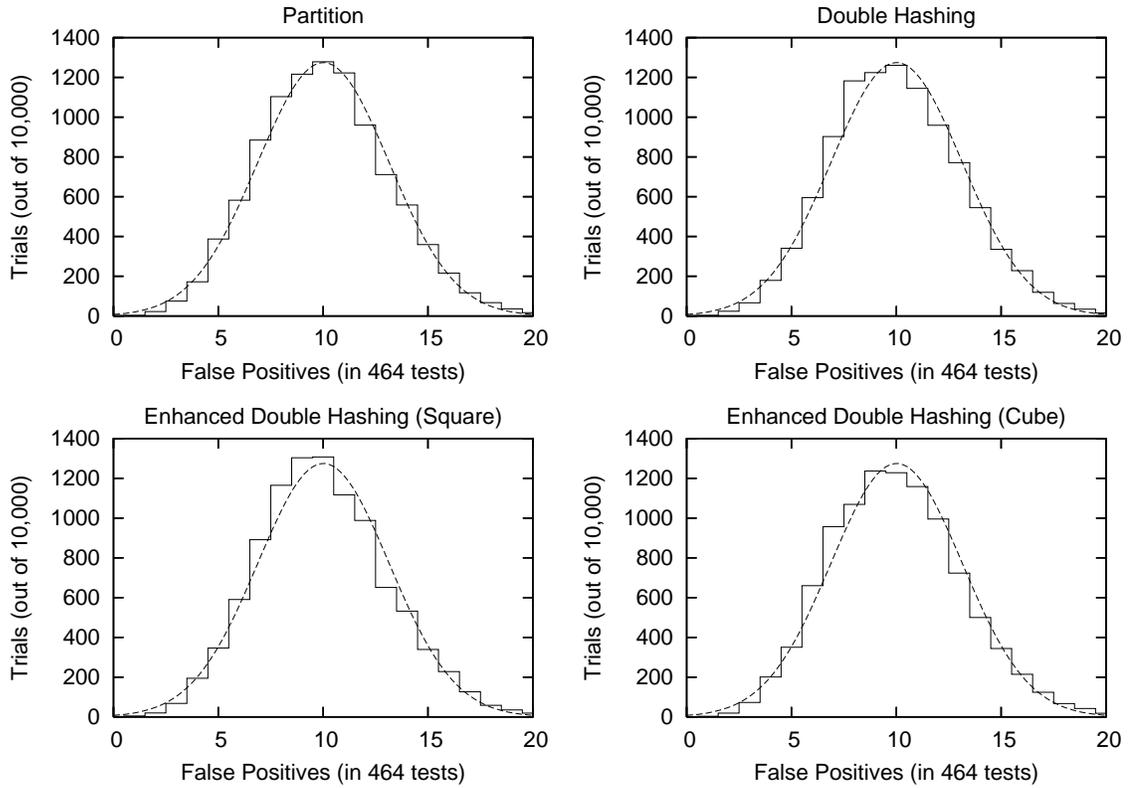


Figure 3.2: Estimate of distribution of Q (for $n = 5000$ and $c = 8$), compared with f .

the set, and this is somewhat larger in the partition scheme. Nevertheless, the difference is sufficiently small that the partition scheme might still be worthwhile in practice if parallelism is desired.

As an aside, Dillinger and Manolios [19, 20] observe that as c grows very large, various enhanced double hashing schemes (including triple hashing, where the g_i 's use a third hash function with a coefficient that is quadratic in the index i) tend to perform slightly better than the regular double hashing scheme. Their results suggest that the difference is most likely due to differences in the constants in the rates of convergence of the various hashing schemes. For the most part, this effect is not noticeable for the Bloom filter configurations that we consider, which are chosen to capture the typical Bloom filter setting where the false positive probability is small enough to be tolerable, but still non-negligible.

In Figure 3.2, we give histograms of the results from our experiments with $n = 5000$ and $c = 8$ for the partition and enhanced double hashing schemes. For this value of c , optimizing for k yields $k = 6$, so we have $p \approx 0.021577$ and $\lceil 10/p \rceil = 464$. In each plot, we compare the results to $f \triangleq 10,000\phi_{464p,464p(1-p)}$, where

$$\phi_{\mu,\sigma^2}(x) \triangleq \frac{e^{-(x-\mu)^2/2\sigma^2}}{\sigma\sqrt{2\pi}}$$

denotes the density function of $N(\mu, \sigma^2)$. As one would expect, given the central limit

theorem result in the fourth part of Theorem 3.6.2, f provides a reasonable approximation to each of the histograms.

3.8 A Modified Count-Min Sketch

We now extend our use of double hashing from Bloom filters to the count-min sketch [14], which is a data structure for estimating statistics of data streams that is closely related to the standard Bloom filter. Specifically, we show how to use double hashing to reduce the number of hash functions in the count-min sketch at the cost of small increase in space. This result is a natural extension of the ideas already introduced in this chapter, and we include it to demonstrate that our techniques are not only useful for Bloom filters, but are also worth consideration for any related data structure. We begin by reviewing the original count-min sketch.

3.8.1 Count-Min Sketch Review

The following is an abbreviated review of the description given in [14]. A count-min sketch takes as input a stream of *updates* (i_t, c_t) , starting from $t = 1$, where each *item* i_t is a member of a universe $U = \{1, \dots, n\}$, and each *count* c_t is a positive number. (Extensions to negative counts are possible; we do not consider them here for convenience.) The state of the system at time T is given by a vector $\vec{a}(T) = (a_1(T), \dots, a_n(T))$, where $a_j(T)$ is the sum of all c_t for which $t \leq T$ and $i_t = j$. We generally drop the T when the meaning is clear.

The count-min sketch consists of an array *Count* of width $w \triangleq \lceil e/\epsilon \rceil$ and depth $d \triangleq \lceil \ln 1/\delta \rceil$: $\text{Count}[1, 1], \dots, \text{Count}[d, w]$. Every entry of the array is initialized to 0. In addition, the count-min sketch uses d hash functions chosen independently from a pairwise independent family $\mathcal{H} : \{1, \dots, n\} \rightarrow \{1, \dots, w\}$.

The mechanics of the count-min sketch are extremely simple. Whenever an update (i, c) arrives, we increment $\text{Count}[j, h_j(i)]$ by c , for $j = 1, \dots, d$. Whenever we want an estimate of a_i (called a *point query*), we compute

$$\hat{a}_i \triangleq \min_{j=1}^d \text{Count}[j, h_j(i)].$$

The fundamental result of count-min sketches is that for every i ,

$$\hat{a}_i \geq a_i \quad \text{and} \quad \Pr(\hat{a}_i \leq a_i + \epsilon \|\vec{a}\|) \geq 1 - \delta,$$

where the norm is the L_1 norm. Surprisingly, this very simple bound allows for a number of sophisticated estimation procedures to be efficiently and effectively implemented on count-min sketches. The reader is once again referred to [14] for details.

3.8.2 Using Fewer Hash Functions

We now show how the improvements to Bloom filters discussed previously in this chapter can be usefully applied to count-min sketches. Our modification maintains all of

the essential features of count-min sketches, but reduces the required number of pairwise independent hash functions to $2\lceil(\ln 1/\delta)/(\ln 1/\epsilon)\rceil$. We expect that, in many settings, ϵ and δ will be related, so that only a constant number of hash functions will be required; in fact, in many such situations only two hash functions are required.

We describe a variation of the count-min sketch that uses just two pairwise independent hash functions and guarantees that

$$\hat{a}_i \geq a_i \quad \text{and} \quad \Pr(\hat{a}_i \leq a_i + \epsilon\|\vec{a}\|) \geq 1 - \epsilon.$$

Given such a result, it is straightforward to obtain a variation that uses

$$2\lceil(\ln 1/\delta)/(\ln 1/\epsilon)\rceil$$

pairwise independent hash functions and achieves the desired failure probability δ : simply build $2\lceil(\ln 1/\delta)/(\ln 1/\epsilon)\rceil$ independent copies of this data structure, and always answer a point query with the minimum estimate given by one of those copies.

Our variation will use d tables numbered $\{0, 1, \dots, d-1\}$, each with exactly w counters numbered $\{0, 1, \dots, w-1\}$, where d and w will be specified later. We insist that w be prime. Just as in the original count-min sketch, we let $\text{Count}[j, k]$ denote the value of the k th counter in the j th table. We choose hash functions h_1 and h_2 independently from a pairwise independent family $\mathcal{H} : \{0, \dots, n-1\} \rightarrow \{0, 1, \dots, w-1\}$, and define $g_j(x) = h_1(x) + jh_2(x) \bmod w$ for $j = 0, \dots, d-1$.

The mechanics of our data structure are the same as for the original count-min sketch. Whenever an update (i, c) occurs in the stream, we increment $\text{Count}[j, g_j(i)]$ by c , for $j = 0, \dots, d-1$. Whenever we want an estimate of a_i , we compute

$$\hat{a}_i \triangleq \min_{j=0}^{d-1} \text{Count}[j, g_j(i)].$$

We prove the following result:

Theorem 3.8.1. *For the count-min sketch variation described above,*

$$\hat{a}_i \geq a_i \quad \text{and} \quad \Pr(\hat{a}_i > a_i + \epsilon\|\vec{a}\|) \leq \frac{2}{\epsilon w^2} + \left(\frac{2}{\epsilon w}\right)^d.$$

In particular, for $w \geq 2e/\epsilon$ and $\delta \geq \ln 1/\epsilon(1 - 1/2e^2)$,

$$\hat{a}_i \geq a_i \quad \text{and} \quad \Pr(\hat{a}_i > a_i + \epsilon\|\vec{a}\|) \leq \epsilon.$$

Proof. Fix some item i . Let A_i be the total count for all items z (besides i) with $h_1(z) = h_1(i)$ and $h_2(z) = h_2(i)$. Let $B_{j,i}$ be the total count for all items z with $g_j(i) = g_j(z)$, excluding i and items z counted in A_i . It follows that

$$\hat{a}_i = \min_{j=0}^{d-1} \text{Count}[j, g_j(i)] = a_i + A_i + \min_{j=0}^{d-1} B_{j,i}.$$

The lower bound now follows immediately from the fact that all items have nonnegative counts, since all updates are positive. Thus, we concentrate on the upper bound, which we approach by noticing that

$$\Pr(\hat{a}_i \geq a_i + \epsilon \|\vec{a}\|) \leq \Pr(A_i \geq \epsilon \|\vec{a}\|/2) + \Pr\left(\min_{j=0}^{d-1} B_{j,i} \geq \epsilon \|\vec{a}\|/2\right).$$

We first bound A_i . Letting $\mathbf{1}(\cdot)$ denote the indicator function, we have

$$\mathbf{E}[A_i] = \sum_{z \neq i} a_z \mathbf{E}[\mathbf{1}(h_1(z) = h_1(i) \wedge h_2(z) = h_2(i))] \leq \sum_{z \neq i} a_z/w^2 \leq \|\vec{a}\|/w^2,$$

where the first step follows from linearity of expectation and the second step follows from the definition of the hash functions. Markov's inequality now implies that

$$\Pr(A_i \geq \epsilon \|\vec{a}\|/2) \leq 2/\epsilon w^2.$$

To bound $\min_{j=0}^{d-1} B_{j,i}$, we note that for any $j \in \{0, \dots, d-1\}$ and $z \neq i$,

$$\begin{aligned} \Pr((h_1(z) \neq h_1(i) \vee h_2(z) \neq h_2(i)) \wedge g_j(z) = g_j(i)) \\ &\leq \Pr(g_j(z) = g_j(i)) \\ &= \Pr(h_1(z) = h_1(i) + j(h_2(i) - h_2(z))) \\ &= 1/w, \end{aligned}$$

so

$$\mathbf{E}[B_{j,i}] = \sum_{z \neq i} a_z \mathbf{E}[\mathbf{1}((h_1(z) \neq h_1(i) \vee h_2(z) \neq h_2(i)) \wedge g_j(z) = g_j(i))] \leq \|\vec{a}\|/w,$$

and so Markov's inequality implies that

$$\Pr(B_{j,i} \geq \epsilon \|\vec{a}\|/2) \leq 2/\epsilon w.$$

For arbitrary w , this result is not strong enough to bound $\min_{j=0}^{d-1} B_{j,i}$. However, since w is prime, each item z can only contribute to one $B_{k,i}$ (since if $g_j(z) = g_j(i)$ for two values of j , we must have $h_1(z) = h_1(i)$ and $h_2(z) = h_2(i)$, and in this case z 's count is not included in any $B_{j,i}$). In this sense, the $B_{j,i}$'s are negatively dependent (specifically, they are negatively right orthant dependent) [22]. It follows that for any value v ,

$$\Pr\left(\min_{j=0}^{d-1} B_{j,i} \geq v\right) \leq \prod_{j=0}^{d-1} \Pr(B_{j,i} \geq v).$$

In particular, we have that

$$\Pr\left(\min_{j=0}^{d-1} B_{j,i} \geq \epsilon \|\vec{a}\|/2\right) \leq (2/\epsilon w)^d,$$

so

$$\begin{aligned} \Pr(\hat{a}_i \geq a_i + \epsilon \|\vec{a}\|) &\leq \Pr(A_i \geq \epsilon \|\vec{a}\|/2) + \Pr\left(\min_{j=0}^{d-1} B_j, i \geq \epsilon \|\vec{a}\|/2\right) \\ &\leq \frac{2}{\epsilon w^2} + \left(\frac{2}{\epsilon w}\right)^d. \end{aligned}$$

And for $w \geq 2e/\epsilon$ and $\delta \geq \ln 1/\epsilon(1 - 1/2e^2)$, we have

$$\frac{2}{\epsilon w^2} + \left(\frac{2}{\epsilon w}\right)^d \leq \epsilon/2e^2 + \epsilon(1 - 1/2e^2) = \epsilon,$$

completing the proof. \square

3.9 Conclusion

Bloom filters are simple randomized data structures that are extremely useful in practice. In fact, they are so useful that any significant reduction in the time required to perform a Bloom filter operation immediately translates to a substantial speedup for many practical applications. Unfortunately, Bloom filters are so simple that they do not leave much room for optimization.

This chapter focuses on modifying Bloom filters to use less of the only resource that they traditionally use liberally: (pseudo)randomness. Since the only nontrivial computations performed by a Bloom filter are the constructions and evaluations of pseudorandom hash functions, any reduction in the required number of pseudorandom hash functions yields a nearly equivalent reduction in the time required to perform a Bloom filter operation (assuming, of course, that the Bloom filter is stored entirely in memory, so that random accesses can be performed very quickly).

We have shown that a Bloom filter can be implemented with only two random hash functions without any increase in the asymptotic false positive probability. We have also shown that the asymptotic false positive probability acts, for all practical purposes and reasonable settings of a Bloom filter's parameters, like a false positive rate. This result has substantial practical significance, since the analogous result for standard Bloom filters is essentially the theoretical justification for their extensive use.

More generally, we have given a framework for analyzing modified Bloom filters, which we expect will be used in the future to refine the specific hashing schemes that we analyzed in this chapter. We also expect that the techniques used in this chapter will be usefully applied to other data structures, as demonstrated by our modification to the count-min sketch.

In terms of the general goals and methodology outlined in Chapter 1, we have provided a theoretical foundation for a very practical modification to the already very practical Bloom filter family of data structures. Our results are particularly significant when the computational resources required to perform a Bloom filter operation are extremely limited. Furthermore, we have confirmed the essence of our theoretical results through experiments (and noted the prior empirical work of Dillinger and Manolios [19, 20] along

these lines), to ensure that they provide a clear and accurate picture of what one would expect to see in a real application.

Chapter 4

Simple Summaries for Hashing with Choices

4.1 Introduction

In a multiple choice hashing scheme, a hash table is built using the following approach: each item x is associated with hash values $h_1(x), h_2(x), \dots, h_d(x)$, each corresponding to a bucket in the hash table, and the item is placed in one (or possibly more) of the d locations. As discussed in Chapter 2, such schemes are often used to lessen the maximum load (that is, the number of items in a bucket), as giving each item the choice between more than one bucket in the hash table often leads to a significant improvement in the balance of the items [2, 10, 40, 41]. These schemes can also be used to ensure that each bucket contains at most one item with high probability [8]. For these reasons, multiple choice hashing schemes have been proposed for many applications, including network routers [10], peer-to-peer applications [11], and standard load balancing of jobs across machines [39].

Recently, in the context of routers, Song *et al.* [56] suggested that a drawback of multiple choice schemes is that at the time of a lookup, one cannot know which of the d possible locations to check for the item. The natural solution to this problem is to use d lookups in parallel [10]. But while this approach might keep the lookup time the same as in the standard single choice hashing scheme, it generally costs in other resources, such as pin count in the router setting. Song *et al.* [56] provide a framework for avoiding these lookup-related costs while still allowing insertions and deletions of items in the hash table. They suggest keeping a small *summary* in very fast memory (that is, significantly faster memory than is practical to store the much larger hash table) that can efficiently answer queries of the form: “Is x in the hash table, and if so, which of $h_1(x), \dots, h_d(x)$ was actually used to store x ?” A simple illustration is given in Figure 4.1. Of course, items that are not actually in the hash table may yield false positives; otherwise, the summary could be no more efficient than a hash table. The small summary used by Song *et al.* [56] consists of a counting Bloom filter [24, 38]. We review this construction in detail in Section 4.3.

The specific applications that motivate this work are a wide variety of packet processing techniques employed by high-speed routers that rely on the use of a hash table. (For details, see Section 2.2.2.) In these applications, the worst case performance of the

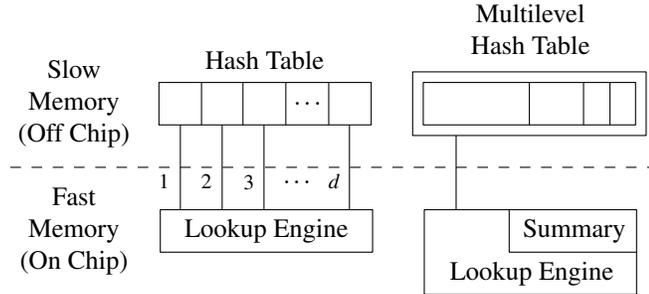


Figure 4.1: A comparative illustration of the hardware and connections in an implementation of a standard multiple choice hash table (left) and a *multilevel hash table* (described in Section 4.5) equipped with a summary data structure (right).

standard lookup, insertion, and even deletion operations can be a bottleneck. In particular, hash tables that merely guarantee good amortized performance may not work well in these settings because it may not be practical to allow any exceptionally time-consuming operations, even if they are rare, since that might require buffering the operations that arrive in the interim. Furthermore, the high cost of computational, hardware, and even energy resources in this setting forces us to seek hash table optimizations that are not only effective, but also extremely amenable to simple, efficient, hardware-based implementations. These observations explain why both we and Song *et al.* focus on the use of multiple choice hash tables, prized for their worst case performance guarantees, along with a summary data structure that allows for a further reduction in the cost of lookups into the hash table.

In this chapter, we suggest three alternative approaches for maintaining summaries for multiple choice hashing schemes. The first is based on interpolation search, and the other two are based on standard Bloom filters or simple variants thereof and a clever choice of the underlying hash table. Our approaches have numerous advantages, including less space for the hash table, similar or smaller space for the summary, and better performance for insertions and deletions. Another advantage of our approaches is that they are very simple to analyze; while [56] provides an analysis for a basic variation of the scheme proposed therein, more advanced versions (which seem to be required for adequate performance) have not yet been analyzed. We believe that the ability to analyze performance is important, as it allows for more informed engineering decisions based on formal guarantees.

An interesting feature of our work is that we use multiple choice hashing schemes that are sharply *skewed*, in the sense that most items are placed according to the first hash function, fewer are placed according to the second, and so on. We show how to take advantage of this skew, providing an interesting principle: dividing the hash table space unequally among the d sub-tables allows for skew tradeoffs that can allow significant performance improvements for the corresponding summary.

4.2 Related Work

As discussed in Chapter 2, there is a lot of work on multiple choice hashing and on Bloom filters. In this chapter, however, our primary starting point is the work of Song *et al.* [56], which introduces an approach for summarizing the locations of items in a hash table that uses multiple hash functions. We review this work in detail in Section 4.3.

We are also influenced by a significant early paper of Broder and Karlin [8]. For n items, they give a construction of a *multilevel hash table* (MHT) that consists of $d = O(\log \log n)$ sub-tables, each with its own hash function, that are geometrically decreasing in size. As described in Chapter 2, an item is always placed in the first sub-table where its hash location is empty, and therefore this hashing scheme is skewed in the sense described above. Our main result can be seen as adding a summary to an MHT, and so we give a more thorough description of MHTs in Section 4.5.

Finally, the problem of constructing summaries for multiple choice hash tables seems closely connected with the work on a generalization of Bloom filters called *Bloomier filters* [13], which are designed to represent functions on a set. In the Bloomier filter problem setting, each item in a set has an associated value; items not in the set have a null value. The goal is then to design a data structure that gives the correct value for each item in the set, while allowing *false positives*, so that a query for an item not in the set may rarely return a non-null value. In our setting, values correspond to the hash function used to place the item in the table. For static hash tables (that is, with no insertions or deletions), current results for Bloomier filters could be directly applied to give a (perhaps inefficient) solution. Limited results exist for Bloomier filters that have to cope with changing function values. However, lower bounds for such filters [13, 47] suggest that we must take advantage of the characteristics of our specific problem setting (e.g., the skew of the distribution of the values) in order to guarantee good performance.

4.3 The Scheme of Song *et al.* [56]

For comparison, we review the summary of [56] before introducing our approaches. The basic scheme works as follows. The hash table consists of m *buckets*, and each item is hashed via d (independent, random) hash functions to d buckets (with multiplicity in the case of collisions). The summary consists of one b -bit *counter* for each bucket. Each counter tracks the number of item hashes to its corresponding bucket.

In the static setting (where the hash table is built once and never subsequently modified), all n items are initially hashed into a preliminary hash table and each is stored in all of its hash buckets. After all items have been hashed, the table is *pruned*; for each item, the copy in the hash bucket with the smallest counter (breaking ties according to bucket ordering) is kept, and all other copies are deleted. Determining the location of an item in the table is now quite easy: one need only compute the d buckets corresponding to the item and choose the one whose corresponding counter is minimal (breaking ties according to bucket ordering). Of course, when looking up an item not in the hash table, there is some chance that all of the examined counters are nonzero, in which case the summary yields a *false positive*. That is, the item appears to be in the hash table until the table is actually

checked at the end of the lookup procedure.

While asymptotics are not given in [56], Song *et al.* strive for parameters that guarantee that there is at most 1 item per bucket with high probability, although their approach could be used more generally. Under this constraint, as we show below, one can achieve $O(n(\log n)\log\log n)$ bits in the summary with $m = \Theta(n\log n)$, $d = \Theta(\log n)$, and $b = \Theta(\log\log n)$; we suspect this bound is tight.

Since the fraction F of nonempty buckets in the hash table before the pruning step satisfies $\mathbf{E}[F] = 1 - (1 - 1/m)^{nd} = 1 - \Omega(1)$, we have $F > (1 + \epsilon)\mathbf{E}[F]$ with probability at most $n^{-\Omega(n)}$, for any constant $\epsilon > 0$ (by a standard martingale argument). This observation tells us that if an $(n + 1)$ -st item is inserted into the hash table just before the pruning step, the probability that all d of its hash buckets are already occupied is at most $n^{-\Omega(n)} + [(1 + \epsilon)\mathbf{E}[F]]^d \leq n^{-c}$ for sufficiently small $\epsilon > 0$, any constant c , sufficiently large n , and some $d = \Theta(\log n)$. This probability is clearly an upper bound on the probability that any particular item hashes to an already occupied bucket. Taking a union bound over all n items tells us that even before pruning, the maximum load is 1 with high probability. Finally, a standard balls-and-bins result (e.g., [42, Lemma 5.1]) tells us that the maximum value of the counters is $O(\log n)$ with high probability, so we may choose $b = \Theta(\log\log n)$.

This basic scheme is not particularly effective. In particular, for a sample configuration considered in [56], there are still a number of buckets with a large number of items. Thus, in practice the scheme seems to have a fairly large probabilistic worst case lookup time. To improve it, Song *et al.* give heuristics to remove collisions in the hash table. The heuristics appear effective, but they are not analyzed. Insertions can be handled readily, but can require relocating previously placed items. Song *et al.* show that the *expected* number of relocations per insertion is constant, but they do not give any high probability bounds on the number of relocations required. Deletions are significantly more challenging under this framework, necessitating additional data structures beyond the summary that require significant memory (for example, a copy of the unpruned hash table, where each item is stored in all of corresponding hash buckets, or a smaller variation called a Shared-node Fast Hash Table) and possibly time; see [56] for details.

4.4 Separating Hash Tables and Their Summaries

Following Song *et al.* [56], we give the following list of high-level goals for our hash table and summary constructions:

- Achieving a maximum load of 1 item per hash table bucket with high probability. (All of our work can be generalized to handle any fixed constant maximum load.)
- Minimizing space for the hash table.
- Minimizing space for the summary.
- Minimizing false positives (generated by the summary) for items not in the hash table.
- Allowing insertions and deletions to the hash table, with corresponding updates for the summary.

More concretely, to understand the tradeoffs between summary space, hash table space, and false positives, it is useful to recall our discussion of router memory technology from Section 2.2.1. In this context, we envision the summary as being implemented in memory that is more expensive and much faster than the memory used for the hash table. For instance, the summary may be implemented with on-chip SRAM and the hash table with off-chip DRAM. From Section 2.2.1, sample access times for on-chip SRAM and off-chip DRAM are 1 ns and 60 ns, respectively. Thus, false positives can cause substantial time penalties. However, reducing false positives necessitates increasing the amount of on-chip SRAM available to the summary, which is limited not just by pure cost-per-bit considerations, but also by chip technology. Of course, the exact tradeoffs involved depend upon the particular application and available memories, and so we only consider these numbers as examples that illustrate the importance of the high-level goals given above.

As a first step, we suggest the following key idea: the summary data structure *need not* correspond to a counter for each bucket in the hash table. That is, we wish to separate the format of the summary structure from the format of the hash table. This change allows us to optimize the hash table and the summary individually. We exploit this additional flexibility in the specific constructions that we present.

The cost of separating the hash table and the summary is additional hashing. With the approach of [56], the hashes of an item are used to access both the summary and the hash table. By separating the formats of the summary and the hash table, we must also separate the roles of the hash functions, and so we must introduce extra hash functions and computation. However, hashing computation is unlikely to be a bottleneck resource in the applications we have in mind, where all of the computations are done in parallel in hardware. Thus, the costs seem reasonable compared to what we will gain, particularly in storage requirements.

A further small disadvantage of separating the summary structure from the hash table is that the summaries we suggest do not immediately tell us if a hash table bucket is empty or not, unlike the summary of [56]. To handle insertions and deletions easily, we therefore require a bit table with one bit per bucket to denote whether each bucket contains an item. Strictly speaking this table is not necessary — we could simply check buckets in the hash table when needed — but in practice this would be inefficient, and hence we add this cost in our analysis of our summary structures.

Having decided to separate the summary structure from the underlying hash table, the next design issue is what underlying hash table to use. We argue that the multilevel hash table of Broder and Karlin [8] offers excellent performance with very small space overhead.

4.5 The Multilevel Hash Table (MHT)

The multilevel hash table (MHT) [8] for representing a set of n elements consists of $d = \log \log n + O(1)$ sub-tables, T_1, \dots, T_d , where T_i has $c_2^{i-1} c_1 n$ buckets that can each hold a single item, for some constant parameters $c_1 > 1$ and $c_2 < 1$. Since the T_i 's are geometrically decreasing in size, the total size of the table is linear (it is bounded by $c_1 n / (1 - c_2)$). For simplicity, we assume that T_1, \dots, T_d place items using independent fully random hash functions h_1, \dots, h_d .

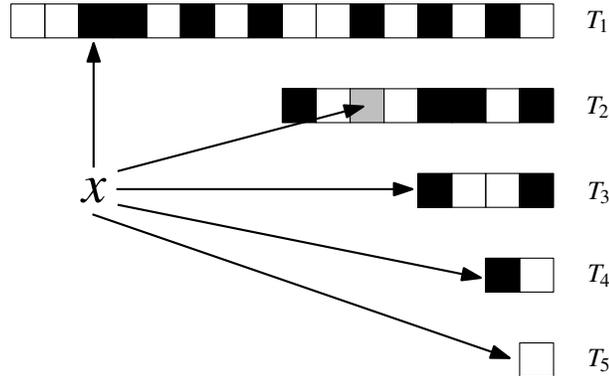


Figure 4.2: Inserting an item x into an MHT. The lines represent the hash locations of x , the black cells represent occupied buckets, the white cells represent unoccupied buckets, and the grey cell represents the bucket in which x is placed. In this example, we place x in $T_2[h_2(x)]$ since that bucket is empty and $T_1[h_1(x)]$ is occupied.

To place an item x in the MHT, we simply find the smallest i for which $T_i[h_i(x)]$ is empty, and place x at $T_i[h_i(x)]$. We illustrate this procedure in Figure 4.2. Of course, this scheme relies on the assumption that at least one of the $T_i[h_i(x)]$'s will always be empty. Following [8], we say that a *crisis* occurs if this assumption fails. By slightly modifying the original analysis, we can show that when $c_1 c_2 > 1$, the probability that there is any crisis is polynomially small.¹ The formal statement and proof of our result can be found in Section 4.11.1.

Finally, we note that deletions can be handled by simply removing items from the hash table. While theoretical bounds are harder to come by when deletions occur, related work shows that multiple choice hashing still does very well [41].

4.5.1 Approximate and Exact Calculations for MHTs

Given a specific configuration of the MHT and the number of items n , one can easily approximate and exactly calculate the probability that a crisis occurs during the sequential insertion of n items into an initially empty MHT. The ability to calculate such numbers is important for making appropriate engineering decisions and is a useful feature of this approach.

A simple approximate calculation can be made by using expectations. In general, if we attempt to hash αn items into a sub-table of size βn , the expected number of nonempty buckets (which is the same as the expected number of items placed in the sub-table) is

$$\beta n [1 - (1 - 1/\beta n)^{\alpha n}] \geq \beta n (1 - \exp(-\alpha/\beta)). \quad (4.1)$$

¹The original Broder-Karlin result shows that crises can be effectively dealt with in certain settings by replacing certain hash functions when a crisis occurs. Since rehashing is not a viable technique in our setting, we consider the occurrence of a crisis to be a serious failure, and so we need our high probability result to theoretically justify using the MHT.

Table 4.1: The approximate and exact expected numbers of items stored in each of the five sub-tables in an MHT for 10k items with $c_1 = 3$ and $c_2 = 1/2$.

	1	2	3	4	5
Approximate	8504.18	1423.70	71.78	0.34	-3.00×10^{-5}
Exact	8504.18	1423.67	71.80	0.35	1.62×10^{-5}

Thus the expected number of items left to be placed in subsequent sub-tables is

$$\alpha n - \beta n [1 - (1 - 1/\beta n)^{\alpha n}] \leq n [\alpha - \beta(1 - \exp(-\alpha/\beta))]. \quad (4.2)$$

Note that the inequalities are quite tight for reasonable n .

To approximate the behavior of the MHT, we can assume that the number of items placed in each sub-table exactly follows (4.1) (or, to give a little wiggle room, the near-tight inequality). Of course, these quantities may deviate somewhat from their expectations (particularly when αn is small), and so these are only heuristic approximations. Once the number of items under consideration is very small, one can use Markov's inequality; if the expected number of items in hashed into a sub-table is $z \ll 1$, then the probability that it is nonzero is at most z .

An exact calculation can be performed similarly. Here, we successively calculate the distribution of the number of items passed to the each sub-table, using the distribution from the previous sub-table. The computation uses the combinatorial fact that when r items are placed randomly into s buckets, the distribution of the number of buckets that remain empty can be calculated. For details, see Section 4.11.3.

In Table 4.1, we compare the results of the heuristic approximation with the exact calculations for a sample MHT. As one would expect, the approximation is extremely accurate when a reasonable number of items are hashed into a sub-table of reasonable size. However, as mentioned above, the approximation becomes less useful as the number of items left to be placed becomes small, since then the random variables of interest become less concentrated around their expectations.

Table 4.1 also shows us that the distribution of the items in the MHT is highly skewed towards the first few tables. This property is extremely important, and it is a recurring theme in this work. Indeed, in Section 4.7.2 we show how to exploit this skew in our summaries, and we propose a slight modification to the original MHT design in order to create more skew to exploit. To illustrate that our approximation technique remains valid under this sort of modification, we compare the results of our heuristic approximation and our exact calculations for a sample modified MHT (which we discuss in more detail when we present more complete numerical results in Section 4.8) in Table 4.2. As before, the approximation is quite accurate until the last sub-table.

4.5.2 The MHT Skew Property

We have mentioned that MHTs have a strong *skew property*, in the sense that the first sub-table contains most of the items, the second sub-table contains most of the rest, and so on. While this can be seen by experimenting with various values in (4.1), we provide a cleaner (albeit looser) presentation based on [8].

Table 4.2: The approximate and exact expected numbers of items stored in each of the five sub-tables in an MHT for 10k items with sub-table sizes 40k, 10k, 5k, 2.5k, 2.5k (where k denotes 1,000).

	1	2	3	4	5
Approximate	8848.07	1088.11	63.42	0.40	-4.80×10^{-5}
Exact	8848.07	1088.08	63.45	0.41	3.37×10^{-5}

Suppose that we insert a set S_0 of items into the MHT, one-by-one. For $i = 1, \dots, d$, let S_i be the set of items that are not placed in tables T_1, \dots, T_i , and let m_i be the size of T_i . First, we note that $|S_i|$ is at most the number of pairwise collisions between elements of S_{i-1} in T_i . Next, we see that given S_{i-1} , there are $\binom{|S_{i-1}|}{2}$ possible pairwise collisions of elements in S_{i-1} , and each of these collisions occurs in T_i with probability $1/m_i$. By linearity of expectation,

$$\mathbf{E}[|S_i| \mid |S_{i-1}|] \leq \binom{|S_{i-1}|}{2} \cdot \frac{1}{m_i} \leq \frac{|S_{i-1}|^2}{2m_i}.$$

Using the heuristic approximation that $|S_i| \leq |S_{i-1}|^2/2m_i$, it is not difficult to show that the $|S_i|$'s decay doubly exponentially when $c_1 c_2 > 1/2$. Indeed, this result is the intuition for the choice of $d = O(\log \log n)$. The result also tells us to expect the distribution of the $|S_i|$'s to be very skewed.

4.6 An Interpolation Search Summary

A straightforward approach to constructing a summary is to hash each item placed in the table to a uniformly distributed b -bit string, for sufficiently large b . We associate each such string with a value (requiring $\log d = \log \log \log n + O(1)$ bits) that indicates what hash function was used for the corresponding item. Searching for an item in the summary can now be done using interpolation search [27], which requires only $O(\log \log n)$ operations on average. (For readers who are unfamiliar with interpolation search, it is essentially the same as binary search, except that instead of choosing the median element as a pivot for comparison, we choose the pivot using linear interpolation. For instance, if we are searching for a number x in a sorted array A of n items drawn from the integers in $[0, 2^b)$, the first pivot is the $\lceil nx/2^b \rceil$ -th element of A . The standard analysis of interpolation search shows that it takes $O(\log \log n)$ expected time to search for a key under the assumption that all elements are uniformly distributed. See [27] for more details.)

Insertions and deletions are conceptually easy; simply add or remove the appropriate string. However, interpolation search is typically defined over a sorted array of strings, and in this case, adding or removing a string in constant time requires a *block copy* operation in hardware to shift the contents of the array as needed. Using more sophisticated data structures, it is possible to implement the summary so that lookups, insertions, and deletions all require $O(\log \log n)$ time with high probability, where the insertion and deletion times are amortized. In certain applications, it may even be possible to strengthen the amortized bounds to worst case bounds. For more details, see [16]. Such data structures

result in a much more complicated summary than the Bloom filter-based constructions presented in Section 4.7, especially if they must be implemented in hardware.

In this summary construction, a *failure* occurs if two items yield the same b bit string. In this case, one might not be able to record the proper value for each item. Therefore b must be chosen to be large enough to make this event extremely unlikely. Of course, b must also be chosen to be large enough so that the false positive probability is also quite small.

We note that two items with the same bit string do not actually constitute a problem if they hash to the same sub-table. For convenience we are choosing to call any such collision a failure here, since allowing any collisions would make handling deletions problematic. In principle, however, we could deal with such collisions if we so desired. One approach would be to record the maximum value associated with each string. In this case, when doing a search, one might have to look at multiple locations in the hash table. For example, if the value 2 is stored with the hash of an item, the item is most likely in the second sub-table, but it might be in the first. Since our goal is to guarantee that only one hash table lookup is necessary for each lookup operation, we do not consider this technique. However, this approach might be useful in some applications and similar ideas are applicable to our other summary constructions as well.

The failure and false positive probabilities for this summary can be computed very easily. The probability of a failure can be calculated using standard probabilistic techniques, as it is just a special case of the birthday paradox [42]. The probability of a false positive, conditioned on no failure occurring (so all n items have distinct b bit strings), is $n/2^b$.

For concreteness, we describe two specific instances of this scheme. Choosing $b = 61$ allows 3 bits for the associated value and for everything to fit into a 64 bit word; 3 bits is enough for 8 hash functions, which should be suitable for most implementations. Setting $n = 100000$ gives a failure probability less than 2.17×10^{-9} and a false positive probability (conditioned on no failure occurring) less than 4.34×10^{-14} . For $n = 10000$, we can achieve similar results for $b = 55$. For these values of n and b , the failure probability is less than 1.39×10^{-9} and the false positive probability (conditioned on no failure occurring) is 2.78×10^{-13} .

This scheme requires only $\Theta(n \log n)$ bits to ensure that failures occur with asymptotically vanishing probability; in this case, false positives occur with vanishing probability as well. In practice, however, the hidden constant factor is nontrivial, and hence the number of bits required can be significantly larger than for other approaches. Also, this approach requires that the application be amenable to a fast implementation of interpolation search. In other words, this approach would only be practical if it were feasible to use a hardware implementation of interpolation search whose particular characteristics were appropriately suited to an application of interest. It is not clear whether any such application exists at this time, especially since we are not aware of any work on hardware implementations of interpolation search. Thus, it seems that this approach is probably impractical. Nevertheless, there are some theoretical advantages of this summary over the others discussed in this chapter, most notably the ability to handle insertions and deletions easily (in certain cases) and the very small false positive probability.

4.7 Bloom Filter-Based MHT Summaries

In this section, we propose summaries that exploit the skew property of MHTs, making extensive use of the theory of Bloom filters. For now we consider insertions only, deferring our discussion of deletions until Section 4.10. We start with an initially empty summary and MHT and insert n items sequentially into both. The summaries presented here never require items to be moved in the MHT, and with high probability, they correctly identify the sub-tables storing each of the n items.

4.7.1 A Natural First Attempt

Our first Bloom filter-based MHT summary can be seen as a simple Bloomier filter that allows insertions. To better illustrate this point, we start by placing our problem in a general setting.

Suppose we have a set of n items, where each item has an integer *type* in the range $[1, \dots, t]$. Our Bloom filter variant consists of m *cells*, where each cell contains a single value in $\{0, 1, \dots, t\}$ (requiring $\log(t + 1)$ bits), and k hash functions (whose domain is the universe of possible items). For convenience, we assume the m cells are divided into k disjoint groups of size m/k , and that each group is the codomain of one hash function. Alternatively, the structure could be built so that all k hash functions hash into the entire set of cells. This decision does not affect the asymptotics. However, the partitioned version is usually easier to implement in hardware, although the unpartitioned version may give a lower false positive probability [9].

Each cell in the structure initially has value 0. When an item is inserted, we hash it according to each of the hash functions to obtain the set of cells corresponding to it. For each of these cells, we replace its value with the maximum of its value and the type of the item. Thus, any cell corresponding to an inserted item gives an overestimate of the type of the item, and if some cell corresponding to an item has value 0, that item is not in the set represented by the structure. The lookup operation is now obvious; to perform a lookup for an item x , we hash x using the k hash functions and compute the minimum z of the resulting counters, and then either declare that x is not represented by the summary (if $z = 0$), or that x has type at most z (if $z > 0$). Note that the lookup operation may give several different kinds of errors: *false positives*, where the summary returns a positive type for an element not in represented set, and *type j failures*, where the structure returns the incorrect type for an element of type j . The analysis of this structure now follows easily from the standard analysis of a Bloom filter [9].

Lemma 4.7.1. *Suppose that we insert a set S of n items into the structure described above. Then the probability that a particular item $x \notin S$ gives a false positive is*

$$(1 - (1 - k/m)^n)^k,$$

and if there are $\beta_j n$ items of type greater than j , then the probability that a specific item of type j causes a failure is

$$\left(1 - (1 - k/m)^{\beta_j n}\right)^k.$$

To use this structure as a summary for an MHT, we simply insert items into the structure as they are inserted into the MHT, and define the type of an item to be the sub-table of the MHT containing the item. (Of course, the type of an item is not well-defined if inserting it into the MHT causes a crisis; that is a different sort of failure that must be considered separately.) A false positive now corresponds to the case where the summary returns a positive type for an item not in the underlying MHT, and a type j failure now corresponds to the case where an item is in T_j in the underlying MHT, but the summary returns some other type when queried with that item. While false positives are not problematic if they appear sufficiently infrequently, we want to avoid any failures in our summary.²

In general, Lemma 4.7.1 can be used in conjunction with a union bound to bound the probability that there are any type j errors; if there are $\alpha_j n$ items of type j , then the probability that any type j failure occurs is at most

$$(\alpha_j n) \left(1 - (1 - k/m)^{\beta_j n}\right)^k \approx (\alpha_j n) \left(1 - e^{-k\beta_j n/m}\right)^k.$$

In our setting, Lemma 4.7.1 demonstrates that the most important tradeoff in constructing the summary is between the probability of a type 1 failure and the false positive probability, which both depend significantly on the numbers of hash functions used in the filter. Following the standard analysis from the theory of Bloom filters, to minimize type 1 failures, we would like $k = (\ln 2)m/\beta_1 n$. Typically this gives a rather large number of hash functions, which may not be suitable in practice. Further, this is far from the optimal number of hash functions to minimize false positives, which is $k = (\ln 2)m/n$, and therefore choosing such a large number of hash functions may make the false positive probability unreasonably high. In general, the choice of the number of hash functions must balance these two considerations appropriately.

There are other significant tradeoffs in structuring the MHT and the corresponding summary. Specifically, one can vary the number of sub-tables and their sizes in the MHT, as well as the size of the summary and the number of hash functions used. Generally, the more hash functions used in the MHT, the smaller the probability of a crisis (up to some point), but increasing the number of hash functions in the MHT increases the number of types, increasing the storage requirement of the summary structure. Moreover, the division of space in the MHT affects not only the crisis probability, but also the number of items of each type, which in turn affects the probability of failure.

As an aside, we note that several bit-level tricks can be used to minimize summary space. For example, three cells taking values in the range $[0, 5]$ can be packed into a single byte easily. Other similar techniques for saving bits can have a non-trivial impact on performance.

Asymptotically, choosing $m = \Theta(n \log n)$, $k = \Theta(\log n)$, and using $\Theta(\log \log \log n)$ bits per cell suffices to have the probability of failure vanish, for a total of

$$\Theta(n(\log n) \log \log \log n)$$

²Technically, we may wish to differentiate between the false positive probability and the false positive rate, as defined in Section 2.1.2, but the distinction is unimportant in practice. See Section 2.1.2 for an explanation.

bits. The constant factors in this approach can be made quite small by taking advantage of skew. We present a complete analysis in Section 4.11.2.

4.7.2 On Skew, and an Improved Construction

Lemma 4.7.1 highlights the importance of skew: the factor of β_j in the exponent drastically reduces the probability of a failure. Alternatively, the factor of β_j can be seen as reducing the space m required to achieve a certain false positive probability by a non-trivial constant factor.

In the construction above, the most likely failure is a type 1 failure; there are many fewer items of types greater than 1, and so there is very little probability for a failure for these items. A natural way to reduce the probability of a type 1 failure is to introduce more skew by making the size of the first sub-table larger (while still keeping linear total size). This can significantly reduce the number of elements of type larger than 1, shrinking β_1 , which leads to dramatic decreases in the total failure probability (the probability that for some j , some item causes a type j failure). That is, if one is willing to give additional space to the MHT, it is usually most sensible to use it in the first sub-table. We use this idea when designing specific instances of our constructions in Section 4.8.

A problem with using a single filter for classifying items of all types is that we lose some control, as in the tradeoff between false positives and type 1 errors. Taking advantage of the skew, we suggest a *multiple Bloom filter* approach that allows more control and in fact uses less space, at the expense of more hashing. Instead of using cells that can take on any of $t + 1$ values, and hence requiring roughly $\log_2(t + 1)$ bits to represent, our new summary consists of multiple Bloom filters, B_0, B_1, \dots, B_{t-1} . The first Bloom filter, B_0 , is simply used to determine whether or not an element is in the MHT; that is, it is a standard, classical Bloom filter for the set of items in the MHT. In convenient terms, it separates items of type greater than or equal to 1 from elements not in the MHT, up to some small false positive probability. (But note that if an element gives a false positive in B_0 , we do not care about the subsequent result.) Next, B_1 is a standard Bloom filter designed to represent the set of items with type greater than or equal to 2. An item that passes B_0 but not B_1 is assumed to be of type 1 (and therefore in the first sub-table of the MHT). A false positive for B_1 on an item of type 1 therefore leads to a type 1 failure, and hence we require an *extremely small* false positive probability for the filter to avoid such a failure. We continue on with B_2, B_3, \dots, B_{t-1} in the corresponding way (so the assumed type of an item x that passes B_0 is the smallest j such that x does not pass B_j , or t if x passes all of B_0, B_1, \dots, B_{t-1}).

Because of the skew, each successive filter can be smaller than the previous one without compromising the total failure probability. The skew is key for this approach to yield a suitably small overall size. Indeed, the total size using multiple Bloom filters will often be less than using a single filter as described in Section 4.7.1; we provide an example in Section 4.8. Further, by separating the filters, one can control the false positive probability and the probability of each type of error quite precisely. Also, by separating each type in this way, at some levels small Bloom filters could be replaced by lists of items, for example using a content addressable memory (CAM).

The only downside of this approach is that it can require significantly more hashing

than the others. For many applications, especially ones where all of the hashing computation is parallelized in hardware, this may not be a bottleneck. Also, it turns out that the calculations in Section 4.8 hold even if the hash functions used by the B_i 's are not independent, as long as all of the hash functions used in any particular B_i are independent. Thus, if ℓ is the least common multiple of the sizes of the codomains of the hash functions, we can just use a few hash functions with codomain $\{0, \dots, \ell - 1\}$, and compute hashes for the B_i 's by evaluating those hash functions modulo the sizes of the desired codomains.

As a further improvement, one might try to combine this technique with a variant of double hashing (as in [20] and Chapter 3) to reduce the total number of hash functions even further. We conducted some preliminary experiments along these lines, but found that despite the encouraging practical and asymptotic results surrounding double hashing, this approach does not seem to be effective in our setting because we require extremely small false positive probabilities. The situation appears to improve as we increase the number of hash functions (triple hashing, quadruple hashing, etc.), but our desire for extremely small false positive probabilities makes it impossible to prove the effectiveness of this modification through experiments. If this approach could somehow be proven viable, however, it would allow for a drastic reduction in the required amount of hashing computation, although it would not reduce the number of bits of the summary that must be examined for each lookup operation.

As for asymptotics, if we knew that for each $j = 1, \dots, d - 1$, there were at most $X_{\geq j}$ items of type at least j , then $\Theta(X_{\geq j} \log n)$ bits in B_j would suffice for the probability of a type j failure to vanish. Since $X_{\geq j} \leq n$ and $d = O(\log \log n)$, the failure probability can be made to vanish with $O(n(\log n) \log \log n)$ bits. However, it turns out that since the first $\log \log n + \Theta(1)$ of the $X_{\geq j}$'s decay doubly exponentially with high probability for a well-designed MHT, we can actually get the failure probability to vanish with $\Theta(n \log n)$ bits. We give the proof in Section 4.11.2.

4.8 Numerical Evaluation (Insertions Only)

In this section, we present constructions of our three summaries for 10,000 and 100,000 items and compare their various storage requirements, false positive probabilities, and failure probabilities. For completeness, we compare with results from Song *et al.* [56]. We continue to work in the setting where there are insertions, but not deletions, because handling deletions introduces too many subtle issues to allow for a straightforward comparison. Nevertheless, this restriction allows for a fair comparison against the scheme in [56], which requires additional structures to handle deletions.

For the MHT summaries, our preliminary goal is to use at most 6 buckets per item; this is less than 1/2 the size of the hash table (in terms of buckets) in [56], and seems like a reasonable objective. In designing the underlying MHTs, the guiding principle is to start with a standard MHT with sub-table sizes decreasing by a factor of 2, then enlarge the first sub-table to generate a lot of additional skew, and then increase the size of the last sub-table to match the second-to-last sub-table, which drives down the crisis probability. From here, we arrive at the following MHTs (where 'k' represents 1,000). For 10k items, there are 5 sub-tables, with sizes 40k, 10k, 5k, 2.5k, and 2.5k, giving a crisis probability less than

1.01×10^{-12} (calculated using the method of Section 4.5.1). For 100k items, there are 6 sub-tables, with sizes 400k, 100k, 50k, 25k, 12.5k, and 12.5k, giving a crisis probability less than 7.78×10^{-16} . Both of these crisis probabilities are dominated by the failure probabilities of the corresponding summaries (except in one case, with 10k items using multiple filters, where the crisis probability is still smaller than the failure probability).

Also, for the MHT summaries discussed in Section 4.7, we do not attempt to optimize all of the various parameters. Instead we simply exhibit parameters that simultaneously perform well with respect to all of the metrics that we consider. Also, we note that it is not practical to exactly compute the false positive and failure probabilities for these schemes. However, it is possible to efficiently compute estimates of these probabilities, and we have built a calculator for this purpose. We believe that our estimates are fairly tight upper bounds when the probabilities are very small, and so we use them as if they were the actual probabilities. For more details, see Section 4.11.3.

We configure the Bloom filter-based MHT summaries as follows. For 10k items, we configure our first summary to have 120k cells and 15 hash functions. When computing the storage requirement, we assume that 3 cells (each taking integral values in $[0,5]$) are packed into a byte. For the multiple Bloom filter summary, we use filters of sizes 106k, 87.5k, 5.5k, 500, and 100 bits, with 7 hash functions for the first filter and 49 for each of the others.³ For 100k items, we configure the first Bloom filter based summary to have 1.2m cells (where ‘m’ represents one million) and 15 hash functions, and here we use three bits for each cell (taking integral values in $[0,6]$). We configure the multiple Bloom filter summary to have filters of sizes 1.06m, 875k, 550k, 1k, and 1k, with 7 hash functions for the first filter and 49 for each of the others.

The results of our calculations are given in Table 4.3. In that table, IS denotes the interpolation search scheme of Section 4.6, SF denotes the single filter scheme of Section 4.7.1, and MBF denotes the multiple Bloom filter scheme of Section 4.7.2. We configure the interpolation search summary according to the examples in Section 4.6. The notation ‘*’ for the Song *et al.* [56] summary denotes information not available in that work. All storage requirements for our summaries include the space for the bit table mentioned in Section 4.4.

In the last column of Table 4.3, note that the sum of the failure and crisis probabilities can be thought of as a bound on the overall probability that a scheme does not work properly. (Also, as mentioned previously, except in the case of 10k items with multiple filters, the failure probability dominates.) As can be seen in the table, interpolation search performs extremely well at the expense of a fairly large summary. The single filter scheme appears comparable to the structure of [56] for 10k items, but uses much less space. The multiple filter scheme allows further space gains with just slightly more complexity. Our schemes also appear quite scalable; for 100k items, we can maintain a ratio of 6 buckets per item in the hash table, with just a slightly superlinear increase in the summary space for our proposed schemes.

For the scheme presented by Song *et al.*, there is no failure probability as we have described, as an item will always be in the location given by the summary. There may,

³For the multiple Bloom filter construction, we use unpartitioned Bloom filters, so the number of hash functions need not divide the size of a filter.

Table 4.3: Numerical Results

Scheme	Hash Table Size (buckets)	Summary Space (bytes)	False Positive Probability	Failure + Crisis Probability
[56]	131072	49152	0.002	*
IS	60000	80000	2.78×10^{-13}	1.39×10^{-9}
SF	60000	47500	0.006	7.64×10^{-10}
MBF	60000	32450	0.006	4.97×10^{-12}

(a) 10k items

Scheme	Hash Table Size (buckets)	Summary Space (bytes)	False Positive Probability	Failure + Crisis Probability
[56]	*	*	*	*
IS	600000	875000	4.34×10^{-14}	2.17×10^{-9}
SF	600000	525000	0.006	7.27×10^{-9}
MBF	600000	379000	0.006	1.38×10^{-11}

(b) 100k items

however, be a crisis, in that some bucket may have more than one item. (Technically, there can be a failure, because they use only three-bit counters with ten hash functions; however, the probability of a failure is very, very small and can be ignored.) They do not have any numerical results for the crisis probability of their scheme when including their heuristics, and hence we leave a ‘*’ in our table of results. However, they do report having found no crisis in one million trials. Finally, we note that [56] does not include results for 100k items. Since it is not clear how to properly configure that scheme for 100k items, we do not attempt to analyze it.

It is worth noting that our improvements in summary space over the scheme in [56] are not as dramatic as the improvement in hash table size. The intuitive explanation for this phenomenon is that the hash table in [56] seems to require $\Theta(n \log n)$ buckets in the hash table, while the MHT requires only $\Theta(n)$ buckets, giving our schemes a factor $\Theta(\log n)$ reduction in the size of the hash table. However, from Sections 4.3, 4.6, 4.7.1, and 4.7.2, we know that if we require the failure probability to vanish, the summary in [56] seems to require $\Theta(n(\log n) \log \log n)$ bits, the interpolation search summary requires $\Theta(n \log n)$ bits, the single filter summary requires $\Theta(n(\log n) \log \log \log n)$ bits, and the multiple Bloom filter summary requires $\Theta(n \log n)$ bits. Thus, while these summaries seem to require fewer bits than the one in [56], the gain appears to be smaller than the factor $\Theta(\log n)$ reduction in hash table size.

4.9 Experimental Validation (Insertions Only)

Ideally, we would be able to directly verify the extremely low failure probabilities given in the previous section through experiments. However, since the probabilities are so small, it is impractical to simulate the construction of the summaries sufficiently many times to accurately estimate the real failure probabilities. We have attempted to validate the calculator we have developed for the summaries based on Bloom filters, and have found that it does give an upper bound in all of our tests. In fact it can be a fairly weak upper bound when the failure probability is very large (greater than 0.1, for example). In all our experiments, we simulated random hashing by fixing hashes for each item using a standard 48-bit pseudorandom number generator.

We have simulated the single filter for 10k items in Table 4.3; in one million simulations, we saw no errors or crises, as predicted by our calculations. We also experimented with a variant on this filter with only 100k counters and 10 hash functions. Our calculations for this filter gave an upper bound on the probability of failure of just over 2.1×10^{-6} ; in one million trials, we had one failure, a natural result given our calculations.

While further large-scale experiments are needed, our experiments thus far have validated our numerical results.

4.10 Deletions

Handling deletions is substantially more difficult than handling insertions. For example, the scheme proposed in [56] for handling deletions requires significant memory; it essentially requires a separate version of the hash table that records all of the hash locations of every item. Moreover, deletions can require significant repositioning of elements in the hash table. To address these issues, we explore two deletion paradigms: lazy deletions and counter-based deletion schemes.

4.10.1 Lazy Deletions

A natural, general approach is to use *lazy* deletions. That is, we keep a *deletion bit* array with one bit for each cell in the hash table, initially 0. When an item is deleted from some bucket b , we simply set the deletion bit corresponding to b to 1. When looking up an item, we treat it as deleted if we find it in a bucket whose deletion bit is 1. When a preset number of deletions occurs, when the total number of items in the hash table reaches some threshold, or after a preset amount of time, we can reconstruct the entire data structure (that is, the hash table, the deletion bit array, and the summary) from scratch using the items in the hash table, leaving out the deleted items. If we want to guarantee good performance whenever there are at most αn deleted items and at most n undeleted items in the hash table, it suffices to simply build our data structures to be able to cope with $(1 + \alpha)n$ items, rebuilding them whenever there are at least $(1 + \alpha)n$ items in the hash table, at least αn of which are marked for deletion. The obvious disadvantage of this approach is that expensive reconstruction operations are necessary, potentially frequently, depending on how often insertions occur. Also, extra space is required to maintain the deleted items until this reconstruction occurs.

However, reconstruction operations are much cheaper than one might expect. Indeed, the time required to perform the MHT reconstruction is essentially determined by the number of items in the MHT that must be moved, and the time required to perform the summary reconstruction is essentially just the time required to scan the MHT and rehash all of the items using the summary hash functions. We find that the MHT skew property ensures that very few items need to be moved in the MHT during a reconstruction, which tells us that the MHT reconstruction is much less expensive than the analogous procedures for other multiple choice hash tables.

As for the scan of the MHT required to rebuild the summary, we note that while this procedure may be somewhat expensive, its simplicity may be an asset, and it may be cheaper than moving the items in the MHT. Also, the cost of this scan can be ameliorated in various ways. For example, if we store the summary hash values of the items in a structure analogous to the MHT (in slow memory), then we only need to scan the hash values of the items when reconstructing the summary, as opposed to scanning the items themselves, which might be much larger. Using the hash reduction techniques discussed in Section 4.7.2 reduces the storage requirement of this data structure, further reducing the cost of the scan.

We now focus on the number of items that must be moved during a reconstruction of the MHT. We consider a natural implementation of the MHT rebuilding process. Before proceeding, recall that the MHT consists of tables T_1, \dots, T_d and corresponding hash functions h_1, \dots, h_d , and that an item x should be placed in $T_j[h_j(x)]$, where j is as small as possible subject to $T_j[h_j(x)]$ being unoccupied. The natural MHT reconstruction algorithm is then as follows: for $i = 1, \dots, d$, iterate over the items in T_i (using the bucket occupancy bit table described in Section 4.4), and for each item x in T_i , determine (again using the bucket occupancy table) if there is some smallest $j < i$ such that $T_j[h_j(x)]$ is empty, and move x to $T_j[h_j(x)]$ if this is the case. (Of course, a bucket is considered empty if its deletion bit is set; we move an item by copying it to its destination and then marking its origin as deleted; we update the occupancy bit table as we go; and at the end of the algorithm, we reset all of the deletion bits to 0.)

Consider, for example, the case where we reconstruct an MHT with $(1 + \alpha)n$ items, exactly αn of which are marked as deleted. For the moment, assume that the deleted items are chosen randomly. While this assumption may be unrealistic, it allows us to analyze the number M of items moved and gives a good indication of general performance. To this end, we let M' be the number of items x that are not deleted and are in some table $j > 1$ such that at least one of the items in $T_1[h_1(x)], \dots, T_{j-1}[h_{j-1}(x)]$ is deleted. We claim that $\mathbf{E}[M']$ can be used as an approximate upper bound on $\mathbf{E}[M]$. Indeed, if, for example, an item y in T_1 is deleted, it is intuitively likely that there are a few items x with $h_1(x) = h_1(y)$, but unlikely that all of those items are moved. However, those items x that are not moved are counted in M' but not M , which suggests that $\mathbf{E}[M']$ is an approximate upper bound for $\mathbf{E}[M]$. The only complication is that an item x in T_j for some $j > 2$ can be moved even if none of $T_1[h_1(x)], \dots, T_{j-1}[h_{j-1}(x)]$ are deleted; one of those items could simply move. But in practice we expect that almost all moves will be from items in the second sub-table to buckets in the first sub-table, and so we expect the effect of this complication to be minimal.

To estimate $\mathbf{E}[M']$, we consider some item x in table $j > 1$. The probability that

x is not deleted is clearly

$$\prod_{i=0}^{\alpha n-1} \left(1 - \frac{1}{(1+\alpha)n-i} \right),$$

and given that x is not deleted, the probability that there is some $i < j$ such that the item at $T_i[h_i(x)]$ is deleted is

$$1 - \frac{\binom{(1+\alpha)n-j}{\alpha n}}{\binom{(1+\alpha)n-1}{\alpha n}} = 1 - \prod_{i=1}^{j-1} \frac{1}{1 + \frac{\alpha n}{n-i}}.$$

From these probabilities and the expected numbers of items in sub-tables when $(1+\alpha)n$ items are inserted (which are obtainable using the method of Section 4.5.1), we can easily compute $\mathbf{E}[M']$. Asymptotic high probability bounds can then be obtained by standard martingale techniques (for example, [42, Section 12.5]).

We give a concrete example using a sample MHT for 10,000 items discussed in Section 4.8. The MHT consists of 5 sub-tables, with sizes 40k, 10k, 5k, 2.5k, and 2.5k, respectively (where k denotes 1,000). We set $\alpha = 0.1$ and $n = 9090$, so that $(1+\alpha)n < 10,000$. Performing the calculations above, we find that $\mathbf{E}[M'] \approx 100.02$. That is, only about 1.1% of the n items are moved on average during an MHT reconstruction! This result shows that periodic MHT reconstructions in the standard lazy deletion scheme are likely to be significantly less expensive than full reconstructions.

To confirm the correctness of our calculations, we estimated the expected number of moves required by an MHT reconstruction in a simple experiment. We averaged the required number of moves over 100,000 trials, where each trial consisted of inserting $(1+\alpha)n$ items into an initially empty MHT, deleting αn of those items at random, and then counting the number of moves required to reconstruct the MHT. The resulting estimate of $\mathbf{E}[M]$ was 96.98, which is smaller than but close to the calculated value, as expected. The minimum and maximum observed values of M were 58 and 113, respectively, demonstrating that M is unlikely to deviate too far above $\mathbf{E}[M']$.

While it may be initially surprising that reconstructing the MHT requires so few moves, there is some simple intuition behind the result. Indeed, Table 4.2 suggests that (approximately) 88% of the $(1+\alpha)n = 1.1n$ items are placed in the first sub-table and the rest are placed in the second sub-table. Under this assumption, an item is moved only if it is initially placed in the second sub-table, it is not deleted, and the item at its hash location in the first sub-table is deleted. But only 12% of the items are initially placed in the second sub-table, and only about $1 - \alpha = 90\%$ of them are not deleted, and of those that remain, only about $\alpha = 10\%$ hash to buckets in the first sub-table that contain deleted items. Thus, the fraction of the n items that are moved is about $(1.1)(0.12)(0.9)(0.1) \approx 1.2\%$ (neglecting the fact that only one item in the second sub-table can be moved into a particular bucket in the first sub-table), closely matching the results above.

One might reasonably wonder how dependent these results are on our assumption that deletions occur randomly. As evidence that the results are fairly robust, we now consider a more pessimistic deletion model. As before, we assume that we perform a reconstruction of an MHT with $(1+\alpha)n$ items, exactly αn of which are marked as deleted. However, rather than assuming that the αn deleted items are chosen randomly from all

$(1 + \alpha)n$ items, we assume that they are chosen randomly from the items in the first sub-table of the MHT. (For a well-designed MHT, it is overwhelming likely that there are at least αn items in the first sub-table.) Then, as before, we focus on the number M of items that must be moved.

To analyze $\mathbf{E}[M]$, we let S_1 denote the set of items not placed in first sub-table. We let M' be as before, and expect that $\mathbf{E}[M']$ is an approximate upper bound on $\mathbf{E}[M]$. Clearly, if $|S_1| > n$, then $M' = n$. Next, consider the case where $|S_1| \leq n$. Then any particular item x of any type $j > 1$ is counted in M' if and only if the item at $T_1[h_1(x)]$ is deleted, which occurs with probability

$$1 - \prod_{i=0}^{\alpha n - 1} \left(1 - \frac{1}{(1 + \alpha)n - |S_1| - i} \right).$$

By linearity of expectation, we conclude that

$$\begin{aligned} \mathbf{E}[M' \mid |S_1|] &= \begin{cases} n & \text{if } |S_1| > n \\ |S_1| \left(1 - \prod_{i=0}^{\alpha n - 1} \left(1 - \frac{1}{(1 + \alpha)n - |S_1| - i} \right) \right) & \text{otherwise.} \end{cases} \end{aligned}$$

which is easily computed. We can then compute the distribution of $|S_1|$ (using the method of Section 4.5.1) and use it to calculate $\mathbf{E}[M']$.

We examined this deletion model for the same MHT and parameters as before. We calculated that $\mathbf{E}[M'] \approx 118.34$, which is approximately 1.3% of the n items. Through an experiment (with 100,000 trials, as before), we also estimated $\mathbf{E}[M]$ as about 114.25. Over the course of the experiment, the smallest and largest observed values of M were 70 and 166, respectively. We concluded that M is reasonably concentrated around $\mathbf{E}[M]$.

We can also predict the above results by slightly modifying the intuitive reasoning for the original deletion model. As before, about 8848 of the $(1 + \alpha)n = 1.1n = 9999$ items are placed in the first sub-table. The probability that any particular item x of the remaining 1151 items must be moved is therefore about $\alpha n / 8848 \approx .1$. Therefore, the fraction of the n items that must be moved is about $(1.1)(1151/9999)(0.1) \approx 1.3\%$ (again neglecting the fact that only one item in the second sub-table can be moved into a particular bucket in the first sub-table).

One might hope to improve these results by allowing items marked for deletion in the MHT to be overwritten by newly inserted items. We do not provide a detailed analysis of this approach, but do give some basic caveats. First, we expect any gains to be minor, given the excellent performance of the basic lazy deletion scheme. Second, one must beware of additional pollution in the summary. In the specific case of our Bloom filter-based summaries, when an item x that is marked for deletion is overwritten by a new item, x cannot be simply removed from the summary (just as a standard Bloom filter does not support deletions). But leaving x in the summary while adding the new item adds noise to the summary, increasing its false positive rate and failure probability. Furthermore, all future queries for x will now result in false positives. In situations where a recently deleted item is likely to be the subject of a lookup query, the latter issue may constitute a

very serious problem. Of course, both summary pollution issues could be ameliorated by increasing the size of the summary and/or adding additional data structures (e.g., to track overwritten items), but such modifications add non-trivial overhead and complications. Therefore, we believe that for most applications, the best approach is likely to be either the standard lazy deletion scheme analyzed above or one of the counter-based schemes that we discuss next in Section 4.10.2.

4.10.2 Counter-Based Deletion Schemes

While the lazy deletions schemes of Section 4.10.1 may be appropriate for many applications, their reliance on periodic reconstructions of the summary might preclude their use in certain situations, such as very high speed data streams. In other words, one might require good worst case time bounds for hash table operations and be unable to settle for the amortized bounds offered by lazy deletion schemes. Of course, in order to achieve good worst case bounds, the underlying hash table must allow for easy deletions. Indeed, an item can be deleted from an MHT simply by removing it from its bucket; no repositioning of the elements is necessary. This observation is another excellent reason for using the MHT as our underlying hash table.

Now, whenever an item is deleted from its bucket in the MHT, our summary must be updated to reflect the deletion. The Bloom filter-based summaries of Section 4.7 can be easily modified so that these updates can be performed quickly. For the single filter summary of Section 4.7.1, we require that each cell now contain one counter for each type, and that each counter tracks the number of items in the MHT of the corresponding type that hash to the cell containing the counter. Both insertions and deletions can now be performed extremely quickly, simply by incrementing or decrementing the appropriate counters. A similar modification works for the multiple Bloom filter summary of Section 4.7.1; we simply replace each Bloom filter by a counting Bloom filter.

Unfortunately, these modified summaries consume much more space than the originals. However, the space requirements of the modified Bloom filter-based summaries can be minimized by aggressively limiting the number of bits used for each counter. Of course, we must guarantee that the probability that some counter overflows is extremely small, since the existence of an overflow can eventually lead to the summary returning an incorrect answer.

Choosing the appropriate number of bits for a counter therefore requires some work. First, we observe that if there are n elements associated with m counters (all initially 0), and for each element we increment c randomly chosen counters, then the distribution of the resulting maximum counter value is the same as the distribution of the maximum load resulting from throwing nc balls randomly into m bins. If nc/m is not too small, we can derive nontrivial high probability bounds for this distribution using the Poisson approximation [42, Section 5.4]. For the Bloom filter-based summaries, this approach allows us to keep the sizes of the counters reasonable while simultaneously ensuring that, with high probability, no false negatives or type j failures occur, for all but the largest couple values of j . This approach is effective until the expected number of items in a table becomes so small that either the variance is too big or the Poisson approximation is inaccurate.

One approach might be to avoid the problems caused by small sub-tables by re-

placing them with a CAM. Alternatively, if we use small sub-tables, the expected number of items that hash to any counter in them is so small each counter can be represented by a bit or two. A detail that must be dealt with is that for a small table, if multiple hash functions for an item being inserted (or deleted) hash to the same counter, that counter should be incremented (or decremented) only once. Otherwise, the insertion of a single item could, with small but non-negligible probability, result in a particular counter being incremented multiple times. This issue is not problematic when we have larger counters and hash tables, and it does introduce some overhead, so we only modify the increment and decrement operations in this way for smaller tables, which are rarely used (by the MHT skew property).

As a concrete example of our design techniques, we modify the multiple bloom filter summary for 10,000 items given in Section 4.8 to use counting Bloom filters with appropriately sized counters. Based on the heuristic calculations described above, we suggest using the modified insertion and deletion operations in the last two filters, using 4 bits per counter in the first three filters, 3 or 4 bits per counter in the fourth filter, and 1 or 2 bits per counter in the last filter. We suspect that the more conservative choices might be necessary to obtain failure probabilities comparable to those in Section 4.8, but that the less conservative ones are still fairly effective.

The two choices give summaries of essentially the same size. For the conservative choices, we can easily build the summary so that it uses 99775 bytes (with 2 4-bit counters per byte in the first four filters, and 4 2-bit counters in the last filter), neglecting the 7500 bytes needed for the bit table described in Section 4.4. The less conservative choices yield a summary that uses at least 99700 bytes (neglecting byte-packing issues for the 3-bit counters). In both cases, the total storage requirement of the summary (including the 7500 byte bit table) is essentially 3.3 times that of the corresponding summary in Section 4.8.

To test the resulting data structure, we instantiated the summary (with 16-bit counters) and the underlying MHT with 10,000 items one million times and recorded the largest counter values v_0, \dots, v_4 ever seen in each of the filters B_0, \dots, B_4 . The results were $v_0 = 12$, $v_1 = 11$, $v_2 = 12$, $v_3 = 4$, and $v_4 = 1$. We concluded that the results of the experiment were consistent with the results of our heuristic analysis. More detailed analysis and more extensive simulations could provide more insight into appropriate values for extremely small failure probabilities comparable to those in Section 4.8.

4.11 Additional Technical Details

We now delve into some of the technical details that we avoid earlier in this chapter. In particular, we derive the asymptotic bound on the crisis probability promised in Section 4.5. We also examine the asymptotics of the Bloom filter-based summaries of Section 4.7. Finally, we give detailed explanations of the calculation methods that we use throughout this chapter.

4.11.1 An Asymptotic Bound on the Crisis Probability

This section is devoted to the following result, which we consider to be the theoretical justification for the MHT's extremely low crisis probabilities.

Theorem 4.11.1. *Suppose we hash n items into an MHT with tables T_1, \dots, T_d (with corresponding fully random hash functions h_1, \dots, h_d), where the size of T_i is $m'_i = \lceil m_i \rceil$ for $m_i = c_2^{i-1} c_1 n$, for any constants $c_1 > 1$ and $c_2 < 1$ with $c_1 c_2 > 1$. Then for any constant $c > 0$, we can choose $d = \log \log n + \Theta(1)$ so that the probability that a crisis occurs is $o(n^{-c})$.*

Proof. We begin the proof with a sequence of lemmas.

Lemma 4.11.1. *Let S_0 denote the set of items being hashed, and for $i = 1, \dots, d$, let S_i denote the set of items not placed in the first i tables. Then for $i \geq 1$ and any $B \geq |S_{i-1}|^2/m_i$, we have $\mathbf{E}[|S_i| \mid |S_{i-1}|] \leq |S_{i-1}|^2/2m_i$ and $\Pr(|S_i| > B \mid |S_{i-1}|) < (e/4)^{B/2}$.*

Proof. Condition on $S_{i-1} = \{x_1, \dots, x_\ell\}$. For $k = 1, \dots, \ell$, let Y_k indicate whether there exists $j < k$ with $h_i(x_j) = h_i(x_k)$, so that $|S_i| = \sum_k Y_k$. Also, let Z_1, \dots, Z_ℓ be independent 0/1 random variables with $\mathbf{E}[Z_k] = \min(1, (k-1)/m_i)$, let $Z = \sum_k Z_k$, and note that $\mathbf{E}[Z] \leq |S_{i-1}|^2/2m_i \leq B/2$. It is easy to see that

$$\begin{aligned} \Pr(Y_k = 1 \mid \{h_i(x_j) : j < k\}) \\ = |\{h_i(x_j) : j < k\}|/m'_i \leq \mathbf{E}[Z_k]. \end{aligned}$$

It follows that $\Pr(Y_k = 1 \mid Y_1, \dots, Y_{k-1}) \leq \mathbf{E}[Z_k]$. Thus,

$$\mathbf{E}[Y_{j_1} \cdots Y_{j_r}] \leq \mathbf{E}[Z_{j_1} \cdots Z_{j_r}]$$

for any j_1, \dots, j_r , implying that $\mathbf{E}[|S_i|^j] \leq \mathbf{E}[Z^j]$ for any integer $j \geq 0$. Now for any $t > 0$,

$$\mathbf{E}[e^{t|S_i|}] = \sum_{j=0}^{\infty} \frac{t^j \mathbf{E}[|S_i|^j]}{j!} \leq \sum_{j=0}^{\infty} \frac{t^j \mathbf{E}[Z^j]}{j!} = \mathbf{E}[e^{tZ}].$$

We can now complete the proof by deriving a Chernoff bound in the usual way (e.g., [42, Theorem 4.4]). \square

Lemma 4.11.2. *Let $\{z_i\}_{i \geq 0}$ be a sequence where $z_0 = n$ and $z_i = z_{i-1}^2/m_i$ for $i \geq 1$. Then for $i \geq 0$, we have $z_i = (1/c_1 c_2)^{2^i - 1} c_1^i n$.*

Proof. The proof is an easy induction on $i \geq 0$. \square

Lemma 4.11.3. *For any events A_1, \dots, A_ℓ ,*

$$\Pr\left(\bigcup_i A_i\right) \leq \sum_i \Pr\left(A_i \mid \bigcap_{j < i} \neg A_j\right).$$

Proof. This result is standard, and the proof is trivial. \square

Lemma 4.11.4. For any $i \geq 1$, we have

$$\Pr(\exists j \leq i : |S_j| > z_j) \leq i(e/4)^{z_i/2}.$$

Proof. Applying Lemmas 4.11.3, 4.11.1, and 4.11.2 gives

$$\begin{aligned} \Pr(\exists j \leq i : |S_j| > z_j) &\leq \sum_{j=1}^i \Pr(|S_j| > z_j \mid |S_{j-1}| \leq z_{j-1}) \\ &\leq \sum_{j=1}^i (e/4)^{z_j/2} \leq i(e/4)^{z_i/2}. \end{aligned}$$

□

We are now ready to prove the theorem. By Lemma 4.11.2, we can choose $r = \log \log n + \Theta(1)$ and obtain

$$2(c+1) \log_{4/e} n \leq z_r = O(\sqrt{n}),$$

so Lemma 4.11.4 implies that

$$\Pr(|S_r| > z_r) < rn^{-c-1} = o(n^{-c}).$$

Since $z_r^2/m_{r+1} = O(\log n)$, Lemma 4.11.1 tells us that we can choose some $w = O(\log n)$ so that

$$\Pr(|S_{r+1}| > w \mid |S_r| \leq z_r) \leq n^{-(c+1)} = o(n^{-c}).$$

Markov's inequality and Lemma 4.11.1 now imply that for $i \geq 1$ and any $w' \leq w$

$$\begin{aligned} \Pr(|S_{r+1+i}| \geq 1 \mid |S_{r+i}| = w') \\ \leq \mathbf{E}[|S_{r+1+i}| \mid |S_{r+i}| = w'] \leq w^2/2m_{r+1+i} = O\left(\frac{\log n}{n}\right). \end{aligned}$$

Since $|S_0| \geq |S_1| \geq \dots \geq |S_d|$,

$$\begin{aligned} \Pr(|S_{r+1+\lceil c+1 \rceil}| \geq 1 \mid |S_{r+1}| \leq w) \\ \leq \prod_{i=1}^{\lceil c+1 \rceil} \Pr(|S_{r+i+1}| > 1 \mid |S_{r+i}| \leq w) \\ \leq O\left(\frac{\log n}{n}\right)^{\lceil c+1 \rceil} = o(n^{-c}). \end{aligned}$$

Setting $d = r + 1 + \lceil c + 1 \rceil$ and applying Lemma 4.11.3 gives

$$\begin{aligned} \Pr(|S_d| \geq 1) &\leq \Pr(|S_r| > z_r) + \Pr(|S_{r+1}| > w \mid |S_r| \leq z_r) \\ &\quad + \Pr(|S_d| \geq 1 \mid |S_{r+1}| \leq w) \\ &= o(n^{-c}), \end{aligned}$$

completing the proof. □

4.11.2 Asymptotics of the Bloom Filter Summaries

This section provides rigorous analyses of the asymptotics for the Bloom filter-based summaries of Section 4.7.

The Single Filter Summary

We show that for any constant $c > 0$, it is possible to construct the single filter summary of Section 4.7.1 so that it has failure probability $o(n^{-c})$ and requires $m = O(n \log n)$ bits and $k = O(\log n)$ hash functions. We assume that the filter is partitioned into k sub-filters of size m/k , one for each hash function, although our analysis can be modified for an unpartitioned filter.

Proceeding, we set $c' = (c+2)/\ln 2$, and then choose the smallest $m \geq c'n \log_{10/7} n$ such that for $k = \lfloor (m/n) \ln 2 \rfloor$, we have that k divides m . The probability that a particular item gives a failure, regardless of its type, is then at most

$$\begin{aligned} (1 - (1 - k/m)^n)^k &\leq \left(1 - e^{-nk/m - nk^2/m^2}\right)^k \\ &\leq \left(1 - e^{-\ln 2 - (\ln 2)^2/n}\right)^k \\ &\leq (0.7)^k \leq (10/7)n^{-c-2}, \end{aligned}$$

where the first step follows from the inequality $1 - x \geq e^{-x-x^2}$ for $x < 1/2$. Taking a union bound over all n items yields a total failure probability of $o(n^{-c})$.

The Multiple Bloom Filter Summary

We now show that for the MHT of Theorem 4.11.1, it is possible to construct a multiple Bloom filter summary with failure probability $o(n^{-c})$ that uses $O(n \log n)$ bits, where $c > 0$ is the same constant as in Theorem 4.11.1.

We continue to use the notation introduced in the statement and proof of Theorem 4.11.1. As in Section 4.7.2, let B_0, \dots, B_{d-1} denote the Bloom filters in the summary. Let b_j denote the size of B_j , and let k_j denote the number of hash functions used by B_j . For simplicity, we assume that each B_j is partitioned into k_j sub-filters of size b_j/k_j , one for each hash function, although our analysis can be modified for unpartitioned filters.

By the same argument as in Section 4.11.2, conditioned on there being at most y_{j-1} elements of type at least j , if $b_j \geq c'y_{j-1} \log_{10/7} n$ for $c' = (c+2)/\ln 2$ and $k_j = \lfloor (b_j/n) \ln 2 \rfloor$, then the probability that any particular item of type j yields a type j failure is at most $(10/7)n^{-(c+2)}$. Taking a union bound over all n items gives an overall failure probability of $o(n^{-c})$.

To complete the proof, it suffices to show that we can choose the y_j 's so that $\sum_{j=0}^{d-1} y_j = O(n)$ and $\Pr(\exists j : |S_j| > y_j) = o(n^{-c})$. To this end, we set $y_0 = n$, followed by

$y_j = (1/c_1c_2)^{2^j-1}n$ for $j = 1, \dots, r$, and then $y_j = y_r$ for $j = r + 1, \dots, d - 1$. Then

$$\begin{aligned} \sum_{j=0}^{d-1} y_j &\leq (d-r)n + \sum_{j=1}^r y_j \\ &\leq (d-r)n + n \sum_{j=1}^{\infty} (1/c_1c_2)^{2^j-1} = O(n), \end{aligned}$$

where we have used the fact that $d-r = O(1)$ and we have bounded the sum by a geometric series. For the high probability result, observe that by definition of the y_j 's,

$$\Pr(\exists j : |S_j| > y_j) = \Pr(\exists 1 \leq j \leq r : |S_j| > y_j).$$

Lemma 4.11.2 implies that $z_j \leq y_j$, and therefore we may apply Lemma 4.11.4 to conclude that this probability is $o(n^{-c})$.

4.11.3 Calculating Various Quantities of Interest

This section describes the calculator that we use in Section 4.8. It is fairly easy to implement, although some care is required to ensure that the computation is efficient and that the memory requirement is reasonable.

Performing the MHT Calculations

Consider an MHT with tables T_1, \dots, T_d , where T_i has size m_i . Let S_0 be a set of size n items hashed into the MHT, and for $i = 1, \dots, d$, let S_i be the set of items not placed in T_1, \dots, T_i . We show how to compute the individual marginal distributions of the $|S_i|$'s. In particular, this allows us to compute the crisis probability of the MHT: $1 - \Pr(|S_d| = 0)$.

First, note that conditioned on $|S_{i-1}|$, the distribution of $|S_{i-1}| - |S_i|$ is the same as the distribution of the number of nonempty bins resulting from randomly throwing $|S_{i-1}|$ balls into m_i bins. Letting $p_{j,m,b}$ denote the probability that randomly throwing j balls into m bins yields exactly b nonempty bins, we have that for $b = 1, \dots, m$,

$$p_{j,m,b} = p_{j-1,m,b-1}(1 - (b-1)/m) + p_{j-1,m,b}(b/m). \quad (4.3)$$

Letting $P_{j,i}[b] = p_{j,m_i,b}$ for $b = 0, \dots, m_i$, we can now compute the individual marginal distributions of the $|S_i|$'s using the following pseudo-code:

```

Set  $\Pr(|S_i| = \ell) = 0$  for  $i = 0, \dots, d$  and  $\ell = 1, \dots, n$ .
Set  $\Pr(|S_0| = n) = 1$ .
for  $i = 1$  to  $d$  do
  Set  $P_{0,i}[0] = 1$  and  $P_{0,i}[j] = 0$  for  $j > 0$ .
  for  $j = 1$  to  $n$  do
    Compute  $P_{j,i}$  from  $P_{j-1,i}$  using (4.3).
    for  $\ell = 1$  to  $j$  do
       $\Pr(|S_i| = \ell) += \Pr(|S_{i-1}| = j) \cdot P_{j,i}[j - \ell]$ 
    end for
  end for

```

end for
end for

Of course, $\Pr(|S_{i-1}| = j)$ is typically negligible for sufficiently large j , so we can optimize the pseudo-code to greatly reduce the number of iterations of the second loop.

The Failure Probability of the Single Filter Summary

Consider an instance of the single filter summary of Section 4.7.1 with m cells and k hash functions. We show to compute upper bounds on the various failure probabilities of the summary that we believe to be nearly tight when those probabilities are small.

We continue to use the notation of Section 4.11.3. First, note that if $|S_{i-1}| = j$ and $|S_i| = \ell$, then the probability that a particular item of type i yields a failure is

$$\left(1 - (1 - k/m)^\ell\right)^k \triangleq q_\ell,$$

and so the conditional probability that any type i failure occurs is at most $(j - \ell)q_\ell$, by a union bound. We believe that this bound is very tight when q_ℓ is small, since other Bloom filter results (specifically, those in Chapter 3) suggest that these $j - \ell$ potential failures are almost independent, and the union bound is extremely accurate for independent events with very small probabilities.

Now, the conditional distribution of $|S_{i-1}| - |S_i|$ given $|S_{i-1}|$ is the same as the distribution of the number of nonempty bins resulting from randomly throwing $|S_{i-1}|$ balls into m_i bins. Therefore, given that $|S_{i-1}| = j$, the probability that any type i failure occurs is at most

$$\sum_{\ell=1}^{j-1} P_{j,i}j - \ellq_\ell \triangleq f_{i,j} \quad (4.4)$$

and so the overall probability that any type i failure occurs is at most

$$\sum_{j=1}^n \Pr(|S_{i-1}| = j) f_{i,j} \triangleq f_i.$$

By another union bound, the total failure probability is at most $\sum_{i=1}^{d-1} f_i \triangleq f$. Once again, we believe that this bound is fairly tight when the f_i 's are small.

We can now compute bounds on the various failure probabilities with the following pseudo-code:

```

for  $i = 1$  to  $d - 1$  do
  Set  $f_i = 0$ .
  for  $j = 1$  to  $n$  do
    Compute  $P_{j,i}$  from  $P_{j-1,i}$  using (4.3).
    Compute  $f_{i,j}$  using (4.4).
     $f_i \text{ += } \Pr(|S_{i-1}| = j) \cdot f_{i,j}$ 
  end for
end for
Compute  $f = \sum_{j=1}^{d-1} f_j$ .

```

As in Section 4.11.3, $\Pr(|S_{i-1}| = j)$ is typically negligible for sufficiently large j , so we can optimize the pseudo-code to greatly reduce the number of iterations of the second loop.

The Failure Probability of the Multiple Bloom Filter Summary

Consider an instance of the multiple Bloom filter summary of Section 4.7.2 with filters B_0, \dots, B_{d-1} , where B_i has b_i bits and k_i hash functions. We show to compute estimates of the various failure probabilities of the summary that we believe to nearly tight upper bounds when those probabilities are small and the number n of items is large.

We continue to use the notation of Section 4.11.3. First, note that if $|S_{i-1}| = j$ and $|S_i| = \ell$, then the probability that a particular item of type i yields a failure is⁴

$$\left(1 - (1 - k_i/b_i)^\ell\right)^{k_i} \triangleq q_{\ell,i},$$

and so the conditional probability that any type i failure occurs is at most $(j - \ell)q_{\ell,i}$, by a union bound. As in Section 4.11.3, we believe that this bound is tight when $q_{\ell,i}$ is small.

Now, the conditional distribution of $|S_{i-1}| - |S_i|$ given $|S_{i-1}|$ is the same as the distribution of the number of nonempty bins resulting from randomly throwing $|S_{i-1}|$ balls into m_i bins. Therefore, given that $|S_{i-1}| = j$, the probability that any type i failure occurs is at most

$$\sum_{\ell=1}^{j-1} P_{j,i}j - \ellq_{\ell,i} \triangleq f_{i,j} \quad (4.5)$$

and so the overall probability that any type i failure occurs is at most

$$\sum_{j=1}^n \Pr(|S_{i-1}| = j) f_{i,j} \triangleq f_i.$$

By another union bound, the total failure probability is at most $\sum_{i=1}^{d-1} f_i \triangleq f$. Once again, we believe that this bound is fairly tight when the f_i 's are small.

We can now compute (approximate) bounds on the various failure probabilities using essentially the same pseudo-code as in Section 4.11.3; we simply use (4.5) instead of (4.4).

4.12 Conclusions and Further Work

We have shown that designing small, efficient summaries to use in conjunction with multiple choice hashing schemes is feasible, improving on the results of [56]. We believe the fact that our summaries can be analyzed to bound performance is a useful characteristic that

⁴Technically, this formula is only valid if we use *partitioned* Bloom filters, as in the single filter summary. Unfortunately, designing a summary is usually easier if we use *unpartitioned* Bloom filters, as in our examples in Section 4.8. These two varieties of Bloom filters are asymptotically equivalent, however, and a partitioned Bloom filter usually has a higher false positive probability than its unpartitioned counterpart [9]. Therefore, we expect this formula to give a nearly tight upper bound.

will ease adoption. In terms of the general goals and techniques outlined in Chapter 1, this work gives a number of theoretical insights into both hash table and summary design with the potential to improve the performance of a real application under resource constraints (specifically, the hardware implementation of a multiple choice hash table with constrained pin count). Furthermore, our techniques allow for precise numerical results that would not be possible using only general theoretical techniques or brute-force simulations.

There are several potential extensions to this work. Our ideas and theoretical results can be extended easily to the case where buckets can hold any constant number of items, but more analysis and experiments must be done. Also, more experimentation could be done to test our summary structures, including large-scale tests with hash functions commonly used in practice, as well as tests for specific applications. A detailed analysis of deletion workloads to determine the effect of and best approach for deletions would also be worthwhile.

Chapter 5

The Power of One Move: Hashing Schemes for Hardware

5.1 Introduction

In this chapter, we once again consider implementing multiple choice hashing schemes in hardware. Unlike in Chapter 4, however, we assume that it is practical to access all of the d hash locations for an item in parallel, so that there is no need for the sort of summary data structure discussed there. Under this assumption, we seek to improve the space utilization of existing hashing schemes. In particular, we consider schemes that allow items already in the hash table to be moved during an insertion operation. Recently, many such schemes have been proposed in the theory literature, and the results are extremely compelling [18, 26, 48, 49]. Thus, it is natural to ask whether these proposed schemes are applicable to a hardware setting.

In general, the immediate answer is no. For example, we consider *cuckoo hashing* [48], which is the simplest multiple choice scheme that allows moves. Using cuckoo hashing, for an initially empty hash table designed to hold n items in $O(n)$ space, there is a non-negligible probability that during the insertion of n items into the table, at least one of those insertions requires $\Omega(\log n)$ moves. This is true despite the fact that, on average, only a constant number of items are moved during an insertion [18, 26, 48]. Unfortunately, in a hardware setting, the worst case time of an insertion operation may be significantly more important than the average time, because the former may essentially determine the complexity of the implementation. In particular, moving items in the hash table, which is generally held off-chip in slow memory, is expensive and must be limited. Only a small constant number of moves — indeed, arguably only one — is acceptable in practice. (Alternatively, we could directly modify the standard cuckoo hashing algorithm, using a queue for move operations to limit the worst case number of moves performed on each item insertion. We consider this approach in [30], and suspect that the appropriate choice of technique depends on the application.)

More concretely, the most critical aspect of the high-performance router setting is the common requirement that algorithms function at extremely fast wire speeds. This speed constraint naturally translates into a limit on the sorts of computation that can be

performed. In particular, memory accesses are extremely expensive, and must therefore be limited. For instance, if a router must handle 40 byte packets (the minimum packet size for TCP, corresponding to an acknowledgment with no payload) at a 40 Gbps wire speed (the OC-768 standard data rate for transmission on optical fiber), then the router has 8 ns to handle a packet. If the fastest memory available to the network processor is on-chip SRAM with an access time of, say, 1 ns, then clearly memory accesses can be made only sparingly. (This example is drawn from the text [59], published in 2004, and so the numbers may be slightly out of date; however, the qualitative picture remains quite accurate.)

A further issue with cuckoo hashing techniques is that the analyses of variants that perform well are currently incomplete. While one can construct proofs that they yield hash tables that hold n items with $O(n)$ space (with high probability), the analyses generally lose significant constant factors [18, 26, 48, 49]. We emphasize that this is not just a theoretical concern, as it makes the design and optimization of such structures essentially dependent upon potentially expensive simulations. The availability of accurate theoretical results, especially in the design and optimization phase, can significantly aid in the practical development of an actual implementation.

The primary purpose of this chapter is to bridge the gap between the existing theory and practice of multiple choice hashing schemes, particularly those that allow moves. We demonstrate simple but very effective multiple choice hashing schemes suitable for implementation in hardware. Specifically:

- We design and analyze schemes that require moving at most one item during each insertion operation. The limitation of just one move suggests that the schemes will be effective in practice. Moreover, we demonstrate that the gains in space utilization are substantial.
- We explicitly consider the availability of a small content addressable memory (CAM), and show how to optimize our schemes accordingly. While CAMs are common in hardware design to cope with rare but problematic cases, they are generally not considered in theoretical analyses.
- We use fluid limit arguments to develop numerical analysis and optimization techniques that give extremely accurate results. Our analysis allows us to consider questions such as the appropriate size for a CAM and the potential benefit of using skewed sub-tables of varying sizes in our construction. Furthermore, our approach provides a solid framework for exploring future hashing schemes.

5.2 The Standard Multilevel Hash Table (MHT), Revisited

As in Chapter 4, the standard multilevel hash table (MHT) [8] is central to our discussion. However, since our perspective on this data structure is slightly different in this chapter, we give an alternate presentation that is more suitable to the sorts of modifications that we consider here.

A standard MHT for representing a set of n items consists of d sub-tables, T_1, \dots, T_d , where T_i has $\alpha_i n$ buckets. Unless otherwise specified, we assume that a bucket can hold

at most one item. For simplicity, we assume that T_1, \dots, T_d place items using independent fully random hash functions h_1, \dots, h_d . To place an item x in the MHT, we simply find the smallest i for which $T_i[h_i(x)]$ is empty, and place x at $T_i[h_i(x)]$. If there is no such i , then we place x onto an *overflow list* L . In general, we visualize T_1, \dots, T_d, L as being laid out from left to right, and we often use this orientation in our discussion. (We note that in this setting, where each bucket holds at most one item, the standard MHT insertion scheme corresponds exactly to the more well-known d -left hashing scheme [10, 44, 60].) Finally, here and in most of this chapter, we concern ourselves only with inserting items into an MHT, although all of what we do can be adapted to deal with deletions. We discuss this further in Section 5.9.

The standard MHT is very amenable to a hardware implementation. We can easily perform a lookup by doing one hash and read for each sub-table in parallel, or sequentially if need be. As in Chapter 4, it may even be practical to use only a single read operation, with the aid of an auxiliary summary data structure that tells us which table to read from. Insertions are similarly easy to perform, as we can store a bit table in faster memory that tells us which buckets are occupied. Thus, the only operation on the actual table required during an insertion is the writing of the item to the proper place.

The prior work on MHTs (notably [8] and Chapter 4) considers implementations where one must be able to insert n items into some MHT with $O(n)$ buckets and be assured that, with very high probability (typically inversely polynomial in n), none will overflow into L ; in effect, there is no overflow list. Indeed, [8] shows that this can be done with an expected constant number of rehashings of the items for $d = \log \log n + O(1)$, and Chapter 4 modifies that analysis for the case where no rehashings are allowed, which is a much more compelling scenario for hardware applications.

We consider a different approach, designed specifically for addressing the issues that arise in hardware applications. (A similar approach, dubbed Filter Hashing, is suggested in [26].) If L is reasonably small (say, at most 64) with overwhelming probability, then we can implement it using a CAM of modest size. In this setting, it becomes useful to consider the following asymptotic regime for theoretical analysis. Rather than setting $d = \log \log n + O(1)$, we fix d to be some small constant. For example, in our analyses, we tend to focus on $d = 4$ because it gives an excellent tradeoff between performance and practicability for the reasonable value $n = 10000$. Now we think of $\alpha_1, \dots, \alpha_d$ as being fixed constants, so that the total size of the MHT is cn buckets, for $c = \sum_{i=1}^d \alpha_i$. We can then imagine inserting n items into the MHT and measuring the fraction that overflow into L . It turns out that, as $n \rightarrow \infty$, this fraction is very sharply concentrated around a constant w that can be calculated from a system of differential equations. In practice, this means that for n items, if w is on the order of, say, $10/n$, then we can implement the MHT using a modest sized CAM to represent L . Furthermore, and much more importantly, this whole approach is so general that we can extend it to deal with many variations on the standard procedure for inserting items. In particular, we use this technique to show that it is possible and practical to make substantial performance improvements to the standard MHT insertion procedure simply by allowing a single item to be moved. This is the main contribution of this chapter.

5.3 Hashing Schemes and Differential Equations

It is well known that, in many situations, the behavior of a randomized hashing scheme can be effectively approximated by a deterministic system [40]. Such an approximation eliminates much of the complexity that arises in many probabilistic analyses, and also makes the results significantly more transparent. We use this approach throughout this chapter.

To illustrate the technique, we consider the standard MHT as described in Section 5.2. Suppose that we start at time 0 with an empty MHT, and then insert n items into it, inserting the j th item at time j/n , so that all items are inserted by time 1. Let $F_i(t)$ denote the fraction of buckets in T_i that are occupied at time t . Now condition on the state of the MHT just after the j th item is inserted, for some $j < n$. Then, for each i , the probability that the $(j+1)$ st item inserted into the MHT ends up in T_i is $(1 - F_i(j/n)) \prod_{k=1}^{i-1} F_k(j/n)$. Formally, letting $\vec{F}(t)$ denote the vector of the $F_i(t)$'s, we have

$$\mathbf{E}[F_i((j+1)/n) - F_i(j/n) \mid \vec{F}(j/n)] = \frac{(1 - F_i(j/n))}{\alpha_i} \prod_{k=1}^{i-1} F_k(j/n).$$

The conditional expectation corresponds to the average change in $F(t)$ over the window of time $[j/n, (j+1)/n]$. Thus, if n is very large, so that changes are comparatively small, then we would expect the following approximation to be valid:

$$\frac{dF_i(t)}{dt} \approx \frac{(1 - F_i(t))}{\alpha_i} \prod_{k=1}^{i-1} F_k(t). \quad (5.1)$$

Indeed, this is essentially the case. More formally, let $\vec{f}(t) = (f_1(t), \dots, f_d(t))$ be the solution to the system of differential equations

$$\frac{df_i}{dt} = \frac{(1 - f_i)}{\alpha_i} \prod_{k=1}^{i-1} f_k$$

with $f_i(0) = F_i(0) = 0$ for each i . Then as $n \rightarrow \infty$ and $\epsilon \rightarrow 0$ we have,

$$\mathbf{Pr} \left(\sup_{t \in [0,1]} \left| \vec{F}(t) - \vec{f}(t) \right| > \epsilon \right) \leq e^{-\Omega(n\epsilon^2)}.$$

Thus, the approximation (5.1) is valid; the f_i 's are extremely accurate approximations of the F_i 's. This is a consequence of an extremely general mathematical result due to Kurtz [33, 34, 55], which justifies not only the discussion here, but also all of the other differential equation approximations that we make in this chapter. This approach is typically referred to as taking the *fluid limit* of the system, or as the *mean-field method*.

Of course, we are glossing over some technical details in this discussion. For example, this form of Kurtz's Theorem requires that the items arrive according to a Poisson process, but this can be dealt with by an application of an appropriate Chernoff bound for Poisson random variables. Also, one must formally check that the conditions of Kurtz's

Theorem are satisfied for this system. This is straightforward, as the hypotheses of the theorem are very general.

Returning to the analysis of the standard MHT, the differential equation approximation immediately predicts the fraction of the n items that overflow into L ; it is just $w \triangleq 1 - \sum_{i=1}^d \alpha_i f_i(1)$. While w is not likely to have a simple form, we can easily compute it numerically using standard mathematical software. Furthermore, thinking of w as a function of $\alpha_1, \dots, \alpha_d$, we can attempt to minimize w subject to the space constraint that $\sum_i \alpha_i \leq c$ for some fixed constant c . In other words, we can look for the best sizes for the sub-tables of the MHT, subject to the constraint that the entire MHT has at most c buckets per item that we wish to insert. Since we can easily evaluate w , we can find good α_i 's by using standard mathematical software for attempting to minimize a black-box function. Such an optimization approach is only possible because the differential equations are so easy to evaluate. If we were using a less efficient method, such as simulation, the optimization procedure would likely be too slow to be of any real use. We use this technique to configure and compare all of our schemes in Section 5.8.

5.4 Allowing One Move: A Conservative Scheme

We have already argued that the standard MHT is very amenable to a hardware implementation. Our results in Section 5.8 also show that it is quite effective. (This in itself is not really new, given the theoretical analysis of [8] and the subsequent theoretical and numerical results of Chapter 4.) Thus, we are now in a position to push the envelope, introducing progressively more complexity into the MHT insertion procedure to reduce the space required while still ensuring that hardware implementations are practical. In particular, as described in the introduction, we are inspired by the existing literature on hashing schemes that allow moves. That inspiration leads us to design alternative MHT insertion procedures that allow a single move per insertion operation, in order to effectively balance performance improvements to the MHT against increased complexity in a hardware implementation. We consider a variety of procedures, in order to gain an understanding of the landscape and how the theory applies.

We start with a fairly conservative scheme. Each bucket of the MHT can be either *marked* or *unmarked*. Initially, all buckets are unmarked. When inserting an item x into the MHT, we find the smallest i such that $T_i[h_i(x)]$ is empty, if there is one. In this case, we simply set $T_i[h_i(x)] = x$. If there is no such i , we find the smallest $j < d$ such that $T_j[h_j(x)]$ is unmarked, if there is one. If there is no such j , we place x on L . If there is such a j , we mark $T_j[h_j(x)]$, set $y = T_j[h_j(x)]$, and look for the smallest $k > j$ such that $T_k[h_k(y)]$ is empty, if there is one. If there is no such k , then we place x on L . If there is such a k , then we set $T_k[h_k(y)] = y$ and $T_j[h_j(x)] = x$.

In words, this scheme tries to place x into a sub-table using the standard MHT scheme. If this fails, it tries to *bump* an item y that it collides with, replacing it and moving y to the right. The item x tries to bump at most one other item. Because of this, we take care to mark an item that we already know cannot be moved to the right successfully to avoid wasting effort. Hence the item that x tries to bump is the leftmost one that we do not know for a fact cannot be bumped.

We say that this scheme is conservative because it is hesitant to move an item. Indeed, the scheme only attempts to move items as the standard MHT insertion procedure breaks down; it makes no attempt to arrange items in advance to prevent this from happening. As such, in practice it is fairly rare for this scheme to actually move an item. We show this in detail in Section 5.8.

As in Section 5.3, suppose that we start at time 0 with an empty MHT, and insert n items at times $1/n, 2/n, \dots, 1$. For convenience, we define the notation $[r] = \{1, \dots, r\}$ for any integer r . For $i \in [d]$, let $F_i(t)$ denote the fraction of buckets in T_i that are occupied at time t , and for $i \in [d-1]$, let $G_i(t)$ denote the fraction of buckets in T_i that are marked. Then a straightforward, if somewhat tedious, probability calculation tells us that the F_i 's and G_i 's can be approximated by the solution of the following system of differential equations (with $f_i(0) = 0$ and $g_i(0) = 0$).

$$\frac{df_i}{dt} = \frac{1 - f_i}{\alpha_i} \left[\prod_{j=1}^{i-1} f_j + \sum_{j=1}^{i-1} \left(\prod_{k=1}^d \begin{cases} g_k & k < j \\ f_k - g_k & k = j \\ f_k & k > j \end{cases} \right) \prod_{k=j+1}^{i-1} f_k \right]$$

$$\frac{dg_i}{dt} = \frac{1}{\alpha_i} \prod_{j=1}^d \begin{cases} g_j & j < i \\ f_j - g_j & j = i \\ f_j & j > i \end{cases}$$

Here we have used an array notation to simplify the presentation of the equations, as the terms of the product are case-dependent on the index. As before, $w \triangleq 1 - \sum_{i=1}^d \alpha_i f_i(1)$ is the asymptotic fraction of items that overflow into L .

We can also use the differential equation approach to determine the asymptotic fraction of insertion operations that require an item to be moved in the MHT. Indeed, if $M(t)$ is the fraction of the n items that are inserted at or before time t and required a move in the MHT, then another calculation tells us that $M(t)$ is approximated by the solution $m(t)$ of the differential equation (with $m(0) = 0$)

$$\frac{dm}{dt} = \sum_{i=1}^{d-1} \left(1 - \prod_{j=i+1}^d f_j \right) \prod_{j=1}^d \begin{cases} g_j & j < i \\ f_j - g_j & j = i \\ f_j & j > i \end{cases}$$

The asymptotic fraction of the n insertion operations that require a move in the MHT is then $m(1)$.

5.5 The Second Chance Scheme

Recall that the scheme in Section 5.4 rarely moves an item in the MHT, even though we have, in principle, allowed ourselves to perform at most one move per insertion operation. Indeed, in many hardware applications, the frequency with which we perform moves may be much less important than the guarantee that we never perform more than one move per insertion. With this in mind, we introduce a scheme that is considerably

```

1: for  $i = 1$  to  $d - 1$  do
2:   if  $T_i[h_i(x)]$  is not full then
3:      $T_i[h_i(x)] \leftarrow x$ 
4:     return
5:   end if
6:    $y \leftarrow T_i[h_i(x)]$ 
7:   if  $T_{i+1}[h_{i+1}(x)]$  is full then
8:     if  $T_{i+1}[h_{i+1}(y)]$  is not full then
9:        $T_{i+1}[h_{i+1}(y)] \leftarrow y$ 
10:       $T_i[h_i(x)] \leftarrow x$ 
11:     return
12:   end if
13: end if
14: end for
15: if  $T_d[h_d(x)]$  is not full then
16:    $T_d[h_d(x)] \leftarrow x$ 
17: else
18:   Add  $x$  to  $L$ 
19: end if

```

Figure 5.1: Pseudocode for inserting an item x in the second chance scheme.

more aggressive in performing moves, while still guaranteeing that no insertion operation requires more than one.

For intuition, consider inserting n items using the standard MHT insertion scheme. It is fairly clear, both from the definition of the scheme and from the differential equations in Section 5.2, that as the items are inserted the sub-tables fill up from left to right, with newly inserted items cascading from T_i to T_{i+1} with increasing frequency as T_i fills up. Thus, it seems that a good way to reduce the overflow from the MHT is to slow down this cascade at every step.

This idea is the basis for our new scheme, which we call the *second chance* scheme. The formal pseudocode is given in Figure 5.1. The basic idea is that whenever an inserted item x cannot be placed in T_i , it checks whether it can be inserted into T_{i+1} . If it cannot be placed there then, rather than simply moving on to T_{i+2} as in the standard scheme, the item x checks whether the item y in $T_i[h_i(x)]$ can be moved to $T_{i+1}[h_{i+1}(y)]$. If this move is possible, then we move y and replace it with x . Thus, we effectively get a *second chance* at preventing a cascade from T_{i+1} to T_{i+2} .

From a hardware implementation perspective, this scheme is much more practical than it may first seem. To see this, consider a standard MHT implementation where we can read and hash one item from each sub-table in parallel. In this setting, we can insert an item x using the second chance scheme by reading and hashing all of items in $T_1[h_1(x)], \dots, T_{d-1}[h_{d-1}(x)]$ in parallel. Once we have the hash values for these items, we can determine exactly how x should be placed in the table using the pseudocode in Figure 5.1.

Alternatively, if we have a hardware implementation that forces us to read and hash these items sequentially, then we can consider storing hash values for the items in a separate table with sub-tables T'_1, \dots, T'_{d-1} , mimicking the MHT, that can be accessed more quickly. Note that, for an item y in T_i , we only need to store the value of $h_{i+1}(y)$ in $T'_i[h_i(y)]$. Thus, when we attempt to insert an item x , we can determine exactly where to place x without reading any cells in the hash table (assuming that we keep a bit table in memory that tracks the cells of the hash table that are occupied). If hash values are much smaller than the items themselves (as is often the case), the space needed for the T' 's is offset by the reduction in size of the MHT.

As before, we can use differential equations to approximate the fraction of items that overflow into L . In the usual way, suppose that we start at time 0 with an empty MHT, and insert n items into it, at times $1/n, 2/n, \dots, 1$. For $i \in [d]$, let $F_i(t)$ denote the fraction of buckets in T_i that are occupied at time t . Perhaps more subtly, for $i \in [d-1]$, we let $G_i(t)$ denote the fraction of buckets in T_i at time t that contain an item y such that $T_{i+1}[h_{i+1}(y)]$ has already been found full in line 8 of Figure 5.1. For such an item y , it is guaranteed that $T_{i+1}[h_{i+1}(y)]$ is occupied at time t and thereafter; since this behavior is different from items where $h_{i+1}(y)$ has never been checked, we must track it separately. Another probability calculation now tells us that the F_i 's and G_i 's can be approximated by the solution of the following system of differential equations (with $f_i(0) = 0$ and $g_i(0) = 0$).

$$\begin{aligned} \frac{df_1}{dt} &= \frac{1 - f_1}{\alpha_1} \\ \frac{df_i}{dt} &= \frac{1 - f_i}{\alpha_i} (f_{i-1} + (f_{i-1} - g_{i-1})f_i) \prod_{j=1}^{i-2} g_j + (f_j - g_j)f_{j+1} \quad \text{for } i = 2, \dots, d \\ \frac{dg_i}{dt} &= \frac{(f_i - g_i)f_{i+1}}{\alpha_i} \prod_{j=1}^{i-1} g_j + (f_j - g_j)f_{j+1} \end{aligned}$$

The asymptotic fraction of items that overflow into L is then just $w \triangleq 1 - \sum_{i=1}^d \alpha_i f_i(1)$.

As before, we can also define $M(t)$ to be the fraction of the n items that are inserted at or before time t and require a move in the MHT. In this case, $M(t)$ is approximated by the solution $m(t)$ of the differential equation (with $m(0) = 0$)

$$\frac{dm}{dt} = \sum_{i=2}^d (f_{i-1} - g_{i-1})f_i(1 - f_i) \prod_{j=1}^{i-2} g_j + (f_j - g_j)f_{j+1}.$$

5.6 The Extreme Second Chance Scheme

In this section, we describe a further enhancement of the second chance scheme, that we call the *extreme second chance scheme*. The idea is fairly intuitive. In the regular second chance scheme, we only consider inserting an item x into T_i if it collides with items $y_1 = T_1[h_1(x)], \dots, y_{i-1} = T_{i-1}[h_{i-1}(x)]$ and for each $j < i - 1$, the bucket $T_{j+1}[h_{j+1}(y_j)]$ is occupied, so that we cannot move y_j to T_{j+1} and place x in T_j . Taking this idea to the extreme, we simply allow for many more possible moves. Thus, in the new scheme, we

only consider inserting an item x into T_i if it collides with items $y_1 = T_1[h_1(x)], \dots, y_{i-1} = T_{i-1}[h_{i-1}(x)]$ and for any $j < k \leq i-1$, the bucket $T_k[h_k(y_j)]$ is occupied, so that we cannot move y_j to T_k and place x in T_j .

More formally, we have the following specification. Suppose we let $y_i = T_i[h_i(x)]$ if $T_i[h_i(x)]$ is not empty. Let z_0 be the index of the leftmost sub-table that is currently empty for x , with the notation that $z_0 = d + 1$ if none of the d choices are available. For each y_i , let z_i be the index of the currently leftmost empty sub-table for y_i . If $z_0 \leq \min_i z_i$, then we place x in the leftmost sub-table with an empty slot for x . Otherwise, let y_j be the item with the smallest value of z_j , breaking ties in favor of the smallest index j . We move y_j to the leftmost sub-table with an empty slot for y_j and put x in its place. Of course, if we cannot insert x using this procedure, then we place x on L .

There are a few negative aspects of this scheme. First, it requires significantly more hashes and other computations than the original second chance scheme. This may or may not be important, depending on the context. Arguably, the gain in space could conceivably be worth the additional complexity. Second, from our perspective, another clear negative is that while our differential equation analysis can be applied to this scheme, the resulting explication of cases leads to a set of equations that is more complicated than the other schemes that we consider in this chapter.

Nevertheless, we can still apply the differential equation technique for the extreme second chance scheme. As before, consider an initially empty MHT, and suppose that n items are inserted into it at times $1/n, 2/n, \dots, 1$. For $i \in [d]$, let $F_i(t)$ denote the fraction of buckets in T_i that are occupied at time t . For $i \leq j \in [d]$, let $G_{i,j}(t)$ denote the fraction of buckets in T_i occupied by an item y for which $T_1[h_1(y)], \dots, T_j[h_j(y)]$, but not $T_{j+1}[h_{j+1}(y)]$, have been found full at time t . (Here we think of new hash values as being sampled as needed, so that we may use the principle of deferred decisions in our analysis (e.g., [42]).) A straightforward (but tedious) probability calculation now tells us that the F_i 's and the $G_{i,j}$'s can be approximated by the solution of the following system of differential equations (with $f_i(0) = 0$ and $g_{i,j}(0) = 0$). Here we use the convention that a sum of zero terms is 0 and a product of zero terms is 1.

$$\begin{aligned}
g_{i,\geq j} &= \sum_{k=j}^d g_{i,k} && \text{for } i < j \in [d] \\
f_i &= g_{i,i} + g_{i,\geq i+1} && \text{for } 1 \leq i < d \\
f_d &= g_{d,d} \\
z_{i,j,k} &= \prod_{\ell=i}^j \sum_{r=\ell}^d g_{\ell,r} \prod_{s=r+1}^k f_s \\
\frac{df_i}{dt} &= \frac{1}{\alpha_i} [z_{1,i-1,i-1} - z_{1,i,i}] \\
\frac{dg_{i,\geq j}}{dt} &= \frac{1}{\alpha_i} z_{1,i-1,j} z_{i+1,j-1,j-1} f_j \sum_{r=i}^{j-1} g_{i,r} \prod_{s=r+1}^{j-1} f_s
\end{aligned}$$

The asymptotic fraction of items that overflow into L is now just $w \triangleq 1 - \sum_{i=1}^d \alpha_i f_i(1)$.

As before, we can also define $M(t)$ to be the fraction of the n items that are inserted at or before time t and require a move in the MHT. In this case, $M(t)$ is approximated by the solution $m(t)$ of the differential equation (with $m(0) = 0$)

$$\frac{dm}{dt} = \sum_{i < j \in [d]} z_{1,i-1,j} z_{i+1,j-1,j-1} f_j (1 - f_j) \sum_{k=i}^{j-1} g_{i,k} \prod_{r=k+1}^{j-1} f_r.$$

5.7 Multiple Items Per Bucket

We can also consider schemes that allow multiple items to be stored in a bucket of the MHT. We do this primarily to illustrate the generality of our methodology, and we emphasize that everything that we do in this section is by way of example; the overall approach can be adapted to a large variety of different settings and schemes.

We focus on the case where a bucket in the MHT can store at most two items, starting with the natural generalization of the standard MHT insertion scheme. Here, an item x is placed in the leftmost sub-table for which $T[h_i(x)]$ is not full, or on L otherwise. Our previous differential equation analysis can now be easily modified. As before, we suppose that we start at time 0 with an empty MHT, and insert n items into it, at times $1/n, 2/n, \dots, 1$. For $i \in [d]$ and $j \in \{0, 1, 2\}$, let $F_{i,j}(t)$ denote the fraction of buckets in T_i that have exactly j items at time t . Another straightforward probability calculation now tells us that the $F_{i,j}$'s can be approximated using the solution of the following system of differential equations (with $f_{i,j}(0) = \mathbf{1}(j=0)$, where $\mathbf{1}(\cdot)$ denotes the indicator function).

$$\begin{aligned} f_{i,\geq j} &= \sum_{k=j}^2 f_{i,k} && \text{for } j = 1, 2 \\ f_{i,0} &= 1 - f_{i,\geq 1} \\ \frac{df_{i,\geq j}}{dt} &= \frac{f_{i,j-1}}{\alpha_i} \prod_{j=1}^{i-1} f_{j,2} && \text{for } j = 1, 2 \end{aligned}$$

In this setting, the asymptotic fraction of items that overflow into L is

$$w \triangleq 1 - \sum_{i=1}^d \sum_{j=1}^2 j \alpha_i f_{i,j}(1).$$

(Buckets containing two items must be multiplied by a factor of two.)

To show how allowing even one move during the insertion procedure can improve performance, we consider a natural modification of the second chance scheme. (The ideas can be extended to larger numbers of items per bucket.) We simply modify the procedure in Figure 5.1 so that when we try to insert an item x into T_i and the buckets $T_i[h_i(x)]$ and $T_{i+1}[h_{i+1}(x)]$ are full, we check both $y \in T_i[h_i(x)]$ to see whether either can be moved to $T_{i+1}[h_{i+1}(y)]$, and in this case, we perform the move and place x in $T_i[h_i(x)]$. Of course, we need to consider the case where both $y \in T_i[h_i(x)]$ can be moved, since then we have to choose between them. For simplicity, we suppose that we move the first y that we check

that can be moved, so that we do not even have to check the other one. Of course, one could also consider the natural variation where we try to minimize the number of items in $T_{i+1}[h_{i+1}(y)]$. As before, this is only an example, and so we focus on our slightly simpler scheme.

We can use differential equations to approximate the fraction of items that overflow into L . As before, suppose that we start at time 0 with an empty MHT, and insert n items into it, at times $1/n, 2/n, \dots, 1$. For $i \in [d]$ and $j \in \{0, 1, 2\}$, let $F_{i,j}(t)$ denote the fraction of buckets in T_i that have exactly j items at time t . For $i \in [d-1]$ and $j \in \{0, 1, 2\}$, let $G_{i,j}(t)$ denote the fraction of buckets in T_i at time t that contain two items, j of which are items y such that $T_{i+1}[h_{i+1}(y)]$ has already been found full in the equivalent of line 8 in Figure 5.1; for such an item y , it is guaranteed that $T_{i+1}[h_{i+1}(y)]$ is full at time t and thereafter. Another straightforward probability calculation now tells us that the $F_{i,j}$'s and $G_{i,j}$'s can be approximated using the solution of the following system of differential equations (with $f_{i,j}(0) = \mathbf{1}(j=0)$ and $g_{i,j}(0) = 0$).

$$\begin{aligned}
f_{i,\geq j} &= \sum_{k=j}^2 f_{i,k} && \text{for } j = 1, 2 \\
g_{i,\geq j} &= \sum_{k=j}^2 g_{i,k} && \text{for } j = 1, 2 \\
f_{i,0} &= 1 - f_{i,\geq 1} \\
g_{i,0} &= f_{i,2} - g_{i,\geq 1} \\
\frac{df_{1,\geq j}}{dt} &= \frac{f_{1,j-1}}{\alpha_1} \\
\frac{df_{i,\geq j}}{dt} &= \frac{1}{\alpha_i} \left[\prod_{r=1}^{i-2} \sum_{s=0}^2 g_{r,s} f_{r+1,2}^{2-s} \right] \sum_{r=0}^2 g_{i-1,r} \sum_{s=0}^{2-r} f_{i,2}^s f_{i,j-1} && \text{for } 2 \leq i \leq d \\
\frac{dg_{i,\geq j}}{dt} &= \frac{1}{\alpha_i} \left[\prod_{r=1}^{i-1} \sum_{s=0}^2 g_{r,s} f_{r+1,2}^{2-s} \right] \left[\sum_{r=0}^{j-1} g_{i,r} f_{i+1,2}^{j-r} \right]
\end{aligned}$$

As before, the asymptotic fraction of items that overflow into L is

$$w \triangleq 1 - \sum_{i=1}^d \sum_{j=1}^2 j \alpha_i f_{i,j}(1).$$

If $M(t)$ is again the fraction of the n items that are inserted at or before time t and require a move in the MHT, then $M(t)$ is approximated by the solution $m(t)$ of the differential equation (with $m(0) = 0$)

$$\frac{dm}{dt} = \sum_{i=2}^d \left[\prod_{r=1}^{i-2} \sum_{s=0}^2 g_{r,s} f_{r+1,2}^{2-s} \right] f_{i,2} \sum_{r=0}^1 g_{i-1,r} \left(1 - f_{i,2}^{2-r} \right).$$

5.8 Evaluation (Insertions Only)

In this section, we use the differential equations previously derived to assess and compare the performance of the six schemes we consider: the standard MHT scheme (Std), the conservative scheme (Cons) of Section 5.4, the second chance scheme (SC), the extreme second chance scheme (SCExt), and the modifications of the standard and second chance schemes for the case of two items per bucket discussed in Section 5.7 (Std2 and SC2, respectively). As mentioned previously, we solve all differential equations numerically using standard mathematical software. Similarly, we use standard numerical optimization procedures to compare schemes in the following way: for each scheme and a particular d and bound on the number of buckets per inserted item in the MHT, we choose the α_i 's in an attempt to minimize w subject to the space constraint. Specifically, we use the NDSolve and NMinimize functions in Mathematica 6.0, with occasional minor modifications to the default behavior of NMinimize (such as trying multiple values for the Method and number of iteration parameters when appropriate). We also verify all of our numerical results through simulation.

For some perspective, the total computation time for all of the numerical results presented in this section was about a week or two on a standard workstation PC obtained in 2004, with another few days spent gathering simulation results. The coding time was also comparably small. We point out these facts to give evidence that this methodology is very practical and, as we shall see shortly, also quite effective.

In what follows, we use the notation that the space of the hash table excluding the CAM is equal to the size of cn items, where c is a constant independent of n . For every scheme presented except the Std2 and SC2 schemes, this is the same as saying that there are cn buckets; because the Std2 and SC2 scheme hold up to two items per bucket, the total number of buckets for the those schemes is $cn/2$.

We start by comparing the overflow rates w of the different schemes for different values of c when $d = 4$. We focus on the choice of $d = 4$ because it achieves an excellent tradeoff between performance and practicality for a hardware implementation for a reasonable range of values of n . We also focus on values of c that provide the big picture of how the schemes compare. As we have suggested, further experimentation for specific cases (different values of c and/or d) would be simple to compute.

The results are displayed in Figure 5.2. As is clear from the figure, the overflow rate drops off exponentially for all schemes as c increases. Also, each increase in complexity to the MHT insertion scheme gives a significant reduction in the resulting overflow rate (under the arguable assumption that SCExt is more complex than Std2, but less complex than SC2). This difference is profound for the SC2 scheme; the additional flexibility from having two items to move at each level offers substantial benefits, at the cost of some hardware complexity and the requirement that two items can be stored in a single bucket. More interestingly, though, the differences between the Std, Cons, SC, and SCExt schemes are enormous as c grows towards 2, and the difference is still significant even for c much closer to 1 (say, $c = 1.2$).

To get a more refined comparison of the schemes, we fix a *target* value of $w = 0.2\%$, and attempt to find the smallest c for each scheme that achieves w using NMinimize, taking c to the nearest hundredth. Roughly speaking, this corresponds to drawing the horizontal

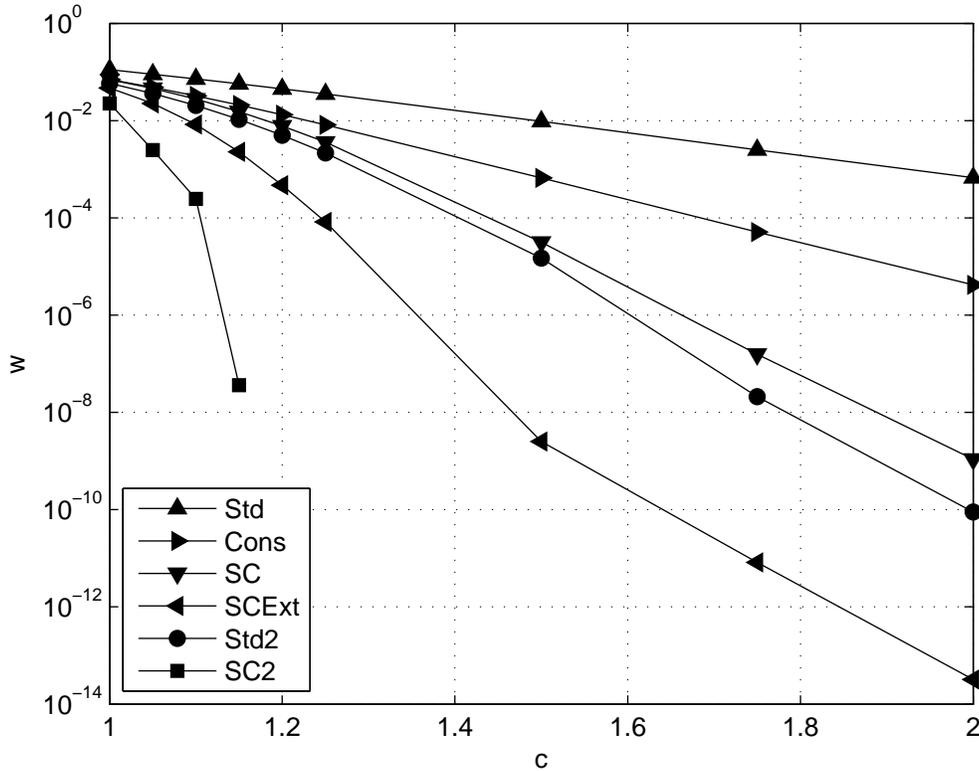


Figure 5.2: Comparison between the asymptotic fractional overflows w and the corresponding space multiplier c for $d = 4$.

line $w = 0.2\%$ in Figure 5.2 and comparing the c coordinates where that line intersects each of the lines corresponding to the different schemes. The value for w is chosen so that for $n = 10000$, which we consider a reasonable size, the expected number of items that overflow is about 20. As we shall see later, the distribution of the number of items that overflow in this case is approximately Poisson(20). The probability that a Poisson(20) random variable exceeds, say, 64, is approximately (by numerical calculation) 3.77×10^{-15} , which is negligible. Thus for these values, we can, in practice, just use a CAM of size 64 to represent the overflow list L .

We present the results of our comparison of the schemes for the target $w = 0.2\%$ for $d \in \{3, 4, 5\}$ in Table 5.1. In that table, we also show the asymptotic fraction m of the n insertion operations that require a move, as well as the values for the α_i 's determined by our optimizations. As mentioned previously, we are principally interested in the results for $d = 4$. The results for the other values for d are of secondary importance and are presented mostly for the sake of comparison with the results for $d = 4$.

Table 5.1 essentially confirms our original intuition concerning the schemes and the overall picture suggested by Figure 5.2. In all cases, Cons gives a significant reduction in space over Std, at the cost of performing moves during a small fraction of insertion

Table 5.1: Scheme comparison for a target $w = 0.2\%$

Scheme	c	m	α_1	α_2	α_3
Std	2.68	0%	1.3880	0.8309	0.4586
Cons	1.78	1.82%	0.7767	0.6062	0.3746
SC	1.62	8.51%	0.7142	0.6400	0.2707
SCExt	1.51	9.90%	0.6554	0.5500	0.3064
Std2	1.53	0%	0.4311	0.2284	0.1042
SC2	1.22	10.0%	0.2668	0.2665	0.0744

(a) $d = 3$

Scheme	c	m	α_1	α_2	α_3	α_4
Std	1.80	0%	0.7867	0.5149	0.3152	0.1782
Cons	1.39	1.66%	0.5214	0.4134	0.2802	0.1774
SC	1.29	12.9%	0.4694	0.4562	0.2512	0.1082
SCExt	1.15	17.8%	0.3774	0.3624	0.2857	0.1276
Std2	1.25	0%	0.2970	0.1735	0.1059	0.0493
SC2	1.05	14.7%	0.2048	0.1961	0.0954	0.0299

(b) $d = 4$

Scheme	c	m	α_1	α_2	α_3	α_4	α_5
Std	1.46	0%	0.544	0.383	0.271	0.162	0.098
Cons	1.24	1.41%	0.410	0.329	0.236	0.162	0.105
SC	1.16	15.3%	0.366	0.367	0.239	0.129	0.058
SCExt	1.06	25.1%	0.223	0.223	0.221	0.220	0.172
Std2	1.14	0%	0.239	0.155	0.096	0.054	0.026
SC2	1.03	16.7%	0.171	0.171	0.107	0.048	0.016

(c) $d = 5$

operations. SC provides a significant further improvement over the space requirement of Cons, at the cost of a (likely reasonable) order of magnitude increase in m . (As indicated in Figure 5.2, this improvement would likely be more dramatic if we considered smaller w , say $w = 2 \times 10^{-4}$, which would correspond to our current setup with $n = 10^5$ instead of the value $n = 10^4$.) SCExt performs more moves and reduces the space requirement even further. The space requirement of SCExt is even superior to that of Std2 for $d = 4$ and $d = 5$, but at the cost of significantly more complexity. Finally, the flexibility of allowing an item to be in either of two locations within a bucket in addition to the power of allowing a single move allows SC2 to outperform SCExt in terms of both space and frequency of insertions requiring a move. Of course, one must remember that this added flexibility comes at a price, as there are now two locations in each bucket to check during a lookup operation.

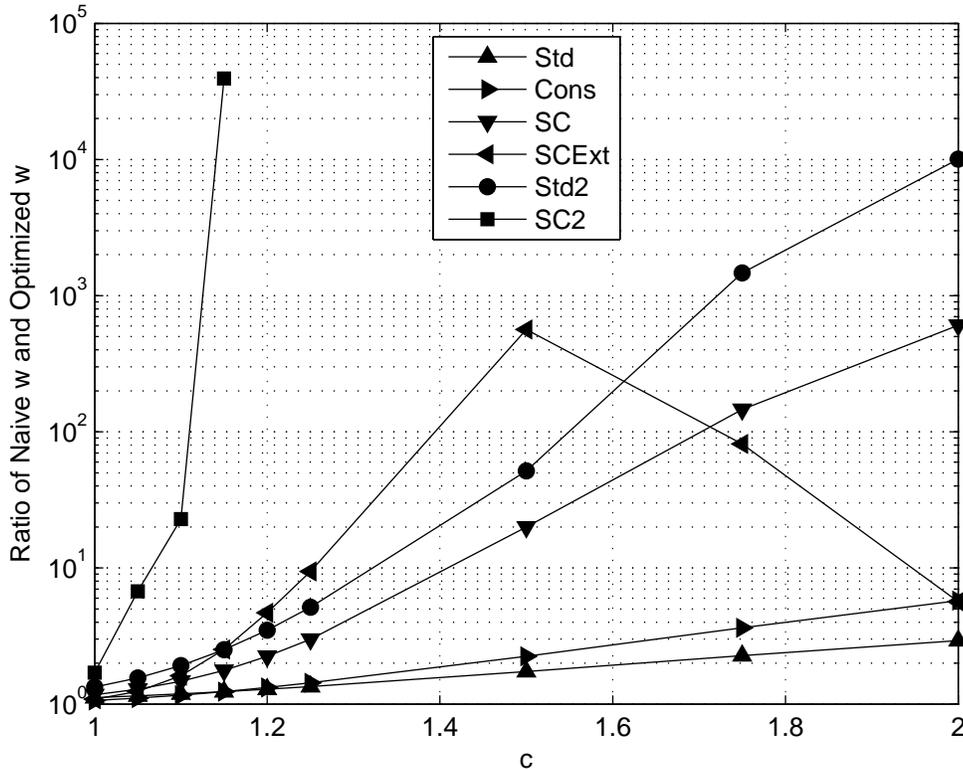


Figure 5.3: Effect on w of optimizing the α_i 's for $d = 4$ and various c values.

Furthermore, as we promised earlier, $d = 4$ gives an excellent tradeoff between the number of hash functions and the performance of the schemes. The space requirements are much worse for $d = 3$, and may not be sufficiently better for $d = 5$ to justify the use of another hash function and accompanying sub-table (assuming that we are using one of the above schemes other than Std or Std2, which do not allow for items to be moved in the MHT).

Of course, while our previous results rely on optimizing the α_i 's, it is likely impractical to use those exact values in a real application. Nevertheless, it is still worthwhile to try to optimize the α_i 's within the domain of practicality. To give a rough illustration of this, we first compute the values of w corresponding to the various schemes and values of c depicted in Figure 5.2 under the naive configuration where $\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = c/4$. We then compare these values of w to the optimized values shown in Figure 5.2, and plot them in Figure 5.3. For all of the schemes except SCExt, the trends are clear. All of these schemes benefit significantly from optimization. The situation for SCExt appears similar, despite the very odd shape of the plot, which we suspect is due to shortcomings in NMinimize. Finally, since it is not entirely clear from the plot, we note that all of the schemes benefit significantly from optimization even for values of c much smaller than 2. For example, for $c = 1.2$, the ratios for Std, Cons, SC, SCExt, and Std2 are 1.29, 1.33, 2.24, 4.69, and 3.49, respectively.

Table 5.2: Effect of optimizing the α_i 's for $d = 4$ and target $w = 0.2\%$

Scheme	Naive c	Optimized c	Ratio
Std	2.00	1.80	1.11
Cons	1.46	1.39	1.05
SC	1.41	1.29	1.09
SCExt	1.20	1.15	1.04
Std2	1.42	1.25	1.14
SC2	1.14	1.05	1.09

To get a more refined look at the effect of optimizing the α_i 's for the various schemes, we once again consider the target value $w = 0.2\%$. We compare the optimized c values from Table 5.1(b) with their corresponding naive values, which are obtained by searching for the smallest value of c that achieves the target $w = 0.2\%$ under the assumption that all of the α_i 's are equal. The results are illustrated in Table 5.2. We see that the savings in space from optimizing the α_i 's is on the order of 10%; it would be slightly higher for smaller values of w . We conclude that optimizing the α_i 's is likely to be worthwhile in practice, if the design naturally allows varying sub-table sizes.

Finally, as our results arise from the fluid limit approximation, it is important to verify them independently through simulation to ensure that they are accurate. (The optimization of the α_i 's are purely numerical; it cannot be efficiently performed through simulation.) We verify all of the results in Figure 5.2 and Tables 5.1 and 5.2 in the following way. For a given scheme and $\alpha_1, \dots, \alpha_d$ we simulate the insertion of $n = 10000$ items into an MHT with d sub-tables, where the size of T_i is $\lfloor \alpha_i n \rfloor$. (We round down for simplicity; the tables are large enough that losing a bucket makes an insubstantial difference.) We keep track of the fraction of items placed in the overflow list L and the fraction of insertion operations resulting in a move, and average the results over 10^5 trials. We sample all hash values using the standard Java pseudorandom number generator.

The simulation results indicate that the differential approximations are highly accurate. For the Std, Cons, SC, and SCExt schemes, the largest relative error for the simulated overflow rates measured against the numerically calculated rates is 0.9% for calculated rates more than 10^{-4} , only 1.5% for calculated rates more than 10^{-6} , and 268% for the remaining rates (which is fairly accurate, considering that the remaining calculated rates are so small compared to the number of trials in the experiment). Similarly, the largest relative error in the fraction of insertions requiring a move is 0.33%. For the Std2 and SC2 schemes, the situation is more complicated, most likely due to the fact that these are the schemes that allow two items to be placed in bucket. The differential equation approximation seems to be very accurate for the first few sub-tables of the MHT, but the relative error degrades significantly from there. For example, for these two schemes, the largest relative error for the simulated overflow rates measured against the numerically calculated rates is 4.8% for calculated rates more than 10^{-4} , significantly more than for the other schemes (but still not very big). Running the simulation again for $n = 10^5$ helps immensely. In this case, the relative errors in the overflow rates are less than 1% when the calculated overflow

rate is at least 10^{-6} . For the SC2 scheme with $n = 10^4$, the largest relative error in the fraction of insertions requiring a move is 0.15%, and for $n = 10^5$, it is 0.012%. (Recall that Std2 never requires any moves.)

Having examined the overflow rate w for the various schemes, we now look at some of the finer properties of the distribution of the items in the T_i 's. Suppose that the differential equations for a scheme tell us that the fraction of the n items inserted into Q is f , where Q is some sub-table of the MHT or L . If the events that each of n items ends up in Q were independent with probability f , then the number of items in Q would be $\text{Binomial}(n, f)$. If f were on the order of λ/n for some small constant λ , then the distribution $\text{Binomial}(n, f)$ would be approximately $\text{Poisson}(\lambda)$. For larger f , we would expect the distribution to be normal around its mean by the central limit theorem.

Obviously, each item does not get inserted into Q with the same probability f ; the probabilities change according to the differential equations. However, under the heuristic assumption that the events at each step are independent with the appropriate probabilities, it is straightforward to generalize the standard generating function proof for the Binomial convergence to the Poisson distribution (e.g., [28, Exercise 5.12.39a]). Hence, if f is on the order of λ/n for some small constant λ , then we expect the distribution of the number of items in Q to be nearly $\text{Poisson}(\lambda)$. Similarly, if f is much larger, then the distribution of the number of items in Q should be approximately normal. Thus, we expect that for a well-configured scheme, the distribution of the number of items in each sub-table of the MHT should be approximately normal, and the number of items in the overflow list L should be approximately Poisson. In fact, the conclusion that the size of L is approximately Poisson is absolutely critical for designing a practical system, so that we can determine the size of the CAM we need. Indeed, recall that the way we choose the target probability $w = 0.2\%$ for $n = 10000$ in Table 5.1 is entirely dependent on such reasoning.

We test the accuracy of this intuition through simulation. For each of the configurations in Table 5.1(b), we run 10^5 trials of the experiment previously outlined, and record, for each trial, the distribution of the $n = 10000$ items over the sub-tables and L . We use this data to test our intuition that the number of items in a particular sub-table is approximately normally distributed, and that the number of items in L is approximately Poisson.

We focus on the second chance scheme, as the results for the other schemes are similar. (The approximations are not quite as good for the SC2 scheme, however, but this is most likely caused by the same rate of convergence issue encountered earlier in our simulation results.) In Figure 5.4, we give a normal probability plot of the number of items in T_4 (the plots for the other sub-tables are similar). Intuitively, the data is plotted so that if the samples were taken independently from a normal distribution, then all of the points would form an approximately straight line with overwhelming probability. Looking at the plot, the data distribution clearly appears to be very close to a normal distribution. Similarly, in Figure 5.5 we plot the empirical cumulative distribution function of the number of items in L and, for comparison, the cumulative distribution function of the $\text{Poisson}(20)$ distribution. The similarity is immediate.

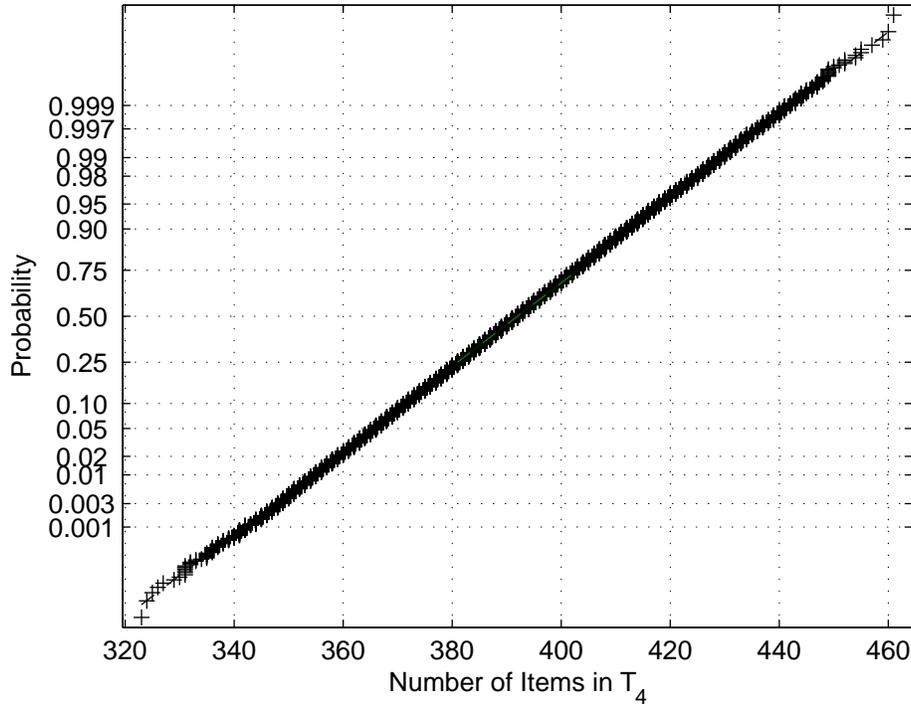


Figure 5.4: Normal probability plot of the number of items in T_4 when simulating the second chance scheme with $d = 4$, $n = 10000$, and target $w = 0.2\%$.

5.9 Deletions

In this section, we adapt our earlier analyses to deal with deletions. Most of the schemes that we consider in this chapter naturally support deletions. In particular, for the standard MHT insertion procedure and the second chance scheme and its variants, we can delete an item simply by removing it from the MHT. The situation for the conservative scheme is more complicated, as the markings on the buckets must be occasionally updated. For simplicity, we omit that scheme from our discussion.

We consider two models for our hash tables when both insertions and deletions are allowed, although we will later see that both models are actually equivalent for our purposes. In both models, we start with an initially empty MHT and overflow list L . The first model is a discrete time stochastic process, where in each of the first n time steps, we insert a new item into the MHT. The remaining time steps consist of inserting a new item into the MHT and then deleting a random item in the system other than the one just inserted. Intuitively, this model corresponds to a pattern of insertions and deletions where the MHT is designed to store n items and is perpetually used at this capacity. If we interpret the state of the MHT at some time t as a vector encoding which positions in the MHT are occupied at time t , then this stochastic process is clearly an irreducible, aperiodic, finite state Markov chain, and thus it is ergodic. We seek to analyze the steady

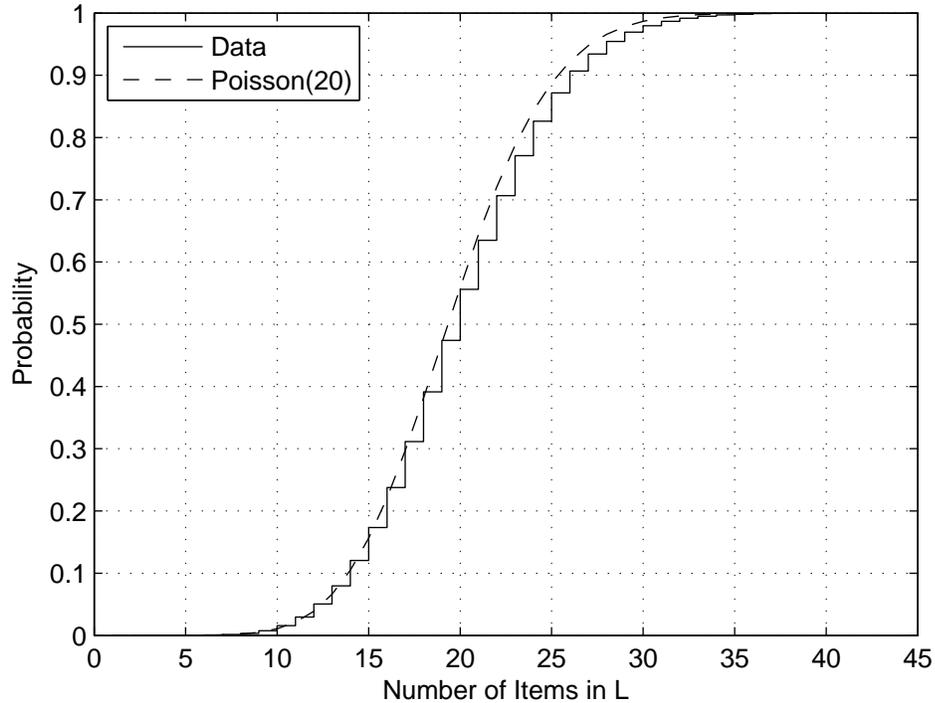


Figure 5.5: Empirical cumulative distribution function of the number of items in L when simulating the second chance scheme with $d = 4$, $n = 10000$, and target $w = 0.2\%$, as compared with the cumulative distribution function of a Poisson(20) random variable.

state behavior of this chain, particularly the expected size of the overflow list.

The second model is a continuous time Markov process where the arrivals of new items are determined according to a Poisson process with rate r and the amount of time that an item remains in the system is exponential with mean λ . It is easy to see that this process is ergodic and that the expected number of items in the steady state is $n \triangleq r/\lambda$. For convenience, we rescale time so that $\lambda = 1$, giving $n = r$. As before, we seek to analyze the steady state behavior of this process, particularly the expected size of the overflow list.

The intuition behind this model is that new items arrive and depart randomly and independently of each other. Indeed, given the widespread use of the Poisson process to model arrivals in other applications, this model may seem more realistic than the first. However, our requirement that the lifetime of an item in the system be exponential is somewhat artificial, and we include it mostly for convenience.

Having introduced and motivated our two models, we now focus on the specific MHT insertion schemes considered in this chapter. We start with the Std scheme in the first model. We embed the discrete time Markov process into continuous time, so that the j -th time step in the discrete chain occurs at time j/n in the embedding. As before, for $i = 1, \dots, d$, we let $\alpha_i n$ denote the size of T_i , where the α_i 's are interpreted as constants with respects to n . Furthermore, we let $F_i(t)$ denote the fraction of buckets in T_i that are

occupied at time t (in the continuous time embedding), and we let $\vec{F}(t)$ denote the vector of the F_i 's. A straightforward calculation then tells us that for any $j \geq n$,

$$\mathbf{E} \left[F_i((j+1)/n) - F_i(j/n) \mid \vec{F}(j/n) \right] = -F_i(j/n) + \frac{1 - F_i(j/n)}{\alpha_i} \prod_{k=1}^{i-1} F_k(j/n).$$

As in Section 5.3, this suggests that we can approximate $F_1(t), \dots, F_d(t)$ by the solution $f_1(t), \dots, f_d(t)$ of the system of differential equations

$$\frac{df_i}{dt} = -f_i + \frac{1 - f_i}{\alpha_i} \prod_{j=1}^{i-1} f_j \quad (5.2)$$

with the $f_i(1)$'s chosen appropriately. If we were so inclined, we could provide a differential equation approximation for the F_i 's for $t \in (0, 1]$ (which would be the same as in Section 5.3), but this is not necessary, because we are only interested in the steady state behavior of the F_i 's, which does not depend on the history of the system before $t = 1$. Thus, for simplicity, we just use (5.2) for all $t > 0$, instead of just $t \geq 1$; we also use this approach for the other schemes that we consider.

From our intuition about the differential equation approximation, we should expect that

$$f_i^* \triangleq \lim_{t \rightarrow \infty} f_i(t) = \lim_{n \rightarrow \infty} \lim_{t \rightarrow \infty} \mathbf{E}[F_i(t)]. \quad (5.3)$$

In this case, $w^* \triangleq 1 - \sum_{i=1}^d f_i^*$ is the asymptotic (as $n \rightarrow \infty$) expected steady state fraction of the n items in L . Such an approximation is useful to us because the f_i^* 's can be approximated numerically by evaluating (5.2) until the values of the $f_i(t)$'s no longer fluctuate much.

Unfortunately, there does not appear to be a general counterpart to Kurtz's theorem that allows us to easily justify (5.3). Nevertheless, there is work on differential equation approximations for queueing systems (e.g., [61, 62, 63]) that suggests that this approach can be formalized, although possibly only on a case-by-case basis. The key property in that work seems to be that all initial conditions are strongly drawn to a unique fixed point of the system of the differential equations. For our purposes, however, it suffices to simply assume that (5.3) is true (as well as its counterparts for the other schemes we consider). Indeed, the validity of our methods will ultimately be established by comparing numerical results with simulations.

Having completed the analysis of the Std scheme in the first model, we now focus on the second model, where new items arrive according to a Poisson process with rate n and the amount of time that an item remains in the system is exponential with mean 1. As before, we let $\alpha_i n$ denote the size of T_i , where the α_i 's are interpreted as constants with respects to n . We let $F_i(t)$ denote the fraction of buckets in T_i that are occupied at time t , and we let $\vec{F}(t)$ denote the vector of the F_i 's. Then given $\vec{F}(t)$, the rate that the i th coordinate grows at time t (in the sense of Poisson processes) is

$$-F_i(t) + \frac{1 - F_i(t)}{\alpha_i} \prod_{j=1}^{i-1} F_j(t).$$

This suggests that the differential equation approximation for the second model is exactly the same as the first model. That is, (5.2) and (5.3) hold in this setting as well. In fact, the same differential equations arise from both methods for all of the schemes that we consider. The intuition here is that both models have the same expected behavior as $n \rightarrow \infty$, and the differential equation approximation depends only the expected behavior of a model. Thus, for our purposes, the two models are equivalent, and we do not distinguish between them in our subsequent analysis.

We now move on to the SC scheme. Unfortunately, the introduction of deletions makes the differential equation approximation for this scheme much more complicated. In particular, as items are inserted into the system, we occasionally learn of *pointers*: items x in some table T_i such that the bucket $T_{i+1}[h_{i+1}](x)$ (called the *destination* of the pointer) is occupied. In our previous analysis of the SC scheme in the absence of deletions, the principle of deferred decisions allowed us to avoid a detailed tracking of pointers. Once deletions are allowed, however, things become more complicated, because we can now have unoccupied buckets that are the destinations of one or more pointers. It is not hard to see that the probability that a particular unoccupied bucket becomes occupied during the insertion of a new item is increasing in the number of pointers having that bucket as a destination. This fact forces us to keep a much more detailed accounting of pointers than before.

Proceeding formally, for $i \in [d]$, we let $F_i(t)$ denote the fraction of buckets in T_i that are occupied at time t . For $i \in [d-1]$, we let $G_i(t)$ be the fraction of buckets in T_i at time t that are occupied by items whose hash location in T_{i+1} is revealed no later than time t . Similarly, for $i \in [d-1]$, we let $O_i(t)$ be the fraction of buckets in T_i at time t that are occupied by an item x whose hash location $h_{i+1}(x)$ in T_{i+1} is revealed no later than time t and $T_{i+1}[h_{i+1}(x)]$ is occupied at time t . As above, whenever there is an item x in some sub-table T_i and $h_{i+1}(x)$ is already revealed, we say that there is a *pointer* from the bucket $T_i[h_i(x)]$ to the bucket $T_{i+1}[h_{i+1}(x)]$. In this case, the bucket $T_{i+1}[h_{i+1}(x)]$ is called the *destination* of the pointer. For $2 \leq i \leq d$ and $j \geq 0$, let $X_{i,j}(t)$ be the fraction of buckets in T_i that are unoccupied and are the destination for exactly j pointers. Similarly, for $2 \leq i \leq d$ and $j \geq 0$, let $Y_{i,j}(t)$ be the fraction of buckets in T_i that are occupied and are the destination for exactly j pointers. Then a straightforward but very tedious calculation tells us that the F_i 's, G_i 's, O_i 's, X_i 's, and Y_i 's are approximated by the following infinite system of differential equations (with the appropriate initial conditions, and uppercase stochastic processes being approximated by their lowercase, deterministic counterparts).

$$\begin{aligned}
 x_{i,\geq 0} &= 1 - f_i && \text{for } i = 2, \dots, d \\
 y_{i,\geq 0} &= f_i && \text{for } i = 2, \dots, d \\
 x_{i,\geq j} &= \sum_{k=j}^{\infty} x_{i,k} && 2 \leq i \leq d, j \geq 1 \\
 y_{i,\geq j} &= \sum_{k=j}^{\infty} y_{i,k} && 2 \leq i \leq d, j \geq 1 \\
 o_i &= \frac{\alpha_{i+1}}{\alpha_i} \sum_{j=1}^{\infty} j y_{i+1,j} && \text{for } i = 1, \dots, d-1
 \end{aligned}$$

$$\begin{aligned}
z_i &= \prod_{j=1}^i o_j + (f_j - g_j)f_{j+1} && \text{for } i = 0, \dots, d-1 \\
\frac{dg_i}{dt} &= -g_i + \frac{z_{i-1}}{\alpha_i}(f_i - g_i)f_{i+1} \\
\frac{df_1}{dt} &= -f_1 + \frac{1 - f_1}{\alpha_1} \\
\frac{df_i}{dt} &= -f_i + \frac{z_{i-2}}{\alpha_i}[f_{i-1}(1 - f_i) + f_i(g_{i-1} - o_{i-1} + (f_{i-1} - g_{i-1})(1 - f_i))] \\
&&& \text{for } i = 2, \dots, d \\
\frac{dx_{i,\geq j}}{dt} &= -jx_{i,j} + y_{i,\geq j} \\
&\quad - z_{i-2} \left[\frac{f_{i-1}x_{i,\geq j}}{\alpha_i} + \frac{f_i(f_{i-1} - g_{i-1})x_{i,\geq j}}{\alpha_i} + \frac{f_i \sum_{k=j}^{\infty} kx_{i,k}}{\alpha_{i-1}} \right] && \text{for } j \geq 1 \\
\frac{dy_{i,\geq j}}{dt} &= -y_{i,\geq j} - jy_{i,j} \\
&\quad + z_{i-2} \left[\frac{f_{i-1}x_{i,\geq j}}{\alpha_i} + \frac{f_i \sum_{k=j}^{\infty} kx_{i,k}}{\alpha_{i-1}} + \frac{f_i(f_{i-1} - g_{i-1})(y_{i,j-1} + x_{i,\geq j-1})}{\alpha_i} \right] && \text{for } j \geq 1
\end{aligned}$$

As before, $w^* = 1 - \sum_{i=1}^d \alpha_i f_i^*$ is the asymptotic (as $n \rightarrow \infty$) steady state fraction of the n items in L , where $f_i^* \triangleq \lim_{t \rightarrow \infty} f_i(t)$. Once again, this statement is heuristic since we do not have a counterpart to Kurtz's theorem for this setting. Also, there are additional theoretical complications introduced by the fact the system of differential equations is infinite. We also note that, for the second model (where items arrive according to a Poisson process), n is only the *expected* number of items in the system in the steady state. Nevertheless, the approximation for w^* is quite accurate in practice, as we shall see in Section 5.10.

Next, we analyze the Std2 scheme. For $i \in [d]$ and $j \in \{0, 1, 2\}$, let $F_{i,j}(t)$ denote the fraction of buckets in T_i that have exactly j items at time t . Another straightforward probability calculation now tells us that the $F_{i,j}$'s can be approximated using the solution of the following system of differential equations (with the appropriate initial conditions).

$$\begin{aligned}
f_{i,\geq j} &= \sum_{k=j}^2 f_{i,k} && \text{for } j = 1, 2 \\
f_{i,0} &= 1 - f_{i,\geq 1} \\
\frac{df_{i,\geq j}}{dt} &= -j f_{i,j} + \frac{f_{i,j-1}}{\alpha_i} \prod_{k=1}^{j-1} f_{i,k} && \text{for } j = 1, 2
\end{aligned}$$

In this setting, the asymptotic (as $n \rightarrow \infty$) steady state fraction of the n items that overflows into L is just $w^* = 1 - \sum_{i=1}^d \sum_{j=1}^2 j \alpha_i f_{i,j}^*$, where $f_{i,j}^* = \lim_{t \rightarrow \infty} f_{i,j}(t)$.

Finally, we consider the SCEExt and Std2 schemes. While the differential equation approximation methods above appear to be applicable to these schemes, their complicated

nature makes this approach impractical. Indeed, these methods do not seem to admit systems of differential equations that can be easily written down, much less numerically analyzed. Specifically, it appears that the number of equations needed for such an approximation is at least exponential in d , and that even for $d = 4$, such a system would be much too large for our numerical techniques to handle.

5.10 Evaluation (Insertions and Deletions)

We now use the differential equation approximations from Section 5.9 to examine the performance of the Std, SC, and Std2 schemes in the presence of deletions using essentially the same methodology as in Section 5.8. As in that section, we first analyze the differential equations using numerical methods, and then we verify the accuracy of our results through simulations.

Before continuing, we note that evaluating the differential equations from Section 5.9 is more complicated than the method in the Section 5.8. The key difference is that, unlike in Section 5.8, it is not sufficient to only evaluate the differential equations up until $t = 1$. Since we are interested in the behavior as $t \rightarrow \infty$, we must examine the differential equations until no variable changes by more than some small amount, up to some maximum time (to ensure that all differential equation calculations are fast, which is crucial for our optimization methods). In our implementation, we evaluate the differential equations in successive calls to the `NDSolve` function of Mathematica 6.0, each of which evaluates the differential equations for 10 time units. (For the infinite system of differential equations corresponding to the SC scheme, we make the system finite by forcing $x_{i,j} = y_{i,j} = 0$ for $j > 20$.) This process terminates when we have reached $t = 30$ or no variable changes by more than 10^{-5} between successive calls. We take the resulting values of the variables as approximations for their limiting values as $t \rightarrow \infty$.

Given this method for approximating limiting values for the differential equations, we can now proceed to compare our schemes as in Section 5.8. For a fixed scheme, space multiplier c , and number of hash functions d , we use the `NMinimize` function of Mathematica 6.0 to try to find the values for $\alpha_1, \dots, \alpha_d$ that minimize the asymptotic (as $n \rightarrow \infty$) steady state overflow w^* subject to the space constraint that $\sum_{i=1}^d \alpha_i \leq c$. We do this for $d = 4$ in Figure 5.6, choosing the values of c in order to give a good high-level picture of how the schemes compare. In this sense, Figure 5.6 is analogous to Figure 5.2 in Section 5.8. We also examine the benefits of optimizing the α_i 's as opposed to the naive setting $\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = c/4$. The results of those calculations are displayed in Figure 5.7, analogously to Figure 5.3 in Section 5.8. Unfortunately, since it now takes much more time to evaluate the differential equations than in Section 5.8, it is not feasible for us to compare the schemes on the basis of a target value for w^* .

As in Section 5.8, Std2 performs better than SC, which performs much better than Std. Furthermore, optimizing the α_i 's still gives a significant decrease in w^* . However, it is worth noting that the overall performance of all of the schemes is much worse than when no deletion operations are allowed. Intuitively, this is because the unoccupied buckets in the MHT that are created by deletions can take a long time to fill, even when one move is allowed. Thus, achieving performance comparable to that of Section 5.8 in scenarios where

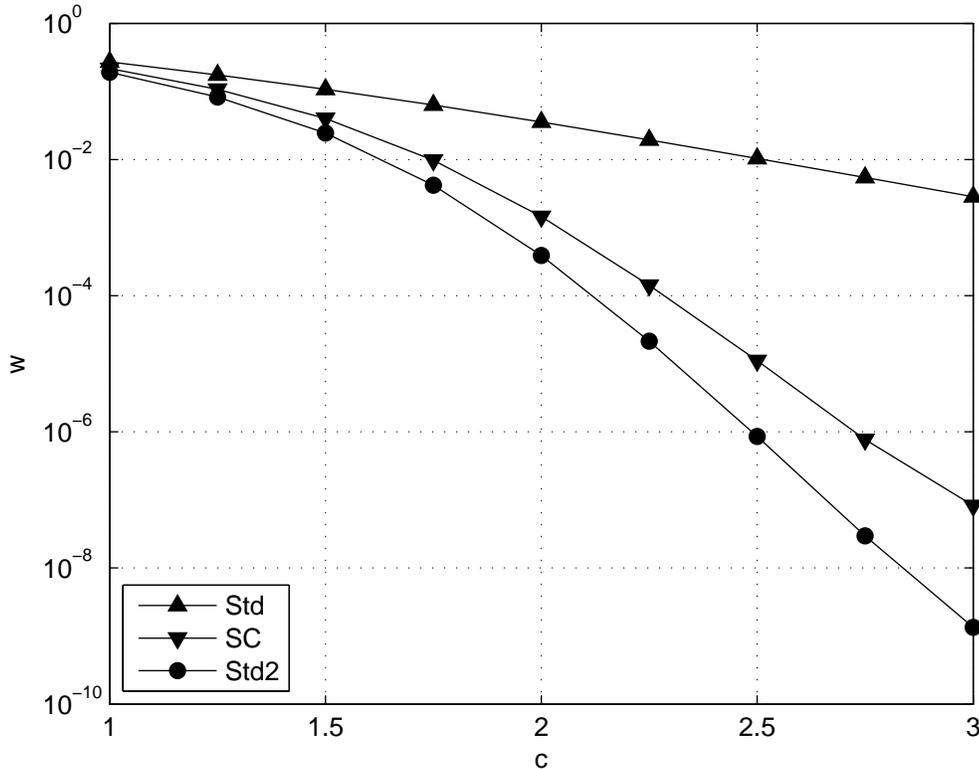


Figure 5.6: Comparison between the asymptotic fractional overflows w and the corresponding space multiplier c for $d = 4$.

deletions are frequent likely requires a different approach.

We now examine the accuracy of our differential equation approximations through simulations. For all of the results in Figures 5.6 and 5.7, we examine each of our deletion models separately. For the first model, given $\alpha_1, \dots, \alpha_4$, we consider an MHT with 4 subtables, where the size of T_i is $\lfloor \alpha_i n \rfloor$. We then simulate the appropriate discrete time Markov chain for $n = 10000$. In each of these simulations, we first insert n items into the MHT, and then perform 3×10^5 alternations of inserting a new item into the MHT and deleting a random item in the system, other than the one just inserted. At this point, we heuristically assume that the chain is in its steady state, and we take the next 10 steps of the chain as being samples from this stationary distribution. We repeat this process 1000 times and average over all samples to obtain our estimates of the steady state behavior of the chain.

The simulation results indicate that the differential equations are accurate, although not as much as in Section 5.8, where there are no deletions. This is not surprising, as the quality of the approximation relies on the degree to which the differential equations track the behavior of the stochastic process over the time interval of interest, and here the time interval is much larger than in Section 5.8 (e.g., 30 time units as opposed to 1). For the Std scheme, the largest relative error for the simulated value of w^* measured against

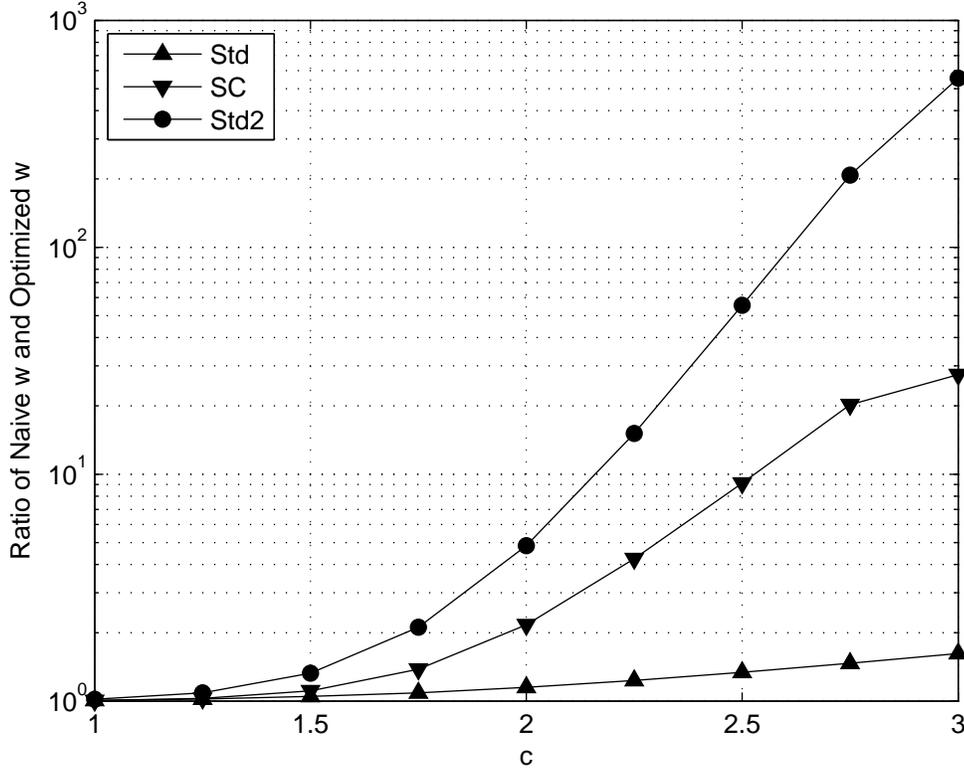


Figure 5.7: Effect on w of optimizing the α_i 's for $d = 4$ and various c values.

the calculated value of w^* is 0.47%. For the SC scheme, the largest relative error for the simulated value of w^* measured against the calculated value of w^* is 1.75% for calculated w^* more than 10^{-3} , about 9.3% for calculated w^* above 10^{-5} , and 100% for the remaining calculated values. For the Std2 scheme, these relative errors are 1.69%, 7.32%, and 236%, respectively. As in Section 5.8, the differential equation approximation is most accurate for the first sub-table of the MHT, slightly less accurate for the second sub-table, and so on. In particular, for the Std, SC, and Std2 schemes, the largest relative error in the simulated values of f_3^* measured against the corresponding calculated values is only 0.18%.

We now turn our attention to the second deletion model, where items arrive according to a Poisson process. As before, we set $n = 1000$ and given $\alpha_1, \dots, \alpha_4$, we let T_i have size $\lfloor \alpha_i n \rfloor$. Here we track Markov chain where state transitions correspond to arrivals or departures of items. We take the first 10 state transitions of this chain after $t = 30$ (in the continuous time process) as coming from the stationary distribution of the Markov process, and we average over 1000 trials to estimate the steady state behavior.

The simulation results are similar to those for the first model. For the Std scheme, the largest relative error for the simulated value of w^* measured against the calculated value of w^* is 0.95%. For the SC scheme, the largest relative error for the simulated value of w^* measured against the calculated value of w^* is 1.55% for calculated w^* more than

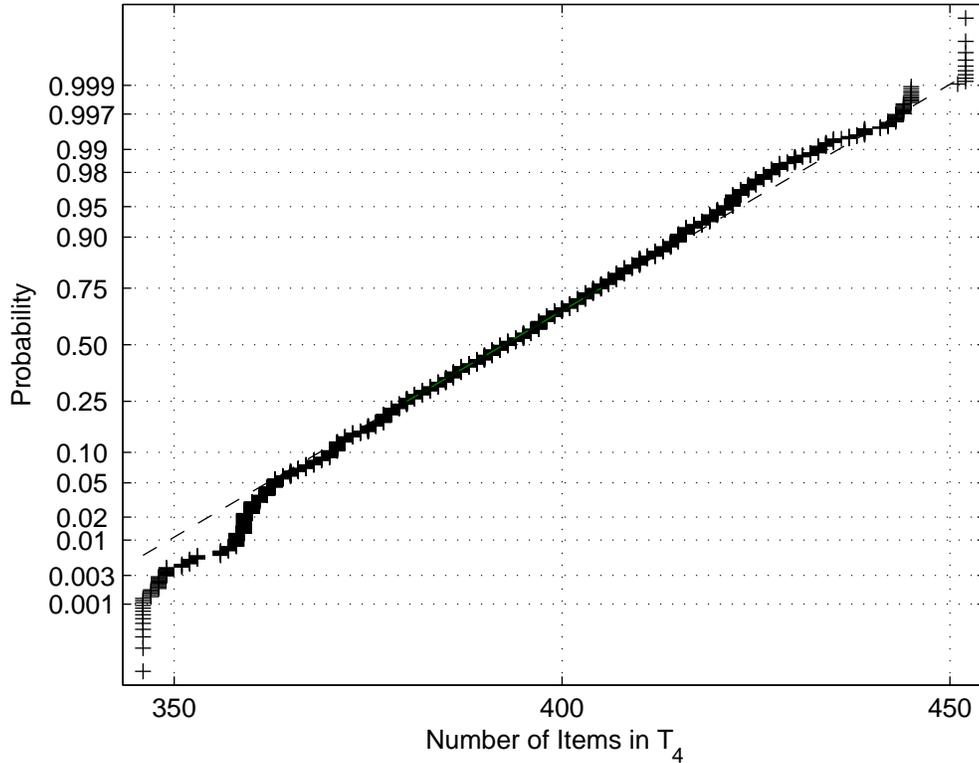


Figure 5.8: Normal probability plot of the number of items in T_4 when simulating the second chance scheme in the first deletion model with $n = 10000$, $c = 2$, and $w^* = 0.146\%$.

10^{-3} , about 14.7% for calculated w^* above 10^{-5} , and 100% for the remaining calculated values. For the Std2 scheme, these relative errors are 0.92%, 10%, and 100%, respectively. As before, the differential equation approximation is most accurate for the first sub-table of the MHT, slightly less accurate for the second sub-table, and so on. In particular, for the Std, SC, and Std2 schemes, the largest relative error in the simulated values of f_3^* measured against the corresponding calculated values is only 0.25%.

Having examined the expected steady state of both deletion models, we now focus on the finer details of the stationary distribution. Intuitively, the situation here should be similar to that in Section 5.8, where we saw that if the MHT is properly configured, then the number of items in each sub-table has an approximately normal distribution, and the size of the overflow list L is approximately Poisson. Indeed, that is essentially what happens here as well, although the results are not as striking as in Section 5.8. To illustrate this phenomenon, we consider the configuration corresponding to the SC scheme with $c = 2$ from Figure 5.6 in the first deletion model with $n = 10000$. Here, the calculated and simulated values of w^* are both approximately 0.15%; we focus on this value of w^* because it is the closest data point that we have to the target value 0.20% that we use throughout Section 5.8. In Figure 5.8, we give a normal probability plot of the number of items in T_4 in

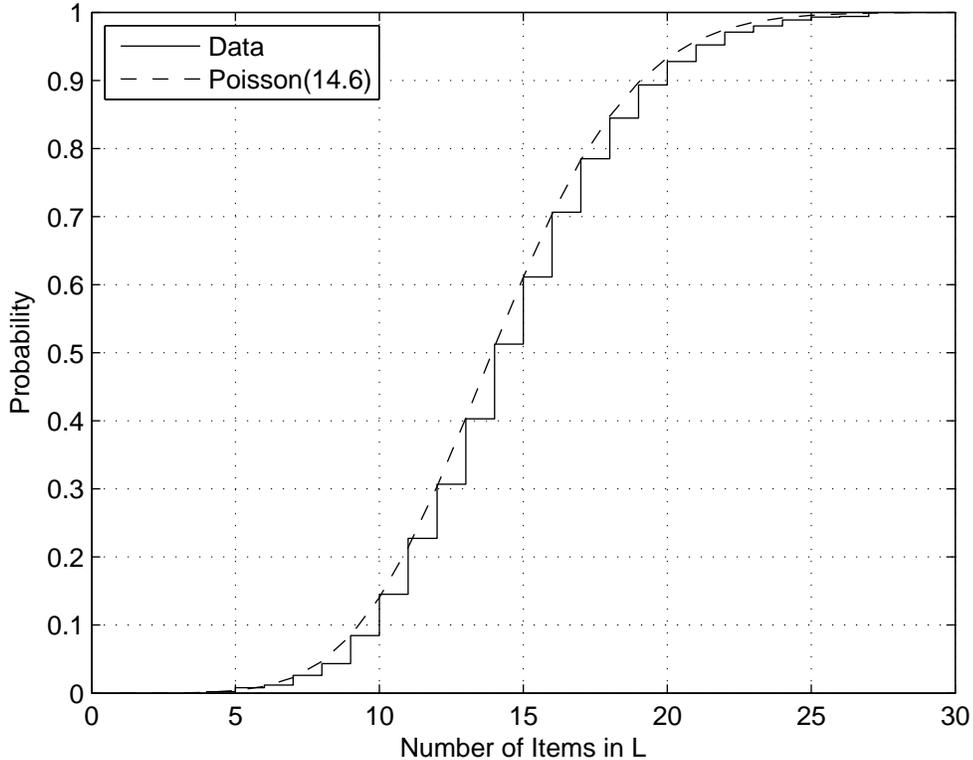


Figure 5.9: Empirical cumulative distribution function of the number of items in L when simulating the second chance scheme in the first deletion model with $n = 10000$, $c = 2$, and $w^* = 0.146\%$, as compared with the cumulative distribution function of a Poisson(14.6) random variable.

the samples drawn during our simulations. In Figure 5.9, we plot the empirical cumulative distribution function of the number of items in L and, for comparison, the cumulative distribution function of a Poisson random variable whose mean is the corresponding value of w^* obtained through simulation. As in Section 5.8, the similarity is immediate.

5.11 Conclusion

We have shown that it is possible and practical to significantly increase the space utilization of a multiple choice hashing scheme by allowing a single move during an insertion procedure. Furthermore, our efforts bridge the theory and practice of such schemes, and provide a solid methodology for future theoretical work, experimental evaluations, and real implementations. In terms of the general contributions of this thesis outlined in Chapter 1, this work illustrates another successful coupling of theoretical foundations (specifically, Kurtz's theorem and the fluid-limit method for analyzing stochastic processes) for intuition

and guiding reasoning, numerical calculations for precision, and simulations for verification.

Bibliography

- [1] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson. The Next Generation of Intel IXP Network Processors. *Intel Technology Journal*, 6(3):6-18, 2002. 13
- [2] Y. Azar, A. Broder, A. Karlin, and E. Upfal. Balanced Allocations. *SIAM Journal on Computing*, 29(1):180-200, 1999. 2, 9, 10, 47
- [3] F. Baboescu and G. Varghese. Scalable Packet Classification. *IEEE/ACM Transactions on Networks*, 13(1):2-14, 2005. 13
- [4] P. Billingsley. *Probability and Measure*. Third edition, John Wiley & Sons, 1995. 21, 22
- [5] B. Bloom. Space/Time Tradeoffs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422-426, 1970. 6
- [6] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines. In *Proceedings of ACM SIGCOMM*, pp. 315-326, 2006. 8
- [7] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. Bloom Filters via d -left Hashing and Dynamic Bit Reassignment. In *Proceedings of the Allerton Conference on Communication, Control and Computing*, 2006. 8
- [8] A. Broder and A. Karlin. Multilevel Adaptive Hashing. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 43-53, 1990. 10, 47, 49, 51, 52, 53, 76, 77, 79
- [9] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4):485-509, 2004. 6, 8, 56, 73
- [10] A. Broder and M. Mitzenmacher. Using Multiple Hash Functions to Improve IP Lookups. *Proceedings of the 20th IEEE International Conference on Computer Communications (INFOCOM)*, pp. 1454-1463, 2001. 13, 47, 77
- [11] J. Byers, J. Considine, and M. Mitzenmacher. Geometric Generalizations of the Power of Two Choices. In *Proceedings of the 16th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 54-63, 2004. 47

-
- [12] J. L. Carter and M. N. Wegman. Universal Classes of Hash Functions. *Journal of Computer and System Sciences*, 18(2):143-154, 1979. 4, 5
- [13] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 30-39, 2004. 8, 49
- [14] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *Journal of Algorithms*, 55(1):58-75, 2005. 8, 16, 42
- [15] H. Cramer. *Random Variables and Probability Distributions*. Third edition, Cambridge University Press, 1970. 39
- [16] E. D. Demaine, T. Jones, and M. Pătraşcu. Interpolation Search for Non-Independent Data. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 522-523, 2004. 54
- [17] M. Dietzfelbinger, T. Hagerup, J. Katajainen, M. Penttonen. A Reliable Randomized Algorithm for the Closest-Pair Problem. *Journal of Algorithms*, 25(1):19-51, 1997. 5
- [18] M. Dietzfelbinger and C. Weidling. Balanced Allocation and Dictionaries with Tightly Packed Constant Size Bins. *Theoretical Computer Science*, 380:(1-2):47-68, 2007. 75, 76
- [19] P. C. Dillinger and P. Manolios. Bloom Filters in Probabilistic Verification. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pp. 367-381, 2004. 2, 15, 25, 26, 41, 45
- [20] P. C. Dillinger and P. Manolios. Fast and Accurate Bitstate Verification for SPIN. In *Proceedings of the 11th International SPIN Workshop on Model Checking of Software (SPIN)*, pp. 57-75, 2004. 2, 15, 25, 26, 41, 45, 59
- [21] B. Donnet, B. Baynat, and T. Friedman. Retouched Bloom Filters: Allowing Networked Applications to Flexibly Trade Off False Positives Against False Negatives. *arxiv, cs.NI/0607038*, 2006. 8
- [22] D. P. Dubhashi and D. Ranjan. Balls and Bins: A Case Study in Negative Dependence. *Random Structures and Algorithms*, 13(2):99-124, 1998. 44
- [23] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. *ACM Transactions on Computer Systems*, 21(3):270-313, 2003. 8
- [24] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, 8(3):281-293, 2000. 5, 8, 47
- [25] A. Feldmann and S. Muthukrishnan. Tradeoffs for Packet Classification. *Proceedings of the Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, vol. 3, pp. 1193-1202, 2000. 13

-
- [26] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis. Space Efficient Hash Tables With Worst Case Constant Access Time. *Theory of Computing Systems*, 38(2):229-248, 2005. 3, 75, 76, 77
- [27] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures In Pascal and C*. Second edition, Addison-Wesley Publishing Company, 1991. 54
- [28] G. Grimmett and D. Stirzaker. *Probability and Random Processes*. Third edition, Oxford University Press, 2001. 29, 30, 91
- [29] K. Ireland and M. Rosen. *A Classical Introduction to Modern Number Theory*. Second edition, Springer-Verlag, 1990. 26, 27
- [30] A. Kirsch and M. Mitzenmacher. Using a Queue to De-amortize Cuckoo Hashing in Hardware. In *Proceedings of the Forty-Fifth Annual Allerton Conference on Communication, Control, and Computing*, 2007. 75
- [31] D. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973. 1, 15
- [32] S. Kumar, J. Turner, P. Crowley, and M. Mitzenmacher. HEXA: Compact Data Structures for Faster Packet Processing. In *Proceedings of the Fifteenth IEEE International Conference on Network Protocols (ICNP)*, pp. 246-255, 2007. 13
- [33] T. G. Kurtz. *Approximation of Population Processes*. Society for Industrial and Applied Mathematics, 1981. 78
- [34] T. G. Kurtz. Solutions of Ordinary Differential Equations as Limits of Pure Jump Markov Processes. *Journal of Applied Probability*, 7:49-58, 1970. 78
- [35] T. V. Lakshman and D. Stiliadis. High-Speed Policy-Based Packet Forwarding Using Efficient Multi-Dimensional Range Matching. In *Proceedings of ACM SIGCOMM*, pp. 203-214, 1998. 13
- [36] G. Lueker and M. Molodowitch. More Analysis of Double Hashing. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing (STOC)*, pp. 354-359, 1988. 16
- [37] J. van Lunteren and T. Engbersen. Fast and Scalable Packet Classification. *IEEE Journal on Selected Areas in Communications*, 21(4):560-571, 2003. 13
- [38] M. Mitzenmacher. Compressed Bloom Filters. *IEEE/ACM Transactions on Networking*, 10(5):613-620, 2002. 8, 34, 47
- [39] M. Mitzenmacher. How Useful Is Old Information? *IEEE Transactions on Parallel and Distributed Systems*, 11(1):6-20, 2000. 47
- [40] M. Mitzenmacher. *The Power of Two Choices in Randomized Load Balancing*. Ph. D. thesis, U.C. Berkeley, 1996. 47, 78

- [41] M. Mitzenmacher, A. Richa, and R. Sitaraman. *The Power of Two Choices: A Survey of Techniques and Results*, edited by P. Pardalos, S. Rajasekaran, J. Reif, and J. Rolim. Kluwer Academic Publishers, Norwell, MA, 2001, pp. 255-312. 9, 47, 52
- [42] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005. 9, 19, 35, 50, 55, 64, 66, 68, 83
- [43] M. Mitzenmacher and S. Vadhan. Why Simple Hash Functions Work: Exploiting the Entropy in a Data Stream. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 746-755, 2008. 6, 16
- [44] M. Mitzenmacher and B. Vöcking. The Asymptotics of Selecting the Shortest of Two, Improved. In *Analytic Methods in Applied Probability: In Memory of Fridrikh Karpelevich*, American Mathematical Society, pp. 165-176, 2003. Edited by Y. Suhov. 77
- [45] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. 23
- [46] A. Östlin and R. Pagh. Uniform Hashing in Constant Time and Linear Space. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing (STOC)*, pp. 622-628, 2003. 5
- [47] A. Pagh, R. Pagh, and S. S. Rao. An Optimal Bloom Filter Replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 823-829, 2005. 8, 49
- [48] R. Pagh and F. Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122-144, 2004. 3, 75, 76
- [49] R. Panigrahy. Efficient Hashing with Lookups in Two Memory Accesses. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 830-839, 2005. 75, 76
- [50] M. V. Ramakrishna. Hashing Practice: Analysis of Hashing and Universal Hashing. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pp. 191-199, 1988. 5
- [51] M. V. Ramakrishna. Practical Performance of Bloom Filters and Parallel Free-Rext Searching. *Communications of the ACM*, 32(10):1237-1239, 1989. 5
- [52] M.V. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient Hardware Hashing Functions for High Performance Computers. *IEEE Transactions on Computers*, 46(12):1378-1381, 1997. 5
- [53] J. P. Schmidt and A. Siegel. The Analysis of Closed Hashing under Limited Randomness. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing (STOC)*, pp. 224-234, 1990. 16

- [54] A. Siegel. On Universal Classes of Extremely Random Constant-Time Hash Functions. *Siam Journal on Computing*, 33(3):505-543, 2004. 5
- [55] A. Shwartz and A. Weiss. *Large Deviations for Performance Analysis: Queues, Communications, and Computing*. Chapman & Hall, 1995. 78
- [56] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing. In *Proceedings of ACM SIGCOMM*, pp. 181-192, 2005. v, 3, 13, 47, 48, 49, 50, 51, 59, 60, 61, 62, 73
- [57] V. Srinivasan and G. Varghese. Fast Address Lookups Using Controlled Prefix Expansion. *ACM Transactions on Computer Systems*, 17(1):1-40, 1999. 13
- [58] M. Thorup. Even Strongly Universal Hashing is Pretty Fast. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 496-497, 2000. 5
- [59] G. Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann Publishers, 2004. 11, 13, 76
- [60] B. Vöcking. How Asymmetry Helps Load Balancing. *Journal of the ACM*, 50(4):568-589, 2003. 2, 9, 10, 77
- [61] N. D. Vvedenskaya, R. L. Dobrushin, and F. I. Karpelevich. Queueing System with Selection of the Shortest of Two Queues: An Asymptotic Approach. *Problems of Information Transmission*, 32(1):15-27, 1996. 94
- [62] N. D. Vvedenskaya and Y. M. Suhov. Dobrushin's Mean-Field Approximation for a Queue with Dynamic Routing. INRIA Research Report 3328, 1997. 94
- [63] N. D. Vvedenskaya and Y. M. Suhov. Functional Equations in Asymptotical Problems of Queueing Theory. *Journal of Mathematical Sciences*, 120(3):1255-1276, 2004. 94
- [64] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. *ACM SIGCOMM Computer Communication Review*, 27(4):25-36, 1997. 13
- [65] G. Wang, W. Gong, and R. Kastner. On the Use of Bloom Filters for Defect Maps in Nanocomputing. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 743-746, 2006. 16
- [66] M. N. Wegman and J. L. Carter. New Hash Functions and Their Use in Authentication and Set Equality. *Journal of Computer and System Sciences*, 22(3):265-279, 1981. 4
- [67] P. Woelfel. Asymmetric Balanced Allocation with Simple Hash Functions. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 424-433, 2006. 6