Stony Brook University

Upper and Lower Bounds for Sorting and Searching in External Memory

A Dissertation Presented
by
Dzejla Medjedovic

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

Dzejla Medjedovic

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

Joseph S. B. Mitchell– Chairperson of Defense
Professor, Department of Applied Mathematics and Statistics

Rob Johnson
Assistant Professor, Department of Computer Science

Martin Farach-Colton – External Member
Professor, Rutgers University

This dissertation is accepted by the Graduate School

Charles Taber
Interim Dean of the Graduate School

Abstract of the Dissertation

Upper and Lower Bounds for Sorting and Searching in External Memory

by

Dzejla Medjedovic

Doctor of Philosophy

in

Computer Science

Stony Brook University

2014

This dissertation presents variants on sorting and searching in external memory.

In the first part of the dissertation, we derive lower and upper bounds on sorting with different-sized records. We show that the record size substantially affects the sorting complexity, and so does the final interleaving of the smaller and larger records in the final sorted sequence: sorting costs more when large and small records are segregated than when they are interleaved in the final sorted order.

In the second part of the dissertation, we study the batched predecessor problem in external memory. Given the underlying sorted set S of size n, and a sorted query Q of size $x \leq n^c$, $0 \leq c < 1$, we study tradeoffs between the searching cost, and the cost to preprocess S. We give lower bounds in three external memory models: the I/O comparison model, I/O pointer-machine model, and the indexability model. Our results show that in the I/O comparison model, the batched predecessor problem needs $\Omega(\log_B n + 1/B)$ I/Os per element if the preprocessing is polynomial; with exponential preprocessing, the problem can be solved faster, in $\Theta((\log_2 n + 1)/B)$.

In the third part of the dissertation, we introduce alternatives to the well-known Bloom filter. The *quotient filter* is designed for RAM, but with better data locality than the Bloom filter. The *buffered quotient filter* and the *cascade filter* are SSD-optimized alternatives to the 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.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my advisor Michael Bender — for teaching me standards, the beauty of algorithms, and to appreciate cilantro; Pablo Montes — for all the times we ran to catch that train; Mayank Goswami — for being my faithful accomplice over the years, and for curing my migraine; Samir and Vesna Selmanovic — for being my home away from home; Megan Tudor and Samira Darvishi — for being such funky ladies; Cindy Scalzo — for being my local mom; Michael Budassi — for staying in a book club with me even though I never finish the book; Raquel Perales — for always being up for coffee; Andrea Massari — for being silly enough to start a food blog with me; Brian Tria — for providing me with a daily dose of sarcasm; Mom and Dad — for everything, but especially for convincing me to come home.

Chapter 1

Introduction

During my PhD career, I worked on variants of sorting and searching in external memory. Sorting and searching are fundamental operations that lie at the backbone of most database applications and are an integral step of many algorithms.

This dissertation studies three variants on these fundamental problems, but with a practical tweak: we change the assumptions of the original problem to reflect the constraints of real-life applications.

The first problem deals with sorting in external memory, but allowing input records to vary in size. The second problem studies the fundamental problem of merging two sorted lists in external memory when the length of one list x is polynomially related to the other list y ($x \leq y^c$, $0 \leq c < 1$). Finally, the third problem deals with designing Bloom filters that scale to large data sets.

List of Publications

1. *Don't Thrash: How to Cache Your Hash on Flash*, HOTSTORAGE 2011, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russel Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok

2. *Don't Thrash: How to Cache Your Hash on Flash*, VLDB 2012, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russel Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P.

Spillane, and Erez Zadok

3. *I/O-Complexity of Sorting with Different-Sized Records*, MASSIVE 2013, Michael A. Bender, Mayank Goswami, Dzejla Medjedovic, Pablo Montes and Kostas Tsichlas

4. *The Batched Predecessor Problem in External Memory*, ESA 2014, Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Dzejla Medjedovic, Pablo Montes and Meng-Tsung Tsai

Chapter 2

I/O-Complexity of Sorting with Different-Sized Keys

2.1 Introduction

Sorting in external memory (or I/O-sorting) has received a great deal of attention in the literature [3]. The Disk-Access Model (DAM) [3], traditionally used to analyze problems in external memory, captures essential aspects of today's computers, where the cost of performing computations in internal memory is subsumed by the cost of transferring the data between the external disk and RAM. In this model, where data is stored on a disk of infinite size, and transferred to internal memory of size M in the blocks of size B, the efficiency of an algorithm is measured by the number of block transfers it performs.

Sorting was among the first problems studied in the DAM model [3]. The optimal I/O-sorting algorithm for N unit-sized records is M/B-way merge sort which merges M/B sorted lists at once by dedicating a block of internal memory to each list's front elements. In one linear scan (N/B I/Os), the number of sorted lists reduces by a factor of M/B, producing a single sorted list in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ block transfers, which matches the information-theoretic lower bound for I/O-sorting.

We derive the lower and upper bounds on the fundamental problem of I/O-sorting when input records can vary in size. Our results show that the complexity of sorting is substantially affected by the record size, and somewhat surprisingly, by the final interleaving of the smaller and larger records in the final sorted sequence: the sorting complexity is higher when large records are clustered together than when they are interspersed with the small records.

In the real-world industrial applications, where data naturally admits irregularity, input records routinely vary in size. Recently, there has been an interest in developing algorithms and data structures that are able to efficiently store and process different-sized data [5, 10, 11, 23, 37, 40, 43]. Most notably, [5] develops lower and upper bounds for sorting strings in external memory.

We model records as *indivisible*, that is, to compare two records, an algorithm needs to bring both into main memory in their entirety (this is consistent with the atomic-key model of [10].) An important consequence of our model is that it implicitly sets different comparison costs for records of different sizes: given that transferring small records in and out of memory is cheaper than transferring large records, the comparisons involving small records are also rendered cheaper than the comparisons involving large records. In addition, large records consume more space in internal memory and thus the algorithms is able to fit fewer large records, reducing the parallelism inherent in batched sorting-like applications.

In this work, we investigate how varying record size affects the sorting complexity in the indivisible-key model.

For the simplified setting where the input contains only two record sizes, the unit size (set S) and the large size w (set L), our results show that the sorting complexity depends on three following problems:

1. Sorting S.
2. Placing L (unsorted) within the sorted order of S.
3. Sorting each of large records that are consecutive in the final sorted sequence.

We derive lower and upper bounds on sorting using the three problems. The main technical contribution is contained in the lower bounds for the second problem, where we demonstrate a novel way to prove information-theoretic lower bounds for problems with different-sized records. To the best of our knowledge, these are the first worst-case lower bounds of this type. Lastly, we generalize the 2-sizes result to

any number of record sizes, and give a structure of an optimal recursive algorithm.

Related work. Lower bounds for external-memory problems are traditionally studied in the Disk-Access Model (DAM) [3, 29, 53], and its sub-models: external comparison trees [3, 6, 15], external algebraic decision trees [26], and external computation trees [8].

The importance of variable-sized keys has been recognized since Knuth [36], who mentions the topic in side notes and exercises, acknowledging a gap between theory and practice in sorting and searching. Most previous work studies external-memory searching and B-trees, in particular [10, 23, 37, 40, 43].

Arge et al. [5] study external-memory string sorting. They show that the I/O complexity depends upon the size of the strings relative to the block size and give matching upper and lower bounds in some cases.

There is also work on batched external-memory dictionary operations for unit-sized keys [4, 7]. Finally, sorting multisets and set partitioning in RAM has connections to the PLE problem [25, 42].

Next, we give a technical overview of our results.

2.2 Problem Definitions and Results

First we study the simplified version of the sorting problem where records can come in only two sizes (*two-sized* sorting); then we generalize the result to input records having any number of sizes (*multiple-sized sorting.*)

2.2.1 Results: Two-Sized Sorting

The input to the two-sized sorting problem are $S = \{s_*\}$ (the small records) and $L = \{\ell_*\}$ (the large records, each of size $1 < w \leq M/2$). A set of large elements forms a *stripe* if for each pair of large elements ℓ_i and ℓ_j in the stripe, there does not exist a small element between ℓ_i and ℓ_j in the final sorted order.

We use the parameter k to denote the number of large-element stripes. Let the large-element stripes be $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_k$, as they are encountered in the ascending

sorted order. The parameters in the complexity analysis of sorting are thus S, L, w, k, and $\{L_i\}_{i=1}^{k}$.

Definition 1 (*Two-Sized Sorting* (2-Sort (N)))**.**
INPUT: an unsorted set of elements $N = S \cup L$. S consists of S unit-size elements, and L consists of L/w elements, each of size w, where $B \leq w \leq M/2$.[1]
OUTPUT: Elements in N, sorted and stored contiguously in external memory.

Definition 2 (*Placement of Large Elements (*PLE (S, L)))**.**
INPUT: The *sorted* set of small elements $\mathcal{S} = \{s_1, s_2, \ldots, s_S\}$, and the *unsorted* set of large elements $\mathcal{L} = \{\ell_1, \ell_2, \ldots, \ell_{L/w}\}$.
OUTPUT: Elements in $\mathcal{S}$ are sorted, and elements in L are sorted according to which stripe they belong to, but arbitrarily ordered within their stripe.

Theorem 3 (**Sorting Lower Bound**)**.** *The worst-case I/O complexity for Two-Sized Sorting is determined by the complexity of the PLE problem. So if* PLE-LOWER(S, L) *is a lower bound of the PLE problem, then we have* SORT-LOWER$(N) =$

$$\Omega\left(\frac{S}{B}\log_{M/B}\frac{S}{B} + \text{PLE-LOWER}(S, L) + \left(\sum_{i=1}^{k}\left(\frac{L_i}{B}\log_{M/w}\frac{L_i}{w}\right) + \frac{L}{B}\right)\right).$$

Theorem 4 (**PLE Lower Bound**)**.** *The worst-case I/O complexity for the PLE problem is* PLE-LOWER$(S, L) =$

$$\Omega\left(\min\left\{\frac{kw}{B}\log_M S + \frac{L}{B}\log_M k, \frac{k}{B}\log S + \frac{L}{wB}\log k + \frac{L}{B}\right\}\right).$$

The sorting lower bound has three terms: the first term corresponds to sorting the small records, the second term corresponds to solving the PLE, and the third term corresponds to sorting each large stripe. The two terms of the PLE lower bound come from analyzing how much information an optimal algorithm learns from 1) inputs of large records and 2) inputs of small records.

[1]The assumption that $w \geq B$ is stated only for the convenience of presentation. Our bounds hold for any $1 < w \leq M/2$.

Theorem 5 (**Sorting Upper Bound**). *There exists a Two-Sized Sorting algorithm having I/O complexity* SORT-UPPER$(N) =$

$$O\left(\frac{S}{B}\log_{M/B}\frac{S}{B} + \text{PLE-UPPER}(S,L) + \left(\sum_{i=1}^{k}\left(\frac{L_i}{B}\log_{M/w}\frac{L_i}{w}\right) + \frac{L}{B}\right)\right).$$

Theorem 6 (**PLE Upper Bound**). *There exists an algorithm for the PLE problem having I/O complexity* PLE-UPPER$(S,L) =$

$$O\left(\min\left\{\frac{L}{B}\log_M S + \frac{S}{B}, \frac{L}{w}\log_B k + k\log_B S + \frac{L}{B} + \frac{S}{B}\right\}\right).$$

The upper bound has a similar structure to the lower bound. The algorithm first sorts the small records, solves the PLE, and sorts the unsorted large stripes. We design two algorithms for PLE: the first algorithm is optimized for the situations when large elements are not too large (when $w \leq B\log M$): it builds a tree data structure with a large fan-out (M), and sweeps large elements through each level of the tree, thus benefiting from a large fan-out. The second algorithm is optimized for large elements that are fairly large ($w > B\log M$). The algorithm processes elements one by one by sending them down two B-trees. This way, the algorithm needs to input the 'heavy' elements only once.

In most cases the PLE upper bounds match the lower bounds. Because the first and the third terms of the sorting bounds are tight, the lower bounds are also automatically tight in all cases when any one of these terms dominates the runtime.

2.2.2 Results: Multiple-Sized Sorting

Next we define the sorting problem for m size classes. The input is set $N = \bigcup_{i=1}^{m} N_i$, where N_i denotes ith size class in the increasing order of element sizes. We first define an i-stripe, the stripe-equivalent for the multiple-sized setting. Denote *i-stripe* as a sequence of keys of size w_i or larger in the final sorted order, uninterrupted by the elements smaller than w_i: for example, a 2-stripe represents one maximal sequence of keys from $N_2, \ldots, N_m$ consecutive in the final sorted order. Denote 2-stripes in the set N as $R_1, \ldots, R_{k_1}$, and the number of distinct size classes present in R_i as m_i.

Definition 7 (*Multiple-Sized Sorting (*m-Sort (N)*)*)**.**
INPUT: The input is $N = \bigcup_{i=1}^{m} N_i$, where w_i denotes the size of the elements in class N_i. Also, $w_1 = 1$, and $w_i \geq B$, for $i > 1$.[2]
OUTPUT: Elements in N, sorted and stored contiguously in external memory.

The essential component of m-Sort (N) is GPLE (S, L) problem, the generalization of PLE to any pair of record sizes.

Theorem 8 (m-Sort (N) **Lower Bound).** *The worst-case I/O complexity for Multiple-Sized Sorting is* m-SORT-LOWER$(N) =$

$$\Omega\left(\frac{N_1}{B}\log_{M/B}\frac{N_1}{B} + \sum_{i=1}^{m}\text{GPLE-LOWER}(N_1, N_i) + \sum_{i=1}^{k_1}(m_i)\text{-SORT-LOWER}(R_i)\right).$$

Theorem 9 (GPLE (S, L) **Lower Bound).** *The worst-case I/O complexity for the* GPLE (S, L) *problem, with key sizes* w_1 *and* w_2*,* $w_1 \leq w_2$*, and* L *forming* k *stripes within* S *is* GPLE-LOWER$(S, L) =$

$$\Omega\left(\min\left\{\frac{kw_2}{B}\log_{M/w_1}\frac{S}{w_1} + \frac{L}{B}\log_{M/w_1}k, \frac{kw_1}{B}\log\frac{S}{w_1} + \frac{Lw_1}{Bw_2}\log k\right\}\right).$$

The lower bound for multiple records first demonstrates that the following subproblems are all individually lower bounds on sorting: sorting the small records, finding the position of every larger class within the small record class, and recursively sorting all 2-stripes. To prove that the *sum* of the subproblems is a lower bound, we argue independence between the subproblems. This lower exhibits the structure of an optimal algorithm, i.e., an algorithm with this structure that optimally solves each subproblem is optimal.

Theorem 10 (m-Sort (N) **Upper Bound).** *There exists an algorithm for Multiple-Sized Sorting with complexity* m-SORT-UPPER$(N) =$

$$\Omega\left(\frac{N_1}{B}\log_{M/B}\frac{N_1}{B} + \sum_{i=1}^{m}\text{GPLE-UPPER}(N_1, N_i) + \sum_{i=1}^{k_1}(m_i)\text{-SORT-UPPER}(R_i)\right).$$

[2]Again, the limitation of $w_i \geq B$ is made only for the ease of presentation of proofs, and our results hold for any record size.

Theorem 11 (GPLE (S, L) **Upper Bound).** *When* $w_1 = 1$ *and* $w_2 \geq B$*, the upper bound for* GPLE (S, L) *becomes the upper bound of* PLE (S, L)*. However, when* $w_1 \geq B$ *and* $w_2 \geq B$*, there exists an algorithm for* GPLE (S, L) *having complexity* GPLE-UPPER$(\mathcal{S}, \mathcal{L}) =$

$$O\left(\min\left\{\frac{L}{B}\log_{M/w_1} S, \frac{kw_1}{B}\log\frac{S}{w_1} + \frac{Lw_1}{Bw_2}\log k + \frac{L}{B}\right\}\right).$$

The structure of results is similar to that of the two-sized sorting problem, in that the optimal algorithm first sorts the smallest keys, performs GPLE(N_1, N_i) for all $i > 1$, and recurses on 2-stripes. Our GPLE algorithms are similar to those for PLE, with one exception: when both record sizes are larger than a block, binary search becomes optimal. The next corollary states that GPLE lower bounds are optimal in the following wide range of parameters:

Corollary 12. If $w_2 > w_1 \log(M/w_1)$,
GPLE-UPPER$(\mathcal{S}, \mathcal{L}) = \Theta($GPLE-LOWER$(\mathcal{S}, \mathcal{L}))$.

2.3 Lower Bounds

The main question we explore in this work is how to derive lower bounds on comparison-based problems when records can vary in size. In this section, we show a novel way to derive the sorting lower bound by decomposing the sorting problem into subproblems in three different layers.

To understand what goes wrong when applying traditional information-theoretic argument for different-sized key problems, first we recall the lower bound argument for sorting unit-sized keys. Using the assumption that all blocks are internally sorted[3], the total number of permutations that could potentially represent the sorted order is $t = \Omega(N!/B^{N/B})$. In one I/O, at most B elements are transferred into main memory that holds at most $M - B$ elements, thus the fraction of permutations that this I/O can eliminate corresponds to the number of ways to place the incoming B elements into the sorted order of elements in memory, and it equals $\binom{M}{B}$. Thus the total number of I/Os required to sort equals $\log t / \log b = \Omega\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$.

[3] Achieved by an initial linear scan of input.

However, the sleek argument for unit-sized records does not carry over to the records of varying sizes for the following reason: the parameter t substantially varies depending on whether the block transfer carries large records or small records, and what are the contents of RAM at the time of the I/O. For example, a transfer of a large element into main memory full of large elements gives only $t = \binom{M/w}{1}/(w/B)$ as a large-element transfer costs w/B. On the other hand, a transfer of B small records into main memory filled with small records gives $t = \binom{M}{B}$. The parameter t can be anything in the range between these two values, for any combination of small and large records being in memory, and inputs of small or large records. If we simply take the highest t, one obtained from the small record inputs, we don't get a tight lower bound—this intuitively makes sense because we can not sort the entire set using only small record comparisons. This brings us to the first layer of decomposition of the sorting problem.

2.3.1 First Layer: Three Types of Comparisons

Here we decompose sorting into three subproblems, based on whether the subproblem requires small-to-small, large-to-large or small-to-large record comparisons. We show two lower bounds based on equal-sized record sorting: sorting all the small records (small-to-small comparisons), and sorting all large stripes (large-to-large comparisons). It is important to extricate these subproblems because they do not directly deal with sorting records of different sizes and are trivially obtained from the Aggarwal and Vitter result (see below). The remaining segment of the problem is achieved using small-to-large record comparisons: this is the PLE problem, which we decompose in the next two levels.

Lemma 13. SORT-LOWER$(N) = \Omega\left(\frac{S}{B}\log_{M/B}\frac{S}{B}\right)$.

Proof: We denote by p the total number of permutations that any algorithm for Two-Sized Atomic-Key Sorting must distinguish between in order to sort. We bound p in terms of the number of large-element stripes k:

$$p \geq \frac{S!}{(B!)^{S/B}}\binom{S-1}{k}\left(\frac{L}{w}!\right)\binom{L/w-1}{k-1}.$$

The four factors that comprise the right side include (1) sorting S (after sorting within the small blocks, the total number of permutations goes down by a factor of

$(B!)^{S/B}$), (2) choosing the k locations for stripes within S, (3) sorting L and (4) forming k large-element stripes by choosing $k-1$ delimiters in the sorted L.

We can assume that the elements in memory are sorted at all times, because maintaining this order requires no additional I/Os. The number of remaining permutations goes down by at most $\binom{M}{B}$ after one memory transfer, thus a lower bound on the number of I/Os to sort is $\text{SORT-LOWER}(N) = \Omega\left(\log p / \log \binom{M}{B}\right)$. Using that $p \geq \frac{S!}{(B!)^{S/B}}$, and that $\log \binom{M}{B} = \Theta(B \log(M/B))$, we get

$$\text{SORT-LOWER}(N) = \Omega\left(\frac{S \log S - S \log B}{B \log(M/B)}\right) = \Omega\left(\frac{S}{B} \log_{M/B} \frac{S}{B}\right),$$

concluding the lemma.

Next we show a generalization of the Aggarwal and Vitter result for records of size $w > 1$.

Theorem 14 (Aggarwal and Vitter [3])**.** *Consider an external-memory algorithm A that sorts the total volume V of V/w elements, each of size w.*

1. *If $1 \leq w < B$, A requires $\Omega\left(\frac{V}{B} \log_{M/B} \frac{V}{B}\right)$ block transfers.*
2. *If $w \geq B$, A requires $\Omega\left(\frac{V}{B} \log_{M/w} \frac{V}{w}\right)$ block transfers.*

Proof: In both cases, we count the total number of possible output permutations and the maximum permutations achievable during a single I/O, or during the input of one element (w/B I/Os), whichever is larger.

1. $1 \leq w < B$: Assume that w divides B. In a linear scan, we can internally sort every block, which restricts the possible output permutations to

$$\frac{(V/w)!}{((B/w)!)^{V/B}}.$$

When a block is input, there are at most $(M-B)/w$ sorted elements in memory. The incoming block contains B/w sorted elements, so the number of remaining output permutations reduces by at most a factor of

$$\binom{(M-B)/w + B/w}{B/w} = \binom{M/w}{B/w}.$$

Thus we get that the algorithm requires at least

$$\Omega\left(\frac{\log\left((V/w)!/((B/w)!)^{V/B}\right)}{\log\binom{M/w}{B/w}}\right)$$

block transfers. Using the same bounds for $\binom{n}{k}$ as in Lemma 13, we get the desired bound.

2. $w \geq B$: Assume for simplicity that w is an integer multiple of B. In this case, there are $(V/w)!$ possible output permutations. One can scan every chunk of size M, but this does not change the bound we present asymptotically ($\log_{M/w} V/w$ changes to $\log_{M/w} V/M$).

   When an element is input, there are at most $(M - w)/w = (M/w) - 1$ (sorted) elements in memory. The input of an element costs w/B I/Os, and this element can go into any one of M/w positions between the elements in memory. Hence the maximum branching factor for one element input is M/w.

   This implies that the number of element inputs is

$$\Omega\left(\frac{(V/w)\log(V/w)}{\log(M/w)}\right) = \Omega\left(\frac{V}{w}\log_{M/w}\frac{V}{w}\right),$$

   and multiplying by the cost of every large element input (w/B) gives us the claimed bound.

Lemma 15.

$$\text{SORT-LOWER}(N) = \Omega\left(\sum_{i=1}^{k}\frac{L_i}{B}\log_{M/w}\frac{L_i}{w}\right).$$

Proof: Consider an instance of sorting where the collection of small elements S is already sorted, and every element from L knows its position within S. To finish sorting, the only useful comparisons are now between the large elements from the same stripe. This reduces to having k independent sorting instances, and by Theorem 14 and a trivial linear-scan bound, the lower bound to sort in this scenario is $\text{SORT-LOWER}(N) = \Omega\left(\sum_{i=1}^{k}\frac{L_i}{B}\log_{M/w}\frac{L_i}{w}\right)$ block transfers.

We remark that Lemmas 13 and 15 give the first and third terms of the lower bound from Theorem 3, and the second term comes from the PLE:

Lemma 16.

$$\text{SORT-LOWER}(N) = \Omega(\text{PLE-LOWER}(S, L)).$$

2.3.2 Second Layer: Interleaving-Sensitive Analysis

Because we are interested in the interleaving-sensitive analysis of the PLE, we express the complexity of the PLE using its three instances:

1. S-k: An instance with only *one large element* in each large-element stripe.
   - Input: Unit-sized elements $s_1, \ldots, s_S$ *(sorted)*, where $s_1 = -\infty$ and $s_S = \infty$, and large elements $\ell_1, \ldots, \ell_k$ (volume kw) *unsorted*.
   - Output: The entire set in the sorted order.
2. k-$\tilde{k}$: An instance with only *one small element* in each small-element stripe.
   - Unit-sized elements $s_1, \ldots, s_{k+1}$ *sorted*, where $s_1 = -\infty$ and $s_{k+1} = \infty$, and large elements $\ell_1, \ldots, \ell_{\tilde{k}}$ (volume $\tilde{k}w$) *unsorted*.
   - Output: For each ℓ_i, output its predecessor and successor in S.
3. k-k: An instance with only *one element in each stripe*, large or small.
   - Input: Unit-sized elements $s_1, \ldots, s_{k+1}$ *sorted*, where $s_1 = -\infty$ and $s_{k+1} = \infty$, and large elements $\ell_1, \ldots, \ell_k$ (volume kw) *unsorted*.
   - Output: The entire set in the sorted order.

The lower bounds for these problems are also lower bounds for the PLE problem, thus we have the following lemma:

Lemma 17.

$$\text{PLE-LOWER}(S, L) = \Omega(S\text{-}k\text{-LOWER}(S, L) + k\text{-}\tilde{k}\text{-LOWER}(S, L) + k\text{-}k\text{-LOWER}(S, L)).$$

2.3.3 Third Layer: Small-Block Input vs. Large-Record Input

In this section, we prove lower bounds for S-k, k-$\tilde{k}$ and k-k by analyzing the amount of information an optimal algorithm learns from a small-block input and from a large-record input. The minimum of the two terms is a lower bound on the

problem. If the large records are very large, then the lower bound comes from solving the problem by predominantly inputting the small records, whereas if the large records are not very large, the lower bound comes from predominantly inputting large records.

The format of lower bounds for S-k, k-$\tilde{k}$ and k-k is as follows: given X as the total number of bits that an algorithm needs to learn in order to solve the problem, all three problems have a lower bound of

$$\min\left(\frac{X}{B}, \frac{Xw}{B\log M}\right).$$

See Theorems 22, 23 and 25 later for the exact expressions. To prove these lower bounds, we design an adversary strategy that ensures the following: in each block transfer of small records, an algorithm learns at most $O(B)$ bits, and in each input of a large record (which costs w/B I/Os), the algorithm learns $O(\log M)$ bits of information.

The proofs of three lower bounds share the same setup and most of the adversary's strategy. We first describe this common framework; at the end of the section, we explain how the main strategy is modified to obtain lower bounds for each individual subproblem.

We capture the information learned at every point of the algorithm by assigning a *search interval* to every large element:

Definition 18 (*Search interval*)**.** A search interval $R(\ell) = (s_i, s_j)$ for a large element ℓ at step t is the narrowest interval of small elements where ℓ can possibly land in the final sorted order, given what the algorithm has learned so far. A search interval $R(s_j)$ is given as an interval (s_i, s_k) where s_i is the predecessor and s_k is the successor of s_j in S.

Mechanics of the Strategy We denote by $\{\ell_i^{p-1}\}_{i=1}^{M/w}$ the set of at most M/w large elements in memory before the tth I/O. The nodes in $\mathcal{T}$ belonging to the set $\{v^{p-1}(\ell_i^{p-1})\}_{i=1}^{M/w}$ have no ancestor-descendant relationships between them. We write S_i^{p-1} to denote $I(v^{p-1}(\ell_i^{p-1}))$, the search interval of ℓ_i^{p-1} at step $p-1$.

Small-block input. Consider the incoming block. We denote n_{pi} as the number of incoming small elements that belong to S_i^{p-1}. These elements divide S_i^{p-1} into

$n_{pi}+1$ parts $\{P_1,\ldots,P_{n_{pi}+1}\}$, some of them possibly empty. The largest of these parts (say P_j) is of size at least $1/(n_{pi}+1)$ times that of S_i^{p-1}. The new search interval of ℓ_i^p is defined to be the highest node in $\mathcal{T}$ such that $I(v)\subset P_j$.

Large element input. On an input of a large element ℓ_{new}^p (with search interval S_{new}^{p-1}), the adversary uses the strategy similar to that one on a small-block input to compare ℓ_{new}^p with the (at most) M small elements present in memory. These M small elements divide S_{new}^{p-1} into at most M parts, and the new search interval of ℓ_{new}^p corresponds to the highest node in $\mathcal{T}$ that contains the largest part.

This is the temporary search interval S_{new}, with the corresponding node v_{new}.

S_{new} can be related to the search intervals of memory-resident large elements in three following ways:

Case 1. The element ℓ_{new}^p shares a node with another large element ℓ_i^p. The conflict is resolved by sending ℓ_{new}^p and ℓ_i^p to the left and right children of v_{new} respectively.

Case 2. The element ℓ_{new}^p has an ancestor in memory. The ancestor is sent one level down, to the child that does not contain v_{new} in its subtree. Thus the conflict is resolved while giving at most $O(1)$ bit.

Case 3. The element ℓ_{new}^p has descendants in memory.

Denote the nodes that are descendants of v_{new} in $\mathcal{T}$ as $v_1,\ldots,v_{M/w}$. Let the corresponding search intervals be $S_1^{p-1},\ldots,S_{M/w}^{p-1}$, respectively. Let $X=\cup_{i=1}^{M/w}S_i^{p-1}$ and $Y=S_{\text{new}}\setminus X$. The set Y is a union of at most M/w intervals, each of which we denote by Y_i. Let Z be the largest interval from the set $\{S_1^{p-1},\ldots,S_{M/w}^{p-1},Y_1,\ldots,Y_{M/w}\}$. Hence, $\|Z\|\geq\|S_{\text{new}}\|/(2M/w)$.

There are two cases to consider. The first case is when $Z=S_i^{p-1}$ for some i. In this case, $S_{\text{new}}=S_i^{p-1}$. In doing this we have given at most $O(\log M)$ bits. Now we proceed as in Case 1 to resolve the conflict with at most $O(1)$ extra bits. Otherwise, if $Z=Y_i$ for some i, then the adversary allots ℓ_{new}^p to the highest node v in $\mathcal{T}$ such that $I(v)\subseteq Z$.

Analysis of the Strategy. We have the following lemmas:

Lemma 19. *On a small-block input, the adversary gives at most $O(\log(n_{pi}+1))$ bits to ℓ_i^p.*

Proof: Observe that

$$\|P_j\| \geq \frac{\|S_i^{p-1}\|}{(n_{pi}+1)}.$$

Divide S_i^{p-1} into $2(n_{pi}+1)$ equal parts (with the last one being possibly smaller). If P_j is equal to the union of two consecutive such parts, there is a node in $\mathcal{T}$ corresponding to P_j, and the adversary has given exactly $\log(n_{pi}+1)$ bits. Otherwise, P_j contains at least one of these parts, for which there is a node $\log(n_{pi}+1)+1$ levels below $v^{p-1}(\ell_i^{p-1})$, which is how many bits the adversary gives in this scenario.

In either case, the maximum number of bits given by the adversary is $O(\log(n_{pi}+1))$, as claimed.

Lemma 20. *On a small-block input, the adversary gives at most $O(B)$ bits.*

Proof: This follows easily from Lemma 19. Let G denote the total number of bits given by the adversary during the input of a block of small elements. It can be seen that $G = \sum_{i=1}^{B}(\log(n_{pi}+1)+1)$. By definition $\sum_{i=1}^{B} n_{pi} = B$, implying that $\sum_{i=1}^{B}\log(n_{ti}+1) \leq B$, which in turn implies that $G < 2B = O(B)$.

Lemma 21. *During the input of a large element, the adversary gives at most $O(\log M)$ bits.*

Proof: The number of bits given due to comparisons with small elements already in memory is $O(\log M)$. In each of the three cases an additional $O(\log(M/w))$ bits are given. Thus, the total number of bits given by the adversary during the I/O of a large element is $O(\log M)$.

Now we are ready to prove the individual lower bounds for k-k, k-$\tilde{k}$ and S-k:

2.3.4 Putting It All Together

S-k Lower Bound. The proof rests on the following action of the adversary: in the very beginning, the adversary gives the algorithm the extra information that the ith largest large element lies somewhere between $s_{(i-1)\alpha}$ and $s_{i\alpha}$, where $\alpha = S/k$.

In other words, the adversary tells the algorithm that the large elements are equally distributed across $\mathcal{S}$, one in each chunk of size S/k in $\mathcal{S}$.

This deems the invariant of large elements in main memory having disjoint search intervals automatically satisfied.

Because any algorithm that solves S-k must achieve $\Omega(k \log(S/k))$ bits of information, we have that

Theorem 22. *The worst-case complexity of S-k is* $\Omega\left(\min\left(\frac{kw}{B}\log_M \frac{S}{k}, \frac{k}{B}\log\frac{S}{k} + \frac{kw}{B}\right)\right)$.

k-$\tilde{k}$ Lower Bound. To solve k-$\tilde{k}$, an algorithm needs to learn $k \log \tilde{k}$ bits of information. Using the adversary strategy we described, we obtain the following lower bound:

Theorem 23. *The worst-case complexity of k-$\tilde{k}$ is* $\Omega\left(\min\left(\frac{\tilde{k}w}{B}\log_M k, \frac{\tilde{k}}{B}\log k + \frac{\tilde{k}w}{B}\right)\right)$.

k-k Lower Bound. To solve k-k, an algorithm needs to learn $k \log k$ bits of information. In the k-k problem, we expect to produce the perfect interleaving of the small and large elements in the final sorted order. That is, *each element lands in its own leaf of $\mathcal{T}$*.

Therefore, the adversary does not posses the freedom to route elements down the tree at all times using the strategy we described. Instead, the strategy is used for a fraction of total bits the algorithm learns, and the remaining fraction is used to make up for the potential imbalance created by sending more elements to the left or to the right. We call these *early* and *late* bits, respectively. Late bits are effectively given away for free by the adversary.

More formally, we define the *node capacity ($c^T(v)$)* as the number of large elements that pass through v during the execution of an algorithm. If the k-k algorithm runs in T I/Os, then the node capacity of v at a level h of $\mathcal{T}$ is designated by $c^T(v) = k/2^h$.

Definition 24 (*early* and *late bits*)**.** A bit gained by a large element ℓ is an *early* bit if, when ℓ moves from v to one of v's children, at most $c^T(v)/4 - 1$ other large elements have already passed through v. The remainder of the bits are *late* bits.

Because a small-block input gives $O(B)$ bits and a large-element input gives $O(\log M)$ bits, and we need to achieve all early bits to solve the problem (there are $(k \log k)/4$ of them), we obtain the following lower bound:

Theorem 25. *The worst-case complexity of k-k is* $\Omega\left(\min\left(\frac{kw}{B}\log_M k, \frac{k}{B}\log k + \frac{kw}{B}\right)\right)$.

Proof of Theorem 4. Here we give details on how to combine the lower bounds of PLE subproblems to obtain the expression from Theorem 4. The lower bounds for k-k, k-$\tilde{k}$ and S-k are each a minimum of two terms; it is safe to add the respective terms as the transition between which term dominates occurs at exactly the same value of w for each of the subproblems.

Adding the terms for the lower bounds of k-k (Theorem 25) and S-k (Theorem 22), provides the $\frac{k}{B}\log S$ and $\frac{kw}{B}\log_M S$ terms in Theorem 4.

Adding the terms for the lower bounds of k-k (Theorem 25) and k-$\tilde{k}$ (Theorem 23), and using that $k + \tilde{k} = L/w$ provides the $\frac{L}{wB}\log S$ and $\frac{L}{B}\log_M S$ terms in Theorem 4. □

Proof of Theorem 3. Adding the terms from sorting small elements (Lemma 13), sorting the large element stripes (Lemma 15) and the PLE (S, L) lower bound (Theorem 4) proves the lower bound in Theorem 3. □

2.4 Algorithms

Our sorting algorithm includes three major steps: (1) sort the small elements, (2) solve the PLE problem, and (3) sort each large-element stripe.

The costs to implement Steps 1 and 3 optimally using a multi-way external merge sort [3] are

$$O\left(\frac{S}{B}\log_{M/B}\frac{S}{B}\right) \quad \text{and} \quad O\left(\frac{L_i}{B}\log_{M/w}\frac{L_i}{w}\right). \tag{1}$$

We give two algorithms to solve Step 2: PLE-DFS and PLE-BFS.

PLE-DFS. PLE-DFS builds a static B-tree T on S, and searches for large elements in T one by one. This approach is preferred in the case of really large elements, and it is better to input them fewer times.

We dynamically maintain a smaller B-tree T' that contains only *border elements* (the two small elements sandwiching each large element in the final sorted order) and has depth at most $\log_B k$. All large elements first travel down T' to locate their stripe. Only those elements for which their stripe has not yet been discovered need to travel down T. After a new stripe is discovered in T, it is then added to T'. The total cost becomes

$$O\left(\frac{L}{w}\log_B k + k\log_B S + \frac{L}{B} + \frac{S}{B}\right). \tag{2}$$

PLE-BFS. Our second algorithm for PLE uses a batch-searching tree with fanout $\Theta(M)$. When a node of the tree is brought into memory, we route all large elements via the node to the next level. We process the nodes of the M-tree level by level so all large elements proceed at an equal pace from the root to leaves. This technique is helpful when large elements are sufficiently small so that bringing them many times into memory does not hurt while they benefit from a large fanout.

The analysis is as follows: at each level of M-tree, the algorithm spends $\Theta(L/B)$ I/Os in large-element inputs. Every node of the tree is brought in at most once, which results in total (S/B) I/Os in small-element inputs. The total number of memory transfers for PLE-BFS then becomes

$$O\left(\frac{L}{B}\log_M S + \frac{S}{B}\right). \tag{3}$$

Proof of Theorem 6. Bounds (2) and (3) imply Theorem 6. □

Proof of Theorem 5. Theorem 6 and the bounds given in (1) imply Theorem 5. □

2.5 Sorting with Multiple Key Sizes

This section gives generalized lower and upper bounds for sorting when the input keys can have any number of key sizes. We focus our attention on a particular class of algorithms—interleaving-based algorithms— that sort *'bottom-up'*: in these algorithms, comparisons are performed between keys of size w_i and higher only after all smaller keys have been sorted. We prove that there exists an algorithm in this class which is optimal. Lastly, we describe the specifics of our upper bound.

We denote by **mSORT** an instance of sorting with keys of m distinct sizes, and the input set to be sorted by $\mathcal{N} = \bigcup_{i=1}^{m} \mathcal{N}_i$, where $\mathcal{N}_i$ denotes the ith size class in the increasing order of element sizes.

Definition 26 (Class of Interleaving-Based Sorting Algorithms for **mSORT** ($\mathcal{C}_m$))**.** Define $\mathcal{C}_m$, $m \geq 2$ inductively as follows:

1. (Base case) $\mathcal{C}_2$ is the class of algorithms that has the following steps:
   - Sort $\mathcal{N}_1$.
   - Solve GPLE($\mathcal{N}_1, \mathcal{N}_2$).
   - Sort each stripe of elements from $\mathcal{N}_2$ independently, i.e., without using any comparisons to elements outside the given stripe.
2. (General case) $\mathcal{C}_m$ is the class of algorithms that sort as follows:
   - Sort $\mathcal{N}_1$.
   - Solve GPLE ($\mathcal{N}_1, \mathcal{N}_i$), $\forall i$, $1 < i \leq m$. Every key lands into one of the resulting stripes $R_1, \ldots, R_k$. Denote the number of distinct key sizes in R_i by m_i, $m_i < m$.
   - For each $1 \leq i \leq k$, sort $\mathcal{R}_i$ using an algorithm from the class $\mathcal{C}_{m_i}$. Each $\mathcal{R}_i$ is sorted separately, without comparisons to the elements outside $\mathcal{R}_i$.

Proof sketch for Theorem 8. We prove the theorem by showing that there exists an optimal algorithm $\mathcal{A}_m$ for **mSORT** such that $\mathcal{A}_m \in \mathcal{C}_m$. Definition 26 formalizes how A_m unravels into a sequence of calls to subroutines GPLE($\mathcal{S}_i, \mathcal{S}_j$) and SORT($\mathcal{S}_k$), where $\mathcal{S}_k$ is an equal-size element set. In order to establish that an algorithm that optimally solves each of these subroutines is also an optimal algorithm for **mSORT**, we need to argue the following:

- Lower bounds for GPLE($\mathcal{S}_i, \mathcal{S}_j$) and SORT($\mathcal{S}_k$) do not increase once the input is restricted from $\mathcal{N}$ to $\mathcal{S}_i \cup \mathcal{S}_j$ and $\mathcal{S}_k$, respectively. (GPLE($\mathcal{S}_i, \mathcal{S}_j$) and SORT($\mathcal{S}_k$) with input $\mathcal{N}$ are trivially lower bounds on **mSORT**.)
- All individual GPLE instances are *mutually independent*, i.e., solving one instance does not help solve the other one faster. We give the same argument for SORT instances.

Our upper bound is an algorithm from the class C_m that implements the subroutines as follows: SORT($\mathcal{S}_k$) uses equal-sized sorting algorithm, and GPLE($\mathcal{S}_i, \mathcal{S}_j$) is implemented as in the two-sized algorithm (see previous section).

However, when both key sizes in the GPLE instance are larger than the cacheline B, the complexity of GPLE changes. It turns out that in this case, GPLE can be solved faster by replacing B-trees in the two-sized algorithm by binary search trees.

Proof of Theorem 9. As in the case of two sizes, we get lower bounds on GPLE (S, L) by deriving lower bounds on the generalized versions of three problems: $(G)k$-k, $(G)k$-$\tilde{k}$ and $(G)S$-k, that are defined analogously. In each of the three subproblems, a sorted set of w_1-size elements is given, and one is required to place an unsorted set of w_2-size elements amongst the sorted set (in $(G)k$-k these k elements are perfectly interleaved, and in $(G)S$-k the k elements go into different slots amongst the small(er) elements). One easily observes

Observation 27. The number of bits required by any algorithm to solve:

1. $(G)k$-k is $\Omega(k \log k)$.
2. $(G)k$-$\tilde{k}$ is $\Omega(\tilde{k} \log k)$.
3. $(G)S$-k is $\Omega(k \log(S/kw_1))$

Our adversary maintains the same invariant as in the proofs of k-k, k-$\tilde{k}$ and S-k; all the large(r) elements (size w_2) in memory have disjoint search spaces. The strategy is randomized in $(G)k$-k, and deterministic in $(G)k$-$\tilde{k}$. This ensures:

Claim 28. In all of the three scenarios, the adversary strategy guarantees:

1. The I/O of a small element (costing w_1/B I/Os) achieves $O(1)$ bits w.h.p.
2. The I/O of a large element (costing w_2/B I/Os) achieves $O(\log(M/w_1))$ bits w.h.p.

Thus, a $\max(1/(w_1/B), \log(M/w_1)/(w_2/B))$ bits can be achieved per I/O. Dividing the number of bits required by this quantity now gives lower bounds on $(G)k$-k, $(G)k$-$\tilde{k}$ and $(G)S$-k, and adding them up gives us the lower bound claimed in Theorem 9.

Generalization to the case when $w_1 < w_2 < B$: The number of bits required by an algorithm for any of the three instances remains unchanged, as that is an information lower bound. It remains to see how the invariant maintained by the adversary limits the information achieved by any algorithm.

The input of a small block contains B/w_1 elements now. Since the large elements in memory have disjoint search spaces, the maximum number of bits achievable by this I/O is B/w_1, which is the case when each of these small elements is a pivot for a unique large element. Thus we get $O(B/w_1)$ bits per I/O.

The input of a large block contains B/w_2 large elements. The memory can contain at most $(M-B)/w_1$ small elements, and so the total number of possible permutations achievable is

$$\begin{aligned} P &= \binom{\frac{M-B}{w_1}+\frac{B}{w_2}}{\frac{B}{w_2}} \left(\frac{B}{w_2}!\right) \\ &< \binom{\frac{M}{w_1}}{\frac{B}{w_2}} \left(\frac{B}{w_2}!\right) \\ &< \left(\frac{ew_2M}{Bw_1}\right)^{B/w_2} \left(\frac{B}{w_2}!\right) \end{aligned}$$

This gives

$$\begin{aligned} \log P &= O\left(\frac{B}{w_2}\log\left(\frac{Mw_2}{Bw_1}\right)+\frac{B}{w_2}\log\left(\frac{B}{w_2}\right)\right) \\ &= O\left(\frac{B}{w_2}\log\left(\frac{M}{w_1}\right)\right) \end{aligned}$$

bits per I/O.

In both cases, the amortized number of bits achieved is:

1. $O(B/w_1)$ bits per I/O, equivalent to $O(1)$ bit per w_1/B I/Os.
2. $\frac{B}{w_2}\log\left(\frac{M}{w_1}\right)$ bits per I/O, equivalent to $\log(M/w_1)$ bits per w_2/B I/Os

□

2.6 Conclusion

The aim of this work is to understand how the complexity of classical external-memory problems such as sorting changes when keys have variable sizes. This

chapter gives the first sorting bounds for variable-size keys, analyzed in terms of final key interleaving. Our results show that when giving interleaving-dependent bounds, a vital component of sorting in the atomic-key setting is an underlying batched searching problem.

Our motivation derives from industrial databases, including TokuDB [52], which routinely sort keys of varying sizes, and our results illustrate that there is a rich set of issues that arise when sorting keys of variable sizes.

Open problems abound. A question of immediate interest is how can these results be extended to cases where keys are typical database rows—splittable in a limited sense. And, how can these results be used to give more precise bounds on sorting strings? Even though the atomic-key model accentuates the dependence of sorting complexity on interleaving, there exist cases in which our algorithms help sort strings faster.

Figure 1: Pseudocode for Two-Sized Sorting

TWO-SIZED-SORT(N)

1 Partition N in two sets S and L of short and long elements, respectively.
2 Sort S
3 $L_1, \ldots, L_k \leftarrow \min$ (PLE-BFS(S, L), PLE-DFS(S, L))
4 Sort each L_i.

PLE-DFS(S, L)

```
 1  Build a B-tree T on S.
 2  T' ← ∅
 3  for each e ∈ L
 4       do Bring e into memory.
 5          found ← SEARCH(T', e)
 6          if not found
 7            then SEARCH(T, e)
 8                 Let x and y be the predecessor and successor of e in S.
 9                 INSERT(T', [x, y])
10          Write e to the corresponding stripe.
```

PLE-BFS(S, L)

1 Build an M-tree T on S.
2 Bring the root of T into memory.
3 Read each long element and send it to the
appropriate child.
4 Recurse in BFS order on each child until a leaf is reached.

Chapter 3

The Batched Predecessor Problem in External Memory

3.1 Introduction

A *static dictionary* is a data structure that represents a set $S = \{s_1, s_2, \ldots, s_n\}$ subject to the following operations:

PREPROCESS(S):	Prepare a data structure to answer queries.
SEARCH(q, S):	Return TRUE if $q \in S$ and FALSE otherwise.
PREDECESSOR(q, S):	Return $\max_{s_i \in S}\{s_i < q\}$.

The traditional static dictionary can be extended to support batched operations. Let $Q = \{q_1, \ldots, q_x\}$. Then, the *batched predecessor* problem can be defined as follows:

BATCHEDPRED(Q, S):	Return $A = \{a_1, \ldots, a_x\}$, where $a_i =$ PREDECESSOR(q_i, S).

In this work we prove lower bounds on the batched predecessor problem in *external memory* [3], that is, when the dictionary is too large to fit into main memory. We study tradeoffs between the searching cost and the cost to preprocess the underlying set S. We present our results in three models: the comparison-based I/O model [3], the pointer-machine I/O model [48], and the indexability model [30,31].

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 [4, 15, 17] are suboptimal.

Our results show that 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 is bounded by a polynomial; on the other hand, the problem *can* be solved asymptotically faster, in $\Theta((\log_2 n)/B)$ I/Os, if we impose no constraints on preprocessing. These bounds stand in marked contrast to single-predecessor queries, where one search costs $\Omega(\log_B n)$ even if preprocessing is unlimited.

We assume that S and Q are sorted. Without loss of generality, Q is sorted because Q's sort time is subsumed by the query time. Without loss of generality, S is sorted, as long as the preprocessing time is slightly superlinear. We consider sorted S throughout. 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 work 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 naïve 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 [4, 15, 17] also take $\Theta(n/B)$ I/Os during a terminal flush phase. We show that with polynomial preprocessing, performing independent searches for each element in Q is optimal, but it is possible to do better for higher preprocessing.

Single and batched predecessor problems in RAM.

In the comparison model, a single predecessor can be found in $\Theta(\log n)$ time using binary search. The batched predecessor problem is solved in $\Theta(x \log(n/x) + x)$ by combining merging and binary search [35, 36]. The bounds for both problems remain tight for any preprocessing budget.

Pătraşcu and Thorup [44] give tight lower bounds for single predecessor queries in the cell-probe model. We are unaware of prior lower bounds for the batched predecessor problem in the pointer-machine and cell-probe models.

Although batching does not help algorithms that rely on comparisons, Karpinski and Nekrich [33] 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 $x = O(\sqrt{\log n})$ ($O(1)$ per element amortized) with superpolynomial preprocessing.

Batched predecessor problem in external memory.

Dittrich et al. [24] 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 $\varepsilon > 0$.

Buffering is a standard technique for improving the performance of external-memory algorithms [4, 15, 17]. By buffering, partial work on a set of operations can share an I/O, thus reducing the per-operation I/O cost. Queries can similarly be buffered. In this work, the number of queries, x, is much smaller than the size, n, of the data structure being queried. As a result, as the partial work on the queries progresses, the query paths can diverge within the larger search structure, eliminating the benefit of buffering.

Goodrich et al. [29] 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 [49].

Results

We first consider the **comparison-based I/O model** [3]. In this model, the problem cannot be solved faster than $\Omega(\log_B n)$ I/Os per element if preprocessing is polynomial. That is, batching queries is not faster than processing them one by one. With exponential preprocessing, the problem can be solved faster, in $\Theta((\log_2 n)/B)$ I/Os per element. We generalize to show a query-preprocessing tradeoff.

Next we study the **pointer-machine I/O model** [48], which is less restrictive than the comparison I/O model in main memory, but more restrictive in external

memory.[1] We show that with preprocessing at most $O(n^{4/3-\varepsilon})$ for constant $\varepsilon > 0$, the cost per element is again $\Omega(\log_B n)$.

Finally, we turn to the more general **indexability model** [30, 31]. This model is frequently used to describe reporting problems, and it focuses on bounding the number of disk blocks that contain the answers to the query subject to the space limit of the data structure; the searching cost is ignored. 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, $r = (n/B)^{\Omega(B/\alpha^2)}$. The lower bounds in this model also hold in the previous two models. This result shows that it is impossible to obtain $O(1/B)$ per element unless the space used by the data structure is exponential, which corresponds to the situation in RAM, where exponential preprocessing is required to achieve $O(1)$ amortized time per query element [33].

The rest of this section formally outlines our results.

Theorem 29 (Lower and upper bound, unrestricted preprocessing, I/O comparison model)**.** *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*

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

I/Os in the worst-case, no matter the preprocessing. There exists a comparison-based algorithm matching this bound.

Traditional information-theoretic techniques give tight sorting-like lower bounds for this problem in the RAM model. In external memory, the analogous approach yields a lower bound of $\Omega\left(\frac{x}{B}\log_{M/B}\frac{n}{x} + \frac{x}{B}\right)$. On the other hand, repeated finger searching in a B-tree yields an upper bound of $O(x\log_B n)$. Theorem 29 shows that both bounds are weak, and that in external memory this problem has a complexity that is between sorting and searching.

[1] 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.

We can interpret results in the comparison model through the amount of information that can be learned from each I/O. For searching, a block input reduces the choices for the target position of the element by a factor of B, thus learning $\log B$ *bits of information*. For sorting, a block input learns up to $\log \binom{M}{B} = \Theta(B \log(M/B))$ bits (obtained by counting the ways that an incoming block can intersperse with elements resident in main memory). Theorem 29 demonstrates that in the batched predecessor problem, the optimal, unbounded-preprocessing algorithm learns B bits per I/O, more than for searching but less than for sorting.

The following theorem captures the tradeoff between the searching and preprocessing: at one end of the spectrum lies a B-tree ($j = 1$) with linear construction time and $\log_B n$ searching cost per element, and on the other end is the parallel binary search ($j = B$) with exponential preprocessing cost and $(\log_2 n)/B$ searching cost. This tradeoff shows that even to obtain a performance that is only twice as fast as that of a B-tree, quadratic preprocessing is necessary. To learn up to $j \log(B/j + 1)$ bits per I/O, the algorithm needs to spend $n^{\Omega(j)}$ in preprocessing.

Theorem 30 (Search-preprocessing tradeoff, I/O comparison model)**.** *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) *in $O((x \log_{B/j+1} n)/j)$ I/Os requires that* PREPROCESSING(S) *use $n^{\Omega(j)}$ blocks of space and I/Os.*

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 following lower bound:

Theorem 31 (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.*

We 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.

Finally, we consider the indexability model [30, 31], where we show:

Theorem 32 (**$r - \alpha$** 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 $\log r = \Omega\left(B \log(n/B)/\alpha^2\right)$.*

A crucial ingredient in our proof is a well-known result from extremal set theory due to Rödl [45]. Partly due to the techniques we use and partly due to the generality of this model, we do not get lower bounds for query time exceeding $Q/\sqrt{B}$, which was possible in the previous two models.

3.2 Batched Predecessor in the I/O Comparison Model

In this section we give the lower bound for when preprocessing is unrestricted. Then we study the tradeoff between preprocessing and the optimal number of I/Os.

3.2.1 Lower Bounds for Unrestricted Space/Preprocessing

We begin with the definition of a search interval.

Definition 33 (*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.

Proof of Theorem 29 (Lower Bound). 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 $\log x$ 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 is $\Omega(x \log(n/xB))$. Along with the scan bound to output the answer, the minimum number of block transfers required to solve the problem is $\Omega\left(\frac{x}{B} \log \frac{n}{xB} + \frac{x}{B}\right)$. □

We devise a matching algorithm (assuming $B \log n < M$), which has $O(n^B)$ preprocessing cost. This algorithm has huge preprocessing costs but establishes that the lower bound from Theorem 29 is tight.

Proof of Theorem 29 (Upper Bound). 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.

In the preprocessing phase, the algorithm produces all $\binom{n}{B}$ possible blocks. The algorithm also constructs a perfectly balanced binary search tree T on S. The former takes at most $B\binom{n}{B}$ I/Os, which is $O(n^B)$, while the latter has a linear cost. The $\binom{n}{B}$ blocks are laid out in a lexicographical order in external memory, and it takes $B \log n$ bits to address the location of any block. □

3.2.2 Preprocessing-Searching Tradeoffs

We give a lower bound on the space required by the batched predecessor problem when the budget for searching is limited. We prove Theorem 30 by proving Theorem 35.

Definition 34. 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 35. *For $x \leq n^{1-\varepsilon}$ ($0 < \varepsilon \leq 1$) and a constant $\gamma > 0$, any algorithm that solves* BATCHEDPRED(Q, S) *in at most $(\gamma x \log n)/(j \log(B/j + 1)) + x/B$ I/Os requires at least $\left(\varepsilon j n^{\varepsilon/2}/2e\gamma B\right)^{\varepsilon j/2\gamma}$ I/Os for preprocessing in the worst case.*

Proof: The proof is by a deterministic adversary argument. 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 $\log x \leq (1 - \varepsilon) \log n$ bits of information. Each element is additionally given half of the number of bits that remain to be learned. This leaves another $T \geq (\varepsilon x \log n)/2$ total bits yet to be discovered. As in the proof of Theorem 29, 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 36. *To learn T bits in at most $(\gamma x \log n)/(j \log(B/j+1))$ I/Os, there must be at least one I/O in which the algorithm learns at least $(j \log(B/j + 1))/a$ bits, where $a = 2\gamma/\varepsilon$.*

If multiple I/Os learn at least $(j \log(B/j+1))/a$ 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 37. *The maximum number of bits an I/O can learn while parallelizing d elements is $d \log(B/d + 1)$.*

Lemma 38. *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 $(j/a - 1) \log (B/(j/a - 1) + 1)$ bits. For all $a \geq 1$ and $j \geq 2$, $\frac{j}{a} \log \left(\frac{B}{j} + 1\right) > \left(\frac{j}{a} - 1\right) \log \left(\frac{B}{j/a-1} + 1\right)$. 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 $q_1, \ldots, q_{j/a}$.

Lemma 39. *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 $(1 - \varepsilon/2) \log n$ bits, $(\varepsilon \log n)/2$ of which were given for free after the initial $(1 - \varepsilon) \log n$. 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. We obtain the following:

Lemma 40. *The number of distinct choices for V at the time of parallelization is at least $n^{j\varepsilon/2a}$.*

Lemma 41. *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 $p_k \in S_k$.*

If the algorithm did not preprocess a block for each vector choice, the adversary could scan all blocks, find a vector for which no block exists, and assign those search intervals to $q_1, \ldots, q_{j/a}$, thus avoiding parallelization.

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

As a consequence, the minimum number of blocks the algorithm needs to preprocess is at least $n^{j\varepsilon/2a} / \binom{B}{j/a} \geq \left(n^{\varepsilon/2}/(eaB/j)\right)^{j/a}$. Substituting for the value of a, we get that the minimum preprocessing is at least $\left(\varepsilon j n^{\varepsilon/2}/2e\gamma B\right)^{\varepsilon j/2\gamma}$.

Algorithms.

An algorithm that runs in $O((x \log n)/j \log(B/j + 1) + x/B)$ 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.

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

Here we analyze the batched predecessor problem in the I/O pointer-machine model. 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. Before proving our theorem, we briefly describe the model.

I/O pointer machine model. The I/O pointer machine model [48] is a generalization of the pointer machine model introduced by Tarjan [51]. Many results in range reporting have been obtained in this model [1, 2].

To answer $\textsc{BatchedPred}(Q, S)$, 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 $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ to represent the n^k blocks and their associated directed pointers. Every algorithm that answers $\textsc{BatchedPred}(Q, S)$ 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 $\mathcal{L}(\mathcal{W})$ 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 $A \subseteq \mathcal{L}(\mathcal{W})$. 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 $\textsc{BatchedPred}(Q, S)$, 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 $f_G(W)$. In other words, the size of the graph containing nodes of W is linear in the minimum pairwise distance within W.

Lemma 43. *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 Sketch. 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 44. *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 $(u, v) \in F$

iff $\Lambda_G(u,v) \in [1,r]$. Then, H has at most $2r|V|B^r$ edges. By Turán's Theorem [50], 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.[2]

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 45. *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 [13], 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.

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 43, 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

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

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 31, where n^k was the size of the entire data structure. We then apply Lemma 44 to obtain that $|W| = \Omega(x)$.

Proof of Theorem 31. 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 $|S_h| \leq n/2$. 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 45, 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 $(u, v) \in F$ iff $\Lambda_{\mathcal{G}'}(u, v) \leq r$. Since the degree of each node in $\mathcal{G}'$ is bounded by $O(B^{\delta+1})$, by Lemma 44, 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 43, we are done. □

3.4 Batched Predecessor in the Indexability Model

This section analyzes the batched predecessor problem in the indexability model [30, 31]. This model is used to analyze reporting problems by focusing on 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.

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 $\min\{\left|\mathcal{B}'\right| : \mathcal{B}' \subseteq \mathcal{B}, A \subseteq \cup\mathcal{B}'\}$, 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.

Every lower bound in this model applies to our previous two models as well. To show the tradeoff between α and r, we use the Redundancy Theorem from [30, 46]:

Theorem 46 (Redundancy Theorem [30, 46])**.** *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 $r \geq \frac{1}{12n} \sum_{i=1}^{m} |A_i|$.*

Proof of Theorem 32. 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 [45]. 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 [45] gives us that

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

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

Theorem 47 (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 [45] proves that for a fixed k and β,

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

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

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.

Chapter 4

Bloom Filters for External Memory

4.1 Introduction

Many databases, storage systems, and network protocols maintain Bloom filters [12] in RAM in order to quickly satisfy queries for elements that do not exist in the database, in external storage, or on a remote network host.

The Bloom filter is the classic example of an approximate membership query data structure (AMQ). A Bloom filter supports insert and lookup operations on a set of keys. For a key in the set, lookup returns "present." For a key not in the set, lookup returns "absent" with probability at least $1 - \varepsilon$, where ε is a tunable false-positive rate. There is a tradeoff between ε and the space consumption. Other AMQs, such as counting Bloom filters, additionally support deletes [14, 27]. For a comprehensive review of Bloom filters, see Broder and Mitzenmacher [16].

Bloom filters work well when they fit in main memory. However, Bloom filters require about one byte per stored data item. Counting Bloom filters—those supporting insertions and deletions [27]—require 4 times more space [14]. Once Bloom filters get larger than RAM, their performance decays because they use random reads and writes, which do not scale efficiently to external storage, such as flash.

This research was supported in part by DOE Grant DE-FG02-08ER25853, NSF Grants CCF-0540897, CNS-0627645, CCF-0634793, CCF-0937829, CCF-0937833, CCF-0937854, CCF-0937860, and CCF-0937822, and Politécnico Grancolombiano.

Results

We present three alternatives to the Bloom filter (BF): the quotient filter (QF), the buffered quotient filter (BQF), and the cascade filter (CF). The QF is designed to run in RAM and the BQF and CF are designed to run on SSD. Unlike the BF, which performs many random writes, these data structures achieve good data locality, and all three support deletions.

The CF is asymptotically more efficient than the BQF at insertions, and thus performs well when the data structure grows much larger than RAM; the BQF is slightly more optimized for queries.

Our evaluation compares the QFs, BQFs, and CFs to BFs and recently proposed BF variants, including buffered Bloom filters (BBF) [18], forest-structured Bloom filters (FBF) [38], and elevator Bloom filters (EBF). For the overview of BF variants, see Section 4.2. 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.

Tables 1 and 2 present 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.)

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.

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 4.5.2 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.

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.

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

Table 1: In-RAM experimental results (operations per second).

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× 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.

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 [19], 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

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 2: On-disk experimental results (operations per second).

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 [19,28,34]. Bloom filters are also used for

deduplication [55], distributed information retrieval [47], network computing [16], stream computing [54], bioinformatics [20, 39], database querying [41], and probabilistic verification [32].

The remainder of this chapter is organized as follows. Section 4.2 describes the Bloom filter and its external-memory variants. Section 4.3 presents the quotient filter and gives a theoretical analysis. Section 4.4 presents the buffered quotient filter and cascade filter. Section 4.5 presents our experiments.

4.2 Bloom Filter and SSD Variants

This section reviews the traditional Bloom filter and its SSD variants.

A Bloom filter B 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 a bit array $B[0\,..\,m-1]$ and k hash functions $h_i : \mathcal{U} \to \{0, \ldots, m-1\}$, where $1 \le i \le k$ and $\mathcal{U}$ 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:

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

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

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

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%.

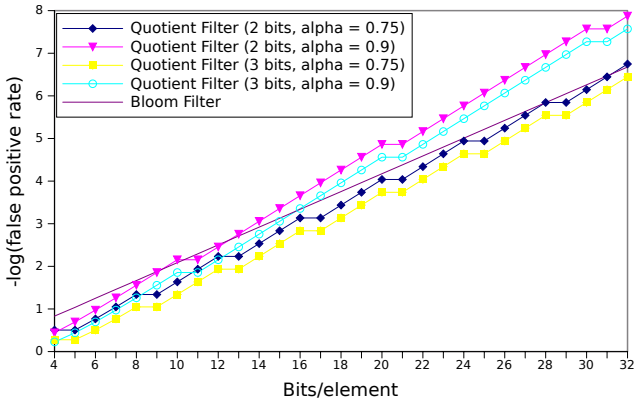


Figure 2: False positive rates for BF and QF. For typical parameters (e.g., 1% false positive rate), QF require about 20% more space than a BF. For extremely low false positive rates, QF use less space than a BF.

Figure 2 shows the BF false positive rate, assuming the optimal number of hash functions, as a function of the number of bits per element.

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. 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.

Researchers have devised several approaches to improve BF scalability:

- *Replacing magnetic disks with SSDs.* SSDs offer random read and write rates superior to those of magnetic disks. 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, 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. 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 [18], queries can also be delayed and buffered in a multithreaded

environment, but the present work 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 [18] and the closely related BloomFlash [22], use this strategy.

- *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 [38] 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.

4.3 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. Finally, we give implementation details, describe potential variations, and analyze asymptotic performance.

The QF represents a multi-set of elements $S \subseteq \mathcal{U}$ 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 $h : \mathcal{U} \to \{0, \ldots, 2^p - 1\}$ is a 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*, suggested by Knuth [36, Section 6.4, exercise 13]; see the open hash table (i.e., hash table with chaining) at the top of Figure 3. 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 $f = f_q 2^r + f_r$.

To reduce the memory required to store the fingerprints and achieve better spatial locality, the hash table is compactly stored in an array $A[0\,..\,m-1]$ of $(r+3)$-bit items, similar to that described by Cleary [21]; see Figure 3, bottom. Each slot in A stores an r-bit remainder along with three meta-data bits, which enable perfect reconstruction of the open hash table.

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.

The three meta-data bits in each slot of A work as follows. For each slot i, we maintain an $is\text{-}occupied$ 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 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.

We define a *cluster* as a sequence of one or more consecutive runs (with no

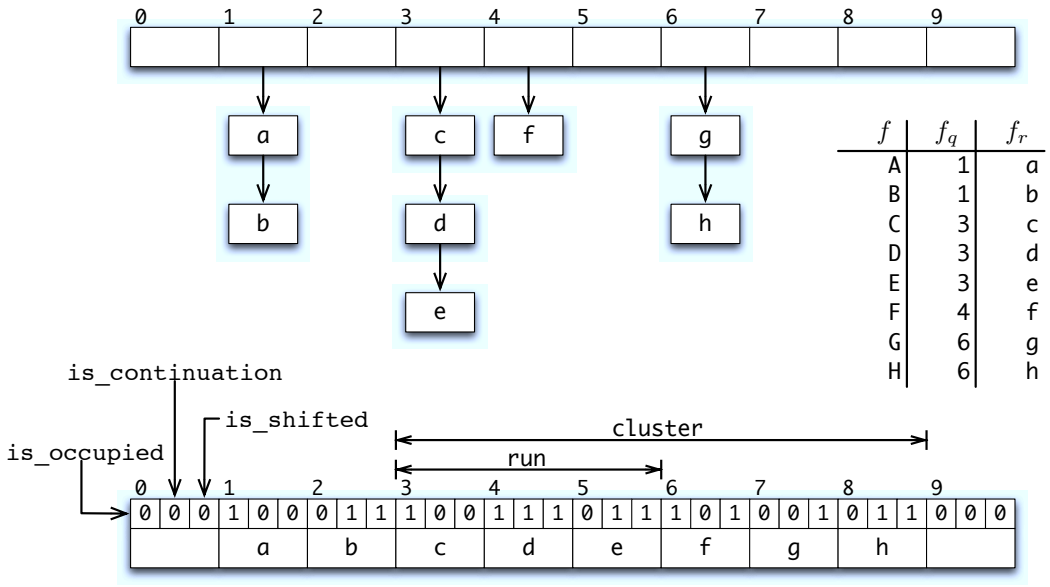


Figure 3: An example quotient filter with 10 slots along with its equivalent open hash table representation. The remainder, f_r, of a fingerprint f is stored in the bucket specified by its quotient, f_q. The quotient filter stores the contents of each bucket in contiguous slots, shifting elements as necessary and using three meta-data bits to enable decoding.

empty slots in between). A cluster is always immediately preceded by an empty slot and its first item is always un-shifted. The decoder only needs to decode starting from the beginning of a cluster. However, rather than decoding, we can perform all operations in place. Figure 4 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.

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 $x \in S$ ($h(x) = h(x')$). We refer to this event as a *hard*

```
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 4: Algorithm for checking whether a fingerprint f is present in the QF A.

collision. Assuming h generates outputs uniformly and independently distributed in $\{0, \ldots, 2^p - 1\}$, the probability of a hard collision is given by

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

Figure 2 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 2 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.

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.

Fact. 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

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

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 5 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.

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

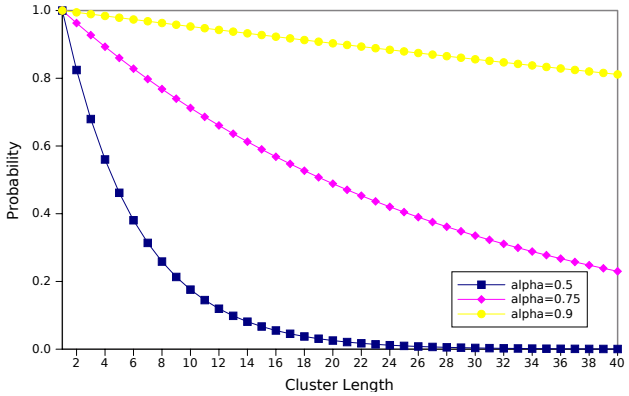


Figure 5: Distribution of cluster sizes for 3 choices of α.

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 [14, 27] 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.

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 $is\text{-}shifted$ bit: if $\neg\, is\text{-}shifted(A[i])$, 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 $is\text{-}shifted$ 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.

4.4 Quotient Filters on Flash

In this section we give two AMQs designed for SSD, the buffered quotient filter and the cascade filter. Both structures use the QF as a building block. The false positive rates of these structures are exactly the same as that of a single QF storing all of the elements.

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 4.2, performance degrades as the filter-to-RAM size increases.

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 COLA [9]; see Figure 6. The CF maintains an in-memory QF, Q_0. In addition, for RAM of size M, the CF maintains $\ell = \log(n/M) + O(1)$ 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.

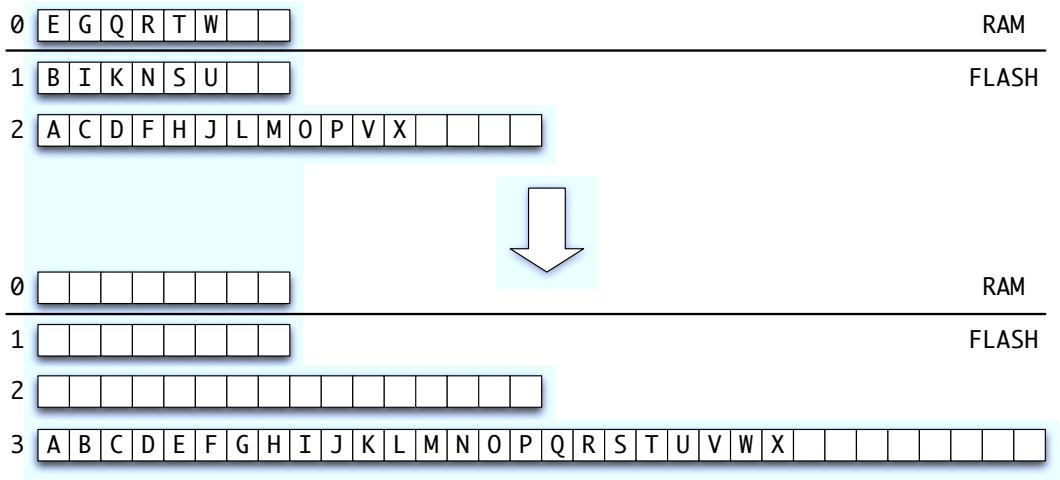


Figure 6: Merging QFs. Three QFs of different sizes are shown above, and they are merged into a single large quotient filter below. The top of the figure shows a CF before a merge, with one QF stored in RAM, and two QFs stored in flash. The three QFs above have all reached their maximum load factors (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.

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 $O(\log(n/M))$ block reads, and an insert requires only $O((\log(n/M))/B)$ amortized block writes/erases, where B is the natural block size of the flash. Typically, $B \gg \log(n/M)$, 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 [9]. This deamortization removes delays caused by merging large QFs.

4.5 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 fanouts 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 occured.

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

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

This section presents the experimental comparison of QF to the BF, with varying false positive rates.

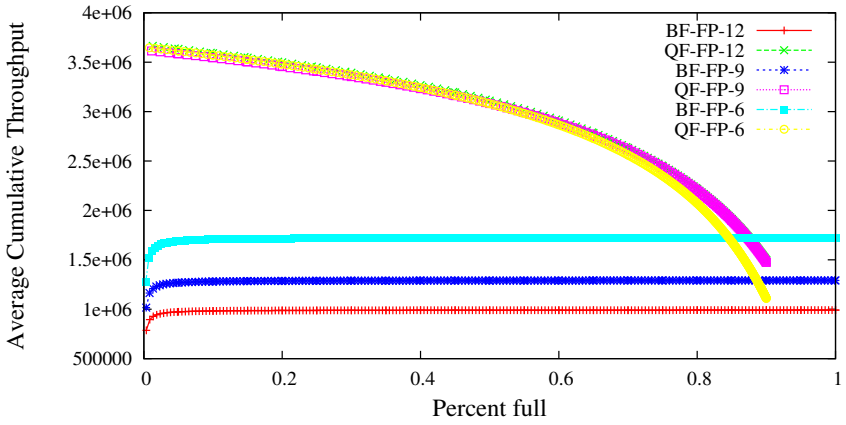
Both data structures were given 2GB of space in RAM and we tested their performance on three false positive rates: $1/64$, $1/512$, and $1/4096$.

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.

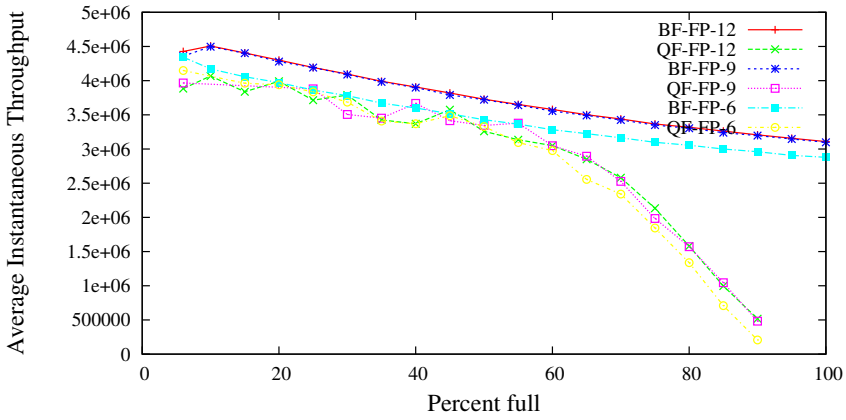
Results Figure 7 shows the insertion, random lookup, and successful lookup throughputs of the BF and quotient filter.

The quotient filter substantially outperforms the BF on insertions until the quotient filter is 80% full. The BF insertion throughput is independent of its ocupancy, 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.

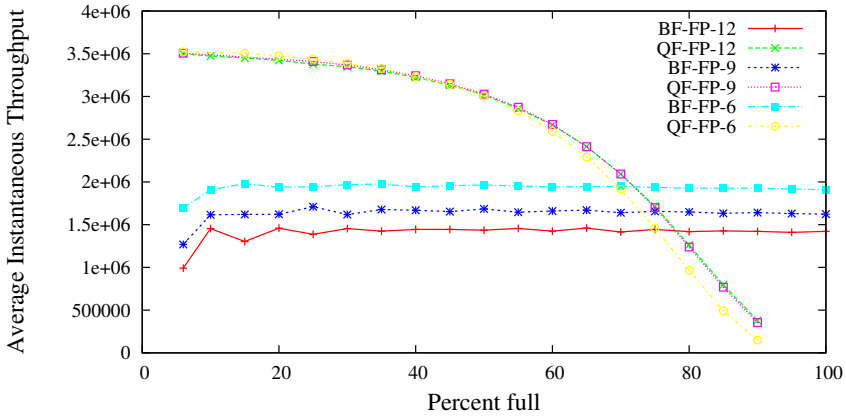
The quotient filter matches the BF random lookup performance until about



(a) Cummulative inserts



(b) Uniform random lookups



(c) Successful lookups

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

	Capacity	
FP rate	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 3: 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.

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.

Table 3 shows the capacity of the BFs and quotient filters in our experiments. As predicted in Figure 2, 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.

4.5.2 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.

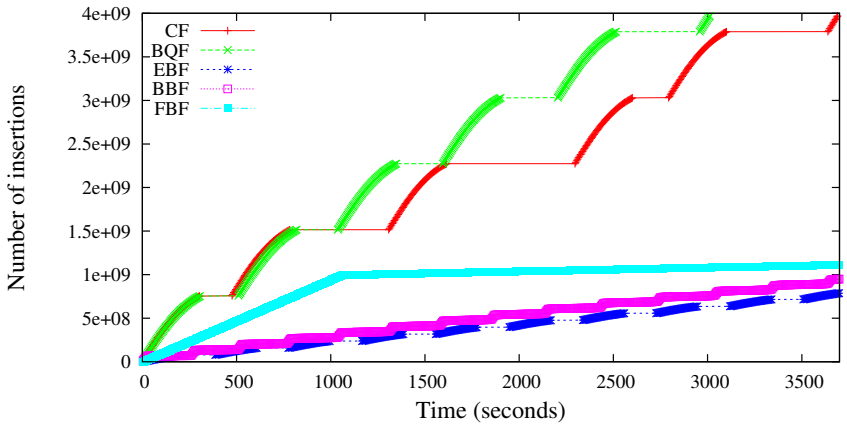
The second experiment, using a RAM-to-filter ratio of $1 : 24$ can be thought of as a "large" experiment. In this case all data structures employ 2GB of RAM buffer, and a 48GB on-disk data structure. As in the previous experiment, 1GB is set aside for the page cache. In this configuration, the CF and 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 Figure 8 shows 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 2 for the overall throughputs.

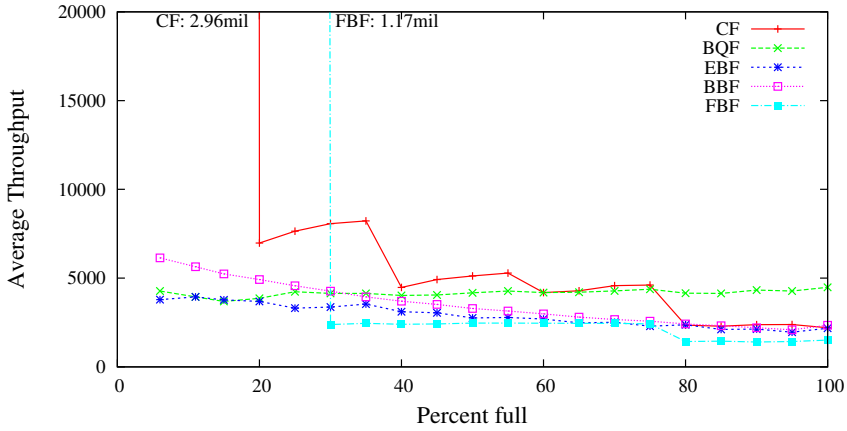
Figures 8(a), 8(b) and 8(c): Small disk experiment In 8(a), the staircase pattern of the CF is due to the merges of the small QFs into a larger quotient filter. The stalls in the BQF performance are due to flushing of the in-RAM quotient filter to the on-disk quotient filter. In 8(b) and 8(c), the lookup 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.

Figures 8(d), 8(e) and 8(f): Large disk experiment In 8(d), the cascade filter outperforms the buffered quotient filter. In 8(e) and 8(f) 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.

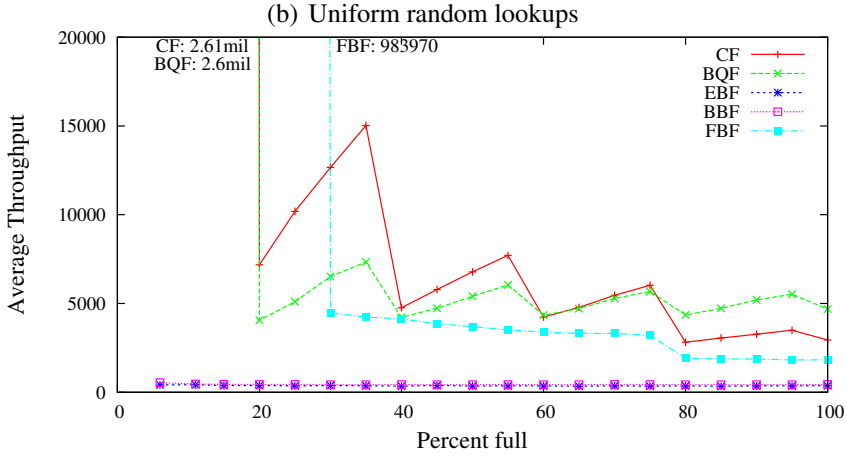
In this experiment, the other three data structures all completed less than 10 percent of the experiment. Figure 8(d) shows their cumulative throughput for the



(a) Inserts



(b) Uniform random lookups



(c) Successful lookups

first 35,000 seconds, but Figures 8(e) and 8(f) 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 $(O(\log(n/M)/B)$ 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 $\log(n/M)$ 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 [9]. 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 2 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 $\log(n/M)$ 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 betwen 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 $4 \approx \log(24)$ active levels, and hence its lookup throughput should be between 1 and 4 times slower than the disk's random read throughput. Figures 8(e) and 8(f) 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.

4.5.3 Cascade Filter: Insert/Lookup Tradeoff

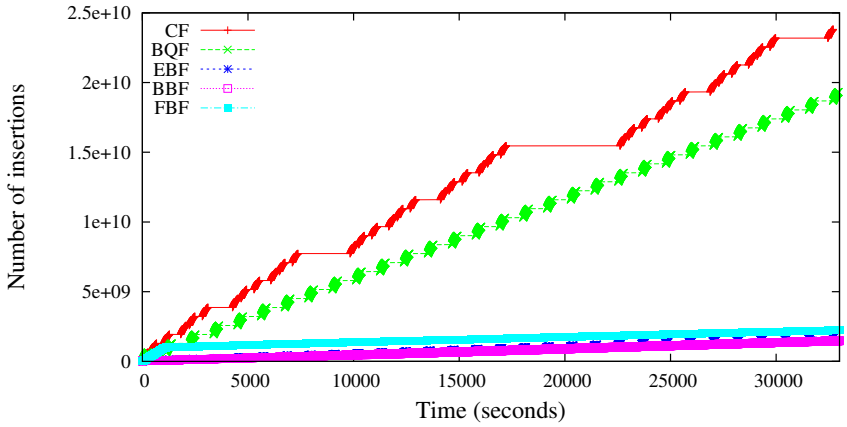
To investigate the effect of the fanout in the CF, we inserted 12 billion items into CFs with the same basic configuration as before: a 2GB buffer and a false positive rate of $1/4096$. After inserting all 12 billion elements, we performed lookups for 60 seconds. We repeated this experiment with CFs for fanouts of 2, 4, and 16. Figure 9 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 9, even a fanout of 16 exceeds the insert performance of all the BF based data structures in our evaluations.

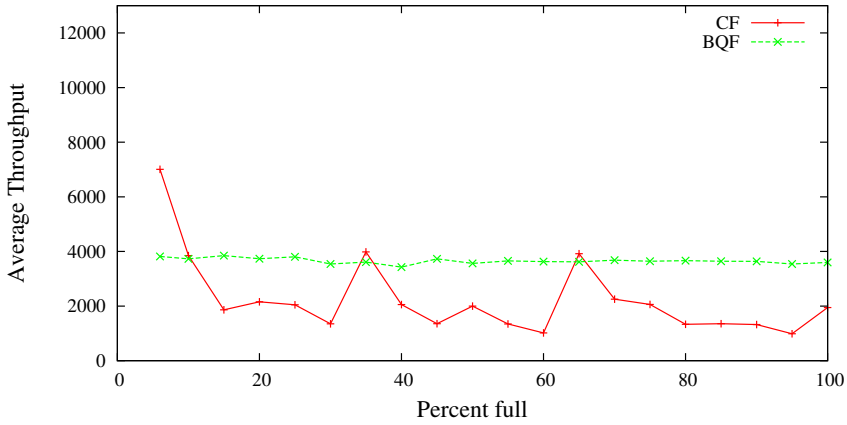
4.5.4 Evaluation Summary

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.

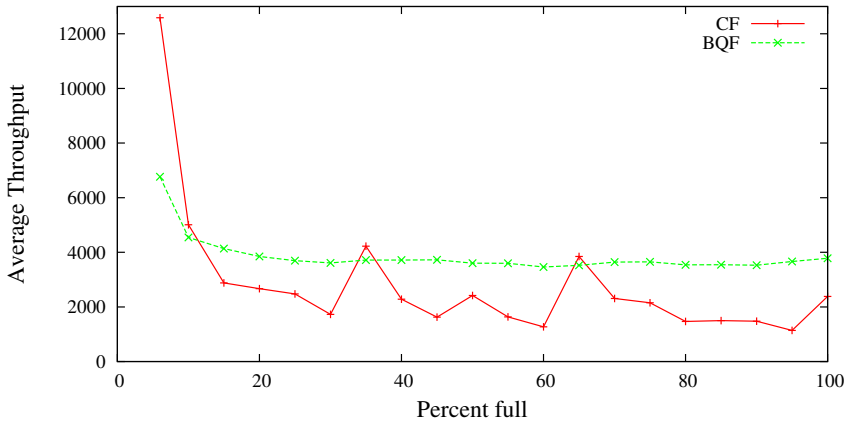
The CF was the most scalable data structure in our experiments. As filter-to-RAM ratio grows, the CF outperforms the BQF. With ratios larger than 24, we expect the CF and the BQF performance to further diverge. When the ratio between the filter and the RAM buffer grows too large, then the flushes that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



(d) Inserts



(e) Uniform random lookups



(f) Successful lookups

Figure 8

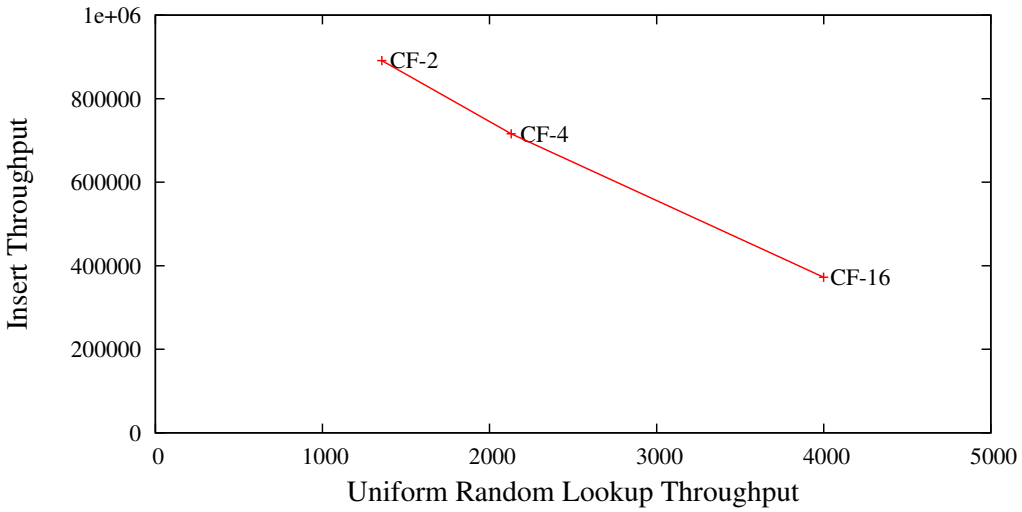


Figure 9: The Cascade Filter Insert/Lookup Tradeoff: Varying fanouts. Higher fanouts foster better lookup performance; lower fanouts optimize the insertion performance.

Bibliography

[1] 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.

[2] 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.

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

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

[5] L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter. On sorting strings in external memory (extended abstract). In *Proc. 29th Annual ACM Symposium on Theory of Computing (STOC)*, pages 540–548, 1997.

[6] L. Arge, M. Knudsen, and K. Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In *In Proc. Workshop on Algorithms and Data Structures (WADS), LNCS 709*, pages 83–94. Springer-Verlag, 1993.

[7] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of i/o-efficient algorithms for multidimensional batched searching problems. In *Proc. 9th Annual ACM-SIAM Aymposium on Discrete Algorithms (SODA)*, 1998.

[8] M. Ben-Or. Lower bounds for algebraic computation trees. In *Proc. 15th Annual ACM Symposium on Theory of Computing (STOC)*, pages 80–86, 1983.

[9] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *SPAA*, pages 81–92, 2007.

[10] M. A. Bender, H. Hu, and B. C. Kuszmaul. Performance guarantees for B-trees with different-sized atomic keys. In *Proc. 29th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 305–316, 2010.

[11] D. K. Blandford and G. E. Blelloch. Compact dictionaries for variable-length keys and data with applications. *ACM T. Algorithms*, 4(2), 2008.

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

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

[14] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting bloom filters. In *Proceedings of the 14th Conference on Annual European Symposium - Volume 14*, ESA'06, pages 684–695, London, UK, UK, 2006. Springer-Verlag.

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

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

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

[18] 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.

[19] 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.

[20] Y. Chen, B. Schmidt, and D. L. Maskell. A reconfigurable Bloom filter architecture for BLASTN. In *ARCS*, pages 40–49, 2009.

[21] J. G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE T. Comput.*, 33(9):828–834, 1984.

[22] B. Debnath, S. Sengupta, J. Li, D. Lilja, and D. Du. Bloomflash: Bloom filter on flash-based storage. In *ICDCS*, pages 635–644, 2011.

[23] G. Diehr and B. Faaland. Optimal pagination of B-trees with variable-length items. *Commun. ACM*, 27(3):241–247, Mar 1984.

[24] W. Dittrich, D. Hutchinson, and A. Maheshwari. Blocking in parallel multisearch problems (extended abstract). In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 98–107, New York, NY, USA, 1998. ACM.

[25] A. Elmasry. Distribution-sensitive set multi-partitioning. In *1st International Conference on the Analysis of Algorithms*, 2005.

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

[27] 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.

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

[29] M. T. Goodrich, J.-J. Tsay, N. C. Cheng, J. Vitter, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry, 1993.

[30] 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. *JOURNAL OF THE ACM*, 49:35–55, 2002.

[31] J. M. Hellerstein, E. Koutsoupias, and C. H. Papadimitriou. On the analysis of indexing schemes. In *Proc. 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 249–256, Tucson, Arizona, 12–15 May 1997.

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

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

[34] 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.

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

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

[37] L. L. Larmore and D. S. Hirschberg. Efficient optimal pagination of scrolls. *Commun. ACM*, 28(8):854–856, Aug. 1985.

[38] G. Lu, B. Debnath, and D. H. C. Du. A forest-structured bloom filter with flash memory. In *MSST*, pages 1–6, 2011.

[39] K. Malde and B. O'Sullivan. Using Bloom filters for large scale gene sequence analysis in Haskell. In *PADL*, pages 183–194, 2009.

[40] E. M. McCreight. Pagination of B*-trees with variable-length records. *Commun. ACM*, 20(9):670–674, Sep 1977.

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

[42] J. I. Munro and P. M. Spira. Sorting and searching in multisets. *SIAM J. Comput.*, 5(1):1–8, 1976.

[43] A. P. Pinchuk and K. V. Shvachko. Maintaining dictionaries: Space-saving modifications of b-trees. In J. Biskup and R. Hull, editors, *Database Theory ICDT '92*, volume 646 of *Lecture Notes in Computer Science*, pages 421–435. Springer Berlin Heidelberg, 1992.

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

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

[46] V. Samoladas and D. P. Miranker. A lower bound theorem for indexing schemes and its application to multidimensional range queries. In *Proc. ACM Symposium on Principles of Database Systemsi (PODS)*, pages 44–51, 1998.

[47] 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.

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

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

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

[51] 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.

[52] Tokutek Inc. TokuDB. `http://www.tokutek.com/`, 2013.

[53] J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv.*, 33(2):209–271, June 2001.

[54] Z. Yuan, J. Miao, Y. Jia, and L. Wang. Counting data stream based on improved counting Bloom filter. In *WAIM*, pages 512–519, 2008.

[55] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST*, pages 18:1–18:14, 2008.