A Streaming Algorithms Approach to Approximating Hit Rate Curves

by

Zachary Drudi

B. Math, University of Waterloo, 2011

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL
STUDIES
(Computer Science)

The University Of British Columbia
(Vancouver)

October 2014

Abstract

In this work, we study systems with two levels of memory: a fixed-size *cache*, and a backing *store*, each of which contain *blocks*. In order to serve an IO request, the block must be in the cache. If the block is already in the cache when it is requested, the request is a *cache hit*. Otherwise it is a *cache miss*, and the block must be brought into the cache. If the cache is full, a block must be evicted from the cache to make room for the new block. A *replacement policy* determines which block to evict. In this work, we consider only the *LRU* policy. An LRU cache evicts the block which was least recently requested.

A *trace* is a sequence of blocks, representing a stream of IO requests. For a given trace, a *hit rate curve* maps cache sizes to the fraction of hits that such a cache would achieve on the trace. Hit rate curves have been used to design storage systems, partition memory among competing processes, detect phases in a trace, and dynamically adjust heap size in garbage-collected applications.

The first algorithm to compute the hit rate curve of a trace over a single pass was given by Mattson et al. in 1970. A long line of work has improved on this initial algorithm. The main contribution of our work is the presentation and formal analysis of two algorithms to approximate hit rate curves. Inspired by recent results in the streaming algorithms community on the distinct elements problem, we use memory efficient *probabilistic counters* to estimate the number of distinct blocks in a subsequence of the trace, which allows us to approximate the hit rate curve using sublinear space. We also formally state some variants of the hit rate curve approximation problem which our algorithms solve, and derive lower bounds on the space complexity of these problems using tools from communication complexity.

Preface

This thesis is the formal analysis complement of the more systems-oriented work published as J. Wires, S. Ingram, N. J. A. Harvey, A. Warfield, and Z. Drudi; Characterizing storage workloads with counter stacks; published in *11th USENIX Symposium on Operating Systems Design and Implementation*. The ideas behind the algorithms described in chapter 3 are common to both works, and were developed together by the above authors.

The analysis in chapters 3, 4, and 5 is original, unpublished work. The implementation used for the experiments in chapter 6 is the same as that analyzed in the published paper mentioned above.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank my supervisor, Nick Harvey, for his encouragement, optimism, and guidance. He is an inexhaustible source of ideas, and his enthusiasm helped me through many an unproductive lull.

I am indebted to Andrew Warfield for encouraging my interest in systems, and taking me on as an intern at Coho Data, where I worked from February to August of 2014.

I would like to thank the remaining members of the team at Coho Data, Stephen Frowe Ingram and Jake Wires. Stephen was the friend at my side who kept me sane whenever I fell into a seemingly bottomless hole of circular dependencies, misconfigurations, and versioning problems. Jake Wires tolerated my absolute ignorance of the tools and techniques used in the systems world, and taught me many practical tricks.

Finally, I would like to thank my family for their long-distance but robust support.

Chapter 1

Introduction

Most computer systems use multiple levels of storage. CPU caches permit very efficient read and write operations, speeding the execution of processes, but are limited in size by their high cost. Main memory is slower, cheaper and larger, but again too expensive to provide all the storage requirements of a system. Magnetic disks or flash drives provide very high capacity storage, but at significantly lower speeds.

Given this memory hierarchy, much work has been done to make the most efficient use of available resources. To simplify the discussion, we assume there are only two levels of memory: a fast, small *cache*, and a larger backing *store*. Both the cache and the store contain data organized into fixed size *blocks*. During execution, a process issues block *requests*. If a requested block is in the cache, we have a cache *hit*, and the process can continue execution. Otherwise, a cache *miss* occurs, and the requested block must be brought into the cache before the process can resume. If the cache is full, a block must be evicted from the cache into the store. A *replacement policy* determines how this block is chosen. To simplify matters, in the following we will just consider the trace, which is the sequence of block requests made by a process, and ignore the process itself.

Belady [4] did early, foundational work on comparing different replacement policies by simulation on request traces. The most widely used replacement policy is *LRU*. LRU stands for Least Recently Used. When a cache miss occurs, the LRU policy evicts the block whose last request occurred before the last request of

any other block in the cache. Although implementing LRU involves dynamically reordering blocks on every request and is somewhat expensive, there are variants of LRU that are both simple and efficient to implement in practice, and are widely used.

Given a replacement policy such as LRU and a trace, the hit rate curve is a function mapping cache sizes to the fraction of hits for that cache size.

Mattson et al. [23] introduced the notion of a stack algorithm, which characterizes a class of replacement policies (including LRU), and gave a simple algorithm that can compute the hit rate curve for a fixed stack algorithm in a single pass over the trace. The algorithm computes the stack distance of each request, and tallies them to produce the hit rate curve after the trace has been processed. The *stack distance* of a request to a block b is the number of distinct blocks that occurred in the trace between the current request to b and the last request to b. The stack distance of a request corresponds to the minimum size LRU cache that would produce a hit on the request. If there is no such earlier request to b, the stack distance is defined to be infinite, as a request to a new block is necessarily a miss regardless of cache size. In the Mattson et al. algorithm, a linked list maintains the blocks that have been previously requested. To process a new request for a block b, the linked list is linearly searched for b. If b is found, its position in the linked list is the stack distance of the request. The block b is then removed from the linked list, and reinserted at the head of the list. If b is not found in the list, then this is the first request for b, and the request is assigned a stack distance of infinity. After the trace is processed, the value of the hit rate curve for a given cache size C is simply the number of stack distances less than or equal to C, normalized by the total number of requests.

While simple, this algorithm is quite inefficient. If the trace has length m and the blocks come from a store of size n, each request will take $O(n)$ time to process. A series of more efficient algorithms has been introduced by authors interested in CPU caches ([5], [26], [28], [1], [13], [27], [34], [15], [25]) . The traces studied are produced by a processor executing a given program, and each request corresponds to a memory location. The cache is the CPU level cache, while the backing store is main memory. For these traces, m is typically on the order of billions, even for short program executions of a minute or less, while n is a fraction of main

memory. In this setting, online algorithms which process each request as it is made by the process are attractive because the trace doesn't need to be recorded. These algorithms must be very efficient or they will impose unacceptable overhead on program execution. Accordingly, past work has frequently focused on time complexity, while space complexity has received comparatively little attention.

In this work, we approach the problem motivated by storage level traces. These traces correspond to requests made to the file system. In this setting, the backing store is the permanent storage medium of the system, which may be a magnetic disk or flash drive. The number of distinct blocks in the backing store, n, may exceed main memory, so we are very sensitive to space usage. Our contribution is the presentation and analysis of approximation algorithms to compute hit rate curves using sublinear space. The main ingredient in our approach is a line of work from the streaming algorithms community on algorithms to count the number of distinct elements in a stream using sublinear space ([16], [2], [3], [14], [17], [21]). Using these counting algorithms as a black box, we can estimate the stack distances of requests in the trace, and produce an approximate hit rate curve. Unlike prior work on approximate hit rate curves, we derive precise error bounds for our algorithms. Furthermore, using results from communication complexity we give lower bounds for the space complexity of algorithms solving several variants of the hit rate curve problem.

Chapter 2

Related Work

Researchers have done a substantial amount of work exploring improved versions of the Mattson algorithm first proposed in [23], as well as entirely new techniques for computing hit rate curves. Here we discuss some of these contributions. We first discuss exact algorithms, and then we describe some recent approximation algorithms.

2.1 Exact Algorithms

Bennett and Kruskal [5] improved on the original Mattson algorithm by replacing the linear list with a k-ary tree. The leaves of the tree represent requests, and store 1 or 0 depending on whether the request corresponds to the last request of the block. Interior nodes store the sum of their children. The last time each block was requested with respect to a given point in the trace is stored in a hash table, which is updated after processing each request. The stack distance of a request is computed by looking up the block to find the time it was last requested, and finding the sum of the corresponding subtree. The time complexity to process a request is thus $O(\log(m))$. This general framework of using a hashtable to store the last access time of each block and an auxilliary tree-based datastructure to compute the stack distance was used by many later authors ([26], [28], [1], [13], [25]).

Almási, Caşcaval and Padua [1] proposed some variants of the Bennett and Kruskal algorithm, counting the number of "holes" (leaves containing 0, instead of

1) instead of the number of 1s in the leaves, and using an interval tree data structure (implemented using either an AVL or red-black tree) to store the locations of the holes. Their modifications were more efficient in practice, although they had the same asymptotic complexity. Sugumar and Abraham [28] used splay trees instead.

Niu et al. presented a parallel algorithm [25]. Given p processors, the trace is split into p chunks, with the kth chunk going to the kth processor. The computation is performed in a series of rounds. In each round, every processor uses a variant of the sequential Mattson algorithm to process its chunk of the trace, storing the stack distances in a local copy of the stack distance histogram. Requests to blocks which occur for the first time in a given chunk may have occured in a previous chunk in the trace. To determine their true stack distances, these requests are added to a queue, and passed to the previous processor at the end of the round. After at most $O(p)$ rounds, all stack distances have been found, and the local histograms are aggregated to produce the complete histogram.

2.2 Approximation Algorithms

By dynamically resizing the tree, Ding and Zhong [13] approximate the hit rate curve by approximating the stack distance of each request with multiplicative error ε. For each block, their algorithms record the time range that the last access belongs to instead of the exact time. The tree structure itself only consumes $O(\log(n)/\varepsilon)$ space, but they still require a hash table mapping each block to the last time in the trace it was requested, requiring $O(n)$ space. The trace can be preprocessed to include this information with each request, at the cost of $O(m \log n)$ time.

Shen et al. augment the algorithm of Ding and Zhong with sampling [34]. The trace is divided into alternating sampling and hibernating intervals. The hash table of last access times is always updated, but the stack distance of a given request is only recorded if the latest prior access occurred in a sampling interval. They also tweak the tree bookkeeping of Ding and Zhong's algorithm by merging all requests from a given hibernating interval into a single leaf node, and record only the number of distinct blocks requested in the interval.

Shen et al. use a probabilistic model to estimate the stack distance histogram [27].

Using a histogram of time distances (the number of requests between two successive requests to the same block) and the number of distinct blocks accessed in the trace, they estimate the stack distance histogram. Their experimental results show good accuracy, but they do not provide analytical error bounds.

Eklov and Hagersten combine trace sampling with approximation of stack distances from reuse distances [15]. Building on their work in [6], their method finds the average stack distance among all reuse windows of a fixed size by using the distribution of forward reuse distances.

Xiang et al. [33], building on earlier work [32], use an approach based on averaging. They define the footprint of a window of the trace to be the number of distinct blocks accessed during the window. For a given window size w, the average footprint is the average number of distinct blocks accessed in windows of size w. They calculate the average footprint of a logarithmic scale of window sizes by recording the reuse distances of requests. Inverting the average footprint function yields an approximation to the unnormalized hit rate curve. While their approach is fast and accurate, their memory requirements are linear in n, and it is unclear if their averaging technique maintains the precise error bounds of an earlier, slower approach measuring all footprints [32].

Chapter 3

Presentation and Analysis of Algorithms

In this chapter we describe our main algorithms and characterize their space usage and accuracy guarantees. We begin by establishing some notation, and introducing the *working set data structure* abstraction. In section 3.3, we present deterministic algorithms for computing hit rate curves. In section 3.4 we modify these using work from the streaming algorithms community. In Theorem 2, our main result of this chapter, we show that the resulting algorithm computes the hit rate curve with additive error ε, and in Corollary 6 we show it uses only $O(\text{poly}(1/\varepsilon, \log(nm)))$ space.

3.1 Preliminaries

To state our results precisely, let us fix some notation. We will use n to represent the size of the store from which blocks are drawn, and m to represent the trace length. We will identify blocks with integers from the set $[n] = \{i : 1 \leq i \leq n\}$.

The set of requested blocks between time t' and strictly before time t is:

$$B(t',t) \;=\; \left\{\, b_i \,:\, i \in [m] \text{ and } t' \leq i < t \,\right\}.$$

At time t, the most recent request for block b_t occurred at time

$$R(t) \;=\; \max\{\, x : x < t \text{ and } b_x = b_t \,\}.$$

We define $R(t) = -\infty$ if b_t was not requested before time t. The stack distance of the request at time t is

$$D(t) \;=\; \begin{cases} |B(R(t),t)| & (\text{if } R(t) > -\infty) \\ \infty & (\text{otherwise}) \end{cases}$$

A cache of size k has a hit at time t if and only if $D(t) \le k$. The *hit rate curve* is the function $C : [n] \to [0,1]$ where $C(k)$ is the hit rate for a cache of size k. Thus

$$C(k) \;=\; |\{\, t \in [m] : D(t) \le k \,\}|/m.$$

In this work we are concerned with computing the hit rate curve at ℓ uniformly-spaced points, where ℓ is a parameter. For simplicity, assume that $n = \ell\Delta$, where Δ $n = \ell\Delta$ is an integer. The *histogram* of D is the function $H : [\ell] \to \mathbb{N}$ where

$$H(i) \;=\; |\{\, t \in [m] : (i-1)\Delta < D(t) \le i\Delta \,\}|. \tag{3.1}$$

The fraction of requests that are hits with a cache of size $x\Delta$ is $\sum_{i=1}^{x} H(i)/m$. The hit rate curve at the desired ℓ uniformly-spaced points is

$$C(x\Delta) \;=\; \sum_{i=1}^{x} H(i)/m \qquad \forall x \in [\ell].$$

3.1.1 Algorithmic guarantees

All algorithms in Sections 3.4 and 4.2 produce a function $\hat{C}$ that satisfies

$$C\big((x-1)\Delta\big) - \varepsilon \;\le\; \hat{C}(x\Delta) \;\le\; C(x\Delta) + \varepsilon \qquad \forall x \in [\ell+1] \quad \text{(Weak-Guarantee)}$$

with high probability. In fact, the algorithms of Sections 3.4.1 and 4.2 actually satisfy the guarantee

$$C\big((x-1)\Delta\big) - \varepsilon x/\ell \;\le\; \hat{C}(x\Delta) \;\le\; C(x\Delta) + \varepsilon x/\ell \qquad \forall x \in [\ell+1]. \tag{Strong-Guarantee}$$

The algorithms of Sections 3.4.2 and 4.2 use space $\text{poly}(\ell, 1/\varepsilon, \log(nm))$.

3.2 The Working Set Data Structure

To streamline our presentation, we introduce an abstract data type called a *working set data structure*. In the following section, we will describe a single algorithm to compute hit rate curves, given an implementation of a working set data structure. Then, we will present a series of implementations of working set data structures.

The notion of a *working set* comes from Denning [11]. The working set of the trace between times t' and t is simply $B(t',t)$. As the stack distance $D(t)$ is defined in terms of $|B(R(t),t)|$, if we knew $|B(t',t)|$ for all t',t we could compute the stack distance for every request in the trace. A working set data structure gives estimates of $|B(t',t)|$, enabling a client to estimate stack distances and thus the hit rate curve.

A working set data structure supports two operations, REGISTER(t,b), which records that block b was requested at time t, and GETWORKINGSETSIZE(t), which estimates the number of distinct blocks requested since time t.

The simplest way to implement a working set data structure is to use a *counter*, which is an abstract data type that computes the number of distinct elements in a data stream. This data type supports two operations, INSERT and QUERY, which returns the number of distinct elements that were inserted. Pseudocode illustrating this is shown in Algorithm 1.

3.3 Deterministic Algorithms for Computing Hit Rate Curves

We are interested in computing the value of C at ℓ uniformly-spaced points, so we begin with Algorithm 2 which computes those values. It can be implemented in $O(m \log n)$ time and $O(n)$ space using the hash table plus tree approach of Bennett and Kruskal [5].

Algorithm 1: An implementation of a working set data structure based on abstract counters.

```
1 c ← 1
2 Function Register(t, b_t):
3     c ← ⌈t/Δ⌉
4     if t ≡ 1 (mod Δ) then
5         Create the new counter 𝒦[c]
6     for i = 1, ..., c do
7         𝒦[i].INSERT(b_t)
8 Function GetWorkingSetSize(t):
9     Return 𝒦[⌈t/Δ⌉].QUERY()
```

Algorithm 2: Algorithm for computing the hit rate curve at $\ell = n/\Delta$ uniformly-spaced points.

```
1 Input: A sequence of requests (b_1, ..., b_m) ∈ [n]^m
2 Initialize the vector H ∈ ℕ^ℓ with zeros
3 for t = 1, ..., m do
4     If D(t) is finite then increment H[⌈D(t)/Δ⌉] by 1
5 ▷ H[i] satisfies condition (3.1).
6 Output the hit rate curve values C(xΔ) = Σ_{i=1}^{x} H[i]/m for x ∈ [ℓ].
```

Next we present Algorithm 3, which is an algorithm to approximate a hit rate curve using a working set data structure. To facilitate compact implementations of that data structure, the algorithm only queries the working set size at specific times $\tau_i = (i-1)\Delta + 1$ for $i = 1, 2, \ldots$. This algorithm also allows the data structure to decline to return an estimate, in which case it returns NULL instead. Consider implementing the working set data structure using exact counters, which compute the number of distinct elements exactly, e.g., using a hash table. Then line 9 in Algorithm 3 will have

$$X_i(t) \;=\; |B(\tau_i, t)| \quad \forall i, t. \tag{3.2}$$

Let us now compare the accuracy of Algorithms 2 and 3 when exact counters are used.

Algorithm 3: An algorithm for approximating the hit rate curve at ℓ uniformly-spaced points, given an implementation $\mathscr{W}$ of a working set data structure.

```
1  Input: A sequence of requests (b_1, ..., b_m) ∈ [n]^m
2  Initialize the vector H ∈ ℕ^ℓ with zeros
3  ▷ For convenience, let τ_i denote (i−1)Δ+1
4  for t = 1, ..., m do
5      ▷ Receive request b_t
6      𝒲.REGISTER(t, b_t)
7      Let c ← ⌈t/Δ⌉
8      for i = 1, ..., c do
9          Let X_i(t+1) ← 𝒲.GETWORKINGSETSIZE(τ_i)
10     for i = 1, ..., c−1 do
11         if X_i(t+1) ≠ NULL and X_{i+1}(t+1) ≠ NULL then
12             Increment H[⌈X_i(t)/Δ⌉] by
               (X_{i+1}(t+1) − X_{i+1}(t)) − (X_i(t+1) − X_i(t))
13     Increment H[⌈X_c(t)/Δ⌉] by 1 − (X_c(t+1) − X_c(t))
14 Output the hit rate curve approximation given by C(xΔ) = Σ_{i=1}^{x} H[i]/m for
   x ∈ [ℓ].
```

Claim 1. *Let C be the hit rate curve computed by Algorithm 2. Let $\hat{C}$ be the hit rate curve computed by Algorithm 3, using Algorithm 1 with exact counters to implement $\mathscr{W}$. Then*

$$C\big((x-1)\Delta\big) \;\le\; \hat{C}(x\Delta) \;\le\; C(x\Delta) \qquad \forall x \in [\ell].$$

Proof. Let H and $\hat{H}$ respectively denote the histograms computed by Algorithms 2 and 3. Note that $b_t \notin B(t',t)$ for $t' > R(t)$ but $b_t \in B(t',t)$ for $t' \le R(t)$. Because of (3.2), we have

$$X_i(t+1) - X_i(t) \;=\; \begin{cases} 1 & (\text{if } R(t) < \tau_i \le t) \\ 0 & (\text{if } 1 \le \tau_i \le R(t)) \end{cases}$$

It follows that the increment in line 12 equals 1 if $\tau_i \le R(t) < \tau_{i+1}$ and otherwise it equals zero. Similarly, the increment in line 13 equals 1 if $\tau_c \le R(t)$. At most

one of these conditions can hold, so for each value of t, Algorithm 3 increments at most one entry of $\hat{H}$. Specifically, if $R(t)$ is finite then the algorithm increments $\hat{H}[\lceil X_{i^*}(t)/\Delta \rceil]$ where $i^* = \lceil R(t)/\Delta \rceil$.

When $R(t)$ is finite, we have $\tau_{i^*} \leq R(t) < \tau_{i^*+1}$. Since $X_{i^*}(t) = |B(\tau_{i^*}, t)|$ and $D(t) = |B(R(t), t)|$, we derive

$$X_{i^*+1}(t) \leq D(t) \leq X_{i^*}(t). \tag{3.3}$$

We also have

$$\begin{aligned} X_{i^*}(t) - X_{i^*+1}(t) \;&=\; |B(\tau_{i^*}, t)| - |B(\tau_{i^*+1}, t)| \\ =\; |B(\tau_{i^*}, t) \setminus B(\tau_{i^*+1}, t)| \;&\leq\; |B(\tau_{i^*}, \tau_{i^*+1})| \;\leq\; \Delta. \end{aligned} \tag{3.4}$$

So by 3.3 and 3.4 we have

$$\left\lceil \frac{X_{i^*}(t)}{\Delta} \right\rceil - 1 \;\leq\; \left\lceil \frac{D(t)}{\Delta} \right\rceil \;\leq\; \left\lceil \frac{X_{i^*}(t)}{\Delta} \right\rceil.$$

Algorithm 2 increments $H[\lceil D(t)/\Delta \rceil]$, whereas Algorithm 3 increments $\hat{H}[\lceil X_{i^*}(t)/\Delta \rceil]$. So

$$\underbrace{\sum_{i=1}^{x} \hat{H}[i]}_{\hat{C}(x\Delta)} \;\leq\; \underbrace{\sum_{i=1}^{x} H[i]}_{C(x\Delta)} \;\leq\; \underbrace{\sum_{i=1}^{x+1} \hat{H}[i]}_{\hat{C}\left((x+1)\Delta\right)}.$$

Rearranging this yields the desired inequality. □

3.4 A Streaming Algorithm for Approximating Hit Rate Curves

In this section we design an improved working set data structure using ideas from streaming algorithms. The working set data structure used in Claim 1 is not efficient because it uses exact counters. Our main idea is to use, as a black box, a streaming algorithm for estimating distinct elements, which we will call a *probabilistic counter*.

Each probabilistic counter has two parameters, n and α. The INSERT operation

takes a value $x \in [n]$ and the QUERY operation reports a value v satisfying

$$|S| \;\le\; v \;\le\; (1+\alpha)|S|. \tag{3.5}$$

An optimal probabilistic counter was developed by Kane et al. [21]. It ensures that (3.5) holds with high probability for $\text{poly}(m)$ queries, and each instantiation uses only $O\big((1/\alpha^2 + \log n)\log m\big)$ bits of space. In practice, the HyperLogLog counter [17] is very simple and has excellent empirical performance.

We assume three simple consistency properties of a counter.

P1: Two consecutive calls to QUERY (without any intervening insertions) return the same value.

P2: Reinserting an item that was previously inserted does not change the value of QUERY.

P3: The values returned by QUERY do not decrease as more elements are inserted.

3.4.1 Implementing Algorithm 3 using probabilistic counters

Now we consider the working set data structure of Algorithm 1 implemented using the optimal probabilistic counter of Kane et al. [21] with accuracy parameter $\alpha = \varepsilon\Delta/n = \varepsilon/\ell$. We will analyze Algorithm 3 with this working set data structure. The data structure will create $\lceil m/\Delta \rceil$ counters, each of which uses $s = O\big((1/\alpha^2 + \log n)\log m\big)$ bits of space. So the total space usage is $O(ms/\Delta)$ bits. In Section 3.4.2 we will modify the data structure to "prune" redundant counters, which reduces the space to $O(\ell s/\varepsilon)$ bits. $\alpha = \varepsilon/\ell$

The following theorem compares the accuracy guarantee of Algorithm 3 using Algorithm 1 with either exact or probabilistic counters.

Theorem 2. *Let C and $\hat{C}$ respectively refer to the hit rate curves produced using exact and probabilistic counters. Then*

$$C\big((x-1)\Delta\big) - 2\alpha x \;\le\; \hat{C}(x\Delta) \;\le\; C(x\Delta) + 2\alpha x \qquad \forall x \in [\ell+1]. \tag{3.6}$$

Furthermore, $\hat{C}$ satisfies the (Strong-Guarantee) *condition on page 9 with respect to the true hit rate curve.*

Proof. Let H and X_i refer to the quantities using exact counters and let $\hat{H}$ and $\hat{X}_i$ refer to the corresponding quantities using probabilistic counters. We require the following claim:

Claim 3. *For any times $a \leq b$ and any index i, we have*

$$X_i(a) - X_{i+1}(a) \geq X_i(b) - X_{i+1}(b).$$

Proof of claim. Recall that $X_i(t) = |B(\tau_i, t)|$. As $\tau_i < \tau_{i+1}$, we get $X_i(t) - X_{i+1}(t) = |B(\tau_i, \tau_{i+1}) \setminus B(\tau_{i+1}, t)|$. Thus

$$\begin{aligned} &X_i(a) - X_{i+1}(a) - (X_i(b) - X_{i+1}(b)) \\ &= |B(\tau_i, \tau_{i+1}) \setminus B(\tau_{i+1}, a)| - |B(\tau_i, \tau_{i+1}) \setminus B(\tau_{i+1}, b)| \ \geq \ 0, \end{aligned}$$

as $B(\tau_{i+1}, a) \subset B(\tau_{i+1}, b)$. □

The histogram H and the hit rate curve C are computed by the increment operation of the ith iteration of the loop on line 10 in algorithm Algorithm 3 while processing the request at time t, for each valid i, t pair. For simplicity, we consider line 13 to be the final iteration of this loop. This increment operation involves only X_i and X_{i+1}. The same is true of $\hat{H}$ and $\hat{C}$, using instead the pair $\hat{X}_i$ and $\hat{X}_{i+1}$. So, to prove (3.6), we will show that the contribution from the pair $\hat{X}_i$ and $\hat{X}_{i+1}$ to $\hat{C}$ approximately equals the contribution from the pair X_i and X_{i+1} to C.

Contribution to C. Fix any $x \in [\ell]$ and recall that $C(x\Delta) = \sum_{j=1}^{x} H[j]/m$. By considering lines 4, 12, and 13 of Algorithm 3 we see that the pair X_i and X_{i+1} can only contribute to $C(x\Delta)$ while $t \leq m$ and $\lceil X_i(t)/\Delta \rceil \leq x$. So, let T_i be the time after the last contribution of X_i and X_{i+1} to $C(x\Delta)$, i.e.,

$$T_i \ = \ \min(\{\, t : X_i(t) > x\Delta \,\} \cup \{m+1\}).$$

At all times $t \geq T_i$, the pair X_i and X_{i+1} does not contribute to $C(x)$.

For the time being, let us assume that $\tau_{i+1} \leq m$. That is, an $(i+1)$th counter is created during trace processing. The contribution to $m \cdot C(x\Delta)$ from the the pair X_i and X_{i+1} is

$$\begin{cases} X_{i+1}(t+1) - X_{i+1}(t) - X_i(t+1) + X_i(t) & \text{(for each time } t \in \{\tau_{i+1}, \ldots, T_i - 1\}) \\ 1 - X_i(t+1) + X_i(t). & \text{(for each time } t \in \{\tau_i, \ldots, \tau_{i+1} - 1\}). \end{cases}$$

Summing up, the total contribution is

$$\begin{aligned} &\sum_{\tau_i \leq t < \tau_{i+1}-1} \big(1 - X_i(t+1) + X_i(t)\big) + \sum_{\tau_{i+1} \leq t < T_i} \big(X_{i+1}(t+1) - X_{i+1}(t) - X_i(t+1) + X_i(t)\big) \\ &= \Delta - X_i(\tau_{i+1}) + X_i(\tau_i) + X_i(\tau_{i+1}) - X_{i+1}(\tau_{i+1}) + X_{i+1}(T_i) - X_i(T_i) \\ &= \Delta + X_{i+1}(T_i) - X_i(T_i) \end{aligned} \tag{3.7}$$

Contribution to $\hat{C}$. Similarly, let $\hat{T}_i = \min(\{t : \hat{X}_i(t) > x\Delta\} \cup \{m+1\})$. Then at all times $t \geq \hat{T}_i$, the pair $\hat{X}_i$ and $\hat{X}_{i+1}$ do not contribute to $\hat{C}(x)$. (This assertion uses property **P3** of the counters.) Summing up as before, the total contribution of the pair $\hat{X}_i$ and $\hat{X}_{i+1}$ to $m \cdot \hat{C}(x\Delta)$ is

$$\Delta + \hat{X}_{i+1}(\hat{T}_i) - \hat{X}_i(\hat{T}_i). \tag{3.8}$$

Upper bound on contribution to $\hat{C}(x\Delta)$. The difference between the contribution of $\hat{X}_i$ and $\hat{X}_{i+1}$ to $m \cdot \hat{C}(x\Delta)$ and the contribution of X_i and X_{i+1} to $m \cdot C(x\Delta)$ is the difference between (3.8) and (3.7), namely

$$\hat{X}_{i+1}(\hat{T}_i) - \hat{X}_i(\hat{T}_i) - X_{i+1}(T_i) + X_i(T_i). \tag{3.9}$$

We now upper bound this quantity. First note that $\hat{T}_i \leq T_i$, by (3.5). Then Claim 3 shows that (3.9) is at most

$$\begin{aligned}
&\hat{X}_{i+1}(\hat{T}_i) - \hat{X}_i(\hat{T}_i) - X_{i+1}(\hat{T}_i) + X_i(\hat{T}_i) \\
&\leq \alpha X_{i+1}(\hat{T}_i) \quad \text{(by (3.5))} \\
&\leq \alpha X_i(\hat{T}_i) \quad \text{(by definition of } X_i \text{ and } X_{i+1}) \\
&\leq \alpha(x\Delta + 1) \quad \text{(since } \hat{T}_i \leq T_i \text{ and by definition of } T_i). \qquad (3.10)
\end{aligned}$$

Lower bound on contribution to $\hat{C}(x\Delta)$. For the lower bound, we must consider the contribution of X_i and X_{i+1} to $C((x-1)\Delta)$. Define

$$T_i' = \min(\{\, t : X_i(t) > (x-1)\Delta \,\} \cup \{m+1\}).$$

Arguing as before, we find that the total contribution of this pair to $m \cdot C((x-1)\Delta)$ is

$$\Delta + X_{i+1}(T_i') - X_i(T_i'). \qquad (3.11)$$

The difference between (3.8) and (3.11) is

$$\hat{X}_{i+1}(\hat{T}_i) - \hat{X}_i(\hat{T}_i) - X_{i+1}(T_i') + X_i(T_i'). \qquad (3.12)$$

Claim 4. $T_i' \leq \hat{T}_i$.

Proof of claim. By definition of T_i', we have $X_i(T_i') \leq (x-1)\Delta + 1$. By definition of α, we have $\alpha n = \varepsilon\Delta < \Delta$. So, by (3.5),

$$\begin{aligned}
\hat{X}_i(T_i') \;&\leq\; (1+\alpha)X_i(T_i') \;\leq\; (1+\alpha)\big((x-1)\Delta + 1\big) \\
&\leq\; (x-1)\Delta + 1 + \alpha n \;<\; x\Delta + 1 \;\leq\; \hat{X}_i(\hat{T}_i).
\end{aligned}$$

By **P3**, the claim is proven. □

Applying Claim 4, we may use Claim 3 to show that (3.12) is at least

$$\begin{aligned}
&\hat{X}_{i+1}(\hat{T}_i) - \hat{X}_i(\hat{T}_i) - X_{i+1}(\hat{T}_i) + X_i(\hat{T}_i) \\
&\geq \; -\alpha X_i(\hat{T}_i) \quad \text{(by (3.5))} \\
&\geq \; -\alpha(x\Delta+1) \quad \text{(since } \hat{T}_i \leq T_i \text{ and by definition of } T_i\text{).}
\end{aligned} \tag{3.13}$$

The last counter. Here we consider the special case where $m < \tau_{i+1}$. In this case, the contribution of X_i to $C(x)$ is

$$\sum_{\tau_i \leq t \leq m} \big(1 - X_i(t+1) + X_i(t)\big) = m - \tau_i + 1 - X_i(m+1). \tag{3.14}$$

Similarly, the contribution of $\hat{X}_i$ to $\hat{C}(x)$ is

$$m - \tau_i + 1 - \hat{X}_i(m+1). \tag{3.15}$$

The bound of $\alpha(x\Delta+1)$ on the absolute value of the difference of (3.15) and (3.14) follows immediately from (3.5). Indeed, we get the bound of $\alpha\Delta$. Note that $T_i = T_i' = m+1$, as $X_i(t) \leq \Delta$ for all $t \in [\tau_i, m]$. Thus the contribution of X_i to $C((x-1)\Delta)$ is equal to the expression in (3.14), and the lower bound follows as well.

Proof of (3.6)*:* We now combine our previous observations to establish (3.6). Recall that $c = \lceil m/\Delta \rceil$ is the total number of counters. Summing (3.10) over all i, we obtain that

$$m\hat{C}(x\Delta) \;\leq\; mC(x\Delta) + \alpha c(x\Delta+1) \;\leq\; mC(x\Delta) + 2\alpha m x.$$

This proves the second inequality of (3.6). The first inequality of (3.6) follows analogously from (3.13).

Proof of (Strong-Guarantee)*:* Let C^* denote the true hit rate curve. Combining (3.6), Claim 1 and the definition of α yields

$$C^*\big((x-2)\Delta\big) - 2\varepsilon x/\ell \;\le\; \hat{C}(x\Delta) \;\le\; C^*(x\Delta) + 2\varepsilon x/\ell.$$

Applying this bound with $\Delta/2$ instead of Δ, and $\varepsilon/2$ instead of ε establishes (Strong-Guarantee). □

3.4.2 Improved space by pruning redundant counters

As requests are processed, adjacent counters may converge to the same value. When this happens, it is not necessary to keep both counters. In this section we use the working set data structure of Algorithm 1 with probabilistic counters, but we delete redundant counters. The new algorithm is shown in Algorithm 4.

Claim 5. *The number of active counters at any point in time is* $O(\ell/\varepsilon)$.

Proof. Due to lines 8 and 9, every $i \in A$ satisfies either $X_{i-1}(t) - X_i(t) > 2\varepsilon\Delta$ or $X_i(t) - X_{i+1}(t) > 2\varepsilon\Delta$. In either case, we must have

$$|B(\tau_{i-1},t)| - |B(\tau_{i+1},t)| > \varepsilon\Delta \quad \forall i \notin \{1,c\}, \tag{3.16}$$

since, by (3.5) and the definition of α,

$$|B(\tau_j,t)| \;\le\; X_j(t) \;\le\; (1+\alpha)|B(\tau_j,t)| \;\le\; |B(\tau_j,t)| + \varepsilon\Delta$$

for every $j \in A$ and every t. Summing (3.16) over i we obtain

$$\begin{aligned}\varepsilon\Delta(|A|-2) &\le \sum_{i\in A\setminus\{1,c\}} \big(|B(\tau_{i-1},t)| - |B(\tau_{i+1},t)|\big) \\ &\le 2\sum_{i=1}^{c-1} \big(|B(\tau_i,t)| - |B(\tau_{i+1},t)|\big) \le 2n.\end{aligned}$$

We conclude that $|A| \le 2 + 2n/\varepsilon\Delta = O(\ell/\varepsilon)$. □

Corollary 6. *The space complexity of Algorithm 3 implemented using Algorithm 4 is* $O\big(\ell(\ell^2/\varepsilon^2 + \log n)\log(m)/\varepsilon\big)$ *bits.*

Algorithm 4: An implementation of a working set data structure incorporating pruning. A is the set of active counters.

$A \leftarrow \emptyset$
Function `Register(`t, b_t`)`**:**
  $c \leftarrow \lceil t/\Delta \rceil$
  if $t \equiv 1 \pmod{\Delta}$ **then**
    Create the new counter $\mathscr{K}[c]$
    $A \leftarrow A \cup \{c\}$
  for $i \in A \setminus \{1, c\}$ **do**
    if

$$\big((i-1 \notin A) \vee (X_{i-1}(t) - X_i(t) \leq 2\varepsilon\Delta)\big) \wedge \big((i+1 \notin A) \vee (X_i(t) - X_{i+1}(t) \leq 2\varepsilon\Delta)\big)$$

    then
      Delete $\mathscr{K}[i]$ and set $A \leftarrow A \setminus \{i\}$
  for $i \in A$ **do**
    $\mathscr{K}[i]$.INSERT(b_t)
Function `GetWorkingSetSize(`t`)`**:**
  Let $i = \lceil t/\Delta \rceil$
  if $i \in A$ **then**
    Return $\mathscr{K}[i]$.QUERY$()$
  else
    Return NULL

Proof. Using the optimal probabilistic counter [21] with the parameter $\alpha = \varepsilon/\ell$ each counter uses space $s = O\big((\ell^2/\varepsilon^2 + \log n)\log m\big)$. The space requirement for the histogram used by Algorithm 3 is $O(\ell \log n)$. By Claim 5, the total space usage is $O\big((\ell/\varepsilon) \cdot s\big)$, as required. □

The next theorem compares the accuracy of Algorithm 3 with two different working set data structures: either Algorithm 1 or Algorithm 4. In both cases we use probabilistic counters.

Theorem 7. *Let C and $\hat{C}$ be respectively the hit rate curve produced from Algorithm 3 using Algorithm 1 or Algorithm 4 to implement the working set data*

structure. Then

$$|C(x\Delta) - \hat{C}(x\Delta)| \;\leq\; 4\varepsilon \qquad \forall x \in [\ell+1].$$

Consequently, $\hat{C}$ satisfies the (Weak-Guarantee) *condition on page 8 with respect to the true hit rate curve.*

Proof. Let H and X_i refer to the quantities computed using Algorithm 1. Let $\hat{H}$ and $\hat{X}_i$ refer to the corresponding quantities computed using Algorithm 4. We will assume that both algorithms are furnished with the same random bits. Consequently,

$$X_i(t) = \hat{X}_i(t) \text{ whenever } \hat{X}_i(t) \neq \text{NULL} \qquad \forall i \in [c], t \in [m]. \tag{3.17}$$

As in the proof of Theorem 2, we fix $x \in [\ell+1]$ and $i \in [c]$ and compare the contribution from the pair of counters X_i and X_{i+1} to $C(x\Delta)$ and the contribution from the pair of counters $\hat{X}_i$ and $\hat{X}_{i+1}$ to $\hat{C}(x\Delta)$.

Let $T_i = \min(\{\, t : X_i(t) > x\Delta \,\} \cup \{m+1\})$. As in the proof of Theorem 2, the pair of counters X_i and X_{i+1} cannot contribute to $C(x\Delta)$ at any time $t \geq T_i$.

Let $\hat{T}_i$ be the first time t at which $\hat{X}_i(t) > x\Delta$, $\{i, i+1\} \nsubseteq A$, or $t \geq m+1$. Then, by considering lines 11-13 of Algorithm 3, we see that the pair of counters $\hat{X}_i$ and $\hat{X}_{i+1}$ cannot contribute to $\hat{C}(x\Delta)$ at any time $t \geq \hat{T}_i$. In the case that $\hat{X}_i(t) > x\Delta$ this follows from property **P3** of $\hat{X}_i$, and in the case that $\{i, i+1\} \nsubseteq A$ this follows because of line 17 of Algorithm 4.

We first assume $\tau_{i+1} \leq m$. Following the argument of (3.7) in the proof of Theorem 2, the difference in contributions to $m \cdot \hat{C}(x\Delta)$ and $m \cdot C(x\Delta)$ is

$$\hat{X}_{i+1}(\hat{T}_i) - \hat{X}_i(\hat{T}_i) - X_{i+1}(T_i) + X_i(T_i). \tag{3.18}$$

Case 1. Suppose that $\hat{T}_i$ is such that $\lceil \hat{X}_i(\hat{T}_i)/\Delta \rceil > x$ or $\hat{T}_i = m+1$. Then we necessarily have $\hat{T}_i = T_i$ due to (3.17). In this case (3.18) is clearly zero.

Case 2. Suppose that $\{i, i+1\} \nsubseteq A$ in iteration $\hat{T}_i$. In this case we might not have $\hat{X}_i(\hat{T}_i) > x\Delta$, but we do have

$$\hat{X}_i(\hat{T}_i) - \hat{X}_{i+1}(\hat{T}_i) \;\leq\; 2\varepsilon\Delta, \tag{3.19}$$

by line 8 of Algorithm 4. Observe that $\hat{T}_i \leq T_i$.

Let $Y_i(t)$ denote $|B(\tau_i, t)|$. Since $Y_i(t) \leq n$ and $\alpha = \varepsilon\Delta/n$, it follows from (3.5) that

$$\begin{aligned} 0 \;\leq\; X_i(t) - Y_i(t) \;\leq\; \varepsilon\Delta \quad \forall i \in [c], t \in [m] \\ 0 \;\leq\; \hat{X}_i(t) - Y_i(t) \;\leq\; \varepsilon\Delta \quad \forall i \in [c], t \leq \hat{T}_i. \end{aligned} \tag{3.20}$$

The first step is to upper bound (3.18).

$$\begin{aligned} &\hat{X}_{i+1}(\hat{T}_i) - \hat{X}_i(\hat{T}_i) - X_{i+1}(T_i) + X_i(T_i) \\ &\leq\; Y_{i+1}(\hat{T}_i) - Y_i(\hat{T}_i) - Y_{i+1}(T_i) + Y_i(T_i) + 2\varepsilon\Delta \quad \text{(by (3.20))} \\ &\leq\; 2\varepsilon\Delta, \end{aligned}$$

by Claim 3, since $\hat{T}_i \leq T_i$. Next we lower bound (3.18).

$$\begin{aligned} &\hat{X}_{i+1}(\hat{T}_i) - \hat{X}_i(\hat{T}_i) - X_{i+1}(T_i) + X_i(T_i) \\ &\geq\; \hat{X}_{i+1}(\hat{T}_i) - \hat{X}_i(\hat{T}_i) - Y_{i+1}(T_i) + Y_i(T_i) - \varepsilon\Delta \quad \text{(by (3.20))} \\ &\geq\; \hat{X}_{i+1}(\hat{T}_i) - \hat{X}_i(\hat{T}_i) - \varepsilon\Delta \quad \text{(by definition of } Y\text{)} \\ &\geq\; -3\varepsilon\Delta \end{aligned}$$

by (3.19). It follows that (3.18) is at most $3\varepsilon\Delta$ in absolute value.

The last counter: Here we examine the special case where $\tau_{i+1} > m$. As $\tau_{i+1} = i\Delta + 1$, at every time $t \in [\tau_i, m]$, $c = i$. Thus the for loop of 7 cannot prune the ith counter. So $\hat{T}_i = T_i$, and the difference between the contributions of counters is zero.

Summing up the contribution to $m \cdot C(x\Delta)$ and $m \cdot \hat{C}(x\Delta)$ from all pairs of counters, we obtain that

$$|m \cdot C(x\Delta) - m \cdot \hat{C}(x\Delta)| \;\leq\; c \cdot 3\varepsilon\Delta \;=\; \lceil m/\Delta \rceil \cdot 3\varepsilon\Delta \;\leq\; 4\varepsilon m.$$

This proves the theorem. □

Chapter 4

A Unified Representation

In the past chapter, we used probabilistic counters as a black box. Here, by examining the implementation of probabilistic counters, we will derive a new algorithm that makes different space tradeoffs. In order to explain this algorithm, we will describe some work from the streaming algorithms community on the distinct elements problem.

4.1 Distinct Elements and Probabilistic Counters

The distinct elements problem, referred to as DISTINCT-ELEMENTS, is the problem of estimating the number of distinct elements in a stream of tokens. There are two parameters: ε is the desired accuracy of the estimate, and δ is the failure probability. Formally, an algorithm solves DISTINCT-ELEMENTS with parameters ε and δ if given a stream S, the algorithm outputs an estimate d such that

$$\Pr((1-\varepsilon)|S| \leq d \leq (1+\varepsilon)|S|) \geq 1-\delta.$$

Using $O(|S|)$ space, it is possible to solve DISTINCT-ELEMENTS deterministically with $\varepsilon = \delta = 0$. However, it is provably impossible to use sublinear space with $\varepsilon = 0$ or $\delta = 0$.

Probabilistic counters are randomized algorithms which use only sublinear space to solve DISTINCT-ELEMENTS. Many probabilistic counters [2, 3, 17, 21] rely on a $\{0,1\}$-matrix M that is updated while processing each item in the stream.

Each item b in the stream is hashed to a binary string σ, and then M is updated based on $\mathrm{lsb}(\sigma)$, the number of trailing zeros in σ. We will call such a matrix M a *bitmatrix*.

The simplest counter [2] uses a single hash function h, and the matrix M has a single column. To process a new stream element b, the algorithm computes $z = \mathrm{lsb}(h(b))$. For each $i \leq z$, M_i is set to 1. After the stream is processed, the algorithm outputs 2^{j^*}, where j^* is the index of the greatest non-zero row. This algorithm produces an $O(1)$ estimate of $|S|$ with failure probability $\leq \sqrt{2}/3$.

Other algorithms refine this estimate by using additional columns and another hash function g, which determines which column to update.

The estimate could be, for example, a function of the average of the lowest non-zero value in each column [14, 17], or the number of non-zero cells below a certain row (Algorithm 3 in [3]).

Many distinct elements algorithms have a fixed failure probability. To reduce the failure probability, a standard method called the median trick is used. Suppose a probabilistic counter solves the distinct elements problem with parameters ε, δ, where $\delta < 1/2$. Now run k independent instantiations of the algorithm on the stream S, and output the median estimate. If the median is greater than $(1+\varepsilon)|S|$, then the estimate of $k/2$ counters exceeded $(1+\varepsilon)|S|$. By a Chernoff bound, this occurs with probability $2^{\Omega(-k)}$. Similarly, we obtain a $2^{\Omega(-k)}$ probability for the event that the median is less than $(1-\varepsilon)|S|$ (further details can be found in [10] and [18]). We will make use of this trick below.

4.2 Counter Packing

If we imagine that Algorithm 1 uses such a counter, and that all counters use the same hash functions h and g, then we see that there is a great deal of redundant state. For example, at time step t we apply the hash functions to the block b_t, then perform the appropriate update on M^c, the bitmatrix corresponding to the most recently created counter. We also loop over all the older counters, and update their bitmatrices as well. However, if $M^c_{i,z} = 1$, it follows that $M^j_{i',z} = 1$ for every $0 \leq i' \leq i$ and $1 \leq j \leq c$, as older counters will have certainly undergone any updates that newer counters have. This observation leads to the following idea: instead of

storing the bitmatrices for all counters separately, we can store a single unified matrix from which all bitmatrices can be computed.

We will keep a single matrix Q to represent a sequence of counters, where the kth counter was started at time $(k-1)\Delta+1$ in the trace. We will maintain Q such that at any time t during stream processing, $Q_{i,j} = r$ means that the bitmatrix of the rth counter has a 1 at position i, j, and for any $r' > r$, the bitmatrix of the r'th counter is 0 at position i, j. By the observation made above, it follows that for any $r' < r$, the r'th bitmatrix has a 1 in position i, j. To extract the bitmatrix corresponding to the r'th counter, $M^{r'}$, we examine each entry of Q. Let $Q_{i,j} = r$. If $r < r'$, then $M^{r'}_{i,j} = 0$. Otherwise, $M^{r'}_{i,j} = 1$. The pseudocode for the algorithm is given in Algorithm 5.

Algorithm 5: An implementation of a working set data structure based on a unified counter representation. Parameterized by a randomized counter algorithm $\mathscr{A}$. The set $\mathscr{Q}$ has many independent copies of the hash functions and the resulting table. We need only $|\mathscr{Q}| = O(\log(1/\delta))$ with $\delta = m^{-3}$.

Data: A collection $\mathscr{Q}$ of pairs $(Q, \mathscr{H})$, where Q is a matrix, $\mathscr{H}$ is a set of hash functions

A randomized counter algorithm $\mathscr{A}$

$c \leftarrow 1$

Function `Register(`t, b_t`)`**:**

  $c \leftarrow \lceil t/\Delta \rceil$

  for $(Q, \mathscr{H}) \in \mathscr{Q}$ **do**

    Update Q using b_t according to $\mathscr{A}$

Function `GetWorkingSetSize(`t'`)`**:**

  for $Q \in \mathscr{Q}$ **do**

    Let $r = \lceil t'/\Delta \rceil$

    Define the bitmatrix M^r by $M^r_{i,j} = \begin{cases} 1 & \text{if } Q_{i,j} - r \geq 0 \\ 0 & \text{otherwise} \end{cases}$

    Feed M^r into counter algorithm $\mathscr{A}$ to obtain estimate R_Q

  Return the median of estimates R_Q

4.3 Analysis

In order to analyze Algorithm 5, we must specify a concrete randomized counter algorithm $\mathscr{A}$. We will use Algorithm 2 from the paper of Bar-Yossef et al. [3]. In this algorithm, each matrix Q has $\log(n)$ rows, and $k = O(1/\alpha^2)$ columns. Each collection $\mathscr{H}$ consists of k t-wise independent hash functions, where t is $O\big(\log(1/\alpha)\big)$. To update Q, for $j \in [k]$, we set $Q_{i,j} = c$ if $\mathrm{lsb}(h_j(b_t)) \geq i$. Given the bitmatrix M, this randomized counter algorithm can produce its estimate.

Claim 8. *The space requirement of Algorithm 5 is* $O\big(\ell^2 \log(n)\log(m)\log(1/\delta)/\varepsilon^2\big)$.

Proof. Each Q has $O(1/\alpha^2)$ columns, $\log n$ rows, and each cell requires $\log m$ space. Thus Q requires $O\big(\log(n)\log(m)/\alpha^2\big)$ space. Each collection $\mathscr{H}$ requires $O\big(\log^2(1/\alpha)\log n\big)$ space, which is negligible. We have $O(\log(1/\delta))$ such pairs $(Q,\mathscr{H})$. Thus the total space requirement is $O\big(\alpha^{-2}\log(n)\log(m)\log(1/\delta)\big)$. Substituting $\alpha = \varepsilon/\ell$ completes the proof. □

Theorem 9. *Algorithm 3 using Algorithm 5 as its working set data structure satisfies* (Strong-Guarantee).

Proof. Let $X_i(t)$ be the result of GETWORKINGSETSIZE(τ_i) at time t, which is an estimate of $|B(\tau_i,t)|$. It suffices to show that $X_i(t) \in [|B(\tau_i,t)|, (1+\alpha)|B(\tau_i,t)|]$ with high probability, in which case the argument of Theorem 2 applies.

For any i,t, by taking the median of $O(\log(1/\delta))$ estimates of $\mathscr{A}$, we have $\Pr[|X_i(t) - |B(\tau_i,t)|| > \alpha|B(\tau_i,t)|] \leq \delta$. At time t, the algorithm computes estimates for t/Δ counters, and thus $O(m^2)$ estimates are taken in total. By a union bound, the probability that any estimate is poor is $\leq \delta m^2$.

There is one subtlety: Theorem 2 assumed counters with only one-sided error, while $\mathscr{A}$ provides counters with two-sided error. If we take $\alpha' = \frac{\alpha}{2+\alpha}$ as the accuracy of $\mathscr{A}$ and divide all estimates by $(1-\alpha')$, we recover one-sided α approximations. □

Chapter 5

Lower Bounds

In this section we prove lower bounds on the space needed by one-pass algorithms to compute approximate hit rate curves. Formally, let $\text{HRC}_{n,m,\varepsilon,\ell}$ be the computational problem in which, given an input sequence in $[n]^m$, one must compute a function $\hat{C}$ satisfying (Weak-Guarantee), where $\Delta = \lfloor n/\ell \rfloor$. Similarly, let $\text{HRC}'_{n,m,\varepsilon,\ell}$ be the analogous problem in which $\hat{C}$ must satisfy (Strong-Guarantee).

While $\text{HRC}_{n,m,\varepsilon,\ell}$ is perhaps a more natural formulation of the hit rate curve problem, Theorems 2 and 9 show that two of our algorithms actually solve $\text{HRC}'_{n,m,\varepsilon,\ell}$. It is useful to analyze the fundamental complexity of both $\text{HRC}_{n,m,\varepsilon,\ell}$ and $\text{HRC}'_{n,m,\varepsilon,\ell}$ in order to understand the limits of our current algorithms and to design improved ones.

5.1 Communication Complexity

In order to prove our lower bounds, we will use communication complexity. The basic model of communication complexity involves two players, Alice and Bob, who are tasked with computing $f(x,y)$, where x is given to Alice and y to Bob. The challenge is to devise a communication protocol such that Alice and Bob exchange the least number of bits to compute $f(x,y)$.

A *k-round protocol* is a communication protocol in which k messages are sent between Alice and Bob. For example, in a 1-round protocol, Bob can compute $f(x,y)$ after receiving a single message from Alice. A *randomized protocol* fur-

nishes Alice and Bob with random bits. In a *private coin protocol*, Alice and Bob each have access to their own random bits and cannot see the other's random bits. In a *public coin protocol*, Alice and Bob have access to the same random bits. For a detailed introduction to communication complexity, consult [22].

In the following, we will construct k-round, public coin protocols.

5.2 Gap Hamming Distance

Our lower bounds are based on reductions from the Gap Hamming Distance (GHD) problem. In $\text{GHD}_{k,t,g}$, Alice and Bob are respectively given vectors $x, y \in \{0,1\}^k$. They are required to determine whether the Hamming distance between x and y, denoted $d(x,y)$, is $\leq t-g$ or $> t+g$, outputting 0 or 1 respectively.

The Gap Hamming Distance problem was first introduced by Woodruff and Indyk in [19]. Their goal was to obtain lower bounds for the space complexity of the Distinct Elements problem. They described a reduction of Gap Hamming Distance to Distinct Elements, and gave a lower bound for the space complexity of Gap Hamming Distance. A subsequent line of work ([30], [20], [31], [7], [8]), culminating in the 2012 paper of Chakrabarti and Regev [9], settled the communication complexity of $\text{GHD}_{n,\sqrt{n},n/2}$ as $\Omega(n)$. The Chakrabarti and Regev paper [9] also contained the following generalization, which will be helpful:

Theorem 10 (Chakrabarti-Regev [9], Proposition 4.4)**.** *Any protocol that solves $GHD_{k,k/2,g}$ with probability $\geq 2/3$ communicates $\Omega(\min\{k, k^2/g^2\})$ bits.*

Since GHD was originally introduced to obtain lower bounds for the distinct elements problem [19], and since hit rate curves fundamentally involve the number of distinct elements, it is not too surprising that reducing GHD to HRC and HRC$'$ is useful.

5.3 HRC Lower Bounds

Theorem 11. *The space complexity of $HRC_{n,n,\varepsilon,\ell}$ is $\Omega(\min\{n, 1/\varepsilon^2\})$.*

Proof. Let $k = n/2 = m/2$, and assume ℓ divides n. Given an instance of $\text{GHD}_{k,k/2,2\varepsilon k}$ with inputs x, y, as well as an algorithm which solves $\text{HRC}_{n,n,\ell,\varepsilon}$ using space s, consider the following protocol:

Alice constructs the set $X = \{j : x_j = 1\} \cup \{k + j : x_j = 0\}$, then runs the algorithm for $\mathrm{HRC}_{n,n,\ell,\varepsilon}$ on the stream of elements of X in arbitrary order. She passes the s bits of state to Bob. Bob constructs the set $Y = \{k + j : y_j = 1\} \cup \{j : y_j = 0\}$, and using the state from Alice, continues the algorithm on the elements of Y. If $\hat{C}((\ell+1)\Delta) \leq 1/4$, he outputs 0, otherwise he outputs 1.

By construction, $d(x,y) = |X \cap Y|$, so $2k \cdot C(\ell\Delta) = d(x,y)$. (Recall that the hit rate curve is normalized by the length of the request sequence, which is $m = 2k$.) Since $\hat{C}$ satisfies (Weak-Guarantee), we have

$$2k \cdot C(\ell\Delta) - 2k\varepsilon \;\leq\; 2k \cdot \hat{C}((\ell+1)\Delta) \;\leq\; 2k \cdot C((\ell+1)\Delta) + 2k\varepsilon \;=\; 2k \cdot C(\ell\Delta) + 2k\varepsilon.$$

Thus the protocol correctly distinguishes the cases $d(x,y) \leq k/2 - 2\varepsilon k$ and $d(x,y) > k/2 + 2\varepsilon k$. Applying Theorem 10, we conclude that $s \in \Omega(\min\{k, (k)^2/(k\varepsilon)^2\} = \Omega(\min\{n, 1/\varepsilon^2\})$. □

Theorem 12. *The space complexity of* $\mathrm{HRC}'_{n,n,\varepsilon,\ell}$ *is* $\Omega(\min\{n/\ell, \ell/\varepsilon^2\})$.

Proof. Let $k = n/2 = m/2$, and assume ℓ divides n. Given an instance of $\mathrm{GHD}_{k,k/2,2\varepsilon k}$ with inputs x, y, as well as an algorithm which solves $\mathrm{HRC}'_{n,n,\ell,\varepsilon}$ using space s, consider the following protocol:

Alice constructs the ℓ sets

$$A_i \;=\; \left\{\; j : x_j = 1 \wedge (i-1)\Delta < j \leq i\Delta \;\right\} \cup \left\{\; j+k : x_j = 0 \wedge (j-1)\Delta < j \leq j\Delta \;\right\}.$$

Bob defines the ℓ sets

$$B_i \;=\; \left\{\; j+k : y_j = 1 \wedge (i-1)\Delta < j \leq i\Delta \;\right\} \cup \left\{\; j : y_j = 0 \wedge (j-1)\Delta < j \leq j\Delta \;\right\}.$$

Alice runs the algorithm for $\mathrm{HRC}'_{n,n,\varepsilon,\ell}$ on A_1, and passes the s_1 bits of algorithm state to Bob, who uses it to run the algorithm on B_1. He sends the resulting s_2 bits of state back to Alice, who uses it to process A_2. Passing the current state back and forth in this manner, after $2\ell - 1$ messages have been exchanged and Bob has finished running the algorithm on B_ℓ, the algorithm has processed the request sequence

$$A_1, B_1, A_2, B_2, \ldots, A_\ell, B_\ell,$$

where the entries of each individual set A_i or B_i may be ordered arbitrarily. Note that the A_i's are pairwise disjoint, as are the B_i's. Furthermore, $A_i \cap B_j = \emptyset$ for $i \neq j$. Thus as $|A_i| = |B_i| = \Delta$ for all i, at every time t we have either $D(t) \leq \Delta$ or $D(t) = \infty$, and so $C(i\Delta) = C(\Delta)$ for all $i \geq 1$. Since $2k \cdot C(\ell\Delta) = d(x,y)$ and $\hat{C}$ satisfies (Strong-Guarantee), we have

$$d(x,y) - 2k\varepsilon/\ell \;\leq\; 2k \cdot \hat{C}(2\Delta) \;\leq\; d(x,y) + 2k\varepsilon/\ell.$$

Thus Bob can solve $\mathrm{GHD}_{k,k/2,2k\varepsilon/\ell}$ as before.

By Theorem 10,

$$(2\ell - 1) \cdot \max_j s_j \;\geq\; \sum_{j=1}^{2\ell-1} s_j \;=\; \Omega(\min\{k, \ell^2/\varepsilon^2\}).$$

So for some j, we have $s_j \in \Omega(\min\{n/\ell, \ell/\varepsilon^2\})$. □

Chapter 6

Experimental Results

Based on the ideas developed in Chapter 3, a prototype was implemented [29]. We compare this prototype against an implementation of the Mattson algorithm [23] on a collection of storage traces [24] released by Microsoft Research in Cambridge.

Xiang et al., the authors of the average footprint paper [33], released an open source implementation of their algorithm [12]. We include the results from a slightly modified version of this implementation in our figures, and discuss some of the strengths and weaknesses of their approach.

6.1 Microsoft Research Traces

The MSR traces record the disk accesses of a collection of 13 different servers over a period of one week. Each trace is a list of records, where each record represents a disk access. Records have fields for the time stamp, server name, disk number, operation type (read or write), disk offset, size of data requested, and latency. In order to process these traces, we filtered out writes and expanded each remaining record into a list of distinct blocks touched by the request. We used a block size of 4 KB.

The smallest trace, `wdev`, has 52,489 distinct blocks and 725,194 requests, while the largest trace, `prxy`, has 523,879 distinct blocks and 358,267,307 requests. The `proj` trace touches the largest volume of data with 324,760,925 distinct blocks, and contains 564,577,120 requests.

		Traces												
		hm	mds	prn	proj	prxy	rsrch	src1	src2	stg	ts	usr	wdev	web
avgfp	average	0.96	0.03	1.71	0.50	0.18	0.59	3.53	0.33	0.02	1.17	0.43	1.10	4.51
	max	4.43	0.57	6.97	4.29	27.91	9.04	44.14	26.46	0.15	15.52	15.56	14.17	35.68
cs	average	0.23	1.06	0.34	1.01	0.98	0.67	0.56	0.74	1.02	1.46	0.26	1.45	1.29
	max	9.71	4.68	8.46	2.54	37.50	16.45	14.55	26.51	1.70	25.41	13.36	22.33	13.91

Table 6.1: Average and maximum error for `avgfp` and `cs`.

In Figure 6.1, we plot the hit rate curves generated by three algorithms on the MSR traces. The `avgfp` algorithm is an implementation of the average footprint technique [33]. The authors released their implementation as open source on github [12]. The original implementation used a statically allocated array of size 512 MB to map blocks to last access times during trace processing, an efficient approach for CPU traces using relatively few distinct memory locations. Given the size of the larger MSR traces, a statically allocated data structure is impossible to use on modern hardware for this purpose, so we altered `avgfp` to use a hash table instead. The `cs` algorithm is a protoype based on the ideas sketched in Chapter 3. Finally, `mattson` is a single-threaded implementation of the Mattson algorithm bundled with parda [25]. We used `mattson` to compute the true hit rate curves to compare against the other algorithms.

To compare the resource usage of the algorithms, we ran each on the trace obtained by merging all MSR traces and sorting by time stamp. The resulting trace, called the `master` trace, is 22 GB, uncompressed. The `avgfp` algorithm processed `master` in 10 minutes using 22 GB of memory, `mattson` in 1 hour using 92 GB, and `cs` in 12 minutes using 0.5 GB. These experiments were run on a Dell PowerEdge R720 with two six-core Intel Xeon processors and 96 GB of RAM.

We give quantitative errors for `avgfp` and `cs` on the MSR traces in Table 6.1. We measured the error of an algorithm by finding the deviation between its curve and the curve produced by `mattson`.

Although Xiang et al. developed their technique with CPU traces in mind [33], their implementation does quite well on most of the MSR traces. Two exceptions are `src1`, with 44.1 maximum and 3.5 average error, and `web`, with 35.7 maximum and 4.5 average. In comparison, `cs` has 14.6 maximum and 0.6 average error

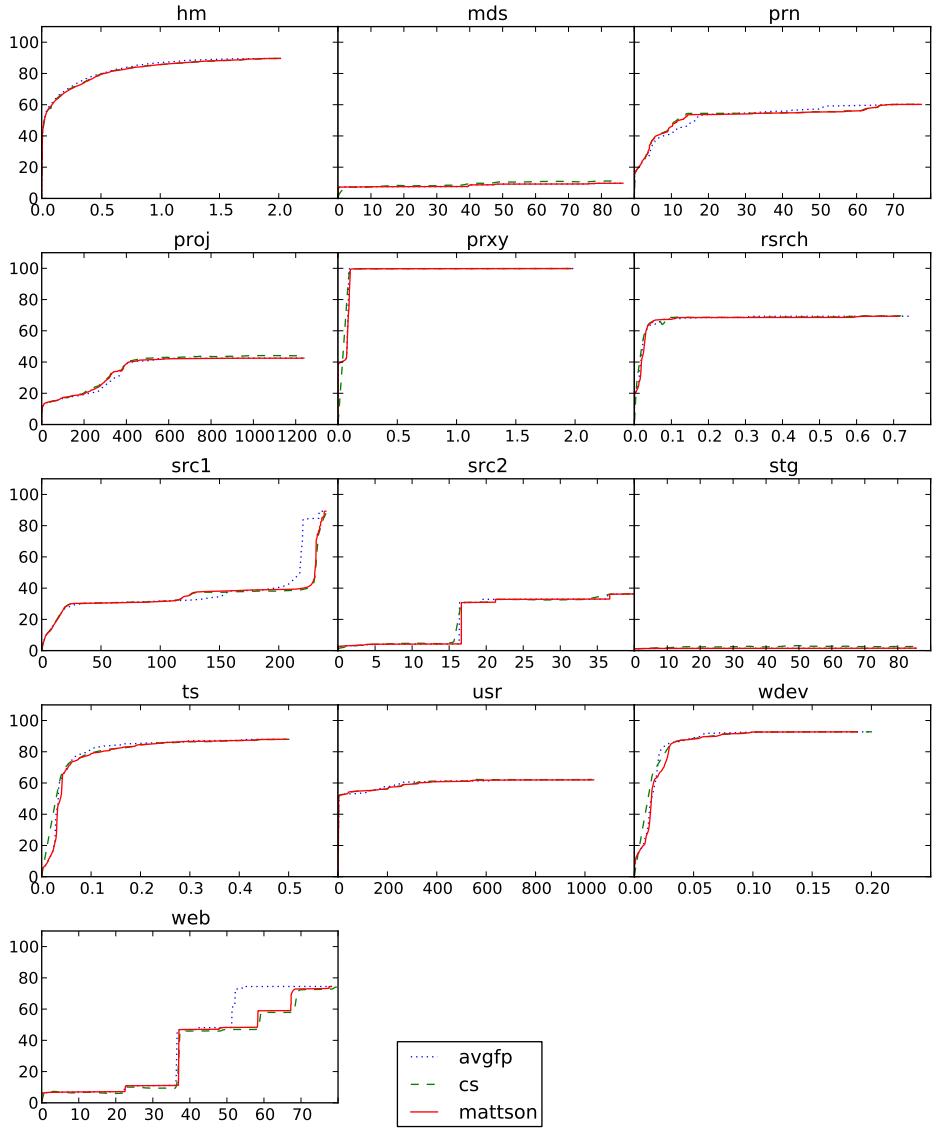

Figure 6.1: Hit rate curves for MSR traces. Cache size in GBs against hit rate percentage.

on src1, and 13.9 maximum and 1.3 average error on web. While both algorithms are less accurate at approximating volatile hit rate curves, avgfp exhibits higher error.

As the staircase plot suggests, web is characterized by multiple working sets of markedly different sizes in different phases of the trace. Such traces can cause problems for the average footprint technique. We investigate this further in the next section.

6.2 Average Footprints and Phase Changes

For each window length w, the average footprint technique computes the average working set size across all windows of length w in the trace. If the trace is marked by distinct phases in which working set size varies drastically, the average working set size may skew the reported stack distances.

To validate this suspicion, we generated an artificial trace, cyclic. cyclic is composed of two distinct phases. In the first phase, the first 10^4 blocks are read in order. This sequential scan is repeated 1000 times. The second phase consists of 10^5 repeated scans of the first 100 blocks. Both phases consist of 10^7 requests, for a total of $2 \cdot 10^7$ requests to 10^5 distinct blocks. The hit rate curves for cyclic as computed by avgfp, cs, and mattson are plotted in 6.2.

As suspected, the average footprint technique has trouble with this trace. For cache sizes between 0.020 GB and 0.035 GB, avgfp reports a hit rate of nearly 100%, while cs and mattson both give about 50%. The average and maximum error for avgfp are 24.6 and 50.0, while for cs they are 0.5 and 41.3. Because the hit rate changes so dramatically for a small change in cache size, large maximum errors are unavoidable for both algorithms (recall Theorem 2 bounds the error for bin x in terms of bins $x-1$ and x).

This trace is highly artificial and cannot be interpreted as a realistic model for system behaviour. However, we believe that it highlights an important limitation in the average footprint approach. Traces with highly pronounced phases do exist, and will cause inaccuracies with the average footprint method, as evidenced by web. We fabricated this trace in order to examine the worst case for the average footprint algorithm.

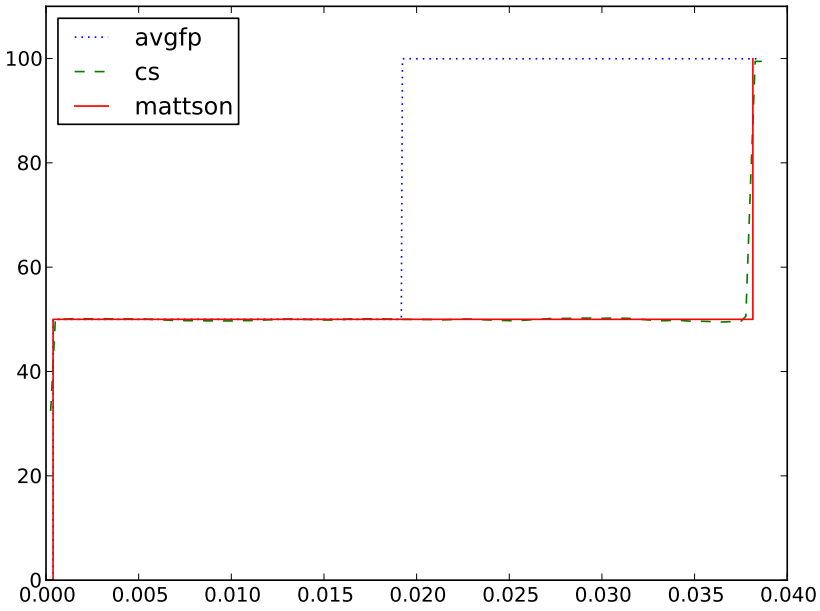

Figure 6.2: Hit rate curves for `cyclic` trace. Cache size in GBs against hit rate percentage.

Chapter 7

Conclusion

In this work, we introduced new algorithms for approximating hit rate curves which differ dramatically from existing approaches. We characterized the space usage of our algorithms, and provided lower bounds on the space complexity of any algorithm which approximates hit rate curves with given error bounds. We also validated an implementation based on our algorithms on the MSR traces, and compared our results to past work.

Our focus was entirely on space efficiency. We did not characterize the time complexity of our algorithms, nor did we provide lower bounds on time complexity. A careful analysis of our algorithms' time usage may result in more efficient algorithms, and is a possible direction for future work.

Our space lower bounds are not satisfactory. In particular, we were not able to obtain any dependence on ℓ for the lower bound of $\text{HRC}_{n,m,\varepsilon,\ell}$ in Theorem 11. We suspect neither Theorem 11 nor Theorem 12 gives tight bounds for their respective problems. We believe a deeper investigation of the space complexity of the hit rate curve problem, besides finding tighter bounds, could result in algorithms with greater space efficiency as well.

Bibliography

[1] G. S. Almási, C. Caşcaval, and D. A. Padua. Calculating stack distances efficiently. In *Proceedings of the 2002 workshop on memory system performance (MSP '02)*, pages 37–43, 2002. → pages 2, 4

[2] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 20–29. ACM, 1996. → pages 3, 22, 23

[3] Z. Bar-Yossef, T. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *Randomization and Approximation Techniques in Computer Science*, pages 1–10. Springer, 2002. → pages 3, 22, 23, 25

[4] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966. → pages 1

[5] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, 1975. → pages 2, 4, 9

[6] E. Berg and E. Hagersten. Statcache: a probabilistic approach to efficient and accurate data locality analysis. In *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*, pages 20–27. IEEE, 2004. → pages 6

[7] J. Brody and A. Chakrabarti. A multi-round communication lower bound for gap hamming and some consequences. In *Computational Complexity, 2009. CCC'09. 24th Annual IEEE Conference on*, pages 358–368. IEEE, 2009. → pages 27

[8] J. Brody, A. Chakrabarti, O. Regev, T. Vidick, and R. De Wolf. Better gap-hamming lower bounds via better round elimination. In *Approximation,*

Randomization, and Combinatorial Optimization. Algorithms and Techniques, pages 476–489. Springer, 2010. → pages 27

[9] A. Chakrabarti and O. Regev. An optimal lower bound on the communication complexity of gap-hamming-distance. *SIAM Journal on Computing*, 41(5):1299–1317, 2012. → pages 27

[10] A. Chakrabarti et al. Cs49: Data stream algorithms lecture notes. http://www.cs.dartmouth.edu/~ac/Teach/CS49-Fall11/Notes/lecnotes.pdf, 2012. Retrieved 2014-07-17. → pages 23

[11] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968. → pages 9

[12] C. Ding. Program locality analysis tool. https://github.com/dcompiler/loca, 2014. Retrieved 2014-07-03. → pages 30, 31

[13] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *PLDI*, pages 245–257. ACM, 2003. → pages 2, 4, 5

[14] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In *Algorithms-ESA 2003*, pages 605–617. Springer, 2003. → pages 3, 23

[15] D. Eklov and E. Hagersten. StatStack: Efficient modeling of LRU caches. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 55–65. IEEE, 2010. → pages 2, 6

[16] P. Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985. → pages 3

[17] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *DMTCS Proceedings*, 0(1), 2008. → pages 3, 13, 22, 23

[18] N. J. A. Harvey. Cpsc 536n: Randomized algorithms lecture notes. http://www.cs.ubc.ca/~nickhar/W12/, 2012. Retrieved 2014-08-16. → pages 23

[19] P. Indyk and D. Woodruff. Tight lower bounds for the distinct elements problem. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 283–288. IEEE, 2003. → pages 27

[20] T. Jayram, R. Kumar, and D. Sivakumar. The one-way communication complexity of hamming distance. *Theory of Computing*, 4(1):129–135, 2008. → pages 27

[21] D. M. Kane, J. Nelson, and D. P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 41–52. ACM, 2010. → pages 3, 13, 19, 22

[22] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, Cambridge, 1997. → pages 27

[23] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970. → pages 2, 4, 30

[24] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008. → pages 30

[25] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan. Parda: A fast parallel reuse distance analysis algorithm. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1284–1294. IEEE, 2012. → pages 2, 4, 5, 31

[26] F. Olken. Efficient methods for calculating the success function of fixed-space replacement policies. Technical report, Lawrence Berkeley Lab., CA (USA), 1981. → pages 2, 4

[27] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. In *POPL*, pages 55–61. ACM, 2007. → pages 2, 5

[28] R. A. Sugumar and S. G. Abraham. Multi-configuration simulation algorithms for the evaluation of computer architecture designs. *Ann Arbor, MI*, 1993. → pages 2, 4, 5

[29] J. Wires, S. Ingram, N. J. A. Harvey, A. Warfield, and Z. Drudi. Characterizing storage workloads with counter stacks. In *OSDI*, page to appear. USENIX, 2014. accepted. → pages 30

[30] D. Woodruff. Optimal space lower bounds for all frequency moments. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 167–175. Society for Industrial and Applied Mathematics, 2004. → pages 27

[31] D. P. Woodruff. The average-case complexity of counting distinct elements. In *Proceedings of the 12th International Conference on Database Theory*, pages 284–295. ACM, 2009. → pages 27

[32] X. Xiang, B. Bao, T. Bai, C. Ding, and T. Chilimbi. All-window profiling and composable models of cache sharing. *ACM SIGPLAN Notices*, 46(8): 91–102, 2011. → pages 6

[33] X. Xiang, B. Bao, C. Ding, and Y. Gao. Linear-time modeling of program working set in shared cache. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 350–360. IEEE, 2011. → pages 6, 30, 31

[34] Y. Zhong and W. Chang. Sampling-based program locality approximation. In *Proceedings of the 7th international symposium on Memory management*, pages 91–100. ACM, 2008. → pages 2, 5