Stony Brook University

Theoretical and Practical Aspects of Compact Data Structures for Range and Membership Queries in Sparse Sets

A Dissertation Presented

by

Pablo Montes Arango

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

August 2014

Stony Brook University

The Graduate School

Pablo Montes Arango

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation.

Michael A. Bender – Dissertation Advisor
Associate Professor, Department of Computer Science

Rob Johnson – Chairperson of Defense
Assistant Professor, Department of Computer Science

Jie Gao
Associate Professor, Department of Computer Science

Martín Farach-Colton – External Member
Professor, Department of Computer Science
Rutgers University

This dissertation is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Dissertation

Theoretical and Practical Aspects of Compact Data Structures for Range and Membership Queries in Sparse Sets

by

Pablo Montes Arango

Doctor of Philosophy

in

Computer Science

Stony Brook University

2014

There is an increasing need for data structures that efficiently manipulate huge amounts of data. When the data to be represented does not fit into main memory all at once, data structures designed without taking this constraint into account fail to perform as expected. Data structures conceived specifically for this purpose fall into two, not necessarily exclusive, categories. The first consists of data structures designed and analyzed under a model of computation where the cost is given by the amount of data transferred between external and internal memory. The second comprises data structures that make the most out of the internal memory by representing the data using as few space as possible, without hurting the performance of the operations they support.

This dissertation falls in the intersection of both categories: It studies representations that use space close to the information-theoretic lower bound while, at the same time, provide good performance guarantees in terms of the number of memory transfers. The resulting data structures are not only of theoretical interest,

but can also be (and in most cases have been) easily turned into robust, efficient, well tested, and usable implementations, where the constants and low order terms hidden by O notation do not take over the running time in practice.

The focus of this dissertation is on data structures that dynamically maintain a set in order to efficiently answer range and membership queries. Specifically, this work presents two compact data structures: the ***level-based packed memory array***, for range and membership queries, and the ***quotient filter***, for approximate membership queries. Furthermore, three extension to this last data structure are also described: the ***buffered quotient filter***, the ***cascade filter***, and the ***counting quotient filter***.

Finally, this dissertation shows that the standard B-tree data structure is asymptotically optimal for the batched predecessor problem in external memory if the space is a constrain. The tradeoff that quantifies the minimum space required for a given query cost is also presented.

To Alejandro.

Contents

List of Tables

List of Figures

Preface

Chapter 2 is based on unpublished joint work with Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Tsvi Kopelowitz.

Chapter 3 is based on work presented at the 38th International Conference on Very Large Databases (VLDB 2012) and published in the proceedings of the VLDB Endowment under the name "Don't thrash: How to cache your hash on flash" [17]. A preliminary version appeared in the proceedings of the 3rd USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 2011) [18]. This research was done jointly with Michal A. Bender, Martín Farach-Colton, Ron Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pradeep Shetty, Richard P. Spillane, and Erez Zadok.

A version of Chapter 4 was published in the proceedings of the 22nd European Symposium on Algorithms (ESA 2014) under the name "The batched predecessor problem in external memory" [16]. This work was done together with Michael A. Bender, Martín Farach-Colton, Mayank Goswami, Dzejla Medjedovic, and Meng-Tsung Tsai.

Acknowledgements

First and foremost I wish to thank my advisor, professor Michael A. Bender. I have learned so many things from him, but in particular I am very grateful of how he helped me understand and develop my strengths and find what style of research works best for me. Having him as my advisor has been pivotal in realizing what it is that I want to do for the rest of my life.

For this dissertation I would like to thank my committee members: Martín Farach-Colton, Jie Gao, and Rob Johnson, for their time, interest, helpful comments, and insightful questions. Special thanks go to Rob, who has always been available to discuss research ideas, has always provided valuable comments and suggestions, and has helped me get unstuck on more than one occasion.

I would also like to thank all my coauthors: Michael A. Bender, Martín Farach-Colton, Mayank Goswami, Rob Johnson, Joondong Kim, Rusell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Heraldo Memelli, Joseph S. B. Mitchell, Pradeep Shetty, Steven Skiena, Richard P. Spillane, Meng-Tsung Tsai, Charles B. Ward, and Erez Zadok. I know that this is a long list compared to the number of publications, but I can honestly say that I have always enjoyed the the collaborative research process and I have learned something from each and every one of them.

I have to recognize professors Steven Skiena and Joseph Mitchell. Steve is the reason why I came to Stony Brook in the first place and, even though we did not end up working together, his advice in terms of finding an advisor was extremely

helpful. Joe is the kind of professor that I would like to be if I ever become a one. I truly admire him. I am very happy that I managed to reduce both My “Skiena number” and my “Mitchell number” down to 1.

I would like to express my gratitude to Dzejla Medjedovic, for all the time we spent chasing red herrings and wandering into dead ends, and Heraldo Memelli, for all the coffee and the second-hand smoke. Knowing that I was not the only one struggling to get by made the load much more bearable.

Special thanks to all my colleagues in the Algorithms Lab. In particular, I would like to acknowledge Roozbeh Ebrahimi and Sam McCauley. It is a shame that none of the several research projects that we started together crystallized into a publication.

I owe a token of appreciation to the administrative staff at the Department of Computer Science at Stony Brook University: Kathy Germana, Betty Knittweis, and, very fondly, Cindy Scalzo, who has always gone above and beyond to help me in every possible way.

Further, I owe my gratitude to Google for three amazing internship experiences. I had a lot of fun, learned a lot of new and interesting things, and met the most amazing people. I would like to thank my hosts: Vlad Petric, Stacey Gammon, and Yaniv Inbar, and my co-hosts: Nick Taylor and Joe Gregorio.

I am in eternal debt to Politécnico Grancolombiano for giving me the opportunity to pursue a Ph.D. while still maintaining my affiliation to the university. Notably, I would like to thank Javier Arango, for his role in making it a reality, and Rafael García, for always believing in me and for acting as a buffer between me and the university. I regret that things did not work out in the end and I really hope that one day our paths cross again.

At a personal level, I would like to give very special thanks to my family.

To my Parents, because without them I would not be here today (both literally

and figuratively). I know that they do not understand why I decided to embark in this crazy endeavor, but they have always supported and encouraged me nonetheless.

To my sisters and brother, whom I love more than they will ever know. I am really glad to have them in my life.

And last, but certainly not least, to my wife, Angela, for her endless encouragement, optimism and support. Without her by my side I would have probably quit the Ph.D. after the first year. I am also eternally grateful for her willingness to partially fund my refusal to grow up and find a real job.

I dedicate this dissertation to my son, Alejandro, the main reason why I wake up every morning (again, both literally and figuratively). His timely arrival helped me gain perspective on the really important things in life. I hope this work will serve him as an example that he should always pursue his dreams, no matter how unattainable they may seem.

Chapter 1

Introduction

It has been said over and over again: The world is drowning in data [8, 60, 77, 93]. Just to cite a few examples, in 2013 Twitter's IPO filing revealed that 500 million tweets are twitted every day [92], Facebook disclosed that it stores more than 300 petabytes of user data [56], and Skybox Imaging, a startup recently acquired by Google, asserted that each of their satellites generate over a terabyte of data per day [86].

The amount of data produced is growing at an alarming rate. This trend is not expected to stop any time in the near future. According to estimates, the volume of business data worldwide doubles every 1.2 years [51] and data production will be 44 times greater in 2020 than it was in 2009 [1].

It is absolutely impossible to fit these vast amounts of data completely inside the computer's internal memory. As a result, access to data stored in external memory becomes a major performance bottleneck. Data structures and algorithms designed using the traditional RAM model of computation, without taking this limitation into account, fail to perform as expected.

Two major and complementing lines of research have emerged to address this issue. The first is the design and study of data structures and algorithms under a different model of computation, the DAM model [4], in which the performance is

measured by the number of block transfers between external and internal memory. The second is to conceive representations that improve space utilization by squeezing the data so that it uses an amount of space that is "close" to the information-theoretic lower bound, while still allowing efficient operations [68]. This second approach reduces the dependency on slow I/O devices by increasing the amount of data that can can be read/written with one block transfer and, hence, improving spacial locality and reducing the number of disk transfers required to see the data.

This dissertation falls in the intersection of both categories. We are specifically concerned with compact data structures. That is, data structures whose space utilization is at most a constant factor away from the minimum number of bits required to represent the data. At the same time, we are interested in data structures that provide good performance guarantees in terms of the number of memory transfers.

The focus of this dissertation is on dynamic data structures—those allowing insertions and deletions—to maintain a set S from a universe U. We consider data structures specifically designed to efficiently answer range—report all element in S within a given interval—and membership—is a given element x in S—queries.

We strive to provide enough details to make our data structures not only of theoretical interest, but also highly implementable. We regard our data structures as efficient, not only in the theoretical sense, but also in a practical setting. That is because we aim for representations where the constants and low order terms hidden by O notation do not take over the execution running time in practice.

We focus our attention to the common case where, even thought the size of S (the set represented by our data structures) is extremely large, the size of the universe, U, (the set of all possible elements that our data structures could potentially represent) is much larger in comparison. It is in this context that we say that our data structures are specifically designed with sparse sets in mind. It is important to clarify that our data structures still work even when the size of S is comparable

to the size of U (e.g., $|S| = c|U|$ for constant $c \leq 1$). However, they cannot be considered to be compact any more, as their space usage diverges from the optimal number of bits required to represent the data.

1.1 Preliminaries

We begin by formally defining what a compact data structure is and the external memory model of computation.

1.1.1 Space Efficient Data Structures

The study of space-efficient data structures dates back to the mid-70s [52,53]. Since then, there has been a large body of work on data structures with bounded space [24, 31, 40, 68, 79].

A space-efficient data structure is a data structure that uses an amount of space that is "close" to the information theoretic lower bound while remaining competitive with state-of-the-art data structures. Formally speaking, if we let Z be the information-theoretic optimum number of bits to store some data, a data structure is considered to be ***Implicit*** if it uses $Z + O(1)$ bits, ***Succinct*** if it uses $Z + o(Z)$ bits, and ***Compact*** if it uses $O(Z)$ bits. In this dissertation we focust on the last-mentioned category.

1.1.2 External Memory Model

Algorithms and data structures specifically designed for external memory execution are usually modeled using the ***external-memory model*** [4] (also known as the ***I/O model***, or the ***DAM model***). This model is a simplified two-level abstraction of

a computer memory hierarchy that captures the performance bottleneck of accessing data stored in an external memory device. In this model, the internal memory level has limited size, M, and the external memory is unbounded. Transfers between these two levels take place in blocks of size B (B elements are transfered in each block). Any computation can only be performed on data residing in the internal memory. The cost of the algorithm is the number of block transfers (I/Os) performed. The model assumes an optimal cache replacement policy.

1.2 Results

In Chapter 2 we give a compact data structure for the range and membership problem: the ***level-based packed memory array***. The space usage of this data structure is $(1+\varepsilon)n \lg u$ bits, where n is the size of the set being represented and u is the size of the universe of all possible elements that our data structures could potentially represent. With this data structure, membership queries take optimal $O(\lg n)$ comparisons, and range queries take $O(\lg n + k)$ comparisons and $O(\lg(n/B) + k/B)$ block transfers, where k is the size of the output. Our approach yields $O(\lg^2 n)$ amortized cost and $O((\lg^2 n)/B)$ amortized I/Os for insertions, the same as previous approaches. We also give a single unified analysis that covers a broad range of rebalance algorithms and choices of rebalance interval.

Chapter 3 explores the approximate membership problem. We present the ***quotient filter***, a compact data structure with a space usage of $O(n \lg(1/\varepsilon))$ bits for a false positive rate of at most ε. The cost to insert or query an element in this data structure is $O(1)$ in expectation. We also give two new data structures specifically designed for external memory: the ***buffered quotient filter*** and the ***cascade filter***. The first achieves $O\left(\frac{N}{MB}\right)$ I/Os amortized per insert and $O\left(1\right)$ I/Os per query in expectation. The second performs $O\left(\frac{b \log_b(N/M)}{B}\right)$ I/Os amortized per insert and

$O\left(\log_b\left(N/M\right)\right)$ I/Os per query in expectation, where b is the branching factor of the cascade filter, which provides a tradeoff between insert and lookup performance. Lastly, we introduce the ***counting quotient filter*** to handle workloads with lots of duplicates and to efficiently estimate the number of occurrences of a given element.

Finally, in Chapter 4 we show that the batched predecessor problem cannot be solved faster than $\Omega(\log_B n)$ I/Os per element if preprocessing is polynomial. That is, B-trees are asymptotically optimal. We also show that with unbounded space/preprocessing, the problem can be solved faster, in $\Theta((\log_2 n)/B)$ I/Os per element. We generalize to show the following query-preprocessing tradeoff: the search cost of $O(\log_{B/j+1}(n/x)/j)$ per element requires $\Omega(n^{\Omega(j)})$ in preprocessing for query size $x \leq n^c$, where $0 \leq c < 1$.

Chapter 2

Range and Membership Queries

2.1 Introduction

In this chapter we give a new compact data structure to efficiently answer range and membership queries. We call it the ***level-based packed memory array*** given its similarity in spirit to the packed memory array [12, 13]. The main difference is that we do not rebalance regions of the array based on density (ratio of number of elements to size of the subarray) thresholds. In addition, the performance analysis for our insertion algorithm is quite simple and applies to a wide range of strategies for choosing rebalance windows and for rebalancing. We also give a single unified analysis that covers a broad range of rebalance algorithms and choices of rebalance interval.

First, let us formally define the range and membership problems.

Definition 2.1 (Membership Problem)**.** *Maintain a set $S \subset U$, subject to insertions and deletions, to efficiently answer membership queries. That is, given an element $x \in U$, output* TRUE *if $x \in S$ and* FALSE *otherwise. By efficient we mean that the cost to answer a membership query is $O(\lg n)$ comparisons, which is asymptotically optimal in the comparison model.*

Definition 2.2 (Range Query Problem)**.** *Maintain a set $S \subset U$, subject to insertions*

and deletions, to efficiently answer range queries. That is, given a range $[x_\ell, x_r] \subset U$*, output a set* $\{x \in S | x_\ell \leq x \leq x_r\}$*. By efficient we mean that the cost to answer a range query is* $O(\lg n + k)$ *comparisons and* $O(\lg(n/B) + k/B)$ ***I/Os****, where* k *is the size of the output.*

Since we are interested in designing a compact data structure, we now give the information-theoretic minimum number of bits required to solve these problems.

Theorem 2.3 (Entropy lower bound for the membership problem)**.** *Any data structure that represents a static set* S*, of size* n*, from a universe* U*, of size* u*, (*$n < u^{1-\varepsilon}$ *for* $0 < \varepsilon \leq 1$*) requires at least* $\Omega(n \lg u)$ *bits.*

Proof. For any of the $\binom{u}{n}$ possible sets of size n there must be some instance I of the data structure. Let m be the number of bits used by the data structure. An m-bit data structure can represent at most 2^m unique instances. Thus,

$$2^m \geq \binom{u}{n}.$$

Taking logs on both sides we get that

$$\begin{aligned} m &\geq \lg \binom{u}{n} \\ &\geq n \lg \frac{u}{n}. \end{aligned}$$

Finally, under the assumption that $n < u^{1-\varepsilon}$,

$$m = \Omega(n \lg u).$$

□

2.2 Related Work

Itai, Konheim, and Rodeh [67] first showed how to achieve $O(\lg^2 n)$ amortized element moves per insert and delete. The data structure was subsequently deamortized by Willard [94–96], and the deamortized version was later simplified by Bender, Cole, Demaine, Farach-Colton, and Zito [10]. Andersson and La [5] used similar update schemes to maintain dynamic binary trees with height at most an additive $O(1)$ from optimal.

Dietz and Zhang [48,98] gave a $\Omega(\lg^2 n)$ lower bound for *smooth rebalancing*, in which all elements that participate in a rebalance are spread out evenly within their rebalance window (subarray). While evenly spreading out may seem as though it should be the best possible, there are cases where somewhat uneven rebalancing can help [23], meaning that this lower bound, while interesting and powerful, is not the last word. Very recently Bulánek, Koucký, and Saks exhibited a matching lower bound of $\Omega(\lg^2 n)$ without this smooth-rebalancing restriction [33].

The first sparse tables [67, 94–96] rebalance within only those subarrays obtained by implicitly viewing the array as a tree, i.e., the entire array, the left and right halves, the four quarters, and so on. Thus, rebalance windows have special structure, since subarrays are either nested or disjoint. Itai and Katriel [66, 70] and Bender et al. [10] showed that the same rebalancing algorithms can be used for arbitrary subarrays—the subarrays need not be aligned and nested, and need not divide the array into some power of 2.

Over the past decade, there has been interest in sequential-file maintenance because of its application in external-memory and cache-oblivious data structures; see e.g., [13, 14, 29]. The external-memory version of a sparse table (called a packed-memory array (PMA) [13]) supports both upper and lower density thresholds so

that sequential scans of subarrays have an asymptotically optimal number of I/Os. Some external-memory and cache-oblivious structures (e.g., [15, 19, 21]) need the flexibility to choose arbitrarily sized and/or arbitrarily aligned rebalance windows. Sparse tables have also found applications in database query processing [62, 82], sorting [20], and distributed computing [54].

Bender and Hu [22, 23] propose an adaptive packed-memory array (APMA), which enables several common insertion patterns to run using only $O(\lg n)$ element moves per insert/delete, while still supporting the optimal $O(\lg^2 n)$ element moves per insert/delete for arbitrary insertion patterns. The APMA improves performance via slightly uneven (nonsmooth) rebalances, which are biased based upon recent insertions/deletions. One should view the APMA as incomparable with the present k-cursor result. On one hand, the APMA does not require prior knowledge of the number k of insertion points. On the other hand, it is not known to perform better than $O(\lg n)$ element moves per insert/delete and is only known to give performance improvements for a specialized class of insertion patterns.

Sequential-file maintenance is closely related to order maintenance [11,45,47, 91], where the objective is answer order queries (which element comes first) on a dynamic set of elements. Some solutions to order maintenance can be viewed as maintaining a dynamic very sparse array, where n elements are stored in an array of size $\Theta(n^{1+\varepsilon})$. There is a lower bound of $\Omega(\lg n)$ for this problem [33, 46].

2.3 Level-based Packed Memory Array

The most basic version of the level-based packed memory array maintains n elements in sorted order in an array of size $m = (1 + \varepsilon)n$, for $0 < \varepsilon \leq 1$, subject to element insertion (after an existing element) and deletion. We focus only on insertions, because deletions can be supported trivially by marking elements as deleted

("ghost elements") and rebuilding the entire data structure every $n/2$ operations, removing the ghost elements.

We partition the array into $\lceil m/(2 \lg m) \rceil$ ***segments***, each of size $2 \lg m$. Without loss of generality, we assume that n is divisible by $\lg m$, as we can insert up to $\lg m$ "dummy elements" without affecting the performance bounds. Each element in the array is assigned a ***level*** from $\{0, 1, \ldots, \lg m - 1\} \cup \{\infty\}$ in such a way that the following invariants are maintained:

Invariant 2.4. *Every segment contains exactly* $\lg m$ *elements at level* ∞.

Invariant 2.5. *Every segment contains at most one element at each level other than* ∞.

We maintain the level of an element implicitly, based on its position within a segment. In particular, if an element is in an odd array slot, then it has level ∞; if an element is in the i-th even slot within the segment, then it has level i. This construction directly implies Invariant 2.5.

We call a contiguous group of segments an ***interval***, and we define its ***size*** to be the number of segments spanned. For an interval K, of size $|K|$, we quantify its amount of ***dirt*** as the number of elements in K at or below level $\lg |K|$. We say that K is ***dirty*** if it has at least $|K|$ dirt, and ***not dirty*** otherwise. If an interval has zero dirt we say it is ***clean***.

At a high level, upon an insertion of a new element y the algorithm finds an appropriate interval K containing y that is not too dirty (or too clean), and performs a ***cleaning process***. That is, the algorithm ***promotes*** all of its dirt to level $\lg |K|$. Once this promotion is done, all subintervals of K are clean.

2.3.1 Algorithm

We now describe how to insert an element y after an existing element x.

Let s be the segment containing x. We first insert y in s at level 0. Since levels are implicitly maintained based on the position, what we actually do is shift elements in s so that all previously filled positions are still filled, plus the level 0 position.

Next we find an appropriate interval, K, to start a cleaning process. Initially K is the segment containing x. If K is dirty, we double its size (to either side). Otherwise, we start a cleaning process on K.

A cleaning process has to guarantee that, once it is done, Invariants 2.4 and 2.5 are maintained. We also shift elements within K in such a way that at the end of the cleaning process:

1. each level $i > \lg |K|$ has the same number of elements as it had before the cleaning process started,

2. the total number of elements at level $\lg |K|$ is equal to the total dirt in K before the cleaning process started, and

3. there are no elements in K below level $\lg |K|$.

Note that we do, however, change which specific elements are at what levels, as well as which segments contain an element at those levels.

A naive algorithm for a cleaning process that runs in linear time with respect to number of positions in K is as follows. Let d_K be the amount of dirt in K. First pack everything tightly to the left (towards the beginning of the array) while marking all positions representing levels above $\lg |K|$ that were occupied as available. Also choose any d_K positions representing level $\lg |K|$ and make them available. All other slots are unavailable. Next, starting from he rightmost element, spread elements to the right into the next available position.

If the entire array is dirty, we rebuild the entire array increasing its size so that all elements can be promoted to level ∞. Invariants 2.4 and 2.5 hold trivially when the array is rebuilt.

The following lemma implies Invariants 2.4 and 2.5 across cleanings. It also provides the basis for the performance analysis by lower-bounding the number of level increases that occur during a cleaning process.

Lemma 2.6. *Consider a cleaning process on an interval K. Let $k = \lg |K|$. For each level j, let n_j and n'_j be the number of elements in K at level j before and after the cleaning process, respectively. Then for all $j > k$, we have $n'_j = n_j$. For all $j < k$, we have $n'_j = 0$. Finally, $n'_k = \sum_{j \le k} n_j$, meaning that at least $\sum_{j<k} n_j$ promotions occur.*

Proof. Because the number of slots made available is exactly the number of elements, all available slots are occupied when the cleaning process finishes. The claim follows from the choice of slots made available. □

Theorem 2.7. *The level-based packed memory array performs $O(\lg^2 n)$ amortized element moves per insert.*

Proof. A cleaning process for an interval K promotes at least $|K|/2$ and strictly less than $|K|$ elements. The cost of this cleaning process is $O(|K| \lg m)$. Thus, the amortized cost of a cleaning process per promotion is $O(|K| \lg m)/\Theta(|K|) = O(\lg m) = O(\lg n)$. Since an element can only be promoted to $\lg m$ different levels, the amortized cost of inserting an element is $O(\lg^2 n)$. □

2.4 Other Rebalancing Strategies

The region-doubling approach presented in Section 2.3 is already more general than cleaning according to say a logical tree over segments, as employed in the original

sparse table [67], because the cleaning interval can grow arbitrarily in either direction. In this section we give a more general analysis that supports many different implementations.

Theorem 2.8. *Consider any rebalance procedure with the following property: whenever rebalancing an interval consisting of X segments,*

(a) the number of elements promoted is $\Omega(X)$,

(b) the cost to promote all $\Omega(X)$ elements and restore Invariants 2.4 and 2.5 is $O(X \lg m)$, and

(c) the total number of elements at any level is at most $X \lg m$ for level ∞ and X for other levels.

Then the amortized insertion cost into the sparse table is $O(\lg^2 n)$.

Proof. Charge the cost of performing the rebalance to the level increases, for an amortized cost of $O(\lg n)$ per level increase. Since there can be at most $O(n \lg n)$ level increases in total for n elements, or $O(\lg n)$ increase per element, the amortized cost of an insertion becomes $O(\lg^2 n)$ per element for rebalances plus $O(\lg n)$ for inserting into the chunk, which is $O(\lg^2 n)$ in total. □

There are many possible rebalancing procedures that satisfy Theorem 2.8:

Arbitrary windows. As stated before, the region-doubling approach is already more general than cleaning according to say a logical tree over the segments. Thus, always extending the intervals in the same direction is nearly trivial. If we want intervals that only grow to the right, then a slight complication here is that a relatively small interval may extend off the end of the array, and thus we cannot afford to rebuild the entire array. This issue can be resolved by allocating some additional

empty segments at the end of the array. If a rebalance ever extends off the end, all elements in the interval can then be promoted to level ∞, and the appropriate number of segments at the end can be claimed.

Monotonically moving elements. Cleaning such that all elements move in the same direction (e.g., as employed in [15]) basically amounts to growing the region continuously, say "to the right," increasing by 1 chunk at a time, until we find an interval K such that the total number of elements at or below level $\lceil \lg |K| \rceil$ is equal to K. To guarantee one-directional movement, elements need only be packed as tightly to the right as allowed by Invariants 2.4 and 2.5, so that the number of elements in any prefix of the rebalance region is nonincreasing. Given that the slots for levels $1, 2, \ldots, \lg m - 1$ occur in left-to-right order, any promotion moves an element to the right. The analysis of the insertion cost here is already captured by Theorem 2.8. In contrast, one-directional rebalances in the standard density-based algorithm require a more complicated analysis [15, 70].

Nonsmooth rebalances. This approach allows for nonuniform redistribution of elements during a rebalance. Specifically, a cleaning process allows for some arbitrary assignment, so it is allowable for one chunk to receive elements at many levels while others receive elements at few levels. Finally, levels can be increased arbitrarily as long as Invariants 2.4 and 2.5 is not violated. For example, there is nothing wrong with inserting directly at a higher level, so long as the strategy is consistent with Invariants 2.4 and 2.5 and Theorem 2.8.

Chapter 3

Approximate Membership

3.1 Introduction

In Chapter 2 we showed that any data structure that solves the membership problem requires at least $\Omega(n \lg u)$ bits. When n is extremely large, however, even the cost for maintaining an implicit data structure that uses exactly this much space is prohibitive. For some applications, for example, it might be desirable to store the data structure in main memory for efficiency reasons. In this chapter we show that it is actually possible use much less space, as long as you are willing to sacrifice precision.

To be more precise, in this chapter we study the approximate membership problem, where the answers given by the data structure can be wrong. The probability of getting the wrong answer, however, is bounded, small, and tunable. The space usage of the data structure is a function of this probability—the probability of being wrong grows as the space consumption decreases.

We are explicitly concerned in one-sided error data structures. That is, if the data structure asserts that an element *is not* in the set, we can be sure that this is correct. On the other hand, if the data structure claims that an element *is* in the set, then there is a small probability that this is not actually the case. We call this type

of data structures ***approximate membership query*** (AMQ) data structures.

The formal definition of the approximate membership problem is as follows.

Definition 3.9 (Approximate Membership Problem)**.** *Maintain a set $S \subseteq U$, subject to insertions and deletions, to efficiently answer approximate membership queries. That is, given an element $x \in U$, output*

- *If $x \in S$,* TRUE
- *If $x \notin S$*
  - FALSE *with probability at least $1 - \varepsilon$*
  - TRUE *with probability at most ε (a* false positive*)*

In this case we step outside the comparison model and define efficient as $O(1)$ comparisons in expectation.

As stated above, the space requirement for this type of data structures is much less than the requirements for the accurate counterpart, and is a function of its false positive probability.

Theorem 3.10 (Entropy lower bound for the approximate membership problem)**.** *Any data structure that represents a static set S, of size n, from a universe U, of size u, ($n < \sqrt{\varepsilon u}$) that supports approximate membership queries allowing false positives for at most a fraction ε of the universe but no false negatives requires at least $\Omega(n \lg(1/\varepsilon))$ bits.*

Proof. The proof follows Dietzfelbinger and Pagh [49]. An instance I of the data structure corresponds to a set $U_I \subseteq U$—the set of elements for which the data structure answers $x \in S$—with at most $n + \varepsilon(u - n)$ elements. For any set $S \subseteq U$, of size n, there must be some instance of the data structure. An instance of the data

structure can represent at most $\binom{n+\varepsilon(u-n)}{n}$ such sets. Let m be the number of bits used by the data structure. An m-bit data structure can represent at most 2^m unique instances. Thus,

$$\begin{aligned}
2^m &\geq \frac{\binom{u}{n}}{\binom{n+\varepsilon(u-n)}{n}} \\
&\geq \left(\frac{u}{n+\varepsilon(u-n)}\right)^n \\
&= \left(\frac{u}{\varepsilon u+(1-\varepsilon)n}\right)^n \\
&= \left(\frac{1}{\varepsilon}\left(1-\frac{(1-\varepsilon)n}{\varepsilon u+(1-\varepsilon)n}\right)\right)^n \\
&\geq \left(\frac{1}{\varepsilon}\right)^n \left(\frac{1}{e}\right)^{-\frac{(1-\varepsilon)n^2}{\varepsilon u+(1-\varepsilon)n}}
\end{aligned}$$

Taking logs on both sides and assuming that $n < \sqrt{\varepsilon u}$ we get that

$$m \geq n \lg \frac{1}{\varepsilon} - O(1)$$

as required. □

The most well-know AMQ data structure is the Bloom filter [25]. The Bloom filter has found wide application in databases, storage systems, and networks in order to quickly satisfy queries for elements that do not exist in the database, in external storage, or on a remote network host. This chapter first describes the quotient filter, which supports the basic operations of the Bloom filter, achieving roughly comparable performance in terms of space and time, but with better data locality.

Bloom filters work well when they fit in main memory. However, because the Bloom filter performs frequent random reads and writes, it is used almost exclusively in RAM, limiting the size of the sets it can represent. Once Bloom filters get

larger than RAM, their performance decays and, thus, they do not scale efficiently to external storage, such as flash. In this chapter we give two data structures, the buffered quotient filter and the cascade filter, which exploit the quotient filter advantages, but are specifically designed to be efficient in external storage. Both data structures significantly outperform recently-proposed SSD-optimized Bloom filter variants, such as the elevator Bloom filter, buffered Bloom filter, and forest-structured Bloom filter. In experiments, the cascade filter and buffered quotient filter performed insertions 8.6-11 times faster than the fastest Bloom filter variant and performed lookups 0.94-2.56 times faster.

3.1.1 Evaluation Results

Our evaluation compares the QFs, BQFs, and CFs to BFs and recently proposed BF variants, including buffered Bloom filters (BBF) [34], forest-structured Bloom filters (FBF) [74], and elevator Bloom filters (EBF). For the overview of BF variants, see Section 3.2. The BBF and FBF were proposed to address the scaling problems of Bloom filters, in particular, when they spill onto SSDs. The EBF is an extension of the BF, which we include as a baseline.

To differentiate the previously existing structures: The EBF is a straightforward application of buffering to BFs. The BBF uses buffering and hash localization to improve SSD performance. The FBF uses buffering, hash localization, as well as in-RAM buffer-management techniques.

Table 1 presents a summary of our experimental results. To put these numbers in perspective, on an Intel X-25M SSD drive, we measured 3,910 random 1-byte writes per second and 3,200 random 1-byte reads per second. Sequential reads run at 261 MB/s, and sequential writes run at 109 MB/s.

We performed three sets of experiments: in RAM, small-scale on SSD, and

large-scale on SSD. We performed the different SSD experiments because the effectiveness of buffering decreases as the ratio of in-RAM to on-disk data decreases.

In each case, we compared the rate of insertions, the rate of uniform random lookups, which amounts to lookups for elements not in the AMQ, and the rate of successful lookups, that is, lookups of elements present in the AMQ. We make this distinction in lookups because a BF only needs to check an expected two bits for unsuccessful lookups, but k bits for successful lookups when there are k hash functions. (For our error rates, the BF had 6, 9, and 12 hash functions, respectively.)

3.1.1.1 In-RAM Experiments

For our in-RAM experiments, we compare the QF and the BF. The QF is supposed to be used when it is at most 75% full; as Figure 7 shows, the QF performance deteriorates as it fills. Table 1 reports on results when the structures are 75% full.

For inserts, QFs outperform BFs by factors of 1.3× to 2.5×, depending on the false positive rates. For uniform random lookups, BFs are 1.4×-1.6× faster. For successful lookups, there is no clear winner.

3.1.1.2 Small On-SSD Experiments

We compared our two SSD data structures to the three Bloom filter variants. In these experiments, the AMQs were grown so that they are approximately four times the size of RAM. See Section 3.6.2 for details.

We find that both BQF and CF insert at least 4 times faster than other data structures and that BQF is at least twice as fast for lookups as all the other AMQs we measured. In fact, on successful lookups, it runs roughly 11 times better than EBF and BBF.

The BQF is the clear winner for this set of experiments.

(a) In-RAM experimental results (operations per second).

AMQ	**BF**	**QF**	**BF**	**QF**	**BF**	**QF**
False Positive Rate	**0.01**	**0.01**	**0.002**	**0.002**	**0.0002**	**0.0002**
Uniform Random Inserts	1.72 mil	2.44 mil	1.29 mil	2.43 mil	991,000	2.45 mil
Uniform Random Lookups	3.1 mil	2.1 mil	3.35 mil	1.98 mil	3.37 mil	2.13 mil
Successful Lookups	1.93 mil	1.61 mil	1.65 mil	1.7 mil	1.44 mil	1.71 mil

(b) On-disk experimental results (operations per second).

AMQ		**CF**	**BQF**	**EBF**	**BBF**	**FBF**
Small experiment	Uniform Random Inserts	1.075 mil	1.32 mil	205,000	249,000	43,100
	Uniform Random Lookups	2,200	4,480	2,180	2,340	1,510
	Successful Lookups	2,950	4,690	372	441	1,830
Large experiment	Uniform Random Inserts	728,000	576,000			
	Uniform Random Lookups	1,940	3,600			
	Successful Lookups	2,380	3,780			

Table 1: Summary of evaluation results.

3.1.1.3 Large On-SSD Experiments

We ran all AMQs for 35,000 seconds. This was enough time for CF and BQF to insert the full data set. However, BBF, FBF, and EBF were at least 10 times slower for insertions and none of them managed to get through even 10% of the insertion load. We therefore conclude that these data structures are not suitable for such workloads.

We note that this workload was large enough for asymptotics to kick in: the CF was 26% faster than the BQF. BQF still dominates for queries, outperforming CF by at least 60%. Therefore the choice of CF versus BQF depends on the ratio of insertions to queries in a particular workload.

3.1.1.4 Other Considerations

For typical configurations (e.g., a 1% false positive rate) a QF uses about 20% more space than a BF. However, QFs (and BQFs and CFs) support deletion, whereas BFs incur a $4\times$ space blow-up to support deletion, and even then they may fail. QFs

support in-order iteration over the hash values inserted into the filter. Consequently, QFs can be dynamically resized, and two QFs can be merged into a single larger filter using an algorithm similar to the merge operation in merge sort. QF inserts and lookups require a single random write or read. BF inserts require multiple writes, and lookups require two reads on average.

3.1.2 Applications

Write-optimized AMQs, such as the CF and BQF, can provide a performance improvement in databases in which inserts and queries are *decoupled* (i.e., insertion operations do not depend on the results of query operations). Webtable [35], a database that associates domain names of websites with website attributes, exemplifies such a workload. An automated web crawler adds new entries into the database while users independently perform queries. The Webtable workload is decoupled because it permits duplicate entries, meaning that searches for duplicates need not be performed before each insertion.

The system optimizes for a high insertion rate by splitting the database tables into smaller subtables, and searches are replicated across all the subtables. To make searches fast, the system maintains an in-memory Bloom filter for each subtable. The Bloom filter enables the database to avoid I/O to subtables that do not contain the queried element.

The CF and BQF could enable databases, such as Webtable, to scale to larger sizes without a concomitant increase in RAM. SSD-optimized AMQs, such as the CF and BQF, can keep up with the high insertion throughput of write-optimized databases.

Similar workloads to Webtable, which also require fast insertions and independent searches, are growing in importance [35,58,71]. Bloom filters are also used for

deduplication [99], distributed information retrieval [85], network computing [30], stream computing [97], bioinformatics [37, 75], database querying [78], and probabilistic verification [65].

3.2 Related Work

The first known AMQ data structure, introduced more than four decades ago [25], is the Bloom filter (BF). A BF is a lossy, space-efficient representation of a set. It supports two operations: INSERT(B, x) and MAY-CONTAIN(B, x).

A BF B consists of an m-bit array $B[0\,..\,m-1]$, initially all set to 0, and k independent random hash functions $h_i : U \to \{0, \ldots, m-1\}$, where $1 \le i \le k$ and U is the universe of objects that may be inserted into the filter. To insert an item x, the filter sets

$$B[h_i(x)] \leftarrow 1 \qquad \text{for } i = 1, \ldots, k.$$

To test whether an element x may have ever been inserted, the filter checks all the bits that would have been set:

$$\text{MAY-CONTAIN}(B, x) = \bigwedge_{i=1}^{k} B[h_i(x)].$$

If not all of them are set, then we conclude that s is not in the set; otherwise, we assume that that the element is present in the set, although in fact the element could be absent.

The false-positive rate of a BF after inserting n items is approximately

$$(1 - e^{-nk/m})^k.$$

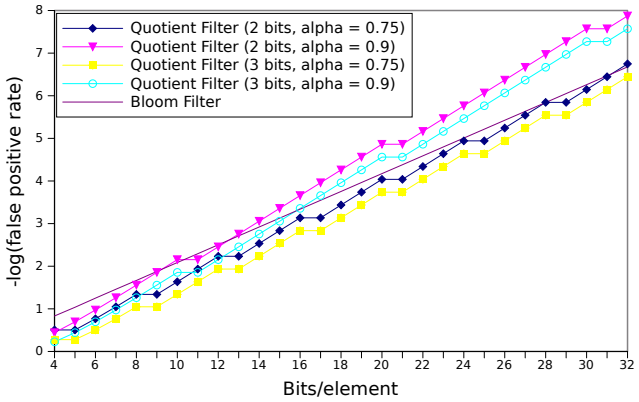


Figure 1: False positive rates for BF and QF. For typical parameters (e.g., 1% false positive rate), QF require about 20% more space than a BF. For extremely low false positive rates, QF use less space than a BF.

This rate is optimized by choosing

$$k = \frac{m}{n} \ln 2,$$

which means that roughly half of the bits in B are set to 1.

As a concrete example, an optimally filled BF with $m = 8n$ (i.e., 1 byte per element) would use six hash functions and can achieve a false positive rate of 1.56%. Figure 1 shows the BF false positive rate, assuming the optimal number of hash functions, as a function of the number of bits per element.

There exist many variants of Bloom filters, optimizing for the false-positive rate, the space-efficiency, or both [36, 42, 61, 76, 80]. For a comprehensive review of Bloom filters, see Broder and Mitzenmacher's survey [30].

BFs has several limitations:

- A BF does not expand to accommodate new elements, so sufficient space for all the elements must be allocated in advance.

- A BF does not support deletions. The data structure can be easily modified to

support deletions by substituting each entry with a counter [57]. This modification substantially increases the space requirements—for most applications, four bits per entry are sufficient to avoid a counter overflow [27].

- A BF does not naturally scale to external storage because of the poor data locality, and consequently they are usually stored in RAM. To illustrate, a BF stored on a rotating disk, with $k = 10$ hash functions, could insert fewer than 20 elements per second.

The Buffered Bloom filter, described by Canim et al. [34], is a Bloom filter variation specifically designed for use on flash. They make two modifications. First, the filter is divided into smaller sub-Bloom filters, each of which is configured to be the size of a virtual page. A given element is represented in exactly one sub-filter chosen at random. Second, each sub-filter has a dedicated in-RAM buffer that collects the read/write requests. Upon filling the buffer, the elements are flushed in bulk to the corresponding sub-filter on flash. This approach improves the locality of otherwise random bit writes that a Bloom filter performs. However, if the flash is much larger than RAM, buffers in RAM will hold only a small fraction of the total sub-filter on flash. Thus, the structure becomes dependent on the random write throughput of the flash, limiting its scalability significantly. Canim et al. report a $3\times$ performance improvement over the traditional Bloom filter on flash.

In general, the following approaches have been used by researchers to improve BF scalability:

- **Replacing magnetic disks with SSDs.** SSDs offer random read and write rates superior to those of magnetic disks, which partly alleviates the performance issues of an on-disk BF. With an off-the-shelf SSD, the traditional BF with $k = 10$ hash functions can achieve roughly 500 inserts per second. High-end devices, such as FusionIO [38], can offer further speedups.

- **Buffering.** Reserve a buffer space in RAM, and cache these updates in the buffer. Flush the buffer as it becomes full. With buffering, multiple bit writes destined for the same SSD block require only one I/O. Similarly, when most pages have a pending update, there is an additional performance gain of the sequential disk write. The elevator Bloom filter implements this strategy. In general, buffering performs well when the ratio between the Bloom filter size and the RAM buffer size is small. As described in [34], queries can also be delayed and buffered in a multithreaded environment, but the present paper measures the performance when queries must be answered immediately.

- **Hash localization.** Improve data locality by directing all hashes of one insertion into a single SSD block. When combined with buffering, this can substantially improve the locality of writes. Queries see a less dramatic improvement in locality. BF variants, such as the buffered Bloom filter [34] and the closely related BloomFlash [44], use this strategy. Localizing hashes to one block moderates the randomness of bit reads/writes across the disk and, in expectation, does not substantially hurt the false positive rate.

- **Multi-layered design.** Maintain multiple on-disk BFs, exponentially increasing in size. Insert only into the largest and most recent BF. This approach effectively reduces the ratio between the RAM size and the active BF by a factor of 2, but increases the search cost, since a search must query all Bloom filters. The forest-structured Bloom filter [74] uses this strategy.

- **Buffer design and flushing policy.** Different buffer management schemes may lead to different performance characteristics. In the BBF, the buffer is equally divided into a number of sub-buffers, each serving updates for a particular SSD block. When a sub-buffer becomes full, its updates are applied

with one I/O. BloomFlash flushes the group of c contiguous sub-buffers that has the most updates, and optimizes for c. The FBF does space stealing between sub-buffers to delay flushing to disk until the RAM is full. A mayor disadvantage of this design is the space overhead necessary to achieve this dynamism, reducing the actual space available to cache the insertions.

3.3 Quotient Filter

In this section we describe the quotient filter, a space-efficient and cache-friendly data structure that delivers all the functionality of the Bloom filter. We explain advantages of the QF over the BF that make the QF particularly suitable to serve as the foundation for our SSD-resident data structures.

The QF represents a multi-set of elements $S \subseteq U$ by storing a p-bit fingerprint for each of its elements. Specifically, the QF stores the multi-set $F = h(S) = \{h(x) \mid x \in S\}$, where $h : U \to \{0, \ldots, 2^p - 1\}$ is a uniform random hash function. To insert an element x into S, we insert $h(x)$ into F. To test whether an element $x \in S$, we check whether $h(x) \in F$. To remove an element x from S, we remove (one copy of) $h(x)$ from F.

Conceptually, we can think of F as being stored in an open hash table T with $m = 2^q$ buckets using a technique called ***quotienting***, first suggested by Knuth [73, Section 6.4, exercise 13] based on the observation that the first q bits of the hash can be inferred by the bucket number in which the hash is stored; see the open hash table (i.e., hash table with chaining) at the top of Figure 2. In this technique a fingerprint f is partitioned into its r least significant bits, $f_r = f \bmod 2^r$ (the remainder), and its $q = p - r$ most significant bits, $f_q = \lfloor f/2^r \rfloor$ (the quotient). To insert a fingerprint f into F, we store f_r in bucket $T[f_q]$. Given a remainder f_r in bucket f_q, the full fingerprint can be uniquely reconstructed as $f = f_q 2^r + f_r$.

In a static setting, we can think of q and r in terms of both, n and ε. Specifically, we let $q = \lg(cn)$, for $1 < c \leq 2$, and $r = \lg(1/\varepsilon)$, where ε is a negative power of two. The constant c specifies the load factor $\alpha = 1/c$ of the hash table. As expected, as c gets closer to one the performance of the QF declines.

To reduce the memory required to store the fingerprints and achieve better spatial locality, we actually use open addressing (closed hashing) instead of a hash table with chaining. As such, the QF can be thought of as an array $A[0\,..\,m-1]$ of r-bit items.

If two fingerprints f and f' have the same quotient ($f_q = f'_q$) we say there is a ***soft collision***. In this case we use linear probing as a collision-resolution strategy. All remainders of fingerprints with the same quotient are stored contiguously in what we call a ***run***. If necessary, a remainder is shifted forward from its original location and stored in a subsequent slot, wrapping around at the end of the array. We maintain the invariant that if $f_q < f'_q$, f_r is stored before f'_r in A, modulo this wrapping.

In order to be able to reconstruct the fingerprints stored in a QF we need to be able to distinguish the original bucket in which a given remainder belongs. This is trivial if we where using an open hash table, but not as much when we use open addressing. To see why, consider a remainder currently in the QF. Just by looking at its current position in the array it is impossible to tell if it is currently stored in its original bucket, or if it has been shifted because of a soft collision.

To solve this, we need to maintain an ***offset*** that keeps track of how far from its original location a remainder is; see Figure 2, bottom. Obviously, maintaining this offsets as regular counters is expensive in terms of space. In Section 3.4 we show one possible way of encoding the offsets using only three extra bits per bucket. We note that it is actually possible to reduce this number down to two bits per bucket, and that this is provably optimal. Designing, implementing, and benchmarking

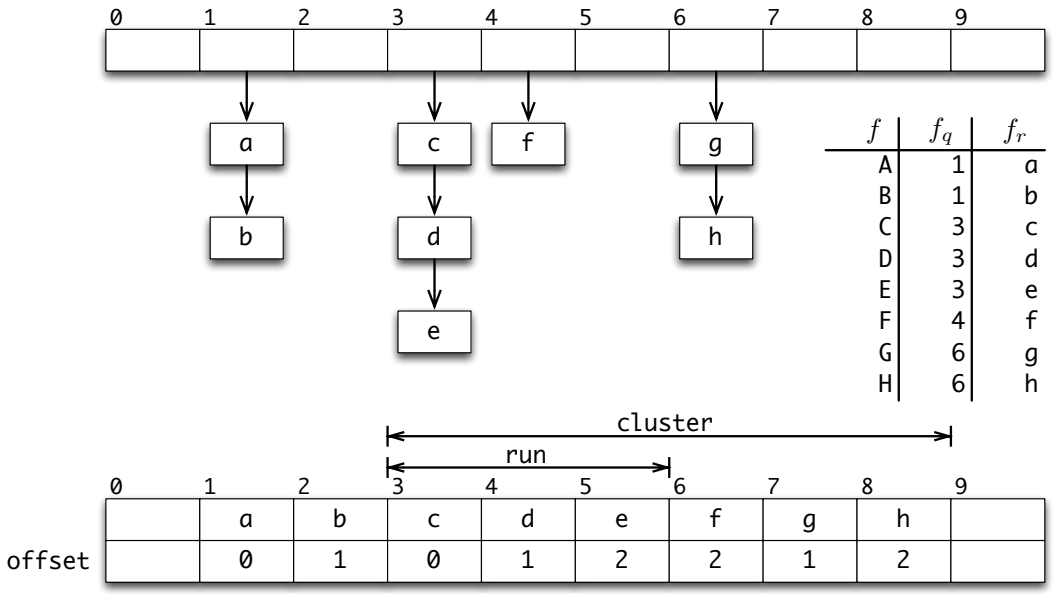


Figure 2: An example quotient filter with 10 slots along with its equivalent open hash table representation. The remainder, f_r, of a fingerprint f is stored in the bucket specified by its quotient, f_q. The quotient filter stores the contents of each bucket in contiguous slots, shifting elements as necessary. We also store an offset to identify the actual quotient of a remainder even after shifting.

other encodings is left as future work.

Theorem 3.11. *The space usage of QF is* $O(n \lg(1/\varepsilon))$.

Proof. Recall that $q = \lg(cn)$ and $r = \lg(1/\varepsilon)$. The QF comprises an array of size 2^q. Each position in the array stores $r + b$ bits, where b is the number of bits used to encode the offsets. As such, the space usage of the QF is

$$\begin{aligned} 2^q(r+o) &= cn(\lg(1/\varepsilon) + o) \\ &= O(n \lg(1/\varepsilon)) + O(n) \end{aligned}$$

□

Theorem 3.12. *The false positive rate of the QF is at most* $\varepsilon = 2^{-r}$.

Proof. Since the QF is just a compact representation of F, its false positive rate is a function of the hash function, h, and the number of items, n, inserted into the filter. In particular, a false positive happens when an element $x' \notin S$ has the same fingerprint as an element $x \in S$ (i.e., $h(x) = h(x')$). We refer to this event as a ***hard collision***.

Assuming h generates outputs uniformly and independently distributed in $\{0, \ldots, 2^p - 1\}$, the probability of a hard collision after a single element has been inserted is clearly $1/2^p$. Conversely, the probability of not having a hard collision in this case is $1 - 1/2^p$.

After $n = \alpha 2^q$ elements have been inserted, the probability of not having a hard collision is

$$\left(1 - \frac{1}{2^p}\right)^n \approx e^{-n/2^p}.$$

Finally, the probability of having a hard collision is given by

$$\begin{aligned} 1 - e^{-n/2^p} &\leq \frac{n}{2^p} \\ &\leq \frac{2^q}{2^p} \\ &= 2^{-r}. \end{aligned}$$

□

Figure 1 shows the false positive rate (on a log scale) for QF as a function of the bits per element. In the figure, α is the load factor of the QF (i.e., the fraction n/m of occupied slots). Figure 1 also shows the false-positive rate for the BF and for a QF variant, described later, that uses only two meta-data bits per slot.

A QF for p-bit fingerprints can have at most 2^p slots, so the size of the fingerprint must be selected in advance based on the number of elements expected to be inserted in the filter and a desired false positive rate. A BF has a similar

limitation—it cannot be expanded to accommodate new items—so sufficient space for all the elements must be allocated in advance. In either case, if the system implementor chooses too small a fingerprint or too small a BF, then the false positive rate will become unacceptably large and the data structure will have to be rebuilt from scratch with better parameters. If the implementor chooses too large of a fingerprint or too large of a BF, then false positive rates will be acceptable, but space will be wasted. Here, however, QF has a significant space advantage. For example, if the implementor overestimates the number of items by a factor of two, then the BF will consume twice the necessary space. The QF, on the other hand, will only consume 1 or 2 extra bits per element which, for typical parameters, will be about 10-20% too much space.

We define a ***cluster*** as a sequence of one or more consecutive runs (with no empty slots in between). A cluster is always immediately preceded by an empty slot and its first item is always un-shifted (i.e., its offset is equal to 0); see Figure 2, bottom.

The time required to perform a lookup, insert, or delete in a QF is dominated by the time to scan backwards and forwards. One such operation need only scan through one cluster. Therefore, we can bound the cost by bounding the size of clusters. The following theorem can be proved by a straightforward application of Chernoff Bounds.

Theorem 3.13. *Let $\alpha \in [0, 1)$. Suppose there are αm items in a quotient filter with m slots. Let*

$$k = (1 + \varepsilon)\frac{\ln m}{\alpha - \ln \alpha - 1}.$$

Then

$$\Pr\left[\textit{there exists a cluster of length} \geq k\right] < m^{-\varepsilon} \xrightarrow{m\to\infty} 0.$$

Proof. Let k be an arbitrary positive integer. We argue that if there is a cluster

spanning slots $i, \ldots, i + k - 1$, then there must be k items that hash into some contiguous region of k slots. Without loss of generality, suppose i is the start of the cluster. Since all items are stored at or after their canonical location, every item stored in slots $i, \ldots, i + k - 1$ must hash into the range $i, \ldots, i + k - 1$. This establishes that some range of k slots has k items that hash into it.

We use Chernoff bounds to limit the probability that k items all hash into the range $i, \ldots, i + k - 1$. Let

$$X_{ij} = \begin{cases} 1 & \text{if item } j \text{ hashes into one of slots } i, \ldots, i + k - 1 \\ 0 & \text{otherwise} \end{cases}$$

and let $X_i = \sum_{j=1}^{m} X_{ij}$. Then $E[X_i] = \alpha m \frac{k}{m} = \alpha k = \mu$.

Recall the multiplicative form of the Chernoff bound: if X is a sum of independent 0/1 random variables with $E[X] = \mu$, then for any $\delta > 0$,

$$\Pr[X > (1 + \delta)\mu] < \left[\frac{e^{\delta}}{(1 + \delta)^{1+\delta}} \right]^{\mu}.$$

Taking $1 + \delta = 1/\alpha$ in this formula yields that

$$\begin{aligned} \Pr[X_i > k] &< \left[\frac{e^{1/\alpha - 1}}{(1/\alpha)^{1/\alpha}} \right]^{\alpha k} \\ &= \frac{e^{k - \alpha k}}{(1/\alpha)^k} \\ &= (\alpha e^{1-\alpha})^k. \end{aligned}$$

Thus, for any particular i,

$$\Pr[X_i > k] < (\alpha e^{1-\alpha})^k.$$

Now observe that

$$\begin{aligned}\Pr[\exists i \text{ s.t. } X_i > k] &< \sum_{i=0}^{m} \Pr[X_i > k] \\ &< m(\alpha e^{1-\alpha})^k.\end{aligned}$$

Thus we need to solve for k such that

$$m(\alpha e^{1-\alpha})^k \xrightarrow{m\to\infty} 0.$$

Taking logs and simple algebra show that if

$$k = (1+\varepsilon)\frac{\ln m}{\alpha - \ln \alpha - 1},$$

then

$$m(\alpha e^{1-\alpha})^k = m^{-\varepsilon} \xrightarrow{m\to\infty} 0.$$

□

For example, with $q = 40$ ($m = 2^{40}$) and $\alpha = 3/4$, the largest cluster in the QF has approximately 736 slots. On average, clusters are $O(1)$ in size. The expected length of a cluster is less than $1/(1 - \alpha e^{1-\alpha})$. For example, with $\alpha = 3/4$, the average cluster length is 27. Figure 3 shows the distribution of cluster sizes for three choices of α. With $\alpha = 1/2$, 99% of the clusters have less than 24 elements.

We have shown that QF offers space and false-positive performance that is comparable to BF, but QF has several significant advantages, as described next.

Cache friendliness. QF lookups, inserts, and deletes require decoding and possibly modifying a single cluster. Since clusters are small, these slots usually fit in

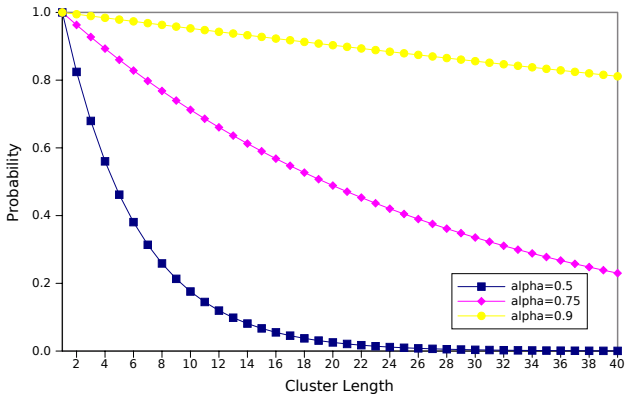


Figure 3: Distribution of cluster sizes for 3 choices of α.

one or two cache lines. On SSD, they usually fit in one disk page, which can be accessed with a single serial read or write. BF inserts, on the other hand, require writing to k random locations, where k is the number of hash functions used by the filter. Similarly, BF lookups require about two random reads on average for absent elements and k for present elements.

In-order hash traversal. As mentioned before, it is possible to reconstruct the exact multi-set of fingerprints inserted into a QF. Furthermore, the QF supports in-order traversal of these fingerprints using a cache-friendly linear scan of the slots in the QF. These two features enable two other useful operations that are not possible with BF: resizing and merging.

Resizing. Like most hash tables, the QF can be dynamically resized—both expanded and shrunk—as items are added or deleted. Unlike hash tables, however, this can be accomplished without the need of rehashing by simply borrowing/stealing one bit from the remainder into the quotient. This can be implemented by iterating over the array while copying each fingerprint into a newly allocated array.

Merging. Similarly, two or more QF can be merged into a single, larger filter using an algorithm similar to that used in merge sort. The merge uses a sequential scan of the two input filters and sequentially writes to the output filter and hence is cache friendly.

Deletes. The QF supports correct deletes while standard Bloom filters do not. In contrast, Counting Bloom filters [27, 57] support probabilistically correct deletions by replacing each bit in a BF with a 4-bit counter, but this incurs a large space overhead and there is still a probability of error.

3.4 Implementation Details

In this section we give details for one possible implementation of QF, as described in [17, 18]. This particular implementation uses 3 meta-data bits per slot. We also briefly describe potential variations.

We begin by analyzing the minimum number of bits per element required to encode the offsets in a particular cluster.

Theorem 3.14. *Any encoding of the offsets for a cluster of size n requires $\Omega(2n)$ bits.*

Proof. The proof is by an information theoretic argument.

First, recall that a cluster is defined as as a sequence of slots with no empty slots in between. Let m be the total number of different possible offset combinations in a cluster of size n. An alternate formulation is to count the total number of ways to place n balls into n bins, with the additional restriction that the number of balls stored in the first k bins is greater or equal than k. Formally, let $x_1, \ldots, x_n$ be non-negative integers representing the number of balls stored in bins $1, \ldots, n$,

respectively. Then m is the number of solutions to the equation $x_1 + \cdots + x_n = n$ such that, for all k ($1 \leq k \leq n$),

$$\sum_{i=1}^{k} x_i \geq k.$$

Using the "stars and bars" method, we consider a string of n stars, representing the balls, and a string of $n-1$ bars, representing separators between two contiguous bins. Then all possible offset combinations can be represented by merging these two strings, and the problem reduces to counting the possible number of ways to merge them. We model the additional constraint by requiring that no initial segment of the merged string has more bars than stars.

Now, if we place an additional bar at the end of all merged strings, this is equivalent to counting the number of Dyck words [39] of length $2n$—strings consisting of n X's and n Y's such that no initial segment of the string has more Y's than X's—which is $C_n = \frac{1}{n+1}\binom{2n}{n}$, the nth Catalan number.

Thus, $m = C_n$ and the total number of bits required to differentiate all m possible offset combinations is $\lg m = 2n + o(n)$, by Stirling's Approximation. □

Our QF implementation is similar in spirit to Cleary Tables [41], a compact representation of hash tables based on quotienting and a bi-directional linear probing scheme utilizing 8 additional meta-data bits per slot—three control bits and five counter bits. Here, each slot in A stores an r-bit remainder along with three meta-data bits, which enable perfect reconstruction of the open hash table; see Figure 4, bottom.

The three meta-data bits in each slot of A work as follows. For each slot i, we maintain an *is-occupied* bit to quickly check whether there exists a fingerprint $f \in F$ such that $f_q = i$. For a remainder f_r stored in slot i, we record whether f_r

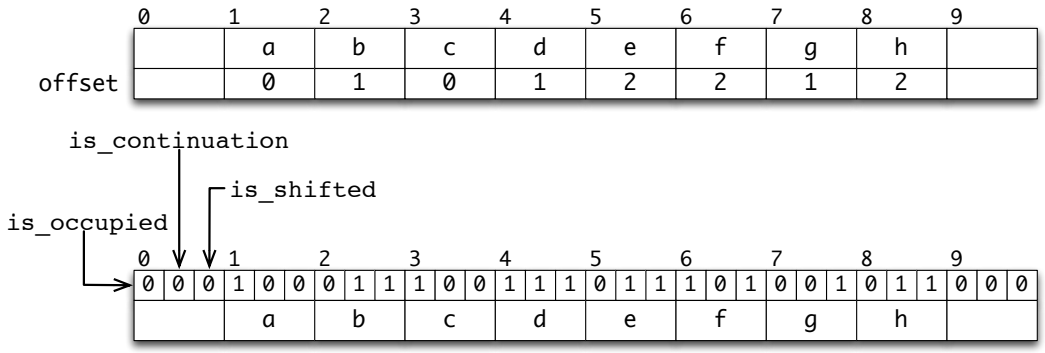


Figure 4: An example quotient filter with 10 slots along with its equivalent representation using three meta-data bits to encode the offsets. Note that this is the same quotient filter as the one in Figure 2

belongs to bucket i (i.e., $f_q = i$) with an $is\text{-}shifted$ bit. Finally, for a remainder f_r stored in slot i, we keep track of whether f_r belongs to the same run as the remainder stored in slot $i-1$ with an $is\text{-}continuation$ bit. Intuitively, the $is\text{-}shifted$ bit, when it is set to 0, tells the decoder the exact location of a remainder in the open hash table representation, the $is\text{-}continuation$ bit enables the decoder to group items that belong to the same bucket (runs), and the $is\text{-}occupied$ bit lets the decoder identify the correct bucket for a run.

In order to reconstruct the fingerprints stored in a QF, we only need to decode starting from the beginning of a cluster. However, rather than decoding, we can perform all operations in place. Figure 5 shows the algorithm for testing whether a fingerprint f might have been inserted into a QF A.

To insert/delete a fingerprint f, we operate in a similar manner: first we mark/unmark $A[f_q]$ as occupied. Next, we search for f_r using the same algorithm as MAY-CONTAIN to find the slot where it should go. Finally, we insert/remove f_r and shift subsequent items as necessary, while updating the other two meta-data bits. We stop shifting items as soon as we reach an empty slot.

```
MAY-CONTAIN(A, f)
    f_q ← ⌊f/2^r⌋              ▷ quotient
    f_r ← f mod 2^r            ▷ remainder
    if ¬ is-occupied(A[f_q])
       then return FALSE
    ▷ walk back to find the beginning of the cluster
    b ← f_q
    while is-shifted(A[b])
          do DECR(b)
    ▷ walk forward to find the actual start of the run
    s ← b
    while b ≠ f_q
          do ▷ invariant: s points to first slot of bucket b
             ▷ skip all elements in the current run
             repeat INCR(s)
               until ¬ is-continuation(A[s])
             ▷ find the next occupied bucket
             repeat INCR(b)
               until is-occupied(A[b])
    ▷ s now points to the first remainder in bucket f_q
    ▷ search for f_r within the run
    repeat  if A[s] = f_r
               then return TRUE
            INCR(s)
      until ¬ is-continuation(A[s])
    return FALSE
```

Figure 5: Algorithm for checking whether a fingerprint f is present in the QF A.

3.4.1 Quotient Filter Variants

We now give space-saving variations on the QF. The QF decoder maintains two pointers: b, a pointer to the current bucket, and s, a pointer to the current slot. The decoder needs to initialize b and s to correct values in order to begin decoding. That is the purpose of the *is-shifted* bit: if $\neg$ *is-shifted*$(A[i])$, then the decoder can

initialize $b = s = i$. There are other ways to initialize b and s:

- Synchronizers. The QF could store a secondary array, $S[0\,..\,(2^q/\ell) - 1]$, of c-bit items. Entry $S[i]$ would hold the offset between bucket $i\ell$ and the slot holding its first element. The decoder can initialize $b = i\ell$ and $s = i\ell + S[i] \bmod 2^q$ for any i. For example, to lookup an element in bucket f_q, the decoder would choose $i = \lfloor f_q/\ell \rfloor$. As a special case, when $S[i] = 2^c - 1$, the offset between bucket $i\ell$ and the slot holding its first element is greater than or equal to $2^c - 1$. The decoder cannot use such entries to begin decoding—it must walk backwards to find the nearest index i such that $S[i] < 2^c - 1$. Since clusters are small, so are the offsets, so we can choose small c (e.g., 5 or 8). The frequency, ℓ, of synchronizers can trade space for decoding speed. The current system, with *is-shifted* bits, is essentially a special case of this scheme with $c = \ell = 1$. By choosing a large ℓ, the per-slot overhead of the QF can be arbitrarily close to two bits.

- Reserved remainders. We can also reserve a special remainder value, e.g. 0, to indicate that a slot is empty, and decoding can begin at an empty slot i with $b = s = i$. This would require only 2 meta-data bits, but reduces the hash space slightly.

- Sorting tricks. Finally, it is possible to indicate empty slots by ordering elements within each bucket and placing "illegal" unordered sequences of elements in empty regions of the QF. In this way, we can achieve exactly two bits of overhead. Decoding in this version is complex and slower.

3.5 Quotient Filter Extensions

In this section we give two AMQs designed for SSD, the buffered quotient filter and the cascade filter, and a data structure designed to efficiently handle duplicate elements that can also be used as a frequency estimator, the Counting Bloom Filter. All three structures use the QF as a building block. The false positive rates of the first two data structures is exactly the same as that of a single QF storing all of the elements.

3.5.1 Buffered Quotient Filter

The BQF uses one QF as the buffer and another QF on the SSD. When the in-RAM QF becomes full, we sequentially iterate over it and flush elements to disk. The QF serves well as a buffer because of its space efficiency and because it allows the flush to iterate sequentially through its fingerprints and write to SSD. Since elements are stored in sequential order, the writes to SSD will also be sequential. Since each flush may write to every page of the on-disk structure, the amortized cost of inserting an item into a BQF of n items with a cache of size M and a block size of B bytes is $O(\frac{n}{MB})$. The BQF is optimized for lookup performance. Most lookups perform one I/O. As with the buffering approaches from Section 3.2, performance degrades as the filter-to-RAM size increases.

3.5.2 Cascade Filter

The CF is optimized for insertion throughput but offers a tradeoff between lookup and insertion speed.

The overall structure of the CF is loosely based on a data structure called the Cache-Oblivious Lookahead Array (COLA) [15]; see Figure 6. The CF maintains

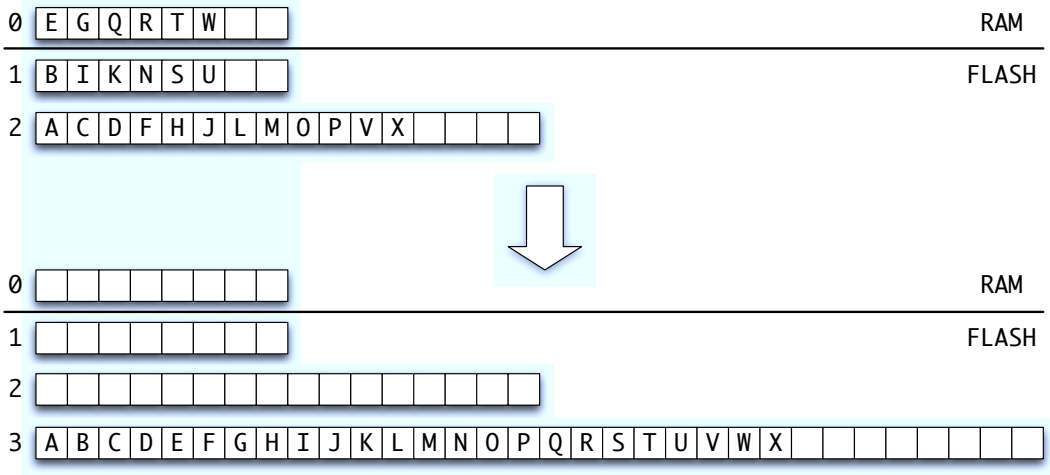


Figure 6: Merging QFs. Three QFs of different sizes are shown above, and they are merged into a single large quotient filter below. The top of the figure shows a CF before a merge, with one QF stored in RAM, and two QFs stored in flash. The three QFs above have all reached their maximum load factors (which is 3/4 in this example). The bottom of the figure shows the same CF after the merge. Now the QF at level 3 is at its maximum load factor, and the QFs at levels 0, 1, and 2 are empty.

an in-memory QF, Q_0. In addition, for RAM of size M, the CF maintains $\ell = \lg(n/M) + O(1)$ in-flash QFs, $Q_1, \ldots, Q_\ell$, of exponentially increasing size. New items are initially inserted into Q_0. When Q_0 reaches its maximum load factor, the CF finds the smallest i such that the elements in $Q_0, \ldots, Q_i$ can be merged into level i. It then creates a new, empty quotient filter Q_i', merges all the elements in $Q_0, \ldots, Q_i$ into Q_i', replaces Q_i by Q_i', and replaces $Q_0, \ldots, Q_{i-1}$ with empty QFs. To perform a CF lookup, we perform a lookup in each nonempty level, which requires fetching one page from each.

It is possible to implement this scheme with different branching factors, b. That is, Q_{i+1} can be b times as large as Q_i. As b increases, the lookup performance increases because there are fewer levels, but the insertion performance decreases because each level may be rewritten multiple times.

The theoretical analysis of CF performance follows from the COLA: a search requires one block read per level, for a total of $O(\lg(n/M))$ block reads; and an insert requires only $O((\lg(n/M))/B)$ amortized block writes/erases, where B is the natural block size of the flash. Typically, $B \gg \lg(n/M)$, meaning the cost of an insertion or deletion is much less than one block write per element. Like a COLA, a CF can be deamortized to provide better worst-case bounds [15]. This deamortization removes delays caused by merging large QFs.

3.5.3 Counting Quotient Filter

One disadvantage of QF is that, if elements are not distributed uniformly, then large clusters can form and, consequently, operations on the QF can become slow. This is a generalized and well-known problem with hash-based containers. If elements inserted into the table are almost always unique, then a good hash function can generate uniformly distributed outputs. The question is how to deal with this issue when some elements are likely to be inserted repeatedly. One possible solution is to check for duplicates whenever an element is inserted, and only allow insertions of new items; however, this would make deletions impossible. Alternatively, the QF can maintain a secondary table, C, of 2^q counters, where entry $C[i]$ indicates the number of times the fingerprint stored in slot i of the QF has been inserted. In this section we analyze this approach.

Interestingly, the CQF can also be used to estimate the number of times that a specific element has been inserted. This problem is normally studied in the area of data stream algorithms under the name frequency estimator. This area of research is characterized by (1) algorithms and data structures whose space is sublinear in both data and input size, (2) the restriction that data can only be seen once or, in some cases, a constant number of times, (3) fast processing time for both updates

and queries, and (4) a small bounded probability of giving a wrong answer, where the factor from which the answer is incorrect is also bounded.

The formal definition of the problem is given below.

Definition 3.15 (Frequency Estimator Problem)**.** *Process a stream of elements $\psi = (x_1, x_2, \ldots)$, where each x_i belongs to a set $S = \{s_1, s_2, \ldots, s_n\}$, so that at any time t (i.e., after the tth element in stream ψ has been processed), we can efficiently estimate the number of occurrences in $\psi_t = (x_1, \ldots, x_t)$ of a given element. Specifically, let $(s_i)_t$ be the actual number of occurrences of element s_i at time t, and $(\widehat{s_i})_t$ the number of occurrences reported by the data structure. Then $(s_i)_t \leq (\widehat{s_i})_t$, and $(\widehat{s_i})_t \leq (s_i)_t + \varepsilon t$ with probability at least $1 - \delta$, where ε and δ are parameters of the data structure.*

Data Structure. Given parameters ε and δ, let $q = \lg(1/\varepsilon)$, $r = \lg(1/\delta)$, and $p = q + r$. The Counting Quotient Filter (CQF) is almost identical to a Quotient Filter (QF) with the exception that each of the 2^q slots store a counter in addition to the remainder and the meta-data bits. The counter keeps track of how many times has the remainder currently stored in that slot been inserted.

When an element x_i from stream ψ arrives we compute a uniformly random p-bit hash $h(x_i)$. Next we partition $h(x_i)$ into its r least significant bits $h_r(x_i) = h(x_i) \bmod 2^r$ (the remainder), and its q most significant bits $h_q(x_i) = \lfloor h(x_i)/2^r \rfloor$ (the quotient). As in QF, we then store $h_r(x_i)$ in bucket $h_q(x_i)$. If this is the first time that $h(x_i)$ is inserted, we initialize its counter to zero. Otherwise, we increment the associated counter by 1.

To compute the estimated number of occurrences of a given element s_i we search the QF for the corresponding hash, $h(s_i)$, and return its associated counter.

Theorem 3.16. *At any time t the estimate $(\widehat{s_i})_t$ reported by CQF is such that*

$$(s_i)_t \leq (\widehat{s_i})_t, \text{ and} \tag{1}$$

$$(\widehat{s_i})_t \leq (s_i)_t + \varepsilon t \text{ with probability at least } 1 - \delta. \tag{2}$$

Proof. First we define some notation. Given two elements $s_i, s_j \in S$, let the indicator random variable $I_{i,j}$ be such that Given two indices $1 \leq i, j \leq n$ let the indicator random variable $I_{i,j}$ be such that

$$I_{i,j} = \begin{cases} 1 & \text{if } s_i \neq s_j \text{ and } h(s_i) = h(s_j), \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

In other words, $I_{i,j}$ is 1 if and only if the hash value for elements s_i and s_j collide. The expected value of $I_{i,j}$ is given by

$$E(I_{i,j}) = Pr[h(s_i) = h(s_j)] \leq \frac{1}{2^p}.$$

For a given element $s_i \in S$, define the variable

$$X_i = \sum_{j=1}^{n} (I_{i,j}(s_j)_t).$$

That is, the number of occurrences of all elements that collide with s_i. By construction, $(\widehat{s_i})_t = (s_i)_t + X_i$, so clearly Inequality (1) holds.

From the definitions given above and by linearity of expectation we can compute the expected value of X_i as

$$
\begin{aligned}
E(X_i) &= E\left(\sum_{j=1}^{n}\left(I_{i,j}(s_j)_t\right)\right) \\
&\leq \sum_{j=1}^{n}\left((s_j)_t E(I_{i,j})\right) \\
&\leq \sum_{j=1}^{n}\frac{(s_j)_t}{2^p} \\
&= \frac{1}{2^p}\cdot\sum_{j=1}^{n}(s_j)_t \\
&= \frac{t}{2^p}.
\end{aligned}
$$

For Inequality (2), observe that

$$
\begin{aligned}
Pr\left[(\widehat{s_i})_t > (s_i)_t + \varepsilon t\right] &= Pr\left[(\widehat{s_i})_t > (s_i)_t + \frac{t}{2^q}\right] \\
&= Pr\left[(s_i)_t + X_i > (s_i)_t + \frac{t}{2^q}\right] \\
&= Pr\left[X_i > \frac{t}{2^q}\right] \\
&= Pr\left[X_i > 2^r E(X_i)\right].
\end{aligned}
$$

Therefore, by Markov's inequality.

$$
\begin{aligned}
Pr\left[(\widehat{s_i})_t > (s_i)_t + \varepsilon t\right] &\leq \frac{1}{2^r} \\
&= \delta.
\end{aligned}
$$

□

The time for both, processing an element and producing the estimate, is $O(1)$,

just as in QF.

Theorem 3.17. *The space usage of CQF is*

$$\frac{1}{\varepsilon} \lg \frac{1}{\delta} + O\left(\frac{1}{\varepsilon}\right).$$

Proof. Let w be the machine word size, where $2^w \geq \max\{t, n\}$, and c be the number of meta-data bits, depending on the QF implementation. Each of the 2^q slots in CQF has $(r + c + w)$ bits, for a total space usage of

$$\begin{aligned} 2^q(r + c + w) &= \frac{1}{\varepsilon}\left(\lg \frac{1}{\delta} + c + w\right) \\ &= \frac{1}{\varepsilon} \lg \frac{1}{\delta} + \frac{c + w}{\varepsilon} \end{aligned}$$

□

Comparison with Count-Min Sketch. The Count-Min Sketch [43] (CMS) for parameters ε and δ can be seen as a Counting Bloom Filter [57] (CBF) with $k = \lceil \ln(1/\delta) \rceil$ pair-wise independent hash functions and $m = \lceil e/\varepsilon \rceil\, k$ counters, where each counter is as large as a machine word, w. The CBF is sliced into k arrays of size $\lceil e/\varepsilon \rceil$, each with its own hash function.

The CMS provides the same guarantees as CQF. That is, the estimate $(\widehat{s_i})_t$ reported is such that $(s_i)_t \leq (\widehat{s_i})_t$ and $(\widehat{s_i})_t \leq (s_i)_t + \varepsilon t$ with probability at least $1 - \delta$. The time to produce the estimate is $O(k)$, the same as the time to process an element. The space usage of CMS is

$$\left\lceil \frac{e}{\varepsilon} \right\rceil \left\lceil \ln \frac{1}{\delta} \right\rceil w = O\left(\frac{1}{\varepsilon} \ln \frac{1}{\delta}\right),$$

which is more than the space usage of CQF for all typical and reasonable values of

ε and δ.

3.6 Evaluation

This section answers the following questions:

1. How does the quotient filter compare to the Bloom filter with respect to in-RAM performance?

2. How do the cascade filter and buffered quotient filter compare to various Bloom filter alternatives on Flash?

3. How does the on-disk performance of the cascade filter and buffered quotient filter change as the database scales out of RAM?

4. How do the different data structures compare on lookup performance? We investigate the performance of both successful lookups and uniform random lookups (which are almost all unsuccessful).

5. What is the insert/lookup tradeoff for the cascade filter with varying fan-outs?

This section comprises three parts. In the first part, we compare the QF and the BF in RAM. We compare the two data structures for three different false positive rates: $1/64 \approx 1\%$, $1/512 \approx 0.2\%$, and $1/4096 \approx 0.02\%$. In the second part, we measure the on-disk performance of the CF, the BQF, the EBF, the BBF and the FBF. Here, we perform experiments with the RAM-to-database size ratios of $1:4$ and $1:24$, which we call small and large experiments, respectively. In the third part, we measure the performance tradeoffs between the insertion and the lookup performance when varying the fanout of the CF. We report results for fan-outs of 2, 4, and 16.

In all experiments, we measure three performance aspects:

Uniform random inserts: Keys are selected uniformly from a large universe.

Uniform random lookups: Keys are selected as before. When performed on an optimally filled AMQ data structure, such queries will report true with probability equal to that of our false positive rate.

Successful lookups: Keys are chosen uniformly at random from one of the keys actually present.

We use an interleaved workload. Every 5% of completed insertions, we spend 60 seconds performing uniform random lookups, followed by 60 seconds performing successful lookups. This way, we can measure the lookup performance at different points of data structure occupancy.

Experimental Setup. We created C++ implementations of all the data structures evaluated in these experiments. Our BF, EBF, BBF, and FBF implementations always uses the optimal number of hash functions. The BBF "page size" parameter controls the amount of space that will be written when buffered data is flushed to SSD. We configured our BBF to use 256KB pages, which is the erasure block size on our SSDs, as recommended by the BBF authors. The analogous FBF parameter is called the "block size", and we configured our FBF implementation to use 256KB blocks. The FBF "page size" governs the size of reads performed during lookups; our FBF implementation used 4KB pages.

Our benchmarking infrastructure generated a 512-bit hash for each item inserted or queried in the data structure. Each data structure could partition the bits in this hash as it needed. For example, a BF configured to use 12 hash functions, each with a 24-bit output, would use 288 bits of the 512-bit hash and discard the rest. We chose 512-bit hashes because many real-world AMQ applications, such as de-duplication services, use cryptographic hashes, such as SHA-512.

We ran our experiments on two identically configured machines, running Ubuntu 10.04.2 LTS. Each machine includes a single-socket Intel Xeon X5650 (6 cores with 1 hyperthread on each core, 2.66GHz, 12MB L2 cache). The machines have 64GB of RAM; to test the out-of-RAM performance, we booted them with 3GB each.

Each machine has a 146.2GB 15KRPM SAS disk used as the system disk and a 160GB SATA II 2.5in Intel X25-M Solid State Drive (SSD) used to store the out-of-RAM part of the data. We use only a 95GB partition of the SSD to minimize the SSD FTL firmware interference. We formatted the 95GB partition as an ext4 filesystem and out-of-RAM data was stored in a 80GB file in that filesystem. We used `dd` to zero the file between each experiment. With this configuration, we could perform 3,910 random 1 byte writes per second, 3,200 1 byte random reads per second, sequential reads at 261 MB/s, and sequential writes at 109 MB/s.

To avoid swapping, we set the Linux swappiness to zero and we monitored `vmstat` output to ensure that no swapping occurred.

Each data structure require different number of bits for their fingerprints. In order to measure the performance independent of the time to compute the fingerprints, we always compute a 512-bit hash for each data structure. We can do this because our data structures require the least number of bits overall.

We implemented all data structure in C++. We did our best effort to follow the description given by the author in their respective papers. In some cases we contacted the authors for advice on implementation details. In any case, when in doubt, we always made whichever assumption would give the most advantage to the other data structures

3.6.1 In-RAM Performance: Quotient Filter vs. Bloom Filter

This section presents the experimental comparison of QF to the BF, with varying false positive rates. Both data structures were given 2GB of space in RAM and we tested their performance on three false positive rates: $1/64$, $1/512$, and $1/4096$. In both experiments, we construct the data structures that can fit the maximum number of elements without violating the false positive rate nor the space requirements. We fill the BF to the maximum occupancy. Because the insertion throughput of the QF significantly deteriorates towards maximum occupancy, we let the QF experiment run up to 90% full.

Results. Figure 7 shows the insertion, random lookup, and successful lookup throughputs of the BF and quotient filter.

The quotient filter substantially outperforms the BF on insertions until the quotient filter is 80% full. The BF insertion throughput is independent of its occupancy, but degrades as the false positive rate goes down, since it has to set more bits for each inserted item. The quotient filter insertion throughput is unaffected by the false positive rate, but it gets slower as it becomes full, since clusters become larger.

The quotient filter matches the BF random lookup performance until about 65% occupancy. The quotient filter performance degrades as its occupancy increases because clusters become longer. The BF performance degrades because the density of 1 bits increases, so the lookup algorithm must, on average, check more bits before it can conclude that an element is not present.

The quotient filter significantly outperforms the BF on successful lookups up to about 75% capacity. The BF performance is independent of occupancy since, in all successful lookups, it must check the same number of bit positions. The quotient filter performance degrades as clusters get larger.

FP rate	Capacity BF	QF (90%)
1/64	1.98 billion	1.71 billion
1/512	1.32 billion	1.29 billion
1/4096	991 million	1.03 billion

Table 2: Capacity of the quotient filter and BF data structures used in our in-RAM evaluation. In all cases, the data structures used 2GB of RAM.

Table 2 shows the capacity of the BFs and quotient filters in our experiments. As predicted in Figure 1, the capacities are almost identical, with the quotient filter more efficient for lower false positive rates.

Overall, the quotient filter outperforms the BF until its occupancy reaches about 70%. The quotient filter requires slightly more space for high false positive rates, and less space for lower false positive rates.

3.6.2 On-disk Benchmarks

We evaluate the insert and lookup performance of CFs, BQFs,EBFs, BBFs and FBFs when they are bigger than RAM. To see how performance of various data structures scales as the RAM-to-filter ratio shrinks, we run two experiments, with RAM-to-filter ratios of $1:4$ and $1:24$. The false positive rate in both experiments is fixed to $f = 1/4096 \approx 0.024\%$, which sets the number of hash functions for the EBF, the BBF and the FBF to $k = 12$, $k = 13$ and $k = 14$, respectively.

We refer to the first experiment, which uses a RAM-to-filter ratio of $1:4$, as the *small* experiment. The RAM buffer size is set to 2GB and the size of data structures on disk is roughly 8GB. The remaining 1GB of RAM is left for the operating system (to use partly as page cache). We inserted 3.97 billion elements into each data structure.

The second experiment, using a RAM-to-filter ratio of $1:24$ can be thought

of as a "large" experiment. In this case all data structures employ 2GB of RAM buffer, and a 48GB on-disk data structure. As in the previous experiment, 1GB is set aside for the page cache. In this configuration, the CF and BQF can hold 23 billion elements, and they can insert them in under 35,000 seconds. All the other data structures were too slow to complete the experiment—we present only partial results obtained after inserting elements for 35,000 seconds.

Results. Figures 8 and 9 show the insertion, random lookup, and successful lookup performance obtained in the small and large experiments. The small CF and BQF experiments completed in about 1 hour. The small EBF and BBF experiments took about 10 hours, and the small FBF experiment took about 25 hours to complete. Consequently, Figure 8(a) only shows the throughput of each data structure through the first hour of the small experiment. See Table 1 for the overall throughputs.

In the large experiment, the other three data structures all completed less than 10 percent of the experiment. Figure 9(a) shows their cumulative throughput for the first 35,000 seconds, but Figures 9(b) and 9(c) do not plot their lookup performance, since the data structures were too slow to obtain this data.

There are two main trends to notice in the insertion throughput graphs: (1) the CF and BQF are orders of magnitude faster than the EBF, BBF, and FBF, and (2) the CF scales better than the BQF. In the small experiment, the BQF outperforms the best BF variant by a factor of 5.2, and slightly outperforms the CF. In the large experiment, the CF performs 11 times more insertions than any of the BF variants, and the BQF performs 9 times more insertions than the BF variants.

The BQF outperforms the CF in the small experiment, but the CF outperforms the BQF in the large experiment, which is consistent with our prediction. Recall that an insert into the BQF requires $O(n/M/B)$ writes, and an insert into the CF

requires $(O(\lg(n/M)/B)$ writes. In the small experiment, $n/M \approx 4$, but in the large experiment, $n/M \approx 24$. Hence, the difference between n/M and $\lg(n/M)$ becomes significant and the CF begins to outperform the BQF. As the size of the database grows, the gap should get larger.

The insertion performance graphs also display the effects of each data structure's buffering strategy. For example, the stalls in the BQF performance correspond to flushing of the full in-RAM QF to the on-disk QF. The stalls become longer as the on-disk QF becomes fuller, making insertions into it more CPU-intensive. The stalls in the CF performance correspond to the merges of QFs. The largest stall is in the middle, where all but the in-RAM QFs are being merged into the largest QF in the CF. There are deamortization techniques, which we did not implement, that can remove such long stalls [15]. The EBF stalls during flushes, too, but each flush takes the same amount of time since BF insertion performance is independent of occupancy. The FBF insertion throughput starts high, during the FBF's in-RAM phase, but drops sharply once data begins spilling to disk. Although it appears to outperform the BBF and EBF in Figure 8(a), Table 1 shows that its overall performance is about 5x less than the BBF and EBF.

The EBF, BBF, and FBF were not able to complete the large experiment, so we cannot compare their overall performance, but we can report their performance on the insertions they completed. The FBF had a cumulative throughput of 67,000 insertions/second during the 35,000 second experiment. The BBF performed 44,600 inserts per second, and the EBF completed 53,000 insertions per second. The CF had a cumulative throughput of 728,000 insertions per second.

The lookup performance graphs support three conclusions: (1) the BQF and CF outperform the BF variants, (2) The BQF performs one random read per lookup, and (3) the CF performs between 1 and $\lg(n/M)$ random reads per lookup. For uniform random lookups, the BQF performance is roughly 1.9 times higher than either

the best BF variant or the CF. The CF uniform random lookup performance is comparable to the EBF and BBF performance, and almost 50% higher than the FBF uniform lookup rate. For successful lookups, the BQF performs 1.6 times better than the CF, 2.5 better than the FBF and 10 to 12 times better than the BBF and the EBF. The FBF maintains the most favorable successful lookup performance among the BF variants. The EBF needs to perform $k = 12$ random reads for each successful lookup, which matches with our results. The BBF is slightly more efficient, due to hash localization and OS prefetching (the lookup indices are sorted.)

The CF always outperforms the BF variants, except under one circumstance. The FBF outperforms the CF when the CF has flushed to disk but the FBF is still operating in RAM. Since the FBF in-RAM phase uses a BF, which is slightly more space efficient than a QF, it can buffer more data before its first flush to disk. Hence the FBF outperforms the CF between 20% and 30% occupancy. Once the FBF flushes to disk, though, it becomes much slower than the CF. Also note that when both the CF and the FBF are operating in RAM, the CF is over twice as fast. Similarly, the BQF outperforms the BF variants once the structures have inserted 30% of the data.

The BQF and CF lookup performance curves match our theoretical analysis. The BQF performance is always around $4,000$ lookups/second, consistent with the conclusion that each BQF lookup requires one random read and the empirical measurement that our SSD can perform about $4,000$ random reads/second. The CF performance also matches theoretical predictions. For example, since the total data set size in the large experiment is 24 times larger than RAM, the CF should have between 1 and $4 \approx \lg(24)$ active levels, and hence its lookup throughput should be between 1 and 4 times slower than the disk's random read throughput. Figures 9(b) and 9(c) match this expectation. The slowest points are at about $1,000$ lookups/second, the fastest at $4,000$ lookups/second.

The lookup figures also reveal several other caching and buffering effects. Lookup throughputs for the CF and BQF exhibit a sawtooth pattern: the upside of the curve is due to populating the in-RAM QF, and thus satisfying a larger fraction of lookups in RAM. Throughput peaks right before the in-RAM QF is flushed – at 20%, 40% 60% and 80% in the small experiment. This effect is also more pronounced for successful lookups, since a successful lookup is more likely to stop in RAM. This effect becomes less significant as more data is inserted, since the data in the buffer becomes a smaller fraction of the inserted elements.

The BBF and the EBF uniform random lookup performance mildly decays as the data structures become fuller. This is due to the on-disk BF having more bits set to one as the occupancy of the filter grows. When the data structure is 100% full, the EBF and BBF need to check 2 bits on average. For EBF, this means 2 random reads; for the BBF, it is slightly less than 2 because two bits from the same subfilter (the erase block) may fall into the same read page. This is confirmed by our results, where the BBF slightly outperforms the EBF in lookups, but both are just above half of the random read throughput of the SSD.

3.6.3 Cascade Filter: Insert/Lookup Tradeoff

To investigate the effect of the fanout in the CF, we inserted 12 billion items into CFs with the same basic configuration as before: a 2GB buffer and a false positive rate of $1/4096$. After inserting all 12 billion elements, we performed lookups for 60 seconds. We repeated this experiment with CFs for fan-outs of 2, 4, and 16. Figure 10 shows the tradeoff between insert and lookup performance in these three experiments.

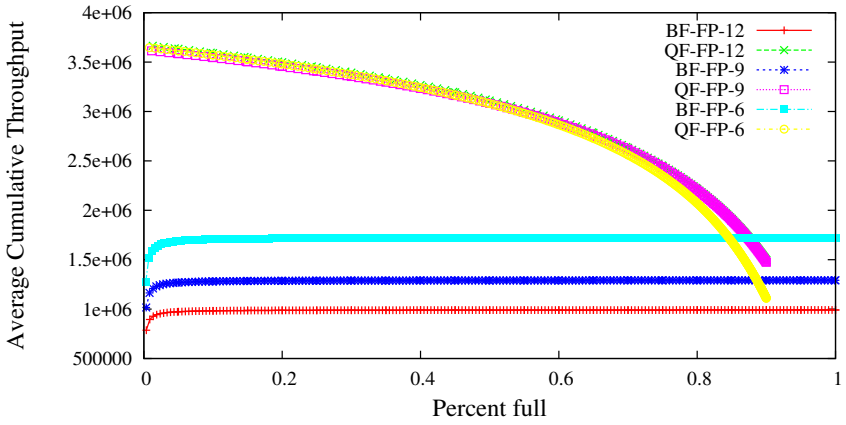
As expected, a higher fanout improves lookup performance, and a lower fanout improves insert performance. High fanouts reduce the number of levels in the CF,

so lookups have fewer levels to check. The drawback of a high fanout is that each level will be written to disk several times, wasting disk bandwidth. According to Figure 10, even a fanout of 16 exceeds the insert performance of all the BF based data structures in our evaluations.

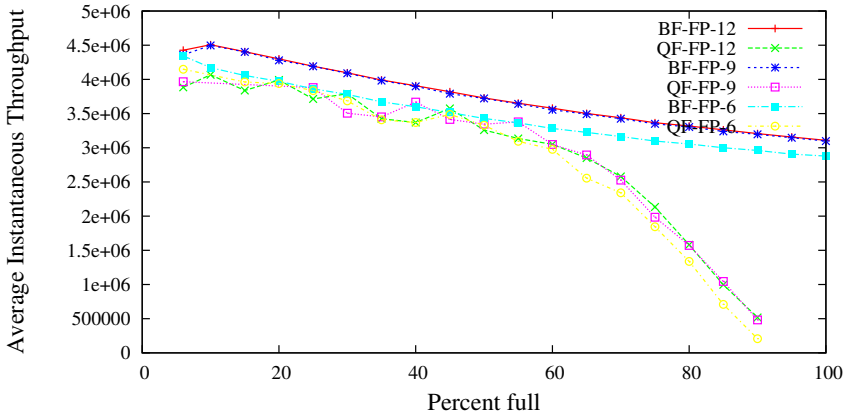
3.6.4 Evaluation Summary

QF-based data structures outperformed BF-based data structures in our evaluation. The QF outperforms the BF, although it uses more space in some configurations. The CF and BQF dramatically outperform all the BF variants. They can perform insertions an order of magnitude faster, and offer comparable or superior lookup performance.

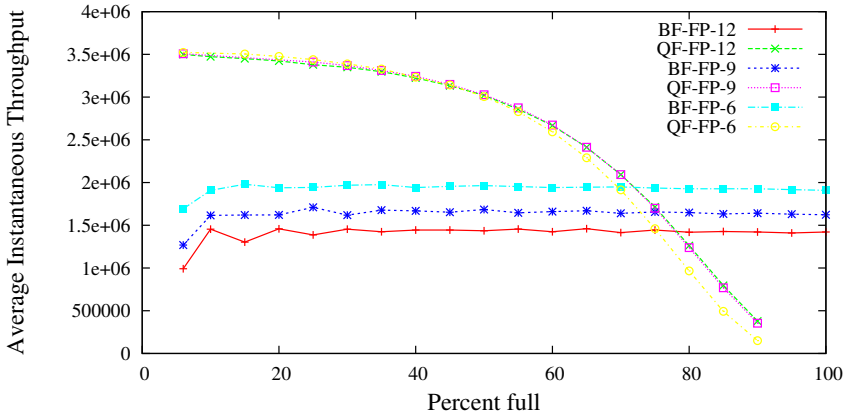
The CF was the most scalable data structure in our experiments. As filter-to-RAM ratio grows, the CF outperforms the BQF. With ratios larger than 24, we expect the CF and the BQF performance to further diverge. When the ratio between the filter and the RAM buffer grows too large, then the flushes that BQF performs become distributed across the large filter, losing some of the space locality.



(a) Cummulative inserts

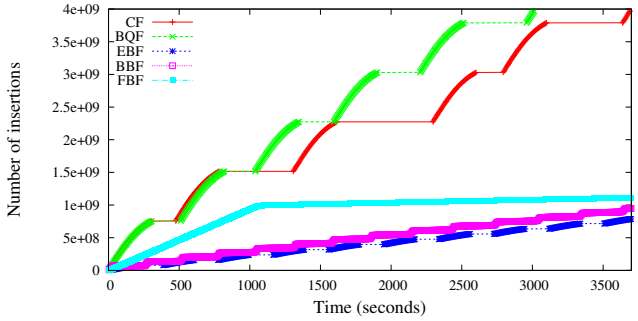


(b) Uniform random lookups

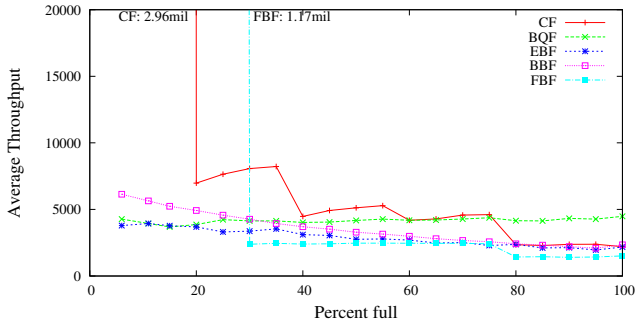


(c) Successful lookups

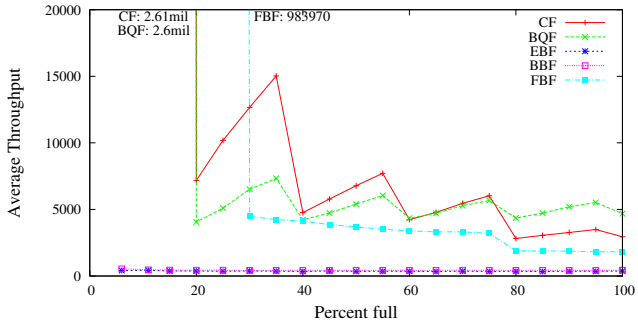
Figure 7: In-RAM Bloom Filter vs. Quotient Filter Performance.



(a) Inserts

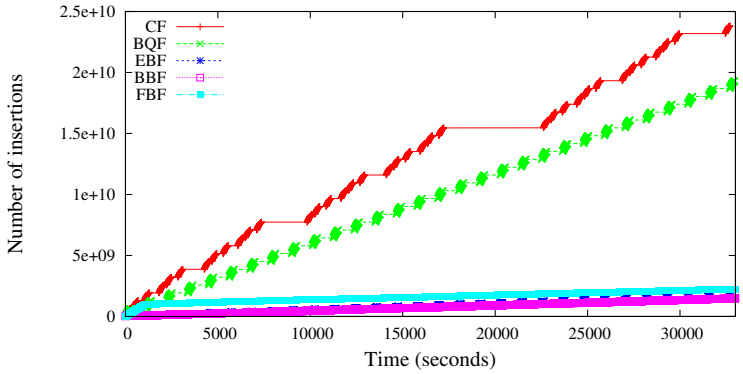


(b) Uniform random lookups

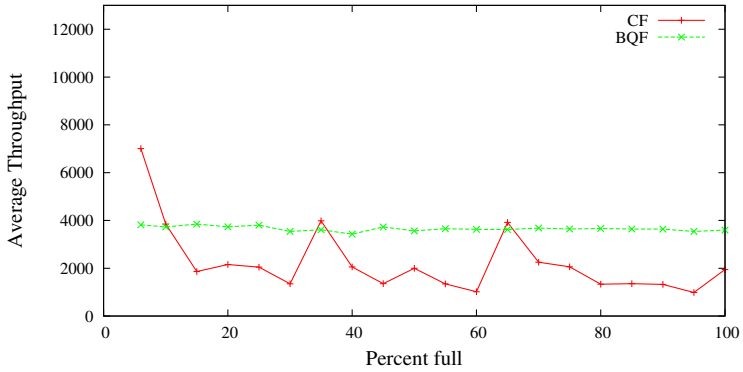


(c) Successful lookups

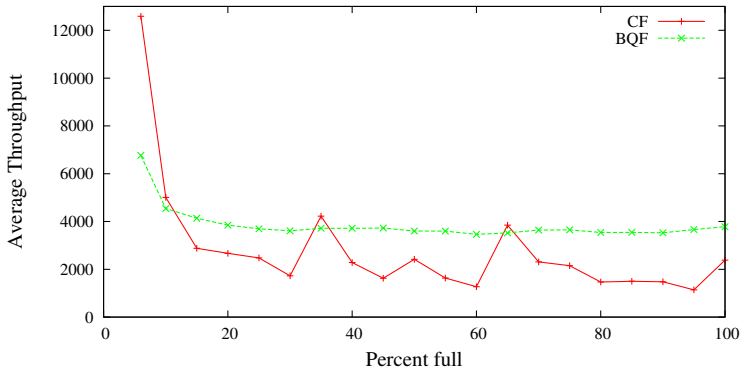
Figure 8: Small disk experiment. In Figure 8(a), the staircase pattern of the CF is due to the merges of the small QFs into a larger quotient filter. The stalls in the BQF performance are due to flushing of the in-RAM quotient filter to the on-disk quotient filter. In Figures 8(b) and 8(c), the lookup performance of the cascade filter depends on the number of full QFs. The BBF and the EBF perform more poorly on the successful lookups, as they need to check 12 bits, performing roughly 12 random reads.



(a) Inserts



(b) Uniform random lookups



(c) Successful lookups

Figure 9: Large disk experiment. In Figure 9(a), the cascade filter outperforms the buffered quotient filter. In Figures 9(b) and 9(c) the cascade filter lookup performance depends on the number of levels it has: at 35 percent and 65 percent, it has only one level, and performs one random read, like buffered quotient filter.

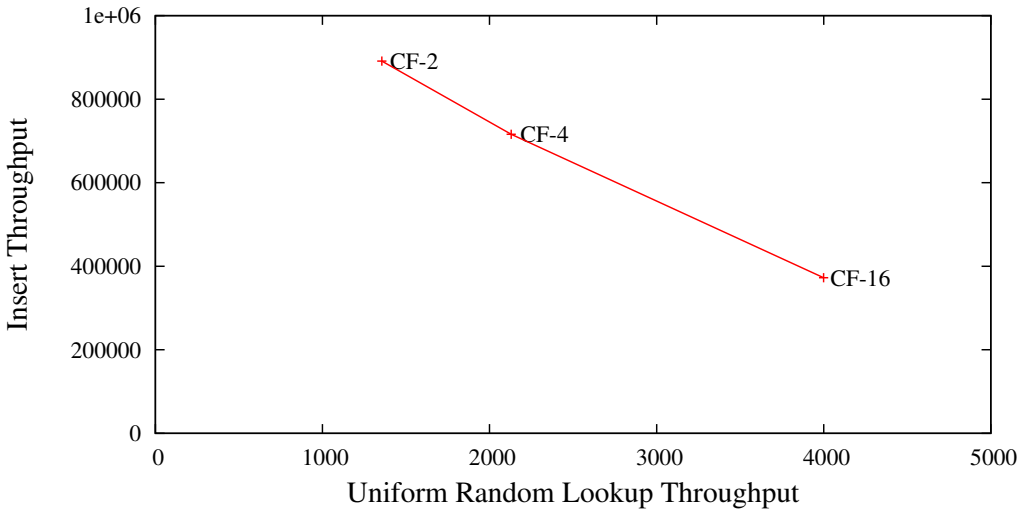


Figure 10: The Cascade Filter Insert/Lookup Tradeoff: Varying fanouts. Higher fanouts foster better lookup performance; lower fanouts optimize the insertion performance.

Chapter 4

Batched Predecessor in External Memory

4.1 Introduction

Buffering is a standard technique for improving the performance of external-memory algorithms. By buffering, partial work on a set of operations can share an I/O, thus reducing the I/O cost per operation. In this chapter we study if buffering queries, so that they are lazily processed in batches, can in fact improve their performance.

The standard ubiquitous data structure to efficiently answer membership queries in external memory is the B-tree [9]. B-trees achieve $O(\log_B n)$ I/Os per query element. It is easy to prove, by a simple information-theoretic argument, that this bound is, indeed, optimal. The natural question then is if it is possible conceive a data structure that performs better when queries are processed in batches.

Let us begin by formally defining the batched membership problem.

Definition 4.18 (Batched Membership Problem)**.** *Represent a static set $S \subset U$ to efficiently answer batched membership queries. That is, given a set $Q = \{q_1, \ldots, q_x\} \subset U$, output a set $C = \{c_1, \ldots, c_x\}$, where $c_i =$* TRUE *if $q_i \in S$ and* FALSE *otherwise. The data structure is created in a preprocessing phase, before any query is performed.*

In this chapter we consider a more general problem—the batched predecessor problem—where, instead of simply returning whether an element is in the set or not, we return the actual element if it is present; if it is not, then we return the element immediately before it in the sorted order. Note that the answer to the batched predecessor problem can be used to give the answer to the batched membership problem in time proportional to the size of the query/answer set: we just have to compare each of the elements of the answer set to its corresponding element in the query set. If both are the same, then we can say that the element is in S; otherwise we say it is not. Hence, we can say that the batched predecessor problem is at least as hard as the batched membership problem.

Definition 4.19 (Batched Predecessor Problem)**.** *Represent a static set $S \subset U$ to efficiently answer batched predecessor queries. That is, given a set $Q = \{q_1, \ldots, q_x\} \subset U$, output a set $A = \{a_1, \ldots, a_x\}$, where $a_i = \max_{s_j \in S} \{s_j \leq q_i\}$. As before, the data structure is created in a preprocessing phase, before any query is performed.*

Our results show that, even in the static setting, buffering does not help. That is, the batched predecessor problem in external memory cannot be solved asymptotically faster than $\Omega(\log_B n)$ I/Os per query element if the preprocessing/space is bounded by a polynomial. We focus on query size $x \leq n^c$, for constant $c < 1$. Thus, the query Q can be large, but is still much smaller than the underlying set S. This query size is interesting because, although there is abundant parallelism in the batched query, common approaches such as linear merges and buffering [6, 28, 32] are suboptimal. On the other hand, Q is too large for the B-tree to be an obvious choice. On the positive side, we show that the problem *can* be solved asymptotically faster, in $\Theta((\log_2 n)/B)$ I/Os, if we impose no constraints on preprocessing/space.

In this chapter we prove lower bounds on the batched predecessor problem in

the comparison-based I/O model [4] We also study tradeoffs between the searching cost and the cost to preprocess the underlying set S.

We assume that S and Q are sorted. Without loss of generality, Q is sorted because the time to sort Q is subsumed by the query time. Similarly, without loss of generality, S is sorted, as long as the preprocessing time is slightly superlinear. We consider sorted S throughout the paper. For notational convenience, we let $s_1 < s_2 < \cdots < s_n$ and $q_1 < q_2 < \cdots < q_x$, and therefore $a_1 \leq a_2 \leq \cdots \leq a_x$.

Given that S and Q are sorted, an alternative interpretation of this chapter is as follows: how can we optimally merge two sorted lists in external memory? Specifically, what is the optimal algorithm for merging two sorted lists in external memory when one list is some polynomial factor smaller than the other?

Observe that the naive linear-scan merging is suboptimal because it takes $\Theta(n/B)$ I/Os, which is greater than the $O(n^c \log_B n)$ I/Os of a B-tree-based solution. Buffer trees [6, 28, 32] also take $\Theta(n/B)$ I/Os during a terminal flush phase. This paper shows that with polynomial preprocessing, performing independent searches for each element in Q is optimal, but it is possible to do better for higher preprocessing.

Finally, we give lower bounds in two other external-memory models: the I/O pointer-machine model, and the indexability model. In the I/O pointer-machine model, we show that with $O(n^{4/3-\varepsilon})$ preprocessing/space for any constant $\varepsilon > 0$, the optimal algorithm cannot perform asymptotically faster than a B-tree. In the indexability model, we exhibit the tradeoff between the redundancy r and access overhead α of the optimal indexing scheme, showing that to report all query answers in $\alpha(x/B)$ I/Os, $\lg r = \Omega((B/\alpha^2)\lg(n/B))$.

4.2 Related work

4.2.1 Single and batched predecessor problems in RAM

In the comparison model, a single predecessor can be found in $\Theta(\lg n)$ time using binary search. The batched predecessor problem is solved in $\Theta(x \lg(n/x) + x)$ by combining merging and binary search [72, 73]. The bounds for both problems remain tight for any preprocessing budget.

Pătraşcu and Thorup [81] give tight lower bounds for single predecessor queries in the cell-probe model. Although batching does not help algorithms that rely on comparisons, Karpinski and Nekrich [69] give an upper bound for this problem in the word-RAM model (bit operations are allowed), which achieves $O(x)$ for all batches of size $x = O(\sqrt{\lg n})$ ($O(1)$ per element amortized) with superpolynomial preprocessing.

4.2.2 Batched predecessor problem in external memory

Dittrich et al. [50] consider multisearch problems where queries are simultaneously processed and satisfied by navigating through large data structures on parallel computers. They give a lower bound of $\Omega(x \log_B(n/x) + x/B)$ under stronger assumptions: no duplicates of nodes are allowed, the ith query has to finish before the $(i+1)$st query starts, and $x < n^{1/(2+\varepsilon)}$, for a constant $\varepsilon > 0$. They do not study the tradeoffs between preprocessing and queries.

Several data structures have been proposed that take advantage of the large block size to buffer insertions in order to amortize their cost [6, 28, 32]. Queries can be similarly buffered and lazily pushed down the tree if the data structure is allowed to answer queries at a later time. However, buffering works well only if the number

of queries is not too small compared to the size of the data structure being queried, or if there are several elements trickling down the same branch of the tree together. Otherwise elements can get stuck in the buffers and flushing them eliminates the benefit of buffering, as each query independently has to at least to finish its walk down a root-to-leaf path in order to find the answer.

Goodrich et al. [59] present a general method for performing x simultaneous external memory searches in $O((n/B + x/B)\log_{M/B}(n/B))$ I/Os when x is large. When x is small, this technique achieves $O(x\log_B(n/B))$ I/Os with a modified version of the parallel fractional cascading technique of Tamassia and Vitter [88].

4.3 Batched Predecessor in the I/O Comparison Model

This section analyzes the batched predecessor problem in the I/O comparison model. First we give the lower bound for the case when preprocessing is unrestricted. Then we study the tradeoff between preprocessing and the optimal number of I/Os.

4.3.1 Lower Bounds for Unrestricted Space/Preprocessing

Now we show two lower bounds for the batched predecessor problem in external memory under the comparison model. The first lower bound is given for pedagogical reasons and is derived using information-theoretic techniques normally used to prove comparison-based lower bounds [4,55]. This first bound, however, is not tight in general. Next we derive a stronger lower bound, assuming assuming unrestricted preprocessing.

Lemma 4.20. *Any algorithm that solves* BATCHEDPRED(Q, S) *requires*

$$\Omega\left(\frac{x}{B}\log_{M/B}\frac{n}{x}+\frac{x}{B}\right)$$

I/Os in the worst case.

Proof. The proof is by an information-theoretic argument. The total number of possible ways to choose the predecessors from S (for the case when they are all distinct for different query elements) is $\binom{n+x}{x}$. One block transfer can reduce the number of candidate interleavings by at most a factor of $\binom{M}{B}$. The lower bound is given by

$$\begin{aligned}\frac{\lg\binom{n+x}{x}}{\lg\binom{M}{B}} &\geq \frac{x\lg(n/x)}{\Theta(B\lg(M/B))}\\ &= \Omega\left(\frac{x}{B}\log_{M/B}\frac{n}{x}\right).\end{aligned}$$

Finally, $\Omega(x/B)$ is a lower bound due to the output size. □

The same lower bound can be derived by relating the number of comparisons needed to solve the batched predecessor problem in RAM with the number of I/Os required to achieve this many comparisons, as described in [7].

Even though the RAM analog of this bound is tight, this lower bound is too weak in external memory. Next we derive a stronger lower bound for this problem, assuming unrestricted preprocessing.

We begin with the definition of a search interval.

Definition 4.21 (***Search interval***)**.** *At step t of an execution, the search interval $S_i^t = [\ell_i^t, r_i^t]$ for an element q_i comprises those elements in S that are still potential values for a_i, given the information that the algorithm has learned so far. When there is no ambiguity, the superscript t is omitted.*

Theorem 4.22 (Lower bound, unrestricted preprocessing)**.** *Let S be a set of size n and Q a set of size $x \leq n^c$ ($0 \leq c < 1$). In the I/O comparison model, computing* BATCHEDPRED(Q, S) *requires*

$$\Omega\left(\frac{x}{B} \lg \frac{n}{xB} + \frac{x}{B}\right)$$

I/Os in the worst-case, regardless of the preprocessing.

Proof. Consider the following problem instance:

1. For all q_i, $|S_i| = n/x$. That is, all elements in Q have been given the first $\lg x$ bits of information about where they belong in S.

2. For all i and j ($1 \leq i \neq j \leq x$), $S_i \cap S_j = \emptyset$. That is, search intervals are disjoint.

We do not charge the algorithm for transferring elements of Q between main memory and disk. This accounting scheme is equivalent to allowing all elements of Q to reside in main memory at all times while still having the entire memory free for other manipulations. Storing Q in main memory does not provide the algorithm with any additional information, since the sorted order of Q is already known.

Now we only consider I/Os of elements in S. Denote a block being input as $b = (b_1, \ldots, b_B)$. Observe that every b_i ($1 \leq i \leq B$) belongs to at most one S_j. The element b_i acts as a **pivot** and helps q_j learn at most one bit of information—by shrinking S_j to its left or its right half.

Since a single pivot gives at most one bit of information, the entire block b can supply at most B bits, during an entire execution of BATCHEDPRED(Q, S).

We require the algorithm to identify the final block in S where each q_i belongs. Thus, the total number of bits that the algorithm needs to learn to solve the problem

BATCHEDPRED(Q, S)

1 Do a linear merge of Q and the ℓth level of T, where $\ell = \lceil \lg x \rceil$.
2 **for** each batch $C_i = \{q_{iB+1}, q_{iB+2}, \ldots, q_{(i+1)B}\}$, set $j = iB + 1$
3 **do** Input the batch C_i.
4 **repeat**
5 Bring in a block $b = (p_j, \ldots, p_{j+B-1})$, where p_k is a median of S_k.
6 Compare each $p_j \in b$ with q_j and adjust S_j accordingly.
7 Construct a B-bit string, β, based on the results of the comparisons.
8 Use β to decide which block to bring in next.
9 **until** the search interval of every element has size at most B.

Figure 11: Batched predecessor algorithm with unlimited space/preprocessing.

is $\Omega(x \lg(n/xB))$. Along with the scan bound to output the answer, the minimum number of block transfers required to solve the problem is $\Omega\left(\frac{x}{B} \lg \frac{n}{xB} + \frac{x}{B}\right)$. □

Next we give a matching algorithm (assuming $B \lg n < M$), which has $O(n^B)$ preprocessing cost. This algorithm is highly impractical given its huge preprocessing costs and space requirements, but serves as an evidence that the lower bound from Theorem 4.22 is tight, unless we pose further restrictions on preprocessing.

Theorem 4.23 (Upper bound, unrestricted preprocessing)**.** *Let S be a set of size n and Q a set of size $x \leq n^c$ ($0 \leq c < 1$). There exists a comparison-based algorithm for* BATCHEDPRED(Q, S) *that performs*

$$\Omega\left(\frac{x}{B} \lg \frac{n}{xB} + \frac{x}{B}\right)$$

I/Os in the worst-case, regardless of the preprocessing.

Proof. In the preprocessing phase, the algorithm constructs a perfectly balanced binary search tree T on S. This step has linear cost. In this phase we also produce all possible blocks of the form $b = (b_1, b_2, \ldots, b_B)$, where each $b_i \in S$ ($1 \leq i \leq B$). In

total, there are $\binom{n}{B}$ such blocks. To produce and output each block, we then require at most $B\binom{n}{B}$ I/Os, which is $O(n^B)$. These blocks are laid out in lexicographical order in external memory. It takes $B \lg n$ bits to address the location of any block.

At a high level, the algorithm processes Q in batches of size B, one batch at a time. A single batch is processed by *simultaneously* performing binary search on all elements of the batch until they find their rank within S.

Figure 11 gives the algorithm in detail. The goal of Step 1 is, for all $q_i \in Q$, to reduce S_i such that $|S_i| = \Theta(n/x)$. The block brought by Step 5 is such that, for each element of the batch, there is one pivot in the block that is a median of its search interval. Step 6 shrinks S_j based on the result of the comparison. The B-bit string constructed in Step 7 represents whether each of the B elements in C_i go into the left/right half of their respective search intervals.

Each of the x/B batches, $C_1, \ldots, C_{x/B}$, is brought once into memory. The cycle in Steps 4–9 takes $\lg(n/xB)$ I/Os until it finds the final block in S where each element in C_i belongs. This gives a total of $(x/B)\lg(n/xB)$ I/Os. Finally, it takes $O(x/B)$ I/Os to output the answer. □

4.3.2 Preprocessing-Searching Tradeoffs

We now give a lower bound on the space required by the batched predecessor problem when the budget for searching is a constraint. The lower bound is posed in terms of a tradeoff where it is clear how the number of I/Os performed in the querying process is affected by restricting the number of blocks generated beforehand.

Definition 4.24. *An I/O containing elements of S is a **j-parallelization I/O** if j distinct elements of Q acquire bits of information during this I/O.*

Theorem 4.25. *Let S be a set of size n and Q a set of size $x \leq n^{1-\varepsilon}$ ($0 < \varepsilon \leq 1$) and let γ be a constant greater than 0. Any algorithm that solves* BATCHEDPRED(Q, S)

in at most

$$\frac{\gamma x \lg n}{j \lg(B/j+1)} + \frac{x}{B}$$

I/Os requires at least

$$\left(\frac{\varepsilon j n^{\varepsilon/2}}{2e\gamma B}\right)^{\varepsilon j/2\gamma}$$

I/Os for preprocessing in the worst case.

Proof. The proof is by a deterministic adversary argument. As before, in the beginning, the adversary partitions S into x equal-sized chunks $C_1, \ldots, C_x$, and places each query element into a separate chunk (i.e., $S_i = C_i$). Now each element knows $\lg x \leq (1-\varepsilon)\lg n$ bits of information. Each element is additionally given half of the number of bits that remain to be learned. This leaves another $T \geq (\varepsilon x \lg n)/2$ total bits yet to be discovered. As in the proof of Theorem 4.22, we do not charge for the inputs of elements in Q, thereby stipulating that all remaining bits to be learned are through the inputs of elements of S.

Lemma 4.26. *To learn T bits in at most*

$$\frac{\gamma x \lg n}{j \lg(B/j+1)}$$

I/Os, there must be at least one I/O in which the algorithm learns at least

$$\frac{j \lg(B/j+1)}{a}$$

bits, where $a = 2\gamma/\varepsilon$.

If multiple I/Os learn at least $(j \lg(B/j+1))/a$ bits, consider the last such I/O during the algorithm execution. Denote the contents of the I/O as $b_i = (p_1, \ldots, p_B)$.

Lemma 4.27. *The maximum number of bits an I/O can learn while parallelizing d elements is $d \lg(B/d + 1)$.*

Proof. Solving the following maximization program, where c_i is the number of pivots dedicated to the ith element parallelized,

$$\max \sum_{i=1}^{d} \lg(c_i + 1) \text{ subject to } \sum_{i=1}^{d} c_i = B,$$

gives that for all i, $c_i = B/d$. □

Lemma 4.28. *The I/O b_i parallelizes at least j/a elements.*

Proof. Given that the most bits an I/O can learn while parallelizing $j/a-1$ elements is

$$\left(\frac{j}{a} - 1\right) \lg\left(\frac{B}{j/a - 1} + 1\right)$$

bits. For all $a \geq 1$ and $j \geq 2$,

$$\frac{j}{a} \lg\left(\frac{B}{j} + 1\right) > \left(\frac{j}{a} - 1\right) \lg\left(\frac{B}{j/a - 1} + 1\right)$$

. Thus, we can conclude that with the block transfer of b_i, the algorithm must have parallelized strictly more than $j/a - 1$ distinct elements. □

We focus our attention on an arbitrarily chosen group of j/a elements parallelized during the transfer of $b_i = \{p_1, \ldots, p_B\}$, which we call $q_1, \ldots, q_{j/a}$.

Observation 4.29. *For every q_u parallelized during the transfer of b_i there is at least one pivot p_v, $1 \leq v \leq B$, such that $p_v \in S_u$.*

Consider the vector $V = (S_1, S_2, \ldots, S_{j/a})$ where S_u denotes the search interval of q_u right before the input of b_i. Each element of Q has acquired at least

$(1-\varepsilon/2)\lg n$ bits, $(\varepsilon \lg n)/2$ of which were given for free after the initial $(1-\varepsilon)\lg n$. For any i, the total number of distinct choices for S_i in the vector V is at least $n^{\varepsilon/2}$, because the element could have been sent to any of these $n^{\varepsilon/2}$-sized ranges in the initial n^{ε} range. Thus, we can observe that

Observation 4.30. *The number of distinct choices for V at the time of parallelization is at least $n^{j\varepsilon/2a}$.*

By a simple adversary argument, we obtain that:

Lemma 4.31. *For each of the $n^{j\varepsilon/2a}$ choices of $V = (S_1, \ldots, S_{j/a})$ (arising from the $n^{\varepsilon/2}$ choices for each S_i), there must exist a block with pivots $p_1, p_2, \ldots, p_{j/a}$, such that $p_k \in S_k$.*

Proof. Assume there exists a vector choice for which there is no block with a pivot from each search interval. The adversary will then make decisions consistent with assigning these search intervals to $q_1, \ldots, q_{j/a}$, and thus avoid parallelization. Note that no natural blocks already contain the combinations of these search intervals, and that such block must be manufactured in the preprocessing phase. □

The same block can serve multiple vector choices, because the block has B elements and we are parallelizing only j/a elements. The next lemma quantifies the maximum number of vectors covered by one block.

Lemma 4.32. *A block can cover at most $\binom{B}{j/a}$ distinct vector choices.*

Proof. Call t_i the number of pivots in the block b that fall in distinct search intervals of the element q_i. Then we are trying to maximize

$$\prod_{i=1}^{j/a} t_i \text{ subject to } \sum_{i=1}^{j/a} t_i = B.$$

This gives that $t_i = aB/j$. By selecting aB/j pivots for each element, we cover $(aB/j)^{j/a}$ distinct vectors. □

As a consequence, the minimum number of blocks the algorithm needs to preprocess is at least

$$\frac{n^{j\varepsilon/2a}}{\binom{B}{j/a}} \geq \left(\frac{n^{\varepsilon/2}}{eaB/j}\right)^{j/a}.$$

Substituting for the value of a, we get that the minimum preprocessing is at least

$$\left(\frac{\varepsilon j n^{\varepsilon/2}}{2e\gamma B}\right)^{\varepsilon j/2\gamma}.$$

□

4.3.2.1 Algorithm

An algorithm that runs in $O((x \lg n)/j \lg(B/j + 1) + x/B)$ I/Os follows an idea similar to the optimal algorithm for unrestricted preprocessing. The difference is that we preprocess $\binom{n}{j}$ blocks, where each block correspond to a distinct combination of some j elements. The block will contain B/j evenly spaced pivots for each element. The searching algorithm uses batches of size j.

4.4 Batched Predecessor in the I/O Pointer-Machine Model

In Section 4.3 we showed that, in the comparison-based I/O model, the batched predecessor problem cannot be solved asymptotically faster than $\Omega(\log_B n)$ I/Os per query element if the preprocessing/space is bounded by a polynomial. A natural

question then is if this is a limitation of the problem itself, or if it is a restriction imposed by the model. In Sections 4.4 and 4.5 we show similar lower bounds in two other external-memory models: the I/O pointer-machine model, and the indexability model. The first model is less restrictive than the comparison I/O model in main memory, but more restrictive in external memory. This is because an algorithm can perform arbitrary computations in RAM, but a disk block can be accessed only via a pointer that has been seen at some point in past. The latter model bounds the number of blocks that an algorithm *must* preprocess/access to report all the query results; the search cost is ignored. The lower bounds in this model also hold in the previous two models.

In this section we focus on the batched predecessor problem in the **I/O pointer-machine model** [87]. This model is a generalization of the pointer machine model introduced by Tarjan [90]. Many results in range reporting have been obtained in this model [2, 3]. In particular, we show that if the preprocessing time is $O(n^{4/3-\varepsilon})$ for any constant $\varepsilon > 0$, then there exists a query set Q of size x such that reporting $\textsc{BatchedPred}(Q, S)$ requires $\Omega(x/B + x \log_B n/x)$ I/Os.

At a high level, in order to show results in the I/O pointer-machine model, we define a graph whose nodes are the blocks on disk of the data structure and whose edges are the pointers between blocks. Since a block has size B, it can contain at most B pointers, and thus the graph is fairly sparse. We show that any such sparse graph has a large set of nodes that are far apart. If the algorithm must visit those well-separated nodes, then it must perform many I/Os. The crux of the proof is that, as the preprocessing increases, the redundancy of the data structure increases, thus making it hard to pin down specific locations of the data structure that must be visited. We show that if the data structure is reasonable in size—in our case $O(n^{4/3-\varepsilon})$—then we can still find a large, well dispersed set of nodes that must be visited, thus establishing the lower bound.

To answer BATCHEDPRED(Q, S), an algorithm preprocesses S and builds a data structure comprised of n^k blocks, where k is a constant to be determined later. We use a directed graph $\mathcal{G} = (V, E)$ to represent the n^k blocks and their associated directed pointers. Every algorithm that answers BATCHEDPRED(Q, S) begins at the start node v_0 in V and at each step picks a directed edge to follow from those seen so far. Thus, the nodes in a computation are all reachable from v_0. Furthermore, each fetched node contains elements from S, and the computation cannot terminate until the visited set of elements is a superset of the answer set A. A node in V contains at most B elements from S and at most B pointers to other nodes.

Let $\mathcal{L}(W)$ be the union of the elements contained in a node set W, and let $\mathcal{N}(a)$ be the set of nodes containing element a. We say that a node set W ***covers*** a set of elements A if $A \subseteq \mathcal{L}(W)$. An algorithm for computing A can be modeled as the union of a set of paths from v_0 to each node in a node set W that covers A.

To prove a lower bound on BATCHEDPRED(Q, S), we show that there is a query set Q whose answer set A requires many I/Os. In other words, for every node set W that covers A, a connected subgraph spanning W contains many nodes. We achieve this result by showing that there is a set A such that, for every pair of nodes $a_1, a_2 \in A$, the distance between $\mathcal{N}(a_1)$ and $\mathcal{N}(a_2)$ is large. That is, all the nodes in $\mathcal{N}(a_1)$ are far from all the nodes in $\mathcal{N}(a_2)$. Since the elements of A can appear in more than one node, we need to guarantee that the node set V of $\mathcal{G}$ is not too large; otherwise the distance between $\mathcal{N}(a_1)$ and $\mathcal{N}(a_2)$ can be very small. For example, if $|V| \geq \binom{n}{2}$, every pair of elements can share a node, and a data structure exists whose minimum pairwise distance between any $\mathcal{N}(a_1)$ and $\mathcal{N}(a_2)$ is 0.

First, we introduce two measures of distance between nodes in any (undirected or directed) graph $G = (V, E)$. Let $d_G(u, v)$ be the length of the shortest (di-)path from node u to node v in G. Furthermore, let $\Lambda_G(u, v) = \min_{w \in V} (d_G(w, u) + d_G(w, v))$. Thus, $\Lambda_G(u, v) = d_G(u, v)$ for undirected graphs,

but not necessarily for directed graphs.

For each $W \subseteq V$, define $f_G(W)$ to be the minimum number of nodes in any connected subgraph H such that (1) the node set of H contains $W \cup \{v_0\}$ and (2) H contains a path from v_0 to each $v \in W$. Observe that $f_G(\{u, v\}) \geq \Lambda_G(u, v)$. The following lemma gives a more general lower bound for $f_G(W)$. It shows that the size of the graph containing nodes of W is linear in the minimum pairwise distance within W.

Lemma 4.33. *For any directed graph $G = (V, E)$ and any $W \subseteq V$ of size $|W| \geq 2$, $f_G(W) \geq r_W|W|/2$, where $r_W = \min_{u,v \in W, u \neq v} \Lambda_G(u, v)$.*

Proof. Consider the undirected version of G, and consider a TSP of the nodes in W. It must have length $r_W|W|$. Any tree that spans W must therefore have size at least $r_W|W|/2$. Finally, $f_G(W)$ contains a tree that spans W. □

Our next goal is to find a query set Q such that every node set W that covers the corresponded answer set A has a large r_W. The answer set A will be an independent set of a certain kind, that we define next. For a directed graph $G = (V, E)$ and an integer $r > 0$, we say that a set of nodes $I \subseteq V$ is ***r-independent*** if $\Lambda_G(u, v) > r$ for all $u, v \in I$ where $u \neq v$. The next lemma guarantees a substantial r-independent set.

Lemma 4.34. *Given a directed graph $G = (V, E)$, where each node has out-degree at most $B \geq 2$, there exists an r-independent set I of size at least*

$$\frac{|V|^2}{|V| + 4r|V|B^r}$$

.

Proof. Construct an undirected graph $H = (U, F)$ such that $U = V$ and $(u, v) \in F$ if and only if $\Lambda_G(u, v) \in [1, r]$. Then, H has at most $2r|V|B^r$ edges. By Turán's

Theorem [89], there exists an independent set of the desired size in H, which corresponds to an r-independent set in G, completing the proof. □

In addition to r-independence, we want the elements in A to occur in few blocks, in order to control the possible choices of the node set W that covers A. We define the ***redundancy*** of an element a to be $|\mathcal{N}(a)|$. Because there are n^k blocks and each block has at most B elements, the average redundancy is $O(n^{k-1}B)$. We say that an element has ***low redundancy*** if its redundancy is at most twice the average. We show that there exists an r-independent set I of size n^{ε} (here ε depends on r) such that no two blocks share the same low-redundancy element. We will then construct our query set Q using this set of low-redundancy elements in this r-independent set.[1]

Finally, we add enough edges to place all occurrences of every low-redundancy element within $\rho < r/2$ of all other occurrences of that element. We show that we can do this by adding few edges to each node, therefore maintaining the sparsity of G. Since this augmented graph also contains a large r-independent set, all the nodes of this set cannot share any low-redundancy elements.

The following lemma shows that nodes sharing low-redundancy elements can be connected with low diameter and small degrees.

Lemma 4.35. *For any $k > 0$ and $m > k$ there exists an undirected k-regular graph H of order m having diameter* $\log_{k-1} m + o(\log_{k-1} m)$.

Proof. In [26], Bollobás shows that a random k-regular graph has the desired diameter with probability close to 1. Thus there exists some graph satisfying the constraints. □

[1] Our construction does not work if the query set contains high redundancy elements, because high redundancy elements might be placed in every block.

Consider two blocks B_1 and B_2 in the r-independent set I above, and let a and b be two low-redundancy elements such that $a \in B_1, b \notin B_1$ and $a \notin B_2, b \in B_2$. Any other pair of blocks B_1' and B_2' that contain a and b respectively must be at least $(r-2\rho)$ apart, since B_i' is at most ρ apart from B_i. By this argument, every node set W that covers A has $r_W \geq (r-2\rho)$. Now, by Lemma 4.33, we get a lower bound of $\Omega((r-2\rho)|W|)$ on the query complexity of Q. We choose $r = c_1 \log_B(n/x)$ and get $\rho = c_2 \log_B(n/x)$ for appropriate constants $c_1 > 2c_2$. This is the part where we require the assumption that $k < 4/3$ as shown in Theorem 4.36, where n^k is the size of the entire data structure. We then apply Lemma 4.34 to obtain that $|W| = \Omega(x)$.

Now we are ready to prove the main theorem of this section.

Theorem 4.36 (Lower bound, I/O pointer-machine model)**.** *Let S be a set of size n. In the I/O pointer-machine model, if* PREPROCESSING(S) *uses $O(n^{4/3-\varepsilon})$ blocks of space and I/Os, for any constant $\varepsilon > 0$, then there exists a constant c and a set Q of size n^c such that computing* BATCHEDPRED(Q, S) *requires $\Omega(x \log_B(n/x) + x/B)$ I/Os.*

Note that in this theorem, c is a function of ε in that, the smaller the preprocessing, the larger the set for which the lower bound can be established.

Proof. We partition S into S_ℓ and S_h by the redundancy of elements in these n^k blocks and claim that there exists $A \subseteq S_\ell$ such that query time for the corresponded Q matches the lower bound.

Let S_ℓ be the set of elements of redundancy no more than $2Bn^k/n$ (i.e., twice of the average redundancy). The rest of elements belong to S_h. By the Markov inequality, we have $|S_\ell| = \Theta(n)$ and $|S_h| \leq n/2$. Let $\mathcal{G} = (V, E)$ represent the connections between the n^k blocks as the above stated. We partition V into V_1 and V_2 such that V_1 is the set of blocks containing some elements in S_ℓ and $V_2 = V \setminus V_1$. Since each block can at most contain B elements in S_ℓ, $|V_1| = \Omega(n/B)$.

Then, we add some additional pointers to $\mathcal{G}$ and obtain a new graph $\mathcal{G}'$ such that, for each $e \in S_\ell$, every pair $u, v \in \mathcal{N}(e)$ has small $\Lambda_{\mathcal{G}'}(u, v)$. We achieve this by, for each $e \in S_\ell$, introducing graph H_e to connect all the n^k blocks containing element e such that the diameter in H_e is small and the degree for each node in H_e is $O(B^\delta)$ for some constant δ. By Lemma 4.35, the diameter of H_e can be as small as

$$\rho \leq \frac{1}{\delta} \log_B |H_e| + o(\log_B |H_e|) \leq \frac{k-1}{\delta} \log_B n + o(\log_B n).$$

We claim that the graph $\mathcal{G}'$ has a $(2\rho + \varepsilon)$-independent set of size n^c, for some constants $\varepsilon, c > 0$. For the purpose, we construct an undirected graph $H(V_1, F)$ such that $(u, v) \in F$ if and only if $\Lambda_{\mathcal{G}'}(u, v) \leq r$. Since the degree of each node in $\mathcal{G}'$ is bounded by $O(B^{\delta+1})$, by Lemma 4.34, there exists an r-independent set I of size

$$|I| \geq \frac{|V_1|^2}{|V_1| + 4r|V|O(B^{r(\delta+1)})} \geq \frac{n^{2-k}}{4rO(B^{r(\delta+1)+2})} = n^c.$$

Then, $r = ((2 - k - c) \log_B n)/(\delta + 1) + o(\log_B n)$. To satisfy the condition made in the claim, let $r > 2\rho$. Hence, $(2 - k - c)/(\delta + 1) > 2(k - 1)/\delta$. Then, $k \to 4/3$ for sufficiently large δ. Observe that, for each $e \in S_\ell$, e is contained in at most one node in I. In addition, for every pair $e_1, e_2 \in S_\ell$ where e_1, e_2 are contained in separated nodes in I, then $\Lambda_{\mathcal{G}'}(u, v) \geq \varepsilon$ for any $u \ni e_1, v \ni e_2$. By Lemma 4.33, we are done. □

4.5 Batched Predecessor in the Indexability Model

This section analyzes the batched predecessor problem in the indexability model [63, 64]. This model is frequently used to analyze reporting problems by focusing on bounding the number of blocks that an algorithm must access to report all the query results. Lower bounds on queries are obtained solely based on how

many blocks were preprocessed. The search cost is ignored—the blocks containing the answers are given to the algorithm for free. Here, the ***redundancy parameter r*** measures the number of times an element is stored in the data structure, and the ***access overhead parameter α*** captures how far the reporting cost is from the optimal.

We show that to report all query answers in $\alpha(x/B)$ I/Os, $r = (n/B)^{\Omega(B/\alpha^2)}$. This result shows that it is impossible to obtain $O(1/B)$ per element unless the space used by the data structure is exponential. This corresponds to the situation in RAM, where exponential preprocessing is required to achieve $O(1)$ amortized time per query element [69].

A ***workload*** is given by a pair $\mathcal{W} = (S, \mathcal{A})$, where S is the set of n input objects, and $\mathcal{A}$ is a set of subsets of S—the output to the queries. An ***indexing scheme*** $\mathcal{I}$ for a given workload $\mathcal{W}$ is given by a collection $\mathcal{B}$ of B-sized subsets of S such that $S = \cup\mathcal{B}$; each $b \in \mathcal{B}$ is called a block.

An indexing scheme has two parameters associated with it. The first parameter, called the ***redundancy***, represents the average number of times an element is replicated (i.e., an indexing scheme with redundancy r uses $r\lceil n/B \rceil$ blocks). The second parameter is called the ***access overhead***. Given a query with answer A, the query time is $\min\{|\mathcal{B}'| : \mathcal{B}' \subseteq \mathcal{B}, A \subseteq \cup\mathcal{B}'\}$, because this is the minimum number of blocks that contain all the answers to the query. If the size of A is x, then the best indexing scheme would require a query time of $\lceil x/B \rceil$. The access overhead of an indexing scheme is the factor by which it is suboptimal. An indexing scheme with access overhead α uses $\alpha\lceil x/B \rceil$ I/Os to answer a query of size x in the worst case.

To show the tradeoff between α and r, we use the Redundancy Theorem (stated below for completeness) from [63, 84]:

Theorem 4.37 (Redundancy Theorem [63,84])**.** *For a workload $\mathcal{W} = (S, \mathcal{A})$ where*

$\mathcal{A} = \{A_1, \cdots, A_m\}$, let $\mathcal{I}$ be an indexing scheme with access overhead $\alpha \leq \sqrt{B}/4$ such that for any $1 \leq i, j \leq m$, $i \neq j$, $|A_i| \geq B/2$ and $|A_i \cap A_j| \leq B/(16\alpha^2)$. Then the redundancy of $\mathcal{I}$ is bounded by $r \geq \frac{1}{12n} \sum_{i=1}^{m} |A_i|$.

Next we prove the main theorem for this section.

Next we prove Theorem 4.38.

Theorem 4.38 (**$r - \alpha$** tradeoff, indexability model)**.** *In the indexability model, any indexing scheme for the batched predecessor problem with access overhead $\alpha \leq \sqrt{B}/4$ has redundancy r satisfying $\lg r = \Omega\left(B \lg(n/B)/\alpha^2\right)$.*

Proof. For the sake of the lower bound, we restrict to queries where all the reported predecessors reported are distinct. To use the redundancy theorem, we want to create as many queries as possible.

Call a family of k-element subsets of S β-sparse if any two members of the family intersect in less than β elements. The size $C(n, k, \beta)$ of a maximal β-sparse family is crucial to our analysis. For a fixed k and β this was conjectured to be asymptotically equal to $\binom{n}{\beta}/\binom{k}{\beta}$ by Erdös and Hanani and later proven by Rödl in [83]. Thus, for large enough n, $C(n, k, \beta) = \Omega(\binom{n}{\beta}/\binom{k}{\beta})$.

We now pick a $(B/2)$-element, $B/(16\alpha^2)$-sparse family of S, where α is the access overhead of $\mathcal{I}$. The result in [83] gives us that

$$C\left(n, \frac{B}{2}, \frac{B}{16\alpha^2}\right) = \Omega\left(\binom{n}{B/\left(16\alpha^2\right)}/\binom{B/2}{B/\left(16\alpha^2\right)}\right).$$

Thus, there are at least $(2n/eB)^{B/(16\alpha^2)}$ subsets of size $B/2$ such that any pair intersects in at most $B/(16\alpha^2)$ elements. The Redundancy Theorem then implies that the redundancy r is greater than or equal to $(n/B)^{\Omega(B/\alpha^2)}$, completing the proof. □

We now describe an indexing scheme that is off from the lower bound by a factor α.

Theorem 4.39 (Indexing scheme for the batched predecessor problem)**.** *Given any $\alpha \leq \sqrt{B}$, there exists an indexing scheme $\mathcal{I}_\alpha$ for the batched predecessor problem with access overhead α^2 and redundancy $r = O((n/B)^{B/\alpha^2})$*

Proof. Call a family of k-element subsets of S β-dense if any subset of S of size β is contained in at least one member from this family. Let $c(n, k, \beta)$ denote the minimum number of elements of such a β-dense family. Rödl [83] proves that for a fixed k and β,

$$\lim_{n\to\infty} c(n, k, \beta)\tbinom{k}{\beta}\tbinom{n}{\beta}^{-1} = 1,$$

and thus, for large enough n, $c(n, k, \beta) = O(\tbinom{n}{\beta}/\tbinom{k}{\beta})$.

The indexing scheme $\mathcal{I}_\alpha$ consists of all sets in a B-element, (B/α^2)-dense family. By the above, the size of $\mathcal{I}_\alpha$ is $O((n/B)^{B/\alpha^2})$.

Given a query answer $A = \{a_1, \cdots, a_x\}$ of size x, fix $1 \leq i < \lceil x/B \rceil$ and consider the B-element sets $C_i = \{a_{(i-1)B}, \cdots, a_{iB}\}$ ($C_{\lceil x/B \rceil}$ may have less than B elements). Since $\mathcal{I}_\alpha$ is an indexing scheme, we are told all the blocks in $\mathcal{I}_\alpha$ that contain the a_is. By construction, there exists a block in $\mathcal{I}_\alpha$ that contains a $1/\alpha^2$ fraction of C_i. In at most α^2 I/Os we can output C_i, by reporting B/α^2 elements in every I/O. The number of I/Os needed to answer the entire answer A is thus $\alpha^2\lceil x/B \rceil$, which proves the theorem. □

Bibliography

[1] Data rEvolution. Technical report, CSC Leading Edge Forum, 2011. `http://www.csc.com/innovation/ds/84818-data_revolution`.

[2] P. Afshani, L. Arge, and K. D. Larsen. Orthogonal range reporting: Query lower bounds, optimal structures in 3-d, and higher-dimensional improvements. In *26th Annual Symposium on Computational Geometry (SoCG)*, pages 240–246, 2010.

[3] P. Afshani, L. Arge, and K. G. Larsen. Higher-dimensional orthogonal range reporting and rectangle stabbing in the pointer machine model. In *28th Annual Symposium on Computational Geometry (SoCG)*, pages 323–332, 2012.

[4] A. Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, 1988.

[5] A. Andersson and T. W. Lai. Fast updating of well-balanced trees. In *2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 111–121, 1990.

[6] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[7] L. Arge, M. Knudsen, and K. Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In *Workshop on Algorithms and Data Structures (WADS)*, pages 83–94, 1993.

[8] P. Backhouse. Drowning in data? *Digital archaeology: bridging method and theory*, page 50, 2006.

[9] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. ACM.

[10] M. A. Bender, R. Cole, E. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *10th European Symposium on Algorithms (ESA)*, pages 152–164, 2002.

[11] M. A. Bender, R. Cole, E. D. Demaine, and M. Farach-Colton. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In *10th Annual European Symposium on Algorithms (ESA)*, volume 2461 of *Lecture Notes in Computer Science*, pages 139–151, 2002.

[12] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 399–409, 2000.

[13] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[14] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. *Journal of Algorithms*, 3(2):115–136, 2004.

[15] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 81–92, 2007.

[16] M. A. Bender, M. Farach-Colton, M. Goswami, D. Medjedovic, P. Montes, and M.-T. Tsai. The batched predecessor problem in external memory. In *Proceedings of the 22nd European Symposium on Algorithms (ESA)*, Wroclaw, Poland, September 2014.

[17] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. *Proceedings of the VLDB Endowment*, 5(11):1627–1637, August 2012.

[18] M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, Portland, Oregon, USA, June 2011.

[19] M. A. Bender, M. Farach-Colton, and B. Kuszmaul. Cache-oblivious string B-trees. In *25th Symposium on Principles of Database Systems (PODS)*, pages 233–242, 2006.

[20] M. A. Bender, M. Farach-Colton, and M. A. Mosteiro. Insertion sort is $O(n \log n)$. *Theory of Computing Systems*, 39(3):391–397, 2006.

[21] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent cache-oblivious B-trees. In *17th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 228–237, 2005.

[22] M. A. Bender and H. Hu. An adaptive packed-memory array. In *25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 20–29, 2006.

[23] M. A. Bender and H. Hu. An adaptive packed-memory array. *ACM Trans. Database Syst.*, 32(4), 2007.

[24] D. K. Blandford. *Compact Data Structures with Fast Queries*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2006.

[25] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communincations of the ACM*, 13(7):422–426, 1970.

[26] B. Bollobás and W. Fernandez de la Vega. The diameter of random regular graphs. *Combinatorica*, 2(2):125–134, 1982.

[27] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting Bloom filters. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA)*, pages 684–695. Springer-Verlag, 2006.

[28] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 546–554, 2003.

[29] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *13th Annual Symposium on Discrete Algorithms (SODA)*, pages 39–48, 2002.

[30] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003.

[31] A. Brodnik and J. I. Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28(5):1627–1640, May 1999.

[32] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 859–860, 2000.

[33] J. Bulánek, M. Koucký, and M. Saks. Tight lower bounds for the online labeling problem. In *Proceedings of the 44th Symposium on Theory of Computing Conference (STOC2)*, pages 1185–1198, 2012.

[34] M. Canim, G. A. Mihaila, B. Bhattacharhee, C. A. Lang, and K. A. Ross. Buffered Bloom filters on solid state storage. In *VLDB ADMS Workshop*, 2010.

[35] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.

[36] 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)*, pages 30–39. Society for Industrial and Applied Mathematics, 2004.

[37] Y. Chen, B. Schmidt, and D. L. Maskell. A reconfigurable Bloom filter architecture for BLASTN. In *Proceedings of the 22nd International Conference on Architecture of Computing Systems (ARCS)*, pages 40–49. Springer-Verlag, 2009.

[38] H. Cheung. TG Video: Fusion io - the power of 1000 hard drives in the palm of your hand. http://www.tgdaily.com/hardware-features/34065-tg-video-fusion-io-the-power-of-1000-hard-drives-in-the-palm-of-your-hand, 2007.

[39] N. Chomsky and M. P. Schützenberger. The Algebraic Theory of Context-Free Languages. In *Computer Programming and Formal Systems*, Studies in Logic, pages 118–161. North-Holland Publishing, Amsterdam, 1963.

[40] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 383–391, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

[41] J. G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE Transactions on Computing*, 33(9):828–834, 1984.

[42] S. Cohen and Y. Matias. Spectral Bloom filters. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 241–252. ACM, 2003.

[43] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, Apr. 2005.

[44] B. Debnath, S. Sengupta, J. Li, D. Lilja, and D. Du. Bloomflash: Bloom filter on flash-based storage. In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS)*, pages 635–644, 2011.

[45] P. F. Dietz. Maintaining order in a linked list. In *Fourteenth Annual ACM Symposium on Theory of Computing (STOC)*, pages 122–127, 1982.

[46] P. F. Dietz, J. I. Seiferas, and J. Zhang. A tight lower bound for on-line monotonic list labeling. In *4th Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 824 of *Lecture Notes in Computer Science*, pages 131–142, 1994.

[47] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *19th Annual Symposium on Theory of Computing (STOC)*, pages 365–372, 1987.

[48] P. F. Dietz and J. Zhang. Lower bounds for monotonic list labeling. In *2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 447 of *Lecture Notes in Computer Science*, 1990.

[49] M. Dietzfelbinger and R. Pagh. Succinct data structures for retrieval and approximate membership. *CoRR*, abs/0803.3693, 2008.

[50] W. Dittrich, D. Hutchinson, and A. Maheshwari. Blocking in parallel multisearch problems (extended abstract). In *10th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 98–107, 1998.

[51] eBay study: How to build trust and improve the shopping experience. `http://knowwpcarey.com/article.cfm?aid=1171`. Accessed: 2014-07-21.

[52] P. Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974.

[53] P. Elias and R. A. Flower. The complexity of some simple retrieval problems. *J. ACM*, 22(3):367–379, 1975.

[54] Y. Emek and A. Korman. New bounds for the controller problem. *Distributed Computing*, 24(3–4):177–186, 2011.

[55] J. Erickson. Lower bounds for external algebraic decision trees. In *16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 755–761, 2005.

[56] Presto: Interacting with petabytes of data at Facebook. https://www.facebook.com/notes/facebook-engineering/presto-interacting-with-petabytes-of-data-at-facebook/10151786197628920. Accessed: 2014-07-21.

[57] 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.

[58] L. Freeman. How NetApp deduplication works - a primer. http://blogs.netapp.com/drdedupe/2010/04/how-netapp-deduplication-works.html, April 2010.

[59] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *1993 IEEE 34th Annual Foundations of Computer Science (FOCS)*, pages 714–723, 1993.

[60] A. Gupta. *Succinct data structures*. PhD thesis, Duke University, 2010.

[61] F. Hao, M. Kodialam, and T. V. Lakshman. Building high accuracy Bloom filters using partitioned hashing. *SIGMETRICS Perform. Eval. Rev.*, 35:277–288, June 2007.

[62] B. He and Q. Luo. Cache-oblivious query processing. In *3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 44–55, 2007.

[63] J. M. Hellerstein, E. Koutsoupias, D. P. Miranker, C. H. Papadimitriou, and V. Samoladas. On a model of indexability and its bounds for range queries. *J. ACM*, 49:35–55, 2002.

[64] J. M. Hellerstein, E. Koutsoupias, and C. H. Papadimitriou. On the analysis of indexing schemes. In *16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 249–256, 1997.

[65] G. J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., 1991.

[66] A. Itai and I. Katriel. Canonical density control. *Inf. Process. Lett.*, 104(6):200–204, 2007.

[67] A. Itai, A. G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *Proc. 8th Internationl Colloquium on Automata, Languages, and Programming (ICALP)*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431, 1981.

[68] G. J. Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, 1988.

[69] M. Karpinski and Y. Nekrich. Predecessor queries in constant time? In *13th Annual European Conference on Algorithms (ESA)*, pages 238–248, 2005.

[70] I. Katriel. Implicit data structures based on local reorganizations. Master's thesis, Technion – Israel Inst. of Tech., Haifa, 2002.

[71] K. Keeton, C. B. Morrey, III, C. A. Soules, and A. Veitch. Lazybase: freshness vs. performance in information management. *SIGOPS Operating Systems Review*, 44(1):15–19, 2010.

[72] M. Knudsen and K. Larsen. I/O-complexity of comparison and permutation problems. Master's thesis, DAIMI, November 1992.

[73] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison Wesley, 1973.

[74] G. Lu, B. Debnath, and D. H. C. Du. A forest-structured bloom filter with flash memory. In *Proceedings of the IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6, 2011.

[75] K. Malde and B. O'Sullivan. Using Bloom filters for large scale gene sequence analysis in Haskell. In *Proceedings of the 11th International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 183–194. Springer-Verlag, 2009.

[76] M. Mitzenmacher. Compressed bloom filters. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing (PODC)*, pages 144–150. ACM, 2001.

[77] M. Mukherjee and L. B. Holder. *Graph-based data mining on social networks*. PhD thesis, University of Texas at Arlington, 2004.

[78] J. K. Mullin. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, 16(5):558–560, 1990.

[79] J. I. Munro. An implicit data structure supporting insertion, deletion, and search in o(log:os2:oen) time. *J. Comput. Syst. Sci.*, 33(1):66–74, 1986.

[80] 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)*, pages 823–829. Society for Industrial and Applied Mathematics, 2005.

[81] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *38th Annual ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006.

[82] V. Raman. Locality-preserving dictionaries: theory and application to clustering in databases. In *18th Symposium on Principles of Database Systems (PODS)*, pages 337–345, 1999.

[83] V. Rödl. On a packing and covering problem. *European Journal of Combinatorics*, 6(1):69–78, 1985.

[84] V. Samoladas and D. P. Miranker. A lower bound theorem for indexing schemes and its application to multidimensional range queries. In *17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 44–51, 1998.

[85] A. Singh, M. Srivatsa, L. Liu, and T. Miller. Apoidea: A decentralized peer-to-peer architecture for crawling the world wide web. In *SIGIR Workshop on Distributed Multimedia Information Retrieval*, pages 126–142, 2003.

[86] Technology: End-to-end technology stack translating photons into information. `http://www.skyboximaging.com/technology`. Accessed: 2014-07-21.

[87] S. Subramanian and S. Ramaswamy. The p-range tree: A new data structure for range searching in secondary memory. In *Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 378–387, 1995.

[88] R. Tamassia and J. S. Vitter. Optimal cooperative search in fractional cascaded data structures. In *Algorithmica*, pages 307–316, 1990.

[89] T. Tao and V. H. Vu. *Additive Combinatorics*. Cambridge University Press, 2009.

[90] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–127, 1979.

[91] A. K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21(1):101–112, 1984.

[92] Twitter's ipo filing shows 215 million monthly active users. `http://abcnews.go.com/Business/twitter-ipo-filing-reveals-500-million-tweets-day/story?id=20460493`. Accessed: 2014-07-21.

[93] J. S. Vitter. *Algorithms and Data Structures for External Memory*. Now Publishers Inc., Hanover, MA, USA, 2008.

[94] D. E. Willard. Maintaining dense sequential files in a dynamic environment (extended abstract). In *14th Annual Symposium on Theory of Computing (STOC)*, pages 114–121, 1982.

[95] D. E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *1986 ACM SIGMOD International Conference on Management of Data*, pages 251–260, 1986.

[96] D. E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Information and Computation*, 97(2):150–204, 1992.

[97] Z. Yuan, J. Miao, Y. Jia, and L. Wang. Counting data stream based on improved counting Bloom filter. In *Proceedings of the 9th International Conference on Web-Age Information Management (WAIM)*, pages 512–519, 2008.

[98] J. Zhang. Density control and on-line labeling problems. Technical report, University of Rochester, Computer Science Department, 1993.

[99] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 18:1–18:14, 2008.