Practical Verified Computation with Streaming Interactive Proofs

Practical Verified Computation with Streaming Interactive Proofs

A dissertation presented

by

Justin R. Thaler

to

The Department of School of Engineering and Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

May 2013

Thesis advisor
Michael Mitzenmacher

Author
Justin R. Thaler

Practical Verified Computation with Streaming Interactive Proofs

Abstract

As the cloud computing paradigm has gained prominence, the need for *verifiable computation* has grown urgent. Protocols for verifiable computation enable a weak client to outsource difficult computations to a powerful, but untrusted, server. These protocols provide the client with a (probabilistic) *guarantee* that the server performed the requested computations correctly, without requiring the client to perform the computations herself.

Surprisingly powerful protocols for verifying outsourced computations were discovered within the computer science theory community in the 1980s and 1990s, in the form of interactive proofs and their brethren. However, these protocols were considered to be primarily of theoretical interest, far too inefficient for actual deployment. This thesis seeks to overturn this viewpoint.

We make progress along two interrelated directions. The first seeks to render interactive proofs suitable for applications involving massive data. To this end, we introduce two new computational models, dubbed *streaming interactive proofs* and *annotated data streams*. In these models, a streaming algorithm (modeling a cloud computing user lacking the resources to store the massive input locally) makes a single pass over an adversarially-ordered input, which is also observed by a powerful but untrusted prover (modeling a cloud computing service). Afterward, the prover sends the user the answer to queries about the stream, and proves that the answers are correct. Our study of these models reveals a rich theory, and we give protocols achieving essentially optimal tradeoffs between proof length and the verifier's

space usage for a variety of significant problems.

The second direction revisits some of the most powerful protocols in the theory literature and substantially improves their efficiency. Specifically, we describe a refinement of a powerful general-purpose interactive proof protocol originally due to Goldwasser, Kalai, and Rothblum (2008). Our refinements reduce the runtime of the prover in this protocol from $\Omega(S^3)$ to $O(S \log S)$, where S is the size of an arithmetic circuit computing the function of interest. We also show how to further reduce the prover's runtime for computations that are sufficiently structured, including arbitrary data parallel computation. We complement our analysis with implementations that demonstrate the genuine scalability of our protocols.

Contents

Title Page i
Abstract iii
Table of Contents v
Bibliographic Note vi
Acknowledgments viii
Dedication x

1 Introduction and Summary **1**

2 Background and Preliminaries **6**
2.1 Interactive Proofs 6
2.1.1 On Interactive Proofs for Languages Versus Functions 8
2.2 Schwartz-Zippel Lemma 8
2.3 Sum-Check Protocol 8

3 Annotations in Data Streams **14**
3.1 Our Contributions 15
3.2 Models, Notation, and Terminology 19
3.2.1 Communication Models 20
3.2.2 Data Stream Models 23
3.2.3 Relationship Between MA Protocols and Schemes 26
3.2.4 Additional Notation and Terminology 27
3.2.5 A Preliminary Lemma: Fingerprints 28
3.3 Index and Selection 29
3.4 A First Result for Frequent Items 34
3.5 Frequency Moments and Generalizations 37
3.5.1 Schemes for Frequency Moments 37
3.5.2 Lower Bounds on Frequency Moments 40
3.6 Frequent Items 43
3.7 Frequency-Based Functions 45
3.7.1 Frequency-Based Functions for Skewed Streams 49
3.8 Set and Multiset Inclusion 50
3.9 Matrix-Vector Multiplication 53
3.10 Graph Problems 57

3.10.1 Counting Triangles via Matrix Multiplication . . . 57
3.10.2 Bipartite Perfect Matching . . . 60
3.10.3 Bipartiteness . . . 61
3.10.4 Connectivity . . . 62
3.10.5 Totally Unimodular Integer Programs . . . 64
3.11 Simulating Non-Streaming Algorithms . . . 72
3.12 Improving the Runtime of the Prover and Experimental Results. . . . 76
3.12.1 Fast Proofs via Fast Fourier Transforms . . . 76
3.12.2 Breaking the Bottleneck . . . 78
3.12.3 Implications . . . 82
3.12.4 Experiments . . . 83
3.13 Related Work . . . 88
3.14 Open Problems . . . 90

4 Annotations for Sparse Data Streams 92
4.1 Introduction . . . 92
4.2 Overview of Results and Techniques . . . 93
4.3 Point Queries, Selection, and Heavy Hitters . . . 97
4.3.1 Upper Bounds . . . 97
4.3.2 Lower Bound . . . 102
4.3.3 Implications of the Lower Bound . . . 104
4.4 Prescient Schemes for Sparse Disjointness and Frequency Moments . . . 107
4.4.1 A Prescient Scheme for Sparse Disjointness . . . 107
4.4.2 A Prescient Scheme for Frequency Moments . . . 110
4.5 An Online Scheme for Frequency Moments . . . 114
4.5.1 An Overview of the Scheme . . . 114
4.5.2 Details of the Scheme . . . 116
4.5.3 A Scheme for MultiIndex: Proof of Lemma ?? . . . 118
4.5.4 Implications of the Online Scheme for Frequency Moments . . . 121
4.6 Graph Problems . . . 124
4.7 Non-strict Turnstile Update Model . . . 127
4.7.1 An Online Scheme . . . 127
4.7.2 An Online AMA Scheme . . . 128
4.8 Discussion and Open Problems . . . 132

5 The GKR Protocol 134
5.1 The GKR Protocol From 10,000 Feet . . . 134
5.2 Notation . . . 136
5.2.1 Technical Outline of the GKR Protocol . . . 137

6 Streaming Interactive Proofs 141
6.1 The Model . . . 141
6.2 Notation . . . 143
6.3 A General Theorem . . . 143
6.4 Second Frequency Moment . . . 146

6.5 Heavy Hitters . . . 154
6.6 Discussion . . . 156

7 Practical Verified Computation with Streaming Interactive Proofs 157
7.1 Overview and Statement of Results . . . 157
7.2 Methodology and Proofs . . . 160
7.2.1 Overview . . . 160
7.2.2 Decomposing $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ as Sums of Variable-Wise Indicator Functions . . . 161
7.2.3 Reducing to Verification of a Single Point . . . 164
7.2.4 Finishing the Proofs of Theorems ?? and ?? . . . 165
7.3 Discussion and Applicability . . . 166
7.3.1 Motivating Problems . . . 166
7.3.2 Circuits For Motivating Problems . . . 167
7.3.3 Circuit Design Issues . . . 173
7.4 Implementation and Experiments . . . 175
7.4.1 Implementation Details . . . 175
7.4.2 Experimental Results . . . 177
7.5 GPU Implementation . . . 179
7.5.1 Overview . . . 180
7.5.2 Architectural considerations . . . 181
7.5.3 GPU Evaluation . . . 185

8 Time-Optimal Interactive Proofs for Circuit Evaluation 193
8.1 Introduction . . . 193
8.1.1 Roadmap for Chapter . . . 196
8.2 Overview of the Ideas . . . 196
8.2.1 Achieving Optimal Prover Runtime for Regular Circuits . . . 196
8.2.2 Verifying General Data Parallel Computations . . . 198
8.2.3 A Special-Purpose Protocol for MatMult . . . 200
8.3 Technical Background: Making $\mathcal{V}$ Fast vs. Making $\mathcal{V}$ Streaming . . . 201
8.4 Time-Optimal Protocols for Circuit Evaluation . . . 202
8.4.1 Protocol Outline and Section Roadmap . . . 202
8.4.2 A Preliminary Lemma . . . 203
8.4.3 Polynomials for Specific Circuits . . . 203
8.4.4 Reusing Work . . . 208
8.4.5 A General Theorem . . . 214
8.5 Experimental Results . . . 221
8.5.1 Summary of Results . . . 221
8.5.2 Details . . . 222
8.6 Verifying General Data Parallel Computations . . . 228
8.6.1 Motivation . . . 229
8.6.2 Overview of the Protocol . . . 230
8.6.3 Technical Details . . . 232

8.7 Extensions 236
8.7.1 Binary Tree of Addition Gates 237
8.7.2 Optimal Space and Time Costs for MatMult 239
8.8 Discussion 246

9 Conclusion **248**
9.1 Related Work 248
9.2 Comparison With Other Approaches 250
9.3 Future Directions 252

Bibliography **254**

A Deferred Proofs from Chapter 8 **263**
A.1 Proof of Theorem ?? 263
A.2 Analysis for Pattern Matching 269

Bibliographic Note

Chapter 1 introduces the contents and structure of this thesis. Chapter 2 reviews relevant background on interactive proofs, including the well-known *sum-check* protocol of Lund, Fortnow, Karloff, and Nisan [77].

The contents of Chapter 3 are primarily derived from the papers *"Annotations in Data Streams"* [30], co-authored with Amit Chakrabarti, Graham Cormode, and Andrew McGregor, which is in submission at the time of writing, and *"Streaming Graph Computations with a Helpful Advisor"* (ESA 2010, *Algorithmica* 2013 [40]), co-authored with Graham Cormode and Michael Mitzenmacher. A preliminary version of [30] by Amit Chakrabarti, Graham Cormode, and Andrew McGregor appeared in ICALP 2009; the full version includes additional and strengthened results, but we have included all results in Chapter 3, including those that appeared in ICALP 2009. The modeling component of Chapter 3 is based on the manuscript *"Annotations for Sparse Data Streams,"* co-authored with Amit Chakrabarti, Graham Cormode, and Navin Goyal, which is in submission at the time of writing. Chapter 3 also contains some experimental work that appeared in *"Practical Verified Computation with Streaming Interactive Proofs"* (ITCS 2012 [39]), co-authored with Graham Cormode and Michael Mitzenmacher.

Chapter 4 is based entirely on the manuscript *"Annotations for Sparse Data Streams,"* co-authored with Amit Chakrabarti, Graham Cormode, and Navin Goyal. Chapter 5 describes background on the general-purpose interactive proof protocol due to Goldwasser, Kalai, and Rothblum [57]. Chapter 6 is based in part on the paper *"Verifying Computations with Streaming Interactive Proofs"* (VLDB 2012 [42]), co-authored with Graham Cormode and Ke Yi. Chapter 7 is based on the papers *"Practical Verified Computation with Streaming Interactive Proofs"* (ITCS 2012 [39]), co-authored with Graham Cormode

and Michael Mitzenmacher, and *"Verifiable Computation with Massively Parallel Interactive Proofs"* (HotCloud 2012 [99]), co-authored with Mike Roberts, Michael Mitzenmacher, and Hanspeter Pfister. Chapter 8 is based on the paper *"Time-Optimal Interactive Proofs for Circuit Evaluation"* (CRYPTO 2013 [97]).

Acknowledgments

My experience in graduate school has been shaped by a number of remarkable people. First and foremost, I am extremely fortunate to have had Michael Mitzenmacher as my adviser. Michael has been a constant source of encouragement and guidance from day one, and unwaveringly generous with his time, despite juggling Area Dean duties for three of my four years in graduate school. While always encouraging me to pursue my own ideas and interests, he has had a huge impact on the problems I've chosen to work on, and he is a big reason for any success I've had in solving them. When telling people that I am his student, Michael likes to suggest that it's actually the other way around, but nothing could be further from the truth. Quite simply, he has been a wonderful adviser.

I am also deeply grateful to Graham Cormode, who has been a mentor, collaborator, friend, and heroic tolerator of nonsensical emails since well before I arrived at Harvard. Graham advised my project during the 2007 REU at DIMACS, and has been a constant source of advice and ideas ever since. It is to him that I owe my interest in streaming algorithms, and his and Michael's influence are in every page of this thesis. A suggestion by Graham during my first week of graduate school set off the entire line of work described here, and he has been along for every step of the journey. Graham has been a delight to work with, and I look forward to continuing to do so for years to come.

I would also like to thank Yiling Chen for serving on my qualifying committee, and Margo Seltzer and Salil Vadhan for serving on my thesis committee: your guidance over the years has been invaluable. Thanks also to Rocco Servedio and Li-Yang Tan, to whom I owe my interests in concrete complexity and learning theory; to Amit Chakrabarti and Andrew McGregor, who I first met back in 2007 while at DIMACS, and who have been role models and collaborators throughout my time in graduate school; to Frank McSherry for hosting me

during an enjoyable internship at Microsoft Research, and to Parikshit Gopalan, who was unceasingly patient in talking to me about a particularly frustrating (some might even say traumatic) problem.

I owe much of the richness of my time at Harvard to the residents of Maxwell Dworkin 138. Kai-Min, Varun, Zhenming, Anna, Colin, Jon, Thomas, Scott, Jiayiang, Jiapeng, Mark, and Tom: I have been fortunate to count you as my friends, collaborators, and even roommates, and I cannot imagine graduate school without you. You made the ride infinitely more fun and productive, and I promise to clean up on the way out. Thanks also to the postdocs on the other side of the wall: Brendan, Andrew, and Karthik.

These last four years would not have been nearly as much fun without the friends who have been with me in Boston (or my occasional second home, New Haven) the whole time: Aaron, Kian, Alicia, Mat, Henry, Emily. Thank you all!

Finally, and especially, to my family: What can I say, other than thank you for 25 years of love and support, and patiently nodding at every mention of the word "logarithm".

This work was made possible by the Department of Defense (DoD) through the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program, and an NSF Graduate Research Fellowship.

To my parents, Florence and Herb, with love and gratitude.

Chapter 1

Introduction and Summary

Protocols for verifiable computation enable a computationally weak *verifier* $\mathcal{V}$ to offload computations to a powerful but untrusted *prover* $\mathcal{P}$. These protocols aim to provide the verifier with a guarantee that the prover performed the requested computations correctly, without requiring the verifier to perform the computations herself.

Surprisingly powerful protocols for verifiable computation were discovered within the computer science theory community several decades ago, in the form of interactive proofs and their brethren: interactive arguments and probabilistically checkable proofs (PCPs). In these protocols, the prover $\mathcal{P}$ solves a problem using her (possibly vast) computational resources, and tells $\mathcal{V}$ the answer. $\mathcal{P}$ and $\mathcal{V}$ then have a conversation, i.e., they engage in a randomized protocol involving the exchange of one or more messages. During this conversation, $\mathcal{P}$'s goal is to convince $\mathcal{V}$ that the answer is correct.

More precisely, any protocol for verifiable computation must satisfy two properties. The first is called *completeness*; this property roughly requires that an honest prover will be able to convince the verifier that the claimed answer is correct. The second property is called *soundness*; this property roughly requires that if the claimed answer is incorrect, then the verifier will reject the answer with high probability no matter what the prover says to try

to convince her otherwise. In the case of interactive proofs, soundness must hold against computationally unbounded provers.

Results quantifying the power of interactive proofs, interactive arguments, and PCPs represent some of the most celebrated results in all of computational complexity theory, but until recently they were considered primarily of theoretical interest, far too inefficient for actual deployment.

However, the surging popularity of cloud computing has brought renewed interest in positive applications of protocols for verifiable computation. A typical motivating scenario is as follows. A business – call it ArthurSystems – processes billions or trillions of transactions a day. The volume is sufficiently high that ArthurSystems cannot or will not store and process the transactions on its own. Instead, it offloads the processing to a commercial cloud computing service – call it MerlinSystems. The offloading of any computation raises issues of trust. ArthurSystems may be concerned about relatively benign errors: perhaps MerlinSystems dropped some of the transactions, executed a buggy algorithm, or experienced an uncorrected hardware fault. Alternatively, ArthurSystems may be more cautious and fear that MerlinSystems is deliberately deceptive or has been externally compromised. Either way, each time ArthurSystems poses a query to MerlinSystems, it may demand that MerlinSystems provide not only the answer but also some proof that the returned answer is correct. This is precisely what protocols for verifiable computation accomplish, with MerlinSystems acting as the prover in the protocol, and ArthurSystems acting as the verifier.

This thesis describes progress toward achieving protocols for verifiable computation that are efficient enough for use in the real world. All of our results hold in variants of the interactive proofs model, meaning they are secure even against computationally unbounded cheating provers.

Structure of this thesis

- Chapter 2 introduces the standard interactive proofs model, discusses seminal work in the area, and reviews prior work that forms the foundation of this thesis, including the well-known *sum-check* protocol of Lund, Fortnow, Karloff, and Nisan [77].

- Chapter 3 introduces the annotated data streams model. In this setting, a computationally weak verifier (modeling ArthurSystems in the scenario of the Introduction), who lacks the resources to store the entire input locally, is given access to a powerful but untrusted prover (modeling the cloud computing service). The verifier must execute within the confines of the restrictive *data streaming* paradigm, i.e., it must process the input sequentially in whatever order it arrives, using space that is substantially sublinear in the total size of the input. The prover is allowed to annotate the data stream as it is read, with the goal of convincing the verifier of the correct answer. The streaming restriction for the verifier fits the cloud computing setting well, as the verifier's streaming pass over the input can occur while uploading data to the cloud.

  In this chapter, we present annotated data streaming protocols achieving non-trivial tradeoffs between the amount of annotation used and the space required to verify it. We also prove lower bounds on such tradeoffs, often nearly matching the upper bounds, via notions related to Merlin-Arthur communication complexity. Our annotated data streaming protocols, as well as our lower bounds, cover classic data stream problems such as selection and frequency moments, and fundamental graph problems such as triangle-freeness and connectivity. Finally, this chapter describes how to exploit sophisticated Fast Fourier Transform algorithms to ensure that the prover in our protocols runs quickly, as well as experimental results demonstrating scalability.

- While many of the protocols of Chapter 3 are optimal, such optimality holds only for

streams whose length is commensurate with the size of the data universe. In contrast, many real-world data sets are relatively sparse, including graphs that contain only $o(n^2)$ edges, and IP traffic streams that contain much fewer than the total number of possible IP addresses, 2^{128} in IPv6. In Chapter 4, we describe protocols that allow both the annotation and the space usage to be sublinear in the total number of stream updates rather than the size of the data universe. On the other hand, we give a lower bound that, for the first time, rules out smooth tradeoffs between annotation and space usage for a specific problem.

- Chapter 5 describes background on the powerful general-purpose interactive proof protocol due to Goldwasser, Kalai, and Rothblum [57].

- Chapter 6 introduces the model of streaming interactive proofs, which extends the annotated data streams model to allow for multiple rounds of interaction between the prover and verifier. This chapter presents some preliminary results on streaming interactive proofs, and in particular reveals an exponential separation between the annotated data stream and streaming interactive proofs models.

- Chapter 7 revisits Goldwasser, Kalai, and Rothblum's general-purpose interactive proof protocol (henceforth GKR protocol), and shows how to reduce the runtime of the prover from $\Omega(S^3)$ to $O(S \log S)$, where S is the size of an arithmetic circuit computing the function of interest. This chapter also describes a full implementation of the protocol, demonstrating much greater scalability than one might have expected. Finally, we describe a parallel implementation of the protocol that leverages Graphics Processing Units (GPUs), and experimentally demonstrate the GKR protocol's substantial amenability to parallelization.

- Chapter 8 describes further refinements and extensions of the GKR protocol. A major

message of this chapter is that the more structure that exists in a computation, the more efficiently it can be verified, and that this structure exists in many real-world computations. Our first refinement applies to circuits whose wiring pattern is sufficiently "regular"; for these circuits, we bring the runtime of the prover down from $O(S \log S)$ as achieved in Chapter 7 to $O(S)$. That is, our prover can evaluate the circuit with a guarantee of correctness, while suffering only a constant-factor blowup in work compared to evaluating the circuit without any guarantee. We argue that our refinements capture a large class of circuits, and complement our theoretical results with experiments on problems such as matrix multiplication and determining the number of distinct elements in a data stream. Experimentally, our refinements yield a prover that is less than 10x slower than a C++ program that simply evaluates the circuit. Leveraging similar techniques, we also describe a protocol targeted at general data parallel computation. Compared to prior work, this protocol can more efficiently verify complicated computations as long as the computation is applied independently to many different pieces of data. Finally, we describe a special-purpose protocol for matrix multiplication that is of interest in its own right.

- Chapter 9 concludes, by comparing the approach to verifiable computation taken in this thesis to several parallel lines of work, and describing some important directions for future work.

Chapter 2

Background and Preliminaries

This chapter introduces the standard interactive proofs model, elaborates on seminal work in the area, and reviews prior work that forms the foundation of this thesis. The main technical tools upon which we draw in subsequent chapters are the well-known *sum-check* protocol of Lund, Fortnow, Karloff, and Nisan [77] (introduced in this chapter), and the powerful general-purpose interactive proof protocol due to Goldwasser, Kalai, and Rothblum [57] (introduced in Chapter 5).

2.1 Interactive Proofs

We give here a standard definition of interactive proofs.

Definition 2.1.1. An interactive proof system for a language $L \subseteq \{0,1\}^*$ consists of a probabilistic polynomial time verifier $\mathcal{V}$ and a prover $\mathcal{P}$ who are given a common input $x \in \{0,1\}^n$. $\mathcal{P}$ and $\mathcal{V}$ exchange a sequence of messages to produce a transcript $t = (\mathcal{V}(r), \mathcal{P})(x)$, where r denotes $\mathcal{V}$'s internal randomness. After the transcript is produced, $\mathcal{V}$ decides whether to output accept or reject based on r, t, and x. We denote by $\text{out}(\mathcal{V}, x, r, \mathcal{P})$ the output of verifier $\mathcal{V}$ on input x given prover strategy $\mathcal{P}$ and that $\mathcal{V}$'s internal randomness is equal to r.

We say the interactive proof system has completeness error δ_c and soundness error δ_s if the following two properties hold.

1. *(Completeness)* There exists a prover strategy $\mathcal{P}$ such that for every $x \in L$, $\Pr[\text{out}(\mathcal{V}, x, r, \mathcal{P}) = \text{accept}] \geq 1 - \delta_c$.

2. *(Soundness)* For every $x \notin L$ and every prover strategy $\mathcal{P}'$, $\Pr[\text{out}(\mathcal{V}, x, r, \mathcal{P}') = \text{accept}] \leq \delta_s$.

We say interactive proof system is valid if $\delta_c, \delta_s \leq 1/3$. The complexity class IP is the class of all languages possessing valid interactive proof systems.

If $\mathcal{V}$ and $\mathcal{P}$ exchange at most m messages for every pair (x, r), we refer to $\lceil m/2 \rceil$ as the *round complexity* of the interactive proof system.

Several clarifying remarks are in order. First, notice that the soundness requirement in Definition 2.1.1 is required to hold even against computationally unbounded provers P'. Second, Definition 2.1.1 implicitly assumes that the total number of messages exchanged by $\mathcal{P}$ and $\mathcal{V}$ is poly(n), as $\mathcal{V}$ must run in poly(n) time over the entire course of the interaction. Third, notice that in an interactive proof system, $\mathcal{V}$'s randomness is internal, and in particular is not visible to the prover.

Interactive proofs were introduced in 1985 by Goldwasser, Micali, and Rackoff [58]. At the same conference, Babai [9] independently introduced the Arthur-Merlin class hierarchy, which captures constant-round interactive proof systems, with the additional requirement that the verifier's randomness is public – that is, visible to the prover. Goldwasser and Sipser [59] subsequently proved that any constant-round private coin interactive proof system can be simulated by a constant-round public coin system.

See the chapter by Arora and Barak [8, Chapter 8] for an excellent overview of foundational work on interactive proofs.

2.1.1 On Interactive Proofs for Languages Versus Functions

Given a language L, let $f_L(x) = 1$ if $x \in L$ and $f_L(x) = 0$ otherwise. Notice that Definition 2.1.1 requires that if $f_L(x) = 1$ then there is some prover that will cause the verifier to accept with high probability, and if $f_L(x) = 0$ then there is no such prover. In particular, for $x \notin L$, Definition 2.1.1 does *not* require there to be a "convincing proof" of the fact that $f_L(x) = 0$.

Later in this thesis, we will consider interactive proofs for non-Boolean valued functions f, and we will typically give protocols that allow the the prover to convince the verifier of the value of $f(x)$ for *all* x. Equivalently, we will give interactive proof systems for the language $L_f = \{(x, y) : y = f(x)\}$.

2.2 Schwartz-Zippel Lemma

Throughout this thesis, we will frequently make use of the following basic property of polynomials. This lemma is commonly known as the Schwartz-Zippel lemma [88, 109].

Lemma 2.2.1. *Let $\mathbb{F}$ be any field, and let $f : \mathbb{F}^m \to \mathbb{F}$ be a nonzero polynomial of total degree at most d. Then on any finite set $S \subseteq \mathbb{F}$,*

$$\Pr_{x \leftarrow S^m}[f(x) = 0] \leq d/|S|.$$

In words, if x is chosen uniformly at random from S^m, then the probability that $f(x) = 0$ is at most $d/|S|$. In particular, any two distinct polynomials of total degree at most d can agree on at most $d/|S|$ fraction of points in S^m.

2.3 Sum-Check Protocol

The sum-check protocol of Lund, Fortnow, Karloff, and Nisan [77] underlies many of the results of this thesis. We present what is now the standard variant of this protocol.

Suppose we are given a v-variate polynomial g defined over a finite field $\mathbb{F}$. The purpose of the sum-check protocol is to compute the sum:

$$H := \sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_v \in \{0,1\}} g(b_1, \ldots, b_v).$$

In applications, this sum will be over a very large number of terms, so the verifier will not have the resources to compute the sum on her own. Instead, she uses the sum-check protocol to force the prover to compute the sum for her.

Remark 1. In full generality, the sum-check protocol can compute the sum $\sum_{b \in B^m} g(b)$ for any $B \subseteq \mathbb{F}$, but we will only require $B = \{0, 1\}$ for most of the applications in this thesis.

For presentation purposes, we assume here that the verifier has oracle access to g, i.e., $\mathcal{V}$ can evaluate $g(r_1, \ldots, r_v)$ for a randomly chosen vector $(r_1, \ldots, r_v) \in \mathbb{F}^v$ with a single query to an oracle, though this will not be the case in applications. In our applications, $\mathcal{V}$ will either be able to efficiently evaluate $g(r_1, \ldots, r_v)$ on her own, or if this is not the case, $\mathcal{V}$ will ask the prover to *tell her* $g(r_1, \ldots, r_v)$, and $\mathcal{P}$ will subsequently prove this claim is correct via further applications of the sum-check protocol.

The protocol proceeds in v rounds as follows. In the first round, the prover sends a polynomial $g_1(X_1)$, and claims that $g_1(X_1) = \sum_{(x_2, \ldots, x_v) \in \{0,1\}^{v-1}} g(X_1, x_2, \ldots, x_v)$. Observe that if g_1 is as claimed, then $H = g_1(0) + g_1(1)$.

Throughout the remainder of this chapter, we use $\deg_i(p)$ to denote the degree of p in variable i. Observe that the polynomial $g_1(X_1)$ has degree $\deg_1(g)$. Hence g_1 can be specified with $\deg_1(g) + 1$ field elements, for example by sending the evaluation of g_1 at each point in the set $\{0, 1, \ldots, \deg_1(g)\}$.

Then, in round $j > 1$, $\mathcal{V}$ chooses a value r_{j-1} uniformly at random from $\mathbb{F}$ and sends r_{j-1} to $\mathcal{P}$. We will often refer to this step by saying that variable $j - 1$ gets *bound* to value r_{j-1}.

In return, the prover sends a polynomial $g_j(X_j)$, and claims that

$$g_j(X_j) = \sum_{(x_{j+1},\ldots,x_v)\in\{0,1\}^{v-j}} g(r_1,\ldots,r_{j-1},X_j,x_{j+1},\ldots,x_v). \tag{2.1}$$

The verifier compares the two most recent polynomials by checking $g_{j-1}(r_{j-1}) = g_j(0) + g_j(1)$, and rejecting otherwise. The verifier also rejects if the degree of g_j is too high: each g_j should have degree $\deg_j(g)$, the degree of variable x_j in g.

In the final round, the prover has sent $g_v(X_v)$ which is claimed to be $g(r_1,\ldots,r_{v-1},X_v)$. $\mathcal{V}$ now checks that $g_v(r_v) = g(r_1,\ldots,r_v)$ (recall that we assumed $\mathcal{V}$ has oracle access to g). If this test succeeds, and so do all previous tests, then the verifier is convinced that $H = g_1(0) + g_1(1)$.

The protocol is summarized in Figure 2.1.

The following proposition formalizes the completeness and soundness properties of the sum-check protocol.

Proposition 2.3.1. *Let g be a v-variate polynomial of total degree at most d defined over a finite field $\mathbb{F}$. For any $H \in \mathbb{F}$, let L be the language of of polynomials g (given as an oracle) such that*

$$H = \sum_{b_1\in\{0,1\}} \sum_{b_2\in\{0,1\}} \cdots \sum_{b_v\in\{0,1\}} g(b_1,\ldots,b_v).$$

The sum-check protocol is an interactive proof system for L with completeness error $\delta_c = 0$ and soundness error $\delta_s \leq vd/|\mathbb{F}|$.

Proof. Completeness is evident: if the prover sends the prescribed polynomial $g_j(X_j)$ at all rounds j, then $\mathcal{V}$ will accept with probability 1.

The proof of soundness is by induction on v. In the case $v = 1$, $\mathcal{P}$'s only message specifies a degree d univariate polynomial $g_1(X_1)$. If $g_1(X_1) \neq g(X_1)$, then by Lemma 2.2.1,

- Fix an $H \in \mathbb{F}$.
- In the first round, $\mathcal{P}$ sends the univariate polynomial

$$g_1(X_1) := \sum_{x_2,\ldots,x_v \in \{0,1\}^{v-1}} g(X_1, x_2, \ldots, x_v),$$

  and sends g_1 to $\mathcal{V}$. $\mathcal{V}$ checks that g_1 is a univariate polynomial of degree at most $\deg_1(g)$, and that $H = g_1(0) + g_1(1)$, rejecting if not.
- $\mathcal{V}$ chooses a random element $r_1 \in \mathbb{F}$, and sends r_1 to $\mathcal{P}$.
- In the jth round, for $1 < j < v$, $\mathcal{P}$ sends to $\mathcal{V}$ the univariate polynomial

$$g_j(X_j) = \sum_{(x_{j+1},\ldots,x_v) \in \{0,1\}^{v-j}} g(r_1, \ldots, r_{j-1}, X_j, x_{j+1}, \ldots, x_v).$$

  $\mathcal{V}$ checks that g_j is a univariate polynomial of degree at most $\deg_j(g)$, and that $g_{j-1}(r_{j-1}) = g_j(0) + g_j(1)$, rejecting if not.
- $\mathcal{V}$ chooses a random element $r_j \in \mathbb{F}$, and sends r_j to $\mathcal{P}$.
- In Round v, $\mathcal{P}$ sends the univariate polynomial

$$g_v(X_v) = g(r_1, \ldots, r_{v-1}, X_v)$$

  to $\mathcal{V}$. $\mathcal{V}$ checks that g_v is a univariate polynomial of degree at most $\deg_v(g)$, rejecting if not.
- $\mathcal{V}$ chooses a random element $r_v \in \mathbb{F}$ and evaluates $g(r_1, \ldots, r_v)$ with a single oracle query to g. $\mathcal{V}$ checks that $g_v(r_v) = g(r_1, \ldots, r_v)$, rejecting if not.
- If $\mathcal{V}$ has not yet rejected, $\mathcal{V}$ halts and accepts.

Figure 2.1: Description of Sum-Check Protocol.

$g_1(r_1) \neq g(r_1)$ with probability at least $1 - d/|\mathbb{F}|$ over the choice of r_1, and hence $\mathcal{V}$'s final check will cause $\mathcal{V}$ to reject with probably at least $1 - d/|\mathbb{F}|$.

Assume by way of induction that for all $v-1$-variate polynomials, the sum-check protocol has soundness error at most $(v-1)d/|\mathbb{F}|$. Suppose $\mathcal{P}$ sends a polynomial $h_1(X_1) \neq g_1(X_1) = \sum_{x_2,\ldots,x_v \in \{0,1\}^{v-1}} g(X_1, x_2, \ldots, x_v)$ in Round 1. Then by Lemma 2.2.1, $h_1(r_1) \neq g_1(r_1)$ with probability at least $1 - d/|\mathbb{F}|$. Conditioned on this event, $\mathcal{P}$ is left to prove the false claim in Round 2 that $h_1(r_1) = \sum_{x_2,\ldots,x_v \in \{0,1\}^{v-1}} g(r_1, x_2, \ldots, x_v)$. Since $g(r_1, x_2, \ldots, x_v)$ is a $v-1$-variate polynomial of total degree d, the inductive hypothesis implies the $\mathcal{V}$ will reject at some subsequent round of the protocol with probability at least $1 - d(v-1)/|\mathbb{F}|$. Therefore, $\mathcal{V}$ will reject with probability at least

$$1 - \Pr[h_1(r_1) \neq g_1(r_1)] - (1 - \Pr[\mathcal{V} \text{ rejects in some Round } j > 1 | h_1(r_1) \neq g_1(r_1)])$$

$$\geq 1 - \frac{d}{|\mathbb{F}|} - \frac{d(v-1)}{|\mathbb{F}|} = 1 - \frac{dv}{|\mathbb{F}|}.$$

□

Discussion of costs. Observe that there is one round in the sum-check protocol for each of the v variables of g. The total communication is $\sum_{i=1}^{v} \deg_i(g) + 1 = v + \sum_{i=1}^{v} \deg_i(g)$ field elements. In particular, if $\deg_i(g) = O(1)$ for all j, then the communication cost is $O(v)$ field elements.

The running time of the verifier over the entire execution of the protocol is proportional to the total communication, plus the cost of a single oracle query to g to compute $g(r_1, \ldots, r_v)$.

Determining the running time of the prover is less straightforward. Recall that $\mathcal{P}$ can specify g_j by sending for each $i \in \{0, \ldots, \deg_j(g)\}$ the value:

$$g_j(i) = \sum_{(x_{j+1},\ldots,x_v) \in \{0,1\}^{v-j}} g(r_1, \ldots, r_{j-1}, i, x_{j+1}, \ldots, x_v). \tag{2.2}$$

An important insight is that the number of terms defining the value $g_j(i)$ in Equation (2.2) falls geometrically with j: in the jth sum, there are only 2^{v-j} terms, each corresponding to a Boolean vector in $\{0,1\}^{v-j}$. Thus, the total number of terms that must be evaluated over the course of the protocol is $\sum_{j=1}^{v} \deg_j(g) 2^{v-j} = O(2^v)$ if $\deg_j(g) = O(1)$ for all j. Consequently, if $\mathcal{P}$ is given oracle access to g, then $\mathcal{P}$ will require just $O(2^v)$ time.

In all of our applications in this thesis, $\mathcal{P}$ will not have oracle access to the truth table of g, and the key to many of our results in Chapters 6, 7, and 8 is to show that in our applications $\mathcal{P}$ can nonetheless evaluate g at all of the necessary points in close to $O(2^v)$ total time.

Chapter 3

Annotations in Data Streams

The surging popularity of commercial cloud computing services has rendered the following scenario increasingly plausible. A business such as ArthurSystems from Chapter 1 processes billions or trillions of transactions a day. The volume is sufficiently high that ArthurSystems cannot or will not store and process the transactions on its own. Instead, it offloads the processing to a commercial cloud computing service – MerlinSystems from Chapter 1.

The offloading of any computation raises issues of trust. ArthurSystems may be concerned about relatively benign errors: perhaps the cloud dropped some of the transactions, executed a buggy algorithm, or experienced an uncorrected hardware fault. Alternatively, ArthurSystems may be more cautious and fear that the cloud operator is deliberately deceptive or has been externally compromised. Either way, each time ArthurSystems poses a query to MerlinSystems, it may demand that the cloud provide not only the answer but also some proof that the returned answer is correct.

In this chapter and the next, we consider a computationally weak *verifier* (modeling ArthurSystems in the above scenario), who lacks the resources to store the entire input locally, and is given access to a powerful but untrusted *prover* (modeling the cloud computing service). The verifier must execute within the confines of the restrictive *data streaming*

paradigm, i.e., it must process the input sequentially in whatever order it arrives, using space that is substantially sublinear in the total size of the input. The prover is allowed to annotate the data stream as it is read, with the goal of convincing the verifier of the correct answer. The streaming restriction for the verifier fits the cloud computing setting well, as the verifier's streaming pass over the input can occur while uploading data to the cloud.

Our approach is naturally related to interactive proofs as introduced in Chapter 2, as well as Merlin-Arthur communication complexity (introduced in Section 3.2) but differs in two important regards. Firstly, the verifier must process both the original data and the advice provided by the helper under the usual restrictions of the data stream model. Secondly, we focus on annotations that can be provided *online*, i.e., annotation that depends only on data that has arrived before the annotation is written. Note that in interactive proofs and Merlin-Arthur communication, it is assumed that the helper is omniscient and that the advice he provides can take into account data held by any of the players. In the stream model, this would correspond to *prescience*, where the annotation in the stream at any particular position may depend on data that is yet to arrive. In contrast, we are primarily interested in designing algorithms with online annotation; this corresponds to a helper who sees the data concurrently with the verifier.

3.1 Our Contributions

Our first contribution in this chapter is to formally define the relevant models: traditional and online Merlin-Arthur communication, and streaming models with either prescient or online annotations. We then investigate the complexity of a range of problems in these models, including selection, frequency moments, and graph problems such as triangle-counting, connectivity, and bipartite perfect matching. Estimating frequency moments in particular has

become a canonical problem when exploring variants of the data stream model such as random order streams [29] and read/write streams [12]. In contrast, we give annotation protocols – which we call schemes – for solving the frequency moments problem and its generalizations *exactly*. The fact that we obtain schemes that solve the exact frequency moments problem is essential, as it allows us to use these schemes as primitives to solve more complicated problems that on their surface appear to have nothing to do with frequency moments.

We now give a detailed overview of the results in this chapter. We use the shorthand "(c_a, c_v)-scheme" for streaming algorithm that uses $O(c_a)$ bits of annotation and requires $O(c_v)$-space; a scheme could be either prescient or online. In general, our streams have length N and consist of tokens from the data universe $[n] := \{1, 2, \ldots, n\}$. In the case of graph streams, we consider tokens from the universe $[n] \times [n]$.

Selection. The problem of finding the median of N values in the range $[n]$ highlights the difference between prescient and online annotation. For arbitrary positive integers x and y, with $xy \geq n$, we present an online $(x \log n, y \log n)$-scheme. Furthermore, we show that this trade-off is optimal up to polylogarithmic factors. In contrast, a trivial $O(\log n)$-space algorithm can verify $O(\log n)$ bits of prescient annotation, implying a prescient $(\log n, \log n)$-scheme.

Frequency Moments and Frequent Items. We next consider properties of $\{f_i\}_{i \in [n]}$ where f_i is the frequency of the token "i" in the stream. For arbitrary positive integers x and y, with $xy \geq n$, we present an online $(\phi^{-1} \log^2 n + x \log n, y \log n)$-scheme that computes the set of tokens whose frequency exceeds ϕN.

For any positive integers x, y with $xy \geq n$, we also present an online $(k^2 x \log n, ky \log n)$-scheme that computes the kth frequency moment $F_k := \sum_i f_i^k$ exactly, where k is a positive integer. This scheme is based on a beautiful, nearly optimal MA communication protocol for

the SET-DISJOINTNESS problem (henceforth, DISJ) developed by Aaronson and Wigderson [3] using algebraic techniques analogous to those in the sum-check protocol presented Chapter 2. We will informally refer to this class of schemes and techniques as *sum-check schemes* and techniques. The tradeoff between annotation length and space usage of our F_k scheme is optimal up to polylogarithmic factors even if the algorithm is allowed to use prescient annotation. To prove this, we present the first Merlin-Arthur communication bounds for multi-party set-disjointness.

Additionally, we generalize the scheme for F_k to any *frequency-based function*, i.e., a function of the form $\sum_{i\in[n]} g(f_i)$ for some $g : \mathbb{Z}_+ \to \mathbb{Z}_+$. Assuming $N = O(n)$, we obtain a prescient $(n^{2/3}\log n, n^{2/3}\log n)$-scheme and an online $(n^{2/3}\log^{4/3} n, n^{2/3}\log^{4/3} n)$ scheme for this important class of functions, as well as improved schemes for skewed data streams.

Matrix-Vector Multiplication Further extending the sum-check techniques, we give an online scheme for multiplying an $n\times n$ matrix with integer entries by an n-dimensional integer vector. Specifically, for any positive integers x and y with $xy \geq n^2$ and $x \geq n$, we give an $(x\log n, y\log n)$ protocol for matrix-vector multiplication. These tradeoffs are optimal up to logarithmic factors, and this scheme serves as a critical primitive in the development of the schemes for graph problems described next.

Graph Problems. We present optimal or nearly-optimal schemes for the following graph problems.

- **Counting Triangles.** For any positive integers x and y, with $xy \geq n^3$, we present an online $(x\log n, y\log n)$-scheme for counting triangles in the graph. We also give an online $(n^2\log n, \log n)$-scheme for this problem.

- **Connectivity and Bipartiteness.** For any positive integers x and y with $x \geq n$ and

$xy \geq n^2$, we present online $(x \log n, y \log n)$-schemes for determining whether a graph is connected or bipartite.

- **Bipartite Perfect Matching.** For any positive integers x and y with $x \geq \log n$ and $xy \geq n^2$, we present online $(x \log n, y \log n)$-schemes for solving bipartite perfect matching. These tradeoffs between annotation length and space usage for the verifier are optimal up to logarithmic factors.

- **Shortest s-t Path.** Given a directed graph G with non-negative integer edge weights and designated nodes s and t, let d be the maximum distance between s and any node reachable from s. For any positive integers x, y such that $xy \geq dn^2$ and $x \geq dn$, we give an online $(x \log n, y \log n)$ protocol for computing the shortest s-t path. These tradeoffs are essentially optimal for small-diameter graphs. The scheme relies on linear programming duality and the total unimodularity of the shortest s-t path problem, in conjunction with sum-check techniques.

- **Minimum Weight Bipartite Perfect Matching (MWBPM).** Suppose we are given a bipartite graph G with non-negative integer edge weights. For any positive integers x, y such that $xy \geq n^3 w_{\max}$ and $x \geq n w_{\max}$, we give an $(x \log n, y \log n)$ protocol for MWBPM, where $w_{\max}$ is the maximum weight of any edge.

Algorithm Simulation. Next, we present a general-purpose online scheme for simulating an arbitrary deterministic algorithm in the standard RAM model. In this scheme, the verifier requires only $\log n$ space, and the annotation length is $O(m \log n + T \log n)$, where T is the runtime of the algorithm, where m is the input length. As an immediate corollary, we derive online $(m \log n, \log n)$-schemes for the following problems on graphs with m edges on n vertices: connectivity, bipartiteness, minimum spanning tree, and maximum matching. We

show that our algorithms are optimal in many cases, in the sense that the annotation length cannot be decreased without increasing the space usage above $O(\log n)$.

Implementation and Experiments. We close the chapter by presenting experimental results based on an implementation of several of our schemes (F_2 and matrix-vector multiplication). Using Fast Fourier Transform techniques, we bring the runtime of the prover in our sum-check schemes down to $O(n \log n)$, from $O(c_a n)$ in a naive implementation, where c_a is the annotation length. Our experimental results demonstrate that our schemes require just a few MBs of space and annotation even when the data universe size is in the trillions, that our verifier additionally runs extremely quickly (processing millions of stream updates per second), and that our FFT-based prover implementation processes several hundred thousand updates per second, allowing the prover implementation to easily scale to streams over data universes of size in the billions.

3.2 Models, Notation, and Terminology

Many of the algorithms (schemes) in Chapter 4 use randomization in subtle ways, making it important to properly formalize several models of computation. While the schemes presented in this chapter use randomization in a more straightforward manner, our careful treatment of randomness as we introduce the models will pay dividends in Chapter 4.

We begin with Merlin–Arthur communication models, a topic first studied by Babai, Frankl and Simon [10], which we eventually use to derive lower bounds. We then turn to annotated data stream models. At the end of the section we set up some notation and terminology for the rest of Chapters 3 and 4.

3.2.1 Communication Models

Let $F : X \times Y \to \{0,1\}$ be a function, where X and Y are both finite sets. This naturally gives a 2-player number-in-hand communication problem, where the first player, Alice, holds an input $x \in X$, and the second player, Bob, holds an input $y \in Y$. The players wish to compute $F(x,y)$ by executing a (possibly randomized) communication protocol that correctly outputs $F(x,y)$ with "high" probability. In Merlin–Arthur communication, there is additionally a "super-player," called Merlin, who knows the entire input (x,y), and can help Alice and Bob by interacting with them. The precise pattern of interaction matters greatly and gives rise to distinct models. Merlin's goal is to get Alice and Bob to output "1" regardless of the actual value of $F(x,y)$, and so Merlin is not to be blindly trusted. Intuitively, one can think of $F(x,y)$ as the indicator function of a property of (x,y), and Merlin's goal is to convince Alice and Bob that the input satisfies the property (even if it does not).

One important departure we make from prior work is that *we allow Merlin to use private random coins* during the protocol. Most prior work on MA (and AM) communication (e.g. [10, 70, 71]) defined Merlin to be deterministic, which does not make a difference in the basic setting where Merlin knows the entire input (x,y) when communicating with Alice and Bob. But in this work we are concerned with "online MA" models, where the distinction does matter, and these online MA models are in close correspondence with the annotated data stream models that are our eventual topic of study.

MA Communication. In a Merlin–Arthur protocol (henceforth, "MA protocol") for F, Merlin begins by sending a help message $\mathfrak{h}(x,y,r_M)$, using a private random string r_M. The help message is seen by both Alice and Bob. Then Alice and Bob (the pair that constitutes the entity "Arthur") run a randomized communication protocol $\mathcal{Q}$, using a public random

string r_A, eventually outputting a bit $\text{out}^{\mathcal{Q}}(x, y, r_A, \mathfrak{h})$. Importantly, r_A is not known to Merlin at the time he sends $\mathfrak{h}$. The protocol $\mathcal{Q}$ is δ_s-sound and δ_c-complete if there exists a function $\mathfrak{h} : X \times Y \times \{0,1\}^* \to \{0,1\}^*$ such that the following conditions hold.

1. If $F(x, y) = 1$ then $\Pr_{r_M, r_A}[\text{out}^{\mathcal{Q}}(x, y, r_A, \mathfrak{h}(x, y, r_M)) = 0] \leq \delta_c$.

2. If $F(x, y) = 0$ then $\forall\, \mathfrak{h}' \in \{0,1\}^* : \Pr_{r_A}[\text{out}^{\mathcal{Q}}(x, y, r_A, \mathfrak{h}') = 1] \leq \delta_s$.

We define $\text{err}(\mathcal{Q})$ to be the minimum value of $\max\{\delta_s, \delta_c\}$ such that the above conditions hold. We define the *help cost* $\text{hcost}(\mathcal{Q})$ to be $1 + \max_{x,y,r_M} |\mathfrak{h}(x, y, r_M)|$ (forcing $\text{hcost} \geq 1$, even for traditional Merlin-free protocols), and the *verification cost* $\text{vcost}(\mathcal{Q})$ to be the maximum number of bits communicated by Alice and Bob over all x, y and r_A. We define $\text{MA}_\delta(F) = \min\{\text{vcost}(\mathcal{Q}) + \text{hcost}(\mathcal{Q}) : \mathcal{Q} \text{ is an MA protocol for } F \text{ with } \text{err}(\mathcal{Q}) \leq \delta\}$, and $\text{MA}(F) = \text{MA}_{1/3}(F)$. The constant $1/3$ is chosen by convention, and can be replaced by any constant in $(0, 1/2)$ without affecting the theory.

Online MA Communication. An online MA protocol is defined to be an MA protocol, as above, but with the communication pattern required to obey the following sequence. (1) Input x is revealed to Alice and Merlin; (2) Merlin sends Alice a help message $\mathfrak{h}_1(x, r_M)$ using a private random string r_M; (3) Input y is revealed to Bob and Merlin; (4) Merlin sends Bob a help message $\mathfrak{h}_2(x, y, r_M)$; (5) Alice sends a message to Bob (this message can depend on a public random string r_A that is not known to Merlin at the time he sends $\mathfrak{h}_1(x, r_M)$ or $\mathfrak{h}_2(x, y, r_M)$), who then gives a 1-bit output. We see this model as the natural MA variant of one-way communication, and the analogy with the gradual revelation of a streamed input should be obvious.

For such a protocol $\mathcal{Q}$, we define $\text{hcost}(\mathcal{Q})$ to be $1 + \max_{x,y,r_M}(|\mathfrak{h}_1(x, r_M)| + |\mathfrak{h}_2(x, y, r_M)|)$ We define soundness, completeness, $\text{err}(\mathcal{Q})$, and $\text{vcost}(\mathcal{Q})$ as for MA. Define $\text{MA}^{\rightarrow}_{\delta}(F) =$

$\min\{\text{hcost}(\mathcal{Q}) + \text{vcost}(\mathcal{Q}) : \mathcal{Q}$ is an online MA protocol for F with $\text{err}(\mathcal{Q}) \leq \delta\}$ and write $\text{MA}^{\rightarrow}(F) = \text{MA}^{\rightarrow}_{1/3}(F)$.

Online AMA Communication. An online AMA protocol is a souped-up version of an online MA protocol, where public random coins can be tossed at the start (and revealed to all parties), before any input is revealed. The number of such coin tosses is added to the vcost of the protocol. This models the cost of an initial round of communication between Arthur (i.e., Alice + Bob) and Merlin. Note that the *second* public random string, used when Alice talks to Bob, does not count towards the vcost.

Multiparty MA Communication. We also will have reason to consider a generalization of the MA communication model to settings in which there are more than two non-Merlin players. Let $f : X_1 \times \cdots \times X_t \rightarrow \{0, 1\}$ be a function, where each X_i is a finite set. This naturally gives a t-player number-in-hand communication problem, where Player i holds an input $x_i \in X_i$ and the players wish to output $f(x_1, \ldots, x_t)$ correctly, with high probability.

Merlin knows the entire input $\mathbf{x} = (x_1, \ldots, x_t)$. Merlin begins by sending a help message $\mathfrak{h}(\mathbf{x}, r_M)$, using a private random string r_M, that is seen by all t other players. Then Players 1 through t run a randomized protocol $\mathcal{Q}$, using a public random string R, eventually outputting a bit $\text{out}^{\mathcal{Q}}(\mathbf{x}, R, \mathfrak{h}(\mathbf{x}, r_M))$. To clarify, R is not known to Merlin at the time he writes $\mathfrak{h}(\mathbf{x}, r_M)$. We define soundness, completeness, $\text{err}(\mathcal{Q})$, $\text{hcost}(\mathcal{Q})$, and $\text{vcost}(\mathcal{Q})$ as for MA.

On Merlin's Use of Randomness. In an MA protocol, Merlin can deterministically choose a help message that maximizes Arthur's acceptance probability. However, Merlin cannot do so in the online MA model, because he does not know the entire input when he talks to Alice. This is why we allow Merlin to use randomness in these definitions.

Several recent papers, including one on which this chapter is based [30, 72], use "online

MA" to mean a more restrictive model where a deterministic Merlin talks only to Bob and not to Alice. With Merlin required to be deterministic, this communication restriction is irrelevant, as Merlin cannot tell Alice anything she does not already know. However, we permit Merlin to be probabilistic, and in this case we do not know that Merlin can avoid talking to Alice. That is, we do not know how to show that for every protocol in which a randomized Merlin talks to Alice, there is a protocol of similar cost in which Merlin does not talk to Alice. Our online MA communication model, which permits Merlin to both be randomized and talk to Alice, may therefore be strictly more powerful than the variant considered in [30, 72] in which Merlin cannot talk to Alice at all.

As noted earlier, our goal in defining the communication models this way is to closely correspond to annotated data stream models. In several of our online schemes in Chapter 4 (see, e.g., Section 4.5), the helper provides initial annotation that specifies a random "hash" function, h, and the completeness guarantee of the subsequent protocol depends crucially on h having "low collision" properties. Since h must be chosen without seeing all of the input, such low collision properties cannot be guaranteed by picking a fixed h in advance. However, if the helper chooses h at random, then we do have such guarantees for each fixed input, with high probability.

3.2.2 Data Stream Models

We now define our annotated data stream models. Recall that a (traditional) data stream algorithm computes a function F of an input sequence $\mathbf{x} \in \mathcal{U}^N$, where N is the number of stream updates, and $\mathcal{U}$ is some data universe, such as $\{0,1\}^b$ or $[n] = \{0, \ldots, n-1\}$: the algorithm uses a limited amount of working memory and has access to a random string. The function F may or may not be Boolean.

An annotated data stream algorithm, or a *scheme*, is a pair $\mathcal{Q} = (\mathfrak{h}, \mathcal{V})$, consisting of a

help function $\mathfrak{h} : \mathcal{U}^N \times \{0,1\}^* \to \{0,1\}^*$ used by a *prover* (henceforth, $\mathcal{P}$) and a data stream algorithm run by a *verifier*, $\mathcal{V}$. Prover $\mathcal{P}$ provides $\mathfrak{h}(\mathbf{x}, r_\mathcal{P})$ as annotation to be read by $\mathcal{V}$. We think of $\mathfrak{h}$ as being decomposed into $(\mathfrak{h}_1, \ldots, \mathfrak{h}_N)$, where the function $\mathfrak{h}_i : \mathcal{U}^N \to \{0,1\}^*$ specifies the annotation supplied to $\mathcal{V}$ after the arrival of the ith token x_i. That is, $\mathfrak{h}$ acts on $\mathbf{x}$ (using $r_\mathcal{P}$) to create an *annotated stream* $\mathbf{x}^{\mathfrak{h},r_P}$ defined as follows:

$$\mathbf{x}^{\mathfrak{h},r_P} := (x_1,\, \mathfrak{h}_1(\mathbf{x}, r_\mathcal{P}),\, x_2,\, \mathfrak{h}_2(\mathbf{x}, r_\mathcal{P}),\, \ldots,\, x_N,\, \mathfrak{h}_N(\mathbf{x}, r_\mathcal{P}))\,.$$

Note that this is a stream over $\mathcal{U} \cup \{0,1\}$, of length $N + \sum_i |\mathfrak{h}_i(\mathbf{x}, r_\mathcal{P})|$. The streaming verifier $\mathcal{V}$, who uses w bits of working memory and has access to a (private) random string $r_\mathcal{V}$, then processes this annotated stream, eventually giving an output $\text{out}^\mathcal{V}(\mathbf{x}^{\mathfrak{h},r_P}, r_\mathcal{V})$.

Prescient Schemes. The scheme $\mathcal{Q} = (\mathfrak{h}, \mathcal{V})$ is said to be δ_s-sound and δ_c-complete for the function F if the following conditions hold:

1. For all $\mathbf{x} \in \mathcal{U}^N$, we have $\Pr_{r_\mathcal{P}, r_\mathcal{V}}[\text{out}^\mathcal{V}(\mathbf{x}^{\mathfrak{h},r_P}, r_\mathcal{V}) \neq F(\mathbf{x})] \leq \delta_c$.

2. For all $\mathbf{x} \in \mathcal{U}^N$, $\mathfrak{h}' = (\mathfrak{h}'_1, \mathfrak{h}'_2, \ldots, \mathfrak{h}'_N) \in (\{0,1\}^*)^N$, we have $\Pr_{r_\mathcal{V}}[\text{out}^\mathcal{V}(\mathbf{x}^{\mathfrak{h}'}, r_\mathcal{V}) \notin \{F(\mathbf{x}), \perp\}] \leq \delta_s$.

If $\delta_c = 0$, the scheme satisfies *perfect completeness*; otherwise it has *imperfect completeness*. An output of "$\perp$" indicates that $\mathcal{V}$ rejects $\mathcal{P}$'s claims in trying to convince $\mathcal{V}$ to output a particular value for $F(\mathbf{x})$.

We note two important things. First, the definition of a scheme allows the annotation $\mathfrak{h}_i(\mathbf{x}, r_\mathcal{P})$ to depend on the entire stream $\mathbf{x}$, thus modeling *prescience*: the advice from the prover can depend on data that the verifier has not seen yet. Second, $\mathcal{P}$ must convince $\mathcal{V}$ of the value of $F(\mathbf{x})$ *for all* $\mathbf{x}$. This is stricter than the traditional definitions of interactive proofs and MA communication complexity (including our own, above) for decision problems, which place different requirements on the cases $F(\mathbf{x}) = 0$ and $F(\mathbf{x}) = 1$. In Chapter 4 (Section

4.6), we briefly consider a relaxed definition of schemes that is in the spirit of the traditional definition.

We define $\mathrm{err}(\mathcal{Q})$ to be the minimum value of $\max\{\delta_s, \delta_c\}$ such that the above conditions are satisfied. We define the *annotation length* $\mathrm{hcost}(\mathcal{Q}) = \max_{\mathbf{x}, r_{\mathcal{P}}} \sum_i |\mathfrak{h}_i(\mathbf{x}, r_{\mathcal{P}})|$, the total size of $\mathcal{P}$'s communications, and the *verification space cost* $\mathrm{vcost}(\mathcal{Q}) = w$, the space used by the verifier $\mathcal{V}$. We say that $\mathcal{Q}$ is a prescient (c_a, c_v)-scheme if $\mathrm{hcost}(\mathcal{Q}) = O(c_a)$, $\mathrm{vcost}(\mathcal{Q}) = O(c_v)$ and $\mathrm{err}(\mathcal{Q}) \leq \frac{1}{3}$.

Online Schemes. We call $\mathcal{Q} = (\mathfrak{h}, \mathcal{V})$ a δ-error online scheme for F if, in addition to the conditions in the previous definition, each function $\mathfrak{h}_i$ depends only on $(x_1, \ldots, x_i)$. We define error, hcost, and vcost as above and say that $\mathcal{Q}$ is an *online* (c_a, c_v)*-scheme* if $\mathrm{hcost}(\mathcal{Q}) = O(c_a)$, $\mathrm{vcost}(\mathcal{Q}) = O(c_v)$, and $\mathrm{err}(\mathcal{Q}) \leq \frac{1}{3}$.

Note that in this thesis we do not always assume that the universe size n and stream length N are polynomially related; it is possible that $N = n^{o(1)}$. This becomes relevant particularly in Chapter 4. Therefore we must be much more careful about logarithmic factors than in some of the publications on which this chapter is based [30, 40].

Notice that the help function can be made deterministic in a prescient scheme, but not necessarily so in an online scheme. This is directly analogous to the situation for MA and online MA communication models, as discussed at the end of Section 3.2.1.

AMA Schemes. We also consider what we call AMA schemes, where there is a common source of public randomness, in addition to the verifier's private random coins. The AMA scheme model is identical to the one considered by Gur and Raz [61], who referred to it as the "Arthur–Merlin streaming model."

An online AMA scheme is identical to a (standard) online scheme, except that the data stream algorithm and help function both have access to a source of public random bits. The

number of random bits used is also counted in both the hcost and the vcost of the scheme.

On Practicality and the Plausibility of Prescience. Although our definition of a scheme allows annotation to be sent after each stream update, with just one exception (the prescient Subset protocol of Lemma 3.8.1), all the schemes we design require annotation only before the start or after the end of the stream. As a practical matter, this avoids the need for fine-grained coordination between the annotation and the data stream, and permits the annotation to be sent to the verifier as an email attachment, or posted on a website for the verifier to retrieve at her convenience.

Online annotation schemes have the appealing property that the prover need not "see into the future" to execute them; at any time t, the prover's message only depends on stream updates that arrived before time t. While the online restriction appears most natural, prescient schemes may still be suitable in some settings, such as when $\mathcal{P}$ has already seen the full input prior to $\mathcal{V}$ beginning to read it. Consider a volunteer computing scenario where the verifier farms out many computations to volunteers, and only inspects a particular input if a volunteer has already looked at that input and claims to have found something interesting. In brief, in some settings the prover may naturally see the input before the verifier, and in this case a prescient scheme will be feasible.

3.2.3 Relationship Between MA Protocols and Schemes

Any prescient (resp. online) (c_a, c_v)-scheme $\mathcal{Q} = (\mathfrak{h}, \mathcal{V})$ for a function F can be converted into an MA (resp. online MA) protocol for F in the natural way: Merlin sends the output of the ith help function $\mathfrak{h}_i$ to Alice—who receives a prefix of the input stream—or Bob, depending on which of the players possesses the ith piece of the input. Alice runs the streaming algorithm $\mathcal{V}$ on her input as well as any annotation she received, and sends the state of the

algorithm to Bob. Bob uses this state to continue running $\mathcal{V}$ on his input and the annotation he received, and then outputs the end result. The hcost of this protocol is at most $c_a \log N$, since Merlin has to specify which stream update i each piece of annotation is associated with, and the vcost of this protocol is at most c_v. Thus, lower bounds on usual (resp. online) MA communication protocols imply related lower bounds on the costs of prescient (resp. online) annotated data stream algorithms.

3.2.4 Additional Notation and Terminology

A data stream specifies an input $\mathbf{x}$ incrementally. Typically, $\mathbf{x}$ can be thought of as a vector (although more generally it may represent a graph or a matrix). Each update in the stream is of the form (i, δ) where $i \in \mathcal{U}$ identifies an element of the data universe, and $\delta \in \mathbb{Z}$ describes the change to the frequency of i. The frequency of universe item i is defined as $f_i(\mathbf{x}) := \sum_{(j_k,\delta_k)\in\mathbf{x}:j_k=i} \delta_k$. We refer to the vector $f(\mathbf{x}) = (f_1(\mathbf{x}), \ldots, f_n(\mathbf{x}))$ as the *frequency vector* of $\mathbf{x}$, where n denotes the size of the data universe. When the stream $\mathbf{x}$ is apparent, we will often omit $\mathbf{x}$ and refer to the frequency vector as $f = (f_1, \ldots, f_n)$. We use $\mathbb{Z}_+$ to denote the set of non-negative integers.

We consider several different update models. In the most general update model, the *non-strict turnstile model*, the δ values may be negative, and so f_i may also be negative. In the *strict turnstile model*, the δ values may be negative, but it is assumed that the frequencies f_i always remain non-negative. In the *insert-only model*, the δ values must be non-negative. Orthogonal to these, in the *unit-update* version of each model, the δ values are assumed to have absolute value 1. Each of our results applies to a subset of these models, and we specify within the statement of each theorem which update models it applies to.

Throughout, n will denote the size of the data universe, N will denote the total number of stream updates, m will denote the total number of items with non-zero frequency at the

end of the stream, and M will refer to the total number of distinct items that ever appear within some stream update. We will refer to N as the *length* of the stream, to m as *sparsity* of the stream, and to M as the *footprint* of the stream. Notice that it is always the case that $m \leq M \leq N$. In the case of insert-only streams, $m = M$, but for streams in the (strict or general) turnstile models it is possible for m to be much smaller than M. Note also that while whenever we talk about "sparse" streams in this thesis, we refer to the relative size of n and m, not the absolute size. Indeed, we assume that m is typically large, too large for $\mathcal{V}$ to store the stream explicitly (else the problems can become trivial).

Throughout this thesis, when analyzing the runtime of the prover or verifier in any protocol, an addition or multiplication within a finite field is assumed to require a single time step.

3.2.5 A Preliminary Lemma: Fingerprints

We often make use of *fingerprint* functions of streams, which enable a streaming verifier to test whether two large streams have the same frequency vector. The verifier chooses a fingerprint function $g(\mathbf{x})$ at random from some family of functions satisfying the property that (over the random selection of the function g),

$$\Pr[g(\mathbf{x}) = g(\mathbf{y}) \mid f(\mathbf{x}) \neq f(\mathbf{y})] < 1/p$$

for a parameter p. Typically, $g(\mathbf{x})$ is an element of a finite field of size poly(p), and hence the number of bits required to store the value $g(\mathbf{x})$ (as well as g itself) is $O(\log p)$. Further, there are known constructions of fingerprint functions where $g(\mathbf{x})$ can be computed in space $O(\log p)$ by a streaming algorithm in the non-strict turnstile update model, as formalized in the following lemma.

Lemma 3.2.1 (Fingerprints)**.** *Given a prime q, let $\mathbb{F}_q$ denote the finite field with q elements. Let $\mathcal{G} = \{g_\alpha : \alpha \in \mathbb{F}_q\}$ denote the family of functions defined as follows:*

$$g_\alpha(f(\mathbf{x})) = \sum_{i=1}^{n} f_i(\mathbf{x})\alpha^i.$$

Then given data streams $\mathbf{x}$ and $\mathbf{y}$ defined over a universe of size n,

$$\Pr_{\alpha \leftarrow \mathbb{F}_q}[g_\alpha(\mathbf{x}) = g_\alpha(\mathbf{y}) \mid f(\mathbf{x}) \neq f(\mathbf{y})] \leq n/q,$$

and for any fixed α, $g_\alpha(f(\mathbf{x}))$ can be computed from the data stream $\mathbf{x}$ by a streaming algorithm in $O(\log q)$ space.

Proof. To prove the inequality, suppose $\mathbf{x}$ and $\mathbf{y}$ are two data streams defined over a universe of size n such that $f(\mathbf{x}) \neq f(\mathbf{y})$. Let $p_{\mathbf{x}}(\alpha) = \sum_{j=1}^{n} f_i(\mathbf{x})\alpha^i$ and $p_{\mathbf{y}}(\alpha) = \sum_{j=1}^{n} f_i(\mathbf{y})\alpha^i$. Since $f(\mathbf{x}) \neq f(\mathbf{y})$, $p_{\mathbf{x}}$ and $p_{\mathbf{y}}$ are distinct polynomials in α of degree n. Because two distinct polynomials of degree n can agree on at most n points, it holds that $\Pr_{\alpha \leftarrow \mathbb{F}_q}[g(\mathbf{x}) = g(\mathbf{y}) \mid f(\mathbf{x}) \neq f(\mathbf{y})] \leq n/q$.

To compute the fingerprint of $f(\mathbf{x})$, a streaming algorithm can store a single element $z \in \mathbb{F}_q$, and process each update (i, δ) via $z \leftarrow z + \delta\alpha^i$. This requires $O(\log q)$ space, and it is easily seen that z equals $g_\alpha(f(\mathbf{x}))$ at the end of the stream. □

Lemma 3.2.1 implies that if the field size q is sufficiently large, say, polynomial in n and m, then $\Pr_{\alpha \leftarrow \mathbb{F}_q}[g(\mathbf{x}) = g(\mathbf{y}) \mid f(\mathbf{x}) \neq f(\mathbf{y})]$ can be made polynomially small while keeping the space usage of the verifier logarithmic in n.

3.3 Index and Selection

In this section, we present an online scheme for the SELECTION problem. Our definition of the SELECTION problem assumes all frequencies $f_i := \sum_{(j_k,\delta_k):j_k=i} \delta_k$ are non-negative, and so this definition is only valid for the strict turnstile update model.

Definition 3.3.1. The SELECTION problem is defined in terms of the quantity $N' = \sum_{i\in[n]} f_i$, the sum of all the frequencies. Given a desired rank $\rho \in [N']$, output an item j from the stream $\mathbf{x} = \langle (j_1, \delta_1), \ldots, (j_N, \delta_N)\rangle$, such that $\sum_{(j_k,\delta_k):j_k<j} \delta_k < \rho$ and $\sum_{(j_k,\delta_k):j_k>j} \delta_k \geq N' - \rho$.

An easy *prescient* $(\log n, \log n)$-scheme is for the helper to give a claimed answer s as annotation at the start of the stream. The verifier need only count how many items in the stream are (a) smaller than s and (b) greater than s. The verifier then outputs s if the rank of s satisfies the necessary conditions, and outputs $\perp$ otherwise.

However, our goal is to present (almost) matching upper and lower bounds when only *online* annotation is allowed. To do this, we first consider the online MA complexity of the communication problem of INDEX: Alice holds a string $x \in \{0,1\}^N$, Bob holds an integer $i \in [N]$, and the goal is for Bob to output $\text{INDEX}(x, i) := x_i$. The lower bound for SELECTION will follow from the lower bound for INDEX and a key idea for the SELECTION upper bound is taken from the communication protocol for INDEX seen in the proof of the following theorem.

Theorem 3.3.2 (Online MA complexity of INDEX)**.** *Let $c_a > 1$ and c_v be integers such that $c_a \cdot c_v \geq N$. There is an online MA protocol $\mathcal{Q}$ for* INDEX*, with* $\text{hcost}(\mathcal{Q}) \leq c_a$ *and* $\text{vcost}(\mathcal{Q}) = O(c_v \log c_a)$*. Futhermore, any online MA protocol $\mathcal{Q}$ for* INDEX *must have* $\text{hcost}(\mathcal{Q})\,\text{vcost}(\mathcal{Q}) = \Omega(N)$*. Thus, in particular,* $\text{MA}^{\rightarrow}(\text{INDEX}) = \tilde{\Theta}(\sqrt{N})$.

Proof. For the lower bound, we use an online MA protocol $\mathcal{Q}$ to build a (Merlin-less) randomized one-way INDEX protocol $\mathcal{Q}'$. Here, a one-way protocol is a one in which Alice sends a message to Bob, with no communication from Bob to Alice.

We first consider the case where Merlin does not send any message to Alice at all and then explain how to modify the proof to cover the case where Merlin sends a message to Alice (possibly based on Merlin's internal randomness r_M) that does not depend on Bob's

input. Let $c_a = \text{hcost}(\mathcal{Q})$. Let $\mathcal{B}(n,p)$ denote the binomial distribution with parameters n and p, and let k be the smallest integer such that $X \sim \mathcal{B}(k, \frac{1}{3}) \Rightarrow \Pr[X > k/2] \leq 2^{-c_a}/3$. A standard Chernoff bound gives $k = \Theta(h)$. Let $a(x, R_A)$ denote the message that Alice sends in $\mathcal{Q}$ when her random string is R_A (notice $a(x, R_A)$ does not depend on any help message $\mathfrak{h}_1(x, r_M)$ from Merlin, since we have assumed no such help message is sent), and let $b(\mathfrak{a}, \mathfrak{h}_2)$ be the bit Bob outputs in $\mathcal{Q}$ upon receiving message $\mathfrak{a}$ from Alice and $\mathfrak{h}_2$ from Merlin. In the protocol $\mathcal{Q}'$, Alice chooses k independent random strings $R_1, \ldots, R_k$ and sends Bob $a(x, R_1), \ldots, a(x, R_k)$. Bob then outputs 1 iff there exists a c_a-bit string $\mathfrak{h}$ such that $\textsc{majority}\,(b(a(x, R_1), \mathfrak{h}_2), \ldots, b(a(x, R_k), \mathfrak{h}_2)) = 1$. Let C be the number of bits communicated in this protocol. Clearly, $C \leq k \cdot \text{vcost}(\mathcal{Q}) = O(\text{hcost}(\mathcal{Q})\,\text{vcost}(\mathcal{Q}))$. We claim that $\mathcal{Q}'$ is a $\frac{1}{3}$-error protocol for INDEX whence, by a standard lower bound (see, e.g., Ablayev [4]), $C = \Omega(N)$.

To prove the claim, consider the case when $x_i = 1$. By the correctness of $\mathcal{Q}$ there exists a suitable help message $\mathfrak{h}_2$ from Merlin that causes $\Pr[b(a(x, R_A), i, \mathfrak{h}_2) = 0] \leq \frac{1}{3}$. Thus, by construction and our choice of k, the probability that Bob outputs 0 in $\mathcal{Q}'$ is at most $2^{-c_a}/3$. Now suppose $x_i = 0$. Then, *every* possible message $\mathfrak{h}_2$ from Merlin satisfies $\Pr[b(a(x, R_A), i, \mathfrak{h}_2) = 1] \leq \frac{1}{3}$. Arguing as before, and using a union bound over all 2^h possible messages $\mathfrak{h}$, we see that Bob outputs 1 with probability at most $2^{c_a} \cdot 2^{-c_a}/3 = \frac{1}{3}$.

Now consider the case in which Merlin sends a message to Alice (possibly based on Merlin's internal randomness r_M) that does not depend on Bob's input. Assume that the soundness probability of the protocol is 1/13-complete (this can be achieved by repeating the whole protocol $O(1)$ times and taking the majority vote, which increases the costs by only constant factors). In this case, we construct a one-way randomized (Merlin-less) communication protocol for INDEX as follows. Alice chooses a random string r_M herself. Since Merlin's message to Alice, $\mathfrak{h}_1(x, r_M)$, does not depend on Bob's input y, Alice can compute

$\mathfrak{h}_1(x, r_M)$ herself. Alice sends to Bob the messages $a(x, R_1, \mathfrak{h}_1(x, r_M)), \ldots, a(x, R_k, \mathfrak{h}_1(x, r_M))$ that she would have sent in the online MA protocol given Merlin's message $\mathfrak{h}_1(x, r_M)$, and Bob outputs 1 if and only if there exists a c_a-bit string $\mathfrak{h}$ that would have caused him to accept on a majority of Alice's messages.

Consider the case when $x_i = 1$. By the correctness of $\mathcal{Q}$, with probability at least 3/4 over the choice of r_M, there exists a suitable help message $\mathfrak{h}_2$ from Merlin that causes $\Pr[b(a(x, R_A, \mathfrak{h}_1(x, r_M)), i, \mathfrak{h}_2) = 0] \leq \frac{1}{3}$ (otherwise, with probability at least $1/4 \cdot 1/3 = 1/12$ over the choice of both r_M and R_A, Merlin will fail to convince Bob to output 1, contradicting the fact that the protocol is 1/13-complete.) Call such a choice of r_M "good". By construction and our choice of k, if r_M is good then the probability that Bob outputs 0 in $\mathcal{Q}'$ is at most $2^{-c_a}/3$. Thus, in the case $x_i = 1$, our one-way randomized communication protocol outputs 1 with probability at least $3/4 - 2^{-c_a}/3 > 2/3$.

In the case $x_i = 0$, the argument that our one-way randomized communication protocol outputs 0 with probability at least 2/3 proceeds exactly as in the case where Merlin did not send any message to Alice, since it holds that for *every* message $\mathfrak{h}_1$ to Alice and *every* possible message $\mathfrak{h}_2$ to Bob, the protocol satisfies $\Pr[b(a(x, R_A, \mathfrak{h}_1), i, \mathfrak{h}_2) = 1] \leq \frac{1}{3}$.

The upper bound follows as a special case of the two-party set-disjointness protocol in [3, Theorem. 7.4] since the protocol there is actually online. We give a more direct protocol, which establishes intuition for our SELECTION result. Write Alice's input string x as $x = y^{(1)} \cdots y^{(v)}$, where each $y^{(j)}$ is a string of at most c_a bits, and fix a prime q with $3c_a < q < 6c_a$. Let $y^{(k)}$ be the substring that contains the desired bit x_i. Merlin sends Bob a string z of length at most c_a, claiming that it equals $y^{(k)}$. Alice picks a random $\alpha \in \mathbb{F}_q$ and sends Bob α and the strings $g_\alpha(y^{(1)}), \ldots, g_\alpha(y^{(v)})$, where g_α is defined as in Lemma 3.2.1. This requires communicating $O(v \log q) = O(v \log c_a)$ bits. Bob checks if $g_\alpha(z) = g_\alpha(y^{(k)})$, outputting $\perp$ if not. If the check passes, Bob assumes that $z = y^{(k)}$, and outputs x_i from z under this

assumption. By Lemma 3.2.1, the error probability is at most $c_a/q < 1/3$. □

It is worth making the following two remarks on the above proof.

1. The above lower bound argument in fact shows that an online MA protocol $\mathcal{Q}$ for an *arbitrary* two-party communication problem F satisfies $\text{hcost}(\mathcal{Q})\,\text{vcost}(\mathcal{Q}) = \Omega(\text{R}^{\rightarrow}(F))$, where $\text{R}^{\rightarrow}(F)$ is the one-way, randomized communication complexity of F. Thus, $\text{MA}^{\rightarrow}(F) = \Omega(\sqrt{\text{R}^{\rightarrow}(F)})$. A similar result was proved by Aaronson [2].

2. The upper bound for INDEX presented above works more or less unchanged when Alice's string is in Σ^N, for an arbitrary finite alphabet Σ. In view of Lemma 3.2.1, one simply needs to choose the prime q such that $3|\Sigma|h < q < 6|\Sigma|h$ to bound the error probability below 1/3. This leads to a protocol $\mathcal{P}$ with $\text{hcost}(\mathcal{P}) \leq h \log |\Sigma|$ and $\text{vcost}(\mathcal{P}) = O(v(\log |\Sigma| + \log h))$. Henceforth, we shall refer to this generalized protocol simply as "the INDEX protocol" — the alphabet Σ will usually be clear from the context.

Theorem 3.3.3. *For all c_a, c_v such that $c_a \cdot c_v \geq n$, there is an online $(c_a \log n, c_v \log n)$-scheme for* SELECTION. *Furthermore, any online (c_a, c_v)-scheme for* SELECTION *must have $c_a \cdot c_v = \Omega(n)$.*

Proof. Conceptually, the verifier builds a vector $\mathbf{r} = (r_1, \ldots, r_n) \in \mathbb{Z}_+^n$ where $r_k = \sum_{j<k} f_k$. This is done by inducing a new stream $\mathbf{x}'$ from the input stream $\mathbf{x}$: each tuple (x_k, δ_k) in A causes virtual tokens $(x_k{+}1, \delta_k), (x_k{+}2, \delta_k), \ldots, (n, \delta_k)$ to be inserted into A'. Then $\mathbf{r} = f(A')$; note that $\|\mathbf{r}\|_1 = O(nN)$. We apply the INDEX protocol to this vector, with $q = \Theta(m^2)$ to retrieve the ranks of elements surrounding the claimed answer s. This information is sufficient to check that s has the claimed rank.

For the lower bound, we use a standard reduction from the INDEX problem. Given the string $x \in \{0,1\}^N$, Alice transforms it into the stream over universe $[2N]$ whose jth tuple is

$(2j - x_j, 1)$, for each j. Given the index $i \in [N]$, Bob transforms it into a stream consisting of i copies of $(2N, 1)$ and $N - i$ copies of $(1, 1)$. Consequently, the median of the combined length-$(2N)$ stream is $2i - x_i$, from which the value of x_i can be recovered. To complete the proof, observe that any online scheme to compute this median would imply an online MA protocol for INDEX with the same cost; and that all players can perform this reduction online without extra space or annotation. □

Notice that in the above scheme the information computed by the verifier is independent of ρ, the rank of the desired element. Therefore these algorithms work even when ρ is revealed at the end of the stream.

3.4 A First Result for Frequent Items

The ϕ-heavy hitters (also known as the frequent items) are those items whose frequency of occurrence in the data stream exceeds a ϕ fraction of the total count $N' = \sum_{i \in [n]} f_i$. This definition assumes all frequencies $f_i := \sum_{(j_k, \delta_k): j_k = i} \delta_k$ are non-negative, and so this definition is only valid for the strict turnstile update model. This problem has a long history in the data streams literature. In the traditional data stream model exact computation of heavy hitters requires linear space [80]. As a result, many algorithms that recover approximate heavy hitters from a data stream have been developed [32, 41].

In order to identify the heavy hitters, a prescient helper can list the set of claimed frequent items, along with their frequencies, for the verifier to check against the stream. But we must also ensure that the helper is not able to omit any items whose frequencies exceed the threshold.

Theorem 3.4.1. *For all c_a, c_v such that $c_a \cdot c_v \geq n$, there is an online $(c_a \phi^{-1} \log^2 n, c_v \log n)$-scheme and a prescient $(\phi^{-1} \log^2 n, \phi^{-1} \log^2 n)$-scheme for demonstrating the ϕ-heavy hitters*

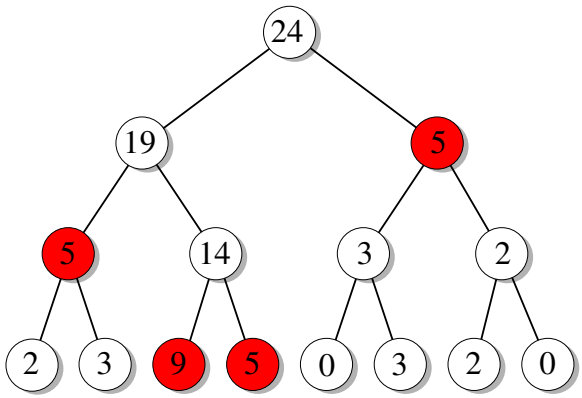


Figure 3.1: Illustration of the witness set introduced in the proof of Theorem 3.4.1.

in the strict turnstile update model.

Proof. Given the threshold $T = \phi N'$, the set of heavy hitters is $\{j : f_j > T\}$. We impose a binary tree $\mathcal{T}$ over the data, whose leaves are the elements of the universe $[n]$, and partition the $(2n-1)$ nodes of $\mathcal{T}$ into c_v *groups* $G_1, \ldots, G_v$, with each $|G_i| \leq 2c_a$. For each node w of $\mathcal{T}$, let $p(w)$ denote the parent of w, and let $L(w)$ denote the set of leaves of the subtree of $\mathcal{T}$ rooted at w. We define $\hat{f}(w) = \sum_{i \in L(w)} f_i$.

The $\hat{f}$-values for the nodes in each group G_i form a vector with entries in $\{0, 1, \ldots, N'\}$. As the verifier processes the stream it maintains an $O(\log n)$-bit basic fingerprint of each such vector; this is easy to do since each token arrival simply causes a linear update to each vector. Once the end of the stream is reached, the helper can then convince the verifier of any $\hat{f}(w)$ value using the INDEX protocol: he simply supplies the vector for the group G_i that contains w, using at most $2c_a \log(N'+1) = O(c_a \log n)$ bits of annotation. In particular, he can identify all the heavy hitters. But he must also convince the verifier that no heavy hitters have been omitted.

To this end, we consider a *witness set*, W, of nodes of $\mathcal{T}$ which together cover the universe. The set W, given threshold T, consists of all leaves ℓ with $\hat{f}(\ell) > T$, plus all nodes u such that $\hat{f}(u) \leq T$ but $\hat{f}(p(u)) > T$. Each node of the latter type is witness to the fact that no

leaves $j \in L(u)$ can have $f_j > T$. The sets $L(u)$ for such u together with $\{j : f_j > T\}$ cover all of $[n]$. Further, because of the lower bound on $\hat{f}(p(u))$, there can be at most $2\phi^{-1}$ such nodes u at any level of $\mathcal{T}$, as the sum of $\hat{f}(w)$ over all nodes w at the parent level is exactly m. Hence $|W| = O(\phi^{-1} \log n)$.

This concept is illustrated in Figure 3.1. The figure shows a frequency distribution of $[2, 3, 9, 4, 1, 3, 2, 0]$. Over these leaves, we impose a binary tree, and for each internal node w in the tree we show $\hat{f}(w)$. With a threshold of $\phi = 0.24$, we seek to find all leaves of weight 6 or above. There is only one such leaf, with weight 9. For the witness set, we also include the fourth leaf, since its parent exceeds the threshold. Other nodes at higher levels in the tree are also included in the witness set when their parent exceeds the threshold but they individually do not. Nodes in the witness set are indicated by a red fill.

The prover presents the verifier with each node u in W, in increasing order of $\min L(u)$, together with a convincing proof of the value of $\hat{f}(u)$. The verifier, besides checking the proofs using the stored fingerprints, checks that the sets $L(u)$ do cover all of $[n]$ (outputting $\perp$ if they do not) and outputs those u that are leaves of $\mathcal{T}$ with $\hat{f}(u) > T$. In total, hcost $= O(|W| \cdot c_a \log n) = O(c_a \phi^{-1} \log^2 n)$ and vcost $= O(c_v \log n)$. Note that the stated vcost does not explicitly account for the verifier storing the $O(\phi^{-1} \log n)$ claimed heavy hitters, as in some settings (e.g., Theorem 3.7.1, later in this chapter) this is not required.

In the prescient case, the helper provides W upfront, which requires $O(|W| \log n) = O(\phi^{-1} \log^2 n)$ bits of annotation. The verifier stores it and then computes all $\hat{f}$-values for nodes in W, checking that these satisfy the requirements on a witness set. In this case, the stated vcost does account for the verifier storing the $O(\phi^{-1} \log n)$ claimed heavy hitters. □

In Section 3.6, we return to this problem and present more involved protocols with a lower cost. Specifically, Theorem 3.6.1 shows how to exploit sum-check techniques to allow the frequencies of *all* items in W to be confirmed with essentially the same cost as a single

item. It also shows how the size of the witness set W can be reduced by a logarithmic factor in the prescient case.

3.5 Frequency Moments and Generalizations

In this section we continue the study of properties of the frequency distribution $\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), \ldots, f_n(\mathbf{x}))$ of a given stream $\mathbf{x}$. In particular, we study the computation of frequency moments, which has a long history in the data streams literature, like the frequent items problem discussed earlier.

Definition 3.5.1. The kth frequency moment of the stream $\mathbf{x}$ is defined as $F_k = F_k(f) := \sum_{j\in[n]} f_j^k = \|f(\mathbf{x})\|_k^k$. Slightly abusing notation, we also define $F_k(\mathbf{v}) := \|\mathbf{v}\|_k^k$ for a vector $\mathbf{v}$.

It is well known that in the traditional data stream model, exact computation of F_k ($k \neq 1$) requires $\Omega(n)$ space. Even constant factor approximation requires $\Omega(n^{1-2/k})$ space for $k \geq 2$ [31].

3.5.1 Schemes for Frequency Moments

We now show a family of schemes that exhibit an optimal tradeoff between verification space and annotation length for the exact computation of F_k as a special case. This scheme is a good example of a *sum-check scheme* as described in Section 3.1, and is based on the Aaronson–Wigderson MA protocol for DISJ [3].

Theorem 3.5.2. *Let $f^{(1)}, \ldots, f^{(\ell)}$ denote the frequency vectors of ℓ data streams, each over the universe $[n]$. Let g be an ℓ-variate polynomial of total degree d over the integers. Let $F = \sum_{i=1}^{n} g(f_i^{(1)}, \ldots, f_i^{(\ell)})$, and let o be an a priori upper bound on $|F|$. Then for positive integers c_a, c_v with $c_a c_v \geq n$, there is an online $(dc_a(\log n + \log o), \ell c_v(\log n + \log o))$-scheme for computing F in the non-strict turnstile update model.*

Proof. We work on $\mathbb{F}_q$, the finite field with q elements, for a suitably large prime q; the choice $q > 2d(n+o)^2$ suffices. $\mathcal{V}$ treats each n-dimensional vector $f^{(j)}$ as a $c_a \times c_v$ array with entries in $\mathbb{F}_q$, using any canonical injection from $[n]$ to $[c_a] \times [c_v]$, and interpreting integers as elements of $\mathbb{F}_q$ in the natural way. Through interpolation, this defines a unique bivariate polynomial $\tilde{f}^{(j)}(X, Y) \in \mathbb{F}_q[X, Y]$ of degree $c_a - 1$ in X and $c_v - 1$ in Y, such that for all $x \in [c_a]$, $y \in [c_v]$, $\tilde{f}^{(j)}(x, y) = f^{(j)}(x, y)$.

The polynomials $\tilde{f}^{(j)}$ can then be evaluated at locations outside $[c_a] \times [c_v]$, so in the scheme $\mathcal{V}$ picks a random position $r \in \mathbb{F}_q$, and evaluates $f^{(j)}(r, y)$ for all $j \in [\ell]$ and $y \in [c_v]$; $\mathcal{V}$ can do this using c_v words of memory per vector $f^{(j)}$ in a streaming manner as follows.

To maintain each $\tilde{f}^{(j)}(r, y)$ for $y \in [c_v]$, note that upon reading a new token $i \in [n]$ in stream j that maps to $(a, b) \in [c_a] \times [c_v]$, the necessary update is of the form $\tilde{f}^{(j)}(r, y) \leftarrow \tilde{f}^{(j)}(r, y) + \chi_{a,b}(r, y)$, where $\chi_{a,b}$ is the Lagrange polynomial

$$\chi_{a,b}(X, Y) \;:=\; \prod_{i \in [c_a] \setminus \{a\}} (X - i)(a - i)^{-1} \cdot \prod_{j \in [c_v] \setminus \{b\}} (Y - j)(b - j)^{-1}\,.$$

Since $\chi_{a,b}(r, y) = 0$ for any $y \in [c_v] \setminus \{b\}$, the verifier need only update the single value $\tilde{f}^{(j)}(r, b)$, by adding $\chi_{a,b}(r, b)$, upon reading this token. Using a table of $O(c_a)$ appropriate precomputed values, this update can be computed quickly. For $c_a \leq \sqrt{n}$, this takes a constant number of arithmetic operations per update without affecting the asymptotic space cost.

Let $\tilde{g}$ denote the total-degree-d polynomial over $\mathbb{F}_q$ that agrees with g at all inputs in $\mathbb{F}_q^\ell$. $\mathcal{P}$ then presents a polynomial $p(X)$ of degree at most $d(c_a - 1)$ that is claimed to be identical to $\sum_{y \in [c_v]} \tilde{g}(\tilde{f}^{(1)}(X, y), \ldots, \tilde{f}^{(\ell)}(X, y))$.

$\mathcal{V}$ checks that $p(r) = \sum_{y \in [c_v]} \tilde{g}\left(\tilde{f}^{(1)}(r, y), \ldots, \tilde{f}^{(\ell)}(r, y)\right)$. If this *sum check* passes, then $\mathcal{V}$ believes $\mathcal{P}$'s claim and accepts $\sum_{x \in [c_a]} p(x)$ as the correct answer. It is evident that this scheme satisfies perfect completeness. The proof of soundness follows from the Schwartz-

Zippel lemma: if $\mathcal{P}$'s claim is false, then

$$\Pr\left[p(r) = \sum_{y\in[c_v]} \tilde{g}\left(\tilde{f}^{(1)}(r,y),\ldots,\tilde{f}^{(\ell)}(r,y)\right)\right] \leq d(c_a-1)/q\,. \qquad \square$$

By setting $\ell = 1$ and $g(f_i) = f_i^k$, we obtain the following immediate corollary.

Theorem 3.5.3. *Suppose c_a and c_v are positive integers with $c_a \cdot c_v \geq n$. Then, for integers $k \geq 1$, there exists an online $(k^2 c_a \log m, kc_v \log m)$-scheme for computing F_k exactly in the non-strict turnstile update model.*

Numerous problems such as computing Hamming distances and inner products, and approximating F_2 and F_∞, can be solved using F_k as a primitive or using related techniques. We proceed to outline the relevant schemes and the results they provide.

Approximate F_2. We obtain an AMA scheme that can approximate F_2 up to a $(1+\varepsilon)$ factor from an integer linear sketch of size $O(1/\varepsilon^2)$ [5, 62, 101]. In particular, if $\mathrm{CS}_w(A)$ denotes a length-w Count-Sketch vector of the stream A built using 4-wise independent hash functions, then $F_2(\mathrm{CS}_w(A))$ estimates $F_2(A)$ with relative error $\varepsilon = w^{-1/2}$ with constant probability [101]. Thus, if the verifier and helper have access to a source of public randomness to define the hash functions used by the sketch, the above F_2 scheme yields an online $(\varepsilon^{-2\alpha}\log m, \varepsilon^{2\alpha-2}\log m)$-scheme for any $\alpha \in [0,1]$. This follows from the combination of sum-check techniques with the observation that the verifier can track linear updates to their sketch efficiently.

Approximate F_∞. Recall that $F_\infty = \max_{j\in[n]} f_j$ and note that $F_\infty^t \leq F_t \leq nF_\infty^t$. Hence, if $t = \log n/\log(1+\varepsilon)$, then $(F_t)^{1/t}$ is at most a factor $1+\varepsilon$ from F_∞. This yields an online $((\frac{1}{\varepsilon}\log n)^2 c_a \log m,\ (\frac{1}{\varepsilon}\log n)c_v \log m)$-scheme for approximating F_∞ for any c_a, c_v such that $c_a \cdot c_v \geq n$. We make use of this scheme in Section 3.7.1.

Inner Product and Hamming Distance. Consider a stream consisting of a string $\mathbf{x} \in \{0,1\}^N$ followed by a string $\mathbf{y} \in \{0,1\}^N$. Exact computation of F_2 implies online schemes for certain functions of $\mathbf{x}$ and $\mathbf{y}$. For example, the inner product $\mathbf{x} \cdot \mathbf{y}$ is $(F_2(\mathbf{x}+\mathbf{y}) - F_2(\mathbf{x}) - F_2(\mathbf{y}))/2$ and the Hamming distance between $\mathbf{x}$ and $\mathbf{y}$ is $|\{i : x_i = 1\}| + |\{i : y_i = 1\}| - 2\mathbf{x} \cdot \mathbf{y}$. Hence we get an online $(c_a \log N, c_v \log N)$-scheme for each of these functions, for every pair of positive integers c_a, c_v satisfying $c_a \cdot c_v \geq N$. Alternately, Theorem 3.5.2 can be used to more directly generate schemes for these problems with the same bounds, by treating $\mathbf{x}$ and $\mathbf{y}$ as two separate data streams, and setting $g(f_i(\mathbf{x}), f_i(\mathbf{y})) = f_i(\mathbf{x}) \cdot f_i(\mathbf{y})$.

3.5.2 Lower Bounds on Frequency Moments

We now present lower bounds on the tradeoffs possible for the exact and approximate computation of the nontrivial frequency moments F_k. The first part of the theorem below shows that the tradeoff given by Theorem 3.5.3 is nearly tight.

Theorem 3.5.4. *Suppose $k \geq 0$ and $k \neq 1$. Let $\mathcal{Q}$ be a (c_a, c_v)-scheme (online or prescient) for computing F_k even in the insert-only unit-update model.*

(1) If $\mathcal{Q}$ computes F_k exactly, then it requires $c_a \cdot c_v = \Omega(n)$.

(2) If $\mathcal{Q}$ approximates F_k up to a constant factor, then it requires $c_a \cdot c_v = \Omega(n^{1-5/k})$.

Proof. Both results follow from lower bounds on the MA complexity of $\textsc{disj}_{n,t} : \{0,1\}^{nt} \to \{0,1\}$, the t-party set disjointness problem, which is defined as follows. The input is a $t \times n$ Boolean matrix, with Player i holding the ith row, for $i \in [t]$. We call an input $\mathbf{x} = (x_{ij})_{i \in [t], j \in [n]}$ *valid* if every column of $\mathbf{x}$ has weight either 0 or 1 or t, and at most one column has weight t. The desired output is

$$\textsc{disj}_{n,t}(\mathbf{x}) \ := \ \neg \vee_{j=1}^{n} \wedge_{i=1}^{t} x_{ij} \, ,$$

i.e., 1 iff the subsets of $[n]$ represented by the rows of $\mathbf{x}$ are disjoint. Note that $\textsc{disj}_{n,t}$ is naturally related to frequency moments: for any valid input $\mathbf{x}$, $F_k(S) \geq t^k$ if $\textsc{disj}_{n,t}(\mathbf{x}) = 0$ and $F_k(S) \leq n$ if $\textsc{disj}_{n,t}(\mathbf{x}) = 1$ where S is the multiset $\{j : x_{ij} = 1\}$. Thus, reductions from $\textsc{disj}_{n,2}$ and $\textsc{disj}_{n,O(n^{1/k})}$ establish the first and second parts of the theorem, respectively, in a straightforward manner.

To complete the proof, we need a lower bound for $\textsc{disj}_{n,t}$ itself. This is given in the next theorem, which generalizes a result by Klauck [70] and also resolves a question of Feigenbaum et al. [50]. □

Theorem 3.5.5. *Let $\mathcal{Q}$ be an ε-error t-party MA protocol for $\textsc{disj}_{n,t}$, where $\varepsilon \leq 1/3$. Then* $\mathrm{hcost}(\mathcal{Q}) \cdot \mathrm{vcost}(\mathcal{Q}) = \Omega(n/t^4)$. *In particular,* $\mathrm{MA}(\textsc{disj}_{n,t}) = \Omega(\sqrt{n}/t^2)$.

Proof. A rectangle is defined as a subset of inputs of the form $\mathcal{X}_1 \times \cdots \times \mathcal{X}_t$, where each $\mathcal{X}_i \subseteq \{0,1\}^n$ is a subset of the set of all possible inputs for Player i. A basic fact about deterministic communication protocols is that the inverse image of any transcript of such a protocol must be a rectangle; this is usually called the *rectangle property*. Let $A = \textsc{disj}_{n,t}^{-1}(1)$ and $B = \textsc{disj}_{n,t}^{-1}(0)$. The following lemma was proved by Alon, Matias and Szegedy [6], generalizing a result due to Razborov [84].

Lemma 3.5.6 (Lemma 3.4 of [6])**.** *There exists a distribution μ over valid inputs such that*

(1) $\mu(A) = \mu(B) = 1/2$, and

(2) every rectangle T satisfies $\mu(T \cap B) \geq (2e)^{-1}\mu(T \cap A) - t2^{-n/2t^4}$. □

Returning to our theorem, assume $t = \omega(n^{1/4})$ since otherwise the bound is trivial. Put $c_a = \mathrm{hcost}(\mathcal{Q})$ and $c_v = \mathrm{vcost}(\mathcal{Q})$. An input $\mathbf{x} \in A$ is said to be *covered* by a message $\mathfrak{h}$ from Merlin if $\Pr_R[\mathrm{out}^{\mathcal{Q}}(\mathbf{x}, R, \mathfrak{h}) = 0] \leq \varepsilon$. By correctness, every such input must be covered, so there exists a help message $\mathfrak{h}^*$ that covers every input in a set $G \subseteq A$, with

$\mu(G) \geq 2^{-c_a}\mu(A) = 2^{-c_a-1}$. Fix Merlin's message in $\mathcal{Q}$ to $\mathfrak{h}^*$ and amplify the correctness of the resulting randomized Merlin-free protocol by repeating it $O(c_a)$ times and taking the majority of the outputs. This gives us a randomized protocol $\mathcal{Q}'$ for $\textsc{disj}_{n,t}$ with communication cost $c = O(c_a \cdot c_v)$ whose error, on every input in $G \cup B$, is at most 2^{-2c_a}.

Let μ' denote the distribution μ conditioned on $G \cup B$. Note that, by condition (1) of Lemma 3.5.6,

$$\forall\, \mathbf{x} \in \{0,1\}^{nt} : \quad \text{either } \mu'(\mathbf{x}) = 0 \text{ or } \mu(\mathbf{x}) \leq \mu'(\mathbf{x}) \leq 2\mu(\mathbf{x})\,. \tag{3.1}$$

By fixing the random coins of $\mathcal{Q}'$ we can obtain a deterministic protocol $\mathcal{Q}''$, for $\textsc{disj}_{n,t}$, that communicates c bits and satisfies $\mathrm{err}_{\mu'}(\mathcal{Q}'') \leq 2^{-2c_a}$. By the rectangle property, there exist disjoint rectangles $T_1, T_2, \ldots, T_{2^c}$ such that $\mathrm{out}^{\mathcal{Q}''}(\mathbf{x}) = 1$ if and only if $\mathbf{x} \in \bigcup_{i=1}^{2^c} T_i$. Therefore

$$\sum_{i=1}^{2^c} \mu'(T_i \cap B) \;\leq\; 2^{-2c_a}\,, \quad \text{and} \tag{3.2}$$

$$\mu'\left(A \setminus \bigcup_{i=1}^{2^c} T_i\right) \;\leq\; 2^{-2c_a}\,. \tag{3.3}$$

By (3.1), we have $\mu'(A) = \mu'(G) \geq \mu(G) \geq 2^{-c_a-1}$. Using (3.1), and a rearrangement of (3.3):

$$\sum_{i=1}^{2^c} \mu(T_i \cap A) \;\geq\; \frac{1}{2}\sum_{i=1}^{2^c} \mu'(T_i \cap A) \;\geq\; \frac{1}{2}\left(\mu'(A) - 2^{-2c_a}\right) \;\geq\; 2^{-c_a-3}\,.$$

Suppose $c \leq n/5t^4$ and n is large enough. Applying condition (2) of Lemma 3.5.6 to each term in the leftmost sum above, we get

$$\sum_{i=1}^{2^c} \mu(T_i \cap B) \;\geq\; \frac{2^{-c_a-3}}{2e} - 2^c t \cdot 2^{-n/2t^4} \;\geq\; 2^{-c_a-6}\,.$$

However, by (3.1) and (3.2), we have $\sum_{i=1}^{2^c} \mu(T_i \cap B) \leq 2^{-2c_a}$, a contradiction. Hence $c_a \cdot c_v = \Omega(c) = \Omega(n/t^4)$. $\square$

3.6 Frequent Items

In this section, we provide further results on finding exact frequent items. Our new results for frequent items improve over Theorem 3.4.1 in two ways: Firstly, we show that in the online case, the frequencies of all items in the witness set can be simultaneously checked with essentially the same cost as checking a single frequency, thereby saving some polynomial factors. Secondly, we show that in the prescient case it is possible to use a more compact witness set relative to Theorem 3.4.1, thereby saving logarithmic factors.

Theorem 3.6.1. *Let $T = \phi N'$, where $N' = \sum_{i\in[n]} f_i$ is the sum of the frequencies of a data stream over a universe of size n. For every pair of positive integers c_a, c_v satisfying $c_a \cdot c_v \geq n$, there is an online $(\phi^{-1}\log^2 n + c_a \log n,\ c_v \log n)$-scheme for finding $\{j : f_j > T\}$ in the strict turnstile update model, as well as a prescient $(\phi^{-1}\log n, \phi^{-1}\log n)$-scheme. Any online or prescient (c_a, c_v)-scheme for this problem, even in the unit-update insert-only model, must have $c_a \cdot c_v = \Omega(n)$.*

Proof. We begin with the online scheme. Let W be the witness set from Theorem 3.4.1. Recall that W is a subset of the nodes of a binary tree $\mathcal{T}$ imposed over the data universe, and in the scheme of Theorem 3.4.1, $\mathcal{P}$ sends to $\mathcal{V}$ a claimed value for $\hat{f}(w) = \sum_{i\in L(w)} f_i$, where $L(w)$ denotes the set of all leaves in the subtree rooted at W.

We show how $\mathcal{V}$ can check that $\hat{f}(w)$ is as claimed for all items $w \in W$. Let z denote the $2n-1$-dimensional vector such that $z_w = 1$ if $w \in W$, and $z_w = 0$ otherwise. Let f^* denote the $2n-1$-dimensional vector such that f^*_w equals the claimed value of $\hat{f}(w)$ if $w \in W$, and $f^*_w = 0$ otherwise. Abusing notation, we will also think of $\hat{f}$ itself as a $2n-1$-dimensional vector such that $\hat{f}_w = \hat{f}(w)$. Then $\hat{f}(w) = f^*_w$ for all $w \in W$ if and only if $0 = \sum_{j\in[2n-1]} z_j(\hat{f}_w - f^*_w)^2$. Theorem 3.5.2 gives an online scheme for computing the quantity $\sum_{j\in[2n-1]} z_j(\hat{f}_w - f^*_w)^2$: within the statement of Theorem 3.5.2, we let $\ell = 3$, $f^{(1)} = \hat{f}$, $f^{(2)} = f^*$, and $f^{(3)} = z$. It is

easy for $\mathcal{V}$ to run the scheme of Theorem 3.5.2 on derived streams defining these three vectors; for any c_a, c_v such that $c_a c_v \geq n$, Theorem 3.5.2 yields a scheme with hcost $= O(c_a \log n)$ and vcost $= O(c_v \log n)$.

Thus, the total hcost of our scheme is $\phi^{-1} \log^2 n + c_a \log n$, where the $\phi^{-1} \log^2 n$ term is the annotation required to specify the items in W and the claimed values for $\hat{f}_w : w \in W$, while the $c_a \log n$ term is the annotation required to check that the claimed $\hat{f}_w$ values are correct. The verifier's space usage is $c_v \log n$, yielding the claimed result.

For the prescient scheme, we specify a witness set that is more succinct than that of Theorem 3.4.1. Consider a binary tree $\mathcal{T}$ whose leaves are the elements of the universe $[n]$, as in Theorem 3.4.1. We will specify a witness set W of size $O(\phi^{-1})$ to identify to identify all leaves j with $f_j > T$; we base W on the concept of *Hierarchical Heavy Hitters* (HHHs) [37]. Below, we refer to the set of Hierarchical Heavy Hitters as H.

We define H inductively, beginning with the leaves and working our way to the root. We include a leaf in H if its frequency exceeds T. Let u be a node at distance l from the root (i.e., at level l of $\mathcal{T}$), and assume inductively that we have determined all HHHs at levels greater than l. Let $H(u)$ denote the set of descendants of u that have been included in H, and let $L(u)$ denote the set of leaves of the subtree rooted at u. Finally, define $S(u) := L(u) \setminus \big(\cup_{v \in H(u)} L(v) \big)$. Intuitively, $S(u)$ is the set of leaves in $L(u)$ that have not already contributed their frequency to an HHH descendant of u. Define the *conditioned count* of u as $g(u) := \sum_{j \in S(u)} f_j$; we include u in H if $g(u) > T$. Observe there are at most ϕ^{-1} items in H since $T = \phi N'$: each leaf contributes its frequency to $g(u)$ for exactly one $u \in H$, and therefore $|H|T \leq \sum_{u \in H} g(u) \leq N'$.

We now define our witness set W as all leaves j in H in addition to all nodes u such that u's parent is in H but u is not in H. Observe that each node $u \in W$ is witness to the fact that no leaves $j \in S(u)$ can have $f_j > T$. We also include the root r in W to account for any

leaves that are not descendants of any node in H. The sets $S(u)$ for $u \in W$ form a partition of $[n]$. Notice that $|W| = O(\phi^{-1})$ since $|H| \leq \phi^{-1}$.

In the prescient scheme, the helper lists all nodes $u \in W$ sorted by the natural order on nodes, and the verifier remembers this information. The verifier may then compute the conditioned count of each $u \in W$ using space $O(|W| \log n) = O(\phi^{-1} \log n)$: each time an item j appears in the stream, the verifier determines the unique $u \in W$ such that $j \in S(u)$ (u is simply the ancestor of j in W farthest from the root), and increments $g(u)$. The verifier checks that $g(j) > T$ for all leaf nodes $j \in W$, and that $g(u) \leq T$ for all internal nodes in W and outputs $\perp$ otherwise. Since the sets $S(u)$ partition $[n]$, this latter check ensures that the helper does not omit any leaves j with $f_j > T$.

We prove the lower bound by an easy reduction from two-party set-disjointness, $\text{DISJ}_{n,2}$. Consider Alice and Bob with respective inputs $x, y \in \{0,1\}^n$. Alice's input x induces a stream A by placing one copy of token j in the stream if $x_j = 1$. Then Bob places one copy of item j in the stream if $y_j = 1$. We may assume Bob knows $|\{j : x_j = 1\}|$, and hence knows the number of non-zero entries N' in the stream; if not Alice can tell Bob $|\{j : x_j = 1\}|$ at an additive cost of logarithmically many bits. Now x and y are disjoint if and only if the set $\{j : f_j > 1 = \phi N'\}$ for $\phi = 1/N'$ is non-empty. Thus, determining the frequent items for $T = 1$ solves two-party set disjointness, proving the bound by Theorem 3.5.5. □

3.7 Frequency-Based Functions

It is natural to ask whether the F_k scheme of Theorem 3.5.3 generalizes to more complicated functions. We demonstrate that this is indeed the case by presenting non-trivial algorithms for the class of all *frequency based functions*. A frequency based function is any function G on frequency vectors $f = (f_1, \ldots, f_n)$ of the form $G(\mathbf{f}) = \sum_{j \in [n]} g(f_j)$ for some

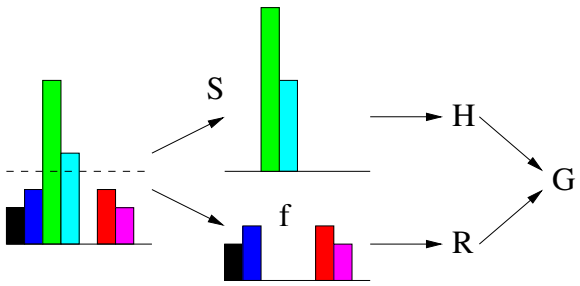


Figure 3.2: Example to illustrate Theorem 3.7.1

$g : \mathbb{Z}_+ \to \mathbb{Z}_+$.

Frequency-based functions have a number of important special cases, including frequency moments, F_0 (the number of distinct items in the stream), and point and range queries on the frequency distribution, and can also be used to compute F_∞, the highest frequency in the frequency vector. These functions occupy an important place in the streaming world: Alon, Matias, and Szegedy asked for a precise characterization of which frequency-based functions can be approximated efficiently in the standard streaming model in their seminal paper [6]. Braverman and Ostrovsky [22] gave a zero-one law for approximating monotonically increasing functions of frequencies that are zero at the origin. This can be contrasted with our result that, in the annotation model, *all* frequency-based functions have non-trivial exact schemes.

Theorem 3.7.1. *Assume $g(x) \leq n^c$ for some constant c, so that each value in the range of g and G can be represented using $O(\log n)$ bits. Suppose $N = O(n)$. Let $G(f) = \sum_{j \in [n]} g(f_j)$ be any frequency-based function. Then G has a prescient $(n^{2/3} \log n, n^{2/3} \log n)$-scheme and an online $(n^{2/3} \log^{4/3} n, n^{2/3} \log^{4/3} n)$-scheme, both in the non-strict turnstile update model.*

Proof. We first describe the prescient scheme. It is natural to attempt to directly apply the scheme of Theorem 3.5.2 (with $\ell = 1$) to the given function g. However, this does not yield a useful result. The problem with this approach is that while the function g within the definition of G may be viewed through polynomial interpolation as a polynomial $\tilde{g}$ over the

integers or the relevant finite field, the degree of $\tilde{g}$ may be large – as large as $2N$, since we need it to hold that $\tilde{g}(x) = g(x)$ for all possible frequencies $x \in \{-N, \ldots, N\}$. If $N = \Omega(n)$, it would be more efficient for the helper to just repeat the stream in sorted order.

The solution is to reduce the degree of $\tilde{g}$ by removing the *heavy hitters* from $\mathbf{x}$ with the aid of the prover. That is, we run the prescient heavy hitters scheme from Theorem 3.6.1 to determine $H := \sum_{j \in S} g(f_j) - |S|g(0)$, where $S := \{j : f_j \geq n^\beta\}$ and $\beta < 1$ is a parameter we will fix later. Note that this requires communication $O((N/n^\beta) \log n) = O(n^{1-\beta} \log n)$ since $N = O(n)$ by assumption. Intuitively, H represents the contribution of the heavy hitters to the frequency-based function, and the verifier then "removes" these items from the stream by setting $f_j = 0$ for all $j \in S$. This ensures that the removed items do not contribute to the sum $R = \sum_{j \in [n]} g(f_j)$. The verifier and prover then run the scheme of Theorem 3.5.2 on the *modified* frequency vector, and the final result is given by $H + R$. From now on, let $\mathbf{f}$ denote this modified vector.

Figure 3.2 gives an illustation of the central idea: the frequency distribution is conceptually split into two pieces, the set of heavy hitters S and the residual distribution $\mathbf{f}$. The contributions of each piece are calculated as H and R respectively, and summed to obtain the answer G.

When running the scheme of Theorem 3.5.2, we exploit the fact that each entry of $\mathbf{f}$ lies in $\{0, 1, \ldots, n^\beta\}$. This lets us use a degree-n^β polynomial $\tilde{g}$ within the scheme of Theorem 3.5.2. For any c_a, c_v such that $c_a \cdot c_v \geq n$, Theorem 3.5.2 yields an online $(n^\beta c_a \log n, c_v \log n)$ scheme for computing $\sum_{i \in [n]} \tilde{g}(\mathbf{f}_i)$.

It remains to show that we can set the parameters c_a, c_v, and β of the above protocol to achieve hcost = vcost = $O(n^{2/3} \log n)$. The help cost is $O(n^{1-\beta} \log n)$ bits for the heavy hitters scheme plus $O(c_a n^\beta \log n)$ bits for the scheme of Theorem 3.5.2. The respective verification costs are $O(n^{1-\beta} \log n)$ and $O(c_v \log n)$. Setting $\beta = \frac{1}{3}$, $c_a = n^{1/3}$, and $c_v = n^{2/3}$ achieves the

desired costs.

In order to achieve an online $(n^{2/3}\log^{4/3} n, n^{2/3}\log^{4/3} n)$-scheme for G, observe that the only place where the above scheme used prescience was to identify heavy hitters. So we simply substitute the online heavy hitters scheme of Theorem 3.6.1, with parameter $\alpha \in [0,1]$, in place of the prescient version. In this case, the help cost is $O(n^{1-\beta}\log^2 n + n^{\alpha}\log n)$ bits for the heavy hitters scheme and $O(c_a n^{\beta}\log n)$ bits for the scheme of Theorem 3.5.2. The respective verification costs are $O(n^{1-\alpha}\log n)$ and $O(c_v \log n)$. Balancing these costs by setting $n^{\beta} = n^{1/3}\log^{2/3} n$, $n^{\alpha} = n^{2/3}$, $c_a = n^{1/3}/\log^{1/3} n$, and $c_v = n^{2/3}\log^{1/3} n$ gives the desired overall costs. □

Applications. Theorem 3.7.1 provides annotation schemes for the problems described below.

- We can compute F_0, the number of items with non-zero count. This follows by observing that F_0 is equivalent to computing $\sum_{i\in[u]} g(f_i)$ for the function g given by $g(0) = 0$ and $g(x) = 1$ for $x > 0$. This yields a prescient $(n^{2/3}\log n, n^{2/3}\log n)$-scheme for F_0, and an online $(n^{2/3}\log^{4/3} n, n^{2/3}\log^{4/3} n)$-scheme.

- More generally, we can compute functions on the inverse distribution, i.e., queries of the form "How many items occur exactly k times in the stream?" We do this by setting $g(k) = 1$ and $g(x) = 0$ for $x \neq k$; here we think of k as being fixed. In the case of $k = 1$, this function is known as *rarity* [45]. One can build on this to compute, e.g., the number of items that occurred between k and k' times, the median of this distribution, etc.

- We obtain a protocol for $F_\infty = \max_{j\in[n]} f_j$, with a little more work. The helper first claims a lower bound ℓ on F_∞ by providing the index of an item with frequency

F_∞, which the verifier checks by running the generalized INDEX protocol from Section 3.3 (see Remark 2 after Theorem 3.3.2). Then the verifier runs the above protocol with $g(x) = 0$ for $x \leq \ell$ and $g(x) = 1$ for $i > \ell$; if $\sum_{j\in[n]} g(f_j) = 0$, then the verifier is convinced that no item has frequency higher than ℓ, and concludes that $F_\infty = \ell$. We therefore achieve a prescient $(n^{2/3}\log n, n^{2/3}\log n)$-scheme and an online $(n^{2/3}\log^{4/3} n, n^{2/3}\log^{4/3} n)$-scheme for F_∞.

3.7.1 Frequency-Based Functions for Skewed Streams

In practice, the frequency distributions of data streams are often skewed, in the sense that a small number of frequent items make up a large portion of the stream. We observe that, if the stream is sufficiently skewed, so that there are few heavy hitters, we can achieve more efficient schemes for frequency-based functions. To see this, notice that in the scheme of Theorem 3.7.1, the verifier, after learning the heavy hitters from the helper, only needs to know an *approximate* upper bound on $F_\infty(A')$, where A' is the stream obtained from the input stream A by deleting all the heavy hitters. That is, the helper only needs to convince the verifier that he has presented "enough" of the true heavy hitters (and their exact frequencies) so that $F_\infty(A') \leq b$ for some upper bound $b = \Theta(n^\beta)$—then we may define $\tilde{g}$ to agree with g on $[b]$, so that the degree of $\tilde{g}$ remains $O(n^\beta)$.

Observe that if there are not many heavy items, the helper can send a list L of heavy hitters and their frequencies (proving the frequencies are truthful as in Theorem 3.6.1) and then appending a proof of an approximate upper bound (within factor $1+\varepsilon$) as per Section 3.5.1 on the quantity $F_\infty(A')$.

It suffices to let ε be any positive constant in order to achieve $b = O(n^\beta)$. When there are fewer than ℓ items with frequency greater than n^β, the INDEX queries, if they are online, require annotation $O(\ell \log n + c_a \log n)$ and space $O(c_v \log n)$ for the verifier, while the

approximate F_∞ scheme requires annotation $O(c_a \log^3 n)$ and space $O(c_v \log^2 n)$. Therefore, we will obtain an $(\ell \log n + c_a \log^3 n, c_v \log^2 n)$ scheme for identifying the set of heavy hitters and an upper bound u on $F_\infty(A')$.

For concreteness, we will analyze the costs of our improved scheme under the assumption that the frequencies of items in the stream follow a Zipfian distribution, a power law distribution that accurately approximates many real-world data sets. Under the Zipfian distribution, the ith largest frequency is (at most) Ni^{-z} for parameter z. Setting this equal to n^β and rearranging, we obtain that there are at most $(N/n^\beta)^{1/z}$ heavy hitters to identify.

Therefore, if $N = \Theta(n)$, we can reduce the cost of the heavy hitters sub-protocol within the scheme of Theorem 3.7.1 to $(n^{(1-\beta)/z} \log n + c_a \text{polylog}\, n, c_v \text{polylog}\, n)$. Adding in the annotation cost of sending the polynomial $\tilde{g} \circ \tilde{f}$, and the space cost to the verifier, the entire scheme therefore requires $\tilde{O}(n^{(1-\beta)/z} + c_a n^\beta)$ annotation and $\tilde{O}(c_v)$ space, where the $\tilde{O}$ notation hides factors polylogarithmic in n. Assume $z \leq 2$. Balancing exponents by setting $\beta = (2-z)/(2+z)$, $c_a = n^{z/(2+z)}$, and $c_v = n/c_a$, we obtain an $(n^{2/(2+z)} \text{polylog}\, n, n^{2/(2+z)} \text{polylog}\, n)$ scheme.

This strictly improves on Theorem 3.7.1 as long as $z > 1$. For example, if $z = 2$, we obtain an online $(n^{1/2} \text{polylog}\, n, n^{1/2} \text{polylog}\, n)$-scheme, which essentially matches the cost of our online scheme for F_2 from Theorem 3.5.3.

3.8 Set and Multiset Inclusion

Building on some of the results and techniques in Section 3.5, we now address a family of abstract problems that involve a helper proving a subset (inclusion) relation to a streaming verifier. Both sets and multisets are of interest. For example, we may need to prove that $A \subseteq B$ for two sets A and B, or we may need to prove that a set A is exactly the support

set of a multiset B. These abstract problems turn out to be common subproblems arising in a number of applications that we shall consider later (see, e.g., Theorems 3.10.5, 3.10.6, and 3.10.7).

Throughout this section, the *size* of a multiset is the number of elements in it, counting multiplicities. A fingerprint of a multiset is a basic fingerprint, as in Definition 3.2.1, of its characteristic (frequency) vector.

Lemma 3.8.1. *Let $A \subseteq [n]$ be a set and $B \subseteq [n]$ a multiset of size t. Let B' be the set formed by removing all duplicate elements from B. Then, given a stream which begins with the elements of A followed by the elements of B, there is a prescient $(t \log n, \log n)$-scheme that establishes whether $B' = A$.*

Proof. As the elements of A are observed in the stream, the helper annotates each $a \in A$ with the multiplicity, f_a, of a in B. Once A has been observed, the helper then lists each element b in the set difference $B' \setminus A$, along with the corresponding multiplicity f_b in B. Obviously there are no such elements iff $B' = A$. From the provided information, the verifier constructs a fingerprint of the multiset in which each $a \in A \cup B'$ appears with multiplicity f_a.

Then, while observing the elements of the multiset B, the verifier incrementally constructs a fingerprint of B, as in Lemma 3.2.1. The verifier accepts iff the two fingerprints match. □

In the remainder of this section, we give three schemes achieving tradeoffs between hcost and vcost for (multi)-set inclusion, in order of generality. First, we give an essentially optimal *online* $(c_a \log n, c_v \log n)$-scheme, for any positive integers c_a and c_v with $c_a \cdot c_v \geq n$, for the special case when B is a set rather than a multiset.

Theorem 3.8.2. *Let $X, Y \subseteq [n]$ be sets. Then given a stream with elements of X and Y arbitrarily interleaved, there is an online $(c_a \log n, c_v \log n)$-scheme for determining whether*

$X \subseteq Y$ for any c_a and c_v such that $c_a \cdot c_v \geq n$. Moreover, any online (c_a, c_v)-scheme requires $c_a \cdot c_v = \Omega(n)$.

Proof. Let $x, y \in \{0,1\}^n$ be the characteristic vectors of X and Y respectively. Notice $F_2(y-x) = |X \Delta Y|$. Thus, $X \subseteq Y$ if and only if $F_2(y-x) = |Y| - |X|$. Consequently, the helper can run the F_2 scheme of Theorem 3.5.3 on the vector $y - x$ to determine if the above equality holds.

The lower bound follows from a straightforward reduction from INDEX. Take $N = n$. Given the string $x \in \{0,1\}^n$, Alice transforms it into the stream over $[n]$ representing the set $Y = \{j : x_j = 1\}$. Given the index $i \in [n]$, Bob transforms it into a stream representing the singleton set $X = \{i\}$. Then $x_i = 1$ if and only if $X \subseteq Y$. □

We now show how to use the result for frequency-based functions to handle duplicated items; in this case X and Y are multisets rather than sets. The next theorem lets us efficiently handle a small number of duplicates.

Theorem 3.8.3. *Let $X, Y \subseteq [n]$ be multisets. Assume k is a known upper bound on the maximum frequency of any element in X or in Y. Then given a stream with elements of X and Y arbitrarily interleaved, there is a online $(kc_a \log n, c_v \log n)$-scheme for determining whether $X \subseteq Y$, for any c_a and c_v with $c_a c_v \geq n$.*

Proof. Let x, y be the characteristic vectors of X and Y respectively. Then $X \subseteq Y$ if and only if $y_i - x_i \geq 0$ for all i. The bound on the maximum frequency implies that $-k \leq y_i - x_i \leq k$ for all $1 \leq i \leq n$. Let $\tilde{g}$ be defined through interpolation as the polynomial of degree $2k$ over the finite field $\mathbb{F}_p$ such that $\tilde{g}(x) = 0$ for $x \in \{0, 1, \ldots, k\}$, and $\tilde{g}(x) = 1$ for $x \in \{-k, -k+1, \ldots, -1\}$. Then $\sum_i \tilde{g}(y_i - x_i) = 0$ if and only if $X \subseteq Y$; intuitively, $\tilde{g}$ acts as an indicator function for the set of possible negative entries in the vector $y - x$. Applying the

polynomial-agreement protocol defined in the proof of Theorem 3.7.1 under this definition of $\tilde{g}$, we obtain a $(kc_a \log n, c_v \log n)$-scheme for checking $X \subseteq Y$ whenever $c_a \cdot c_v \geq n$. $\square$

Finally, we give an online $(n^{2/3} \log n, n^{2/3} \log n)$-scheme for the general multiset inclusion problem, as long as $t = O(n)$.

Theorem 3.8.4. *Let $X, Y \subseteq [n]$ be multisets of size at most t. Then given a stream with elements of X and Y arbitrarily interleaved, there is there is an online $(n^{2/3} \log n, n^{2/3} \log n)$-scheme for determining whether $X \subseteq Y$ assuming $t = O(n)$.*

Proof. Let x, y be the characteristic vectors of X and Y respectively. It holds that $X \subseteq Y$ if and only if $y_i - x_i \geq 0$ for all i. Define $g : \{-t, -t+1, \ldots, 0, 1, \ldots, t\} \to \{0, 1\}$ by $g(x) = 0$ for $x \in \{0, \ldots, t\}$ and $g(x) = 1$ for $x \in \{-t, -t+1, \ldots, -1\}$. The theorem holds by applying the protocol of Theorem 3.7.1 to $G(\mathbf{f})$, where $\mathbf{f}$ is the vector $y - x$ and G is the frequency-based function defined by g. (As stated, the protocol of Theorem 3.7.1 applies only to $g : \mathbb{Z}_+ \to \mathbb{Z}_+$, but it applies without modification to any function g defined on a suitably small domain, such as ours). $\square$

3.9 Matrix-Vector Multiplication

We now give a scheme achieving essentially optimal tradeoffs between annotation length and space usage for multiplying a $b \times c$ integer matrix A by a c-dimensional vector $\mathbf{x}$.

Theorem 3.9.1. *Consider a data stream containing entries of a $b \times c$ matrix A and a c-dimensional vector $\mathbf{x}$, in some arbitrary order, possibly interleaved. We assume that all entries of A and $\mathbf{x}$ are integers of absolute value polynomial in b and c. For any positive integers c_a, c_v such that $c_a c_v \geq c$, there is an online $(bc_a \log(b+c), c_v \log(b+c))$-scheme for computing product $A\mathbf{x}$. Moreover, any (c_a, c_v) protocol requires $c_a \cdot c_v = \Omega(\min(c, b)^2)$ bits for matrices with $\Omega(b \cdot c)$ non-zero entries.*

Proof. We begin with the upper bound. The protocol for verifying inner-products, which follows from Theorem 3.5.2 treats a c dimensional vector (such as a row of A) as a $c_a \times c_v$ array H, where $c_a \cdot c_v \geq c$. This then defines a bivariate polynomial h over a suitably large field $\mathbb{F}_q$, such that h has degree c_a in its first variable and c_v in its second variable, and such that $h(x, y) = H_{x,y}$ for all $(x, y) \in [c_a] \times [c_v]$. For an inner-product between two vectors (such as a row of A and the vector $\mathbf{x}$, treated as arrays H and G respectively), we wish to compute $\sum_{x \in [c_a], y \in [c_v]} G_{x,y} H_{x,y} = \sum_{x \in [c_a], y \in [c_v]} g(x, y)h(x, y)$ for the corresponding arrays G, H and polynomials g, h. These polynomials can then be evaluated at locations outside $[c_a] \times [c_v]$, so in the protocol $\mathcal{V}$ picks a random position r and evaluates $h(r, y)$ and $g(r, y)$ for $1 \leq y \leq c_v$. $\mathcal{P}$ then presents a degree c_a polynomial $p(X)$ which is claimed to be $\sum_{y=1}^{c_v} g(X, y)h(X, y)$. $\mathcal{V}$ checks that $p(r) = \sum_{y=1}^{c_v} g(r, y)h(r, y)$, and if so accepts $\sum_{x=1}^{c_a} p(x)$ as the correct answer.

In Theorem 3.5.2 it is shown how $\mathcal{V}$ can compute $h(r, y)$ efficiently as H is defined incrementally in the stream: each addition of δ to a particular index is mapped to $(x, y) \in [c_a] \times [c_v]$, which causes $h(r, y) \leftarrow h(r, y) + \delta \cdot \chi_{x,y}(r)$, where $\chi_{x,y}$ is a Lagrange polynomial. Equivalently, the final value of $h(r, y)$ over updates in the stream where the jth update is $t_j = (\delta_j, x_j, y_j)$ is $f(r, y) = \sum_{t_j : y_j = y} \delta_j \cdot \chi_{x_j,y}(r)$.

To run this protocol over multiple vectors in parallel naively would require keeping the $h(r, y)$ values implied by each different vector separately, which would be costly, as it would increase *both* the annotation and the space usage by a factor of b relative to a single inner product query. Our observation is that rather than keep these values explicitly, it is sufficient to keep only a fingerprint of these values, using the linearity of fingerprint functions to finally test whether the polynomials provided by $\mathcal{P}$ for each vector together agree with the stored values.

In our setting, the $b \times c$ matrix A implies b bivariate polynomials $h_1, \ldots, h_b$ of degree c_a in the first variable and c_v in the second. We evaluate each polynomial at (r, y) for $1 \leq y \leq c_v$

for the same value of r: since each test is fooled by $\mathcal{P}$ with small probability, the chance that none of them is fooled can be kept high by choosing the field to evaluate the polynomials over to have size polynomial in $b + c$. Thus, conceptually, the parallel invocation of b instances of this protocol require us to store $h_i(r, y)$ for $1 \leq y \leq c_v$ and $1 \leq i \leq b$ (for the b rows of A), as well as $g(r, y)$ for $1 \leq y \leq c_v$ (where g is the polynomial derived from $\mathbf{x}$). Rather than store this set of $b \cdot c_v$ values explicitly, $\mathcal{V}$ instead stores only c_v fingerprints, one for each value of y, where each fingerprint captures the vector b values of $h_i(r, y)$.

From the definition of our fingerprinting function in Lemma 3.2.1, this means over stream updates $t_j = (\delta_j, i_j, x_j, y_j)$ of weight δ_j to row i_j and column indexed by x_j and y_j we compute one fingerprint z_y for each value $y \in [c_v]$:

$$z_y = \sum_{i=1}^{b} h_i(r, y)\alpha^i = \sum_{i=1}^{b} \sum_{t_j : y_j = y, i_j = i} \delta_j \cdot \chi_{x_j, y}(r_j)\alpha^i,$$

where α is chosen uniformly at random from $\mathbb{F}_q$ as in Lemma 3.2.1. Observe that for each y this can be computed incrementally in the stream by storing only r and the current value of z_y.

To verify the correctness, $\mathcal{V}$ receives the b polynomials p_i, one for each row, and incrementally builds a fingerprint z^* of the b-dimensional vector whose ith entry is $p_i(r)$. $\mathcal{V}$ then tests whether

$$\sum_{y=1}^{c_v} z_y g(r, y) = z^*.$$

To see the correctness of this, we expand the left hand side as

$$\begin{aligned}
\sum_{y=1}^{c_v} z_y g(r, y) &= \sum_{y=1}^{c_v} \big(\sum_{i=1}^{b} h_i(r, y)\alpha^i \big) g(r, y) \\
&= \sum_{y=1}^{c_v} \big(\sum_{i=1}^{b} g(r, y) h_i(r, y)\alpha^i \big) \\
&= \sum_{i=1}^{b} \big(\sum_{y=1}^{c_v} g(r, y) h_i(r, y) \big) \alpha^i
\end{aligned}$$

Likewise, if all p_i's are as claimed, then

$$z^* = \sum_{i=1}^{b} p_i(r)\alpha^i = \sum_{i=1}^{b}\Big(\sum_{y=1}^{c_v} g(r,y)h_i(r,y)\Big)\alpha^i$$

Thus, if the p_i's are as claimed, then these two fingerprints will match. Moreover, by the Schwartz-Zippel lemma (Lemma 2.2.1), and the fact that α and r are picked uniformly at random from $\mathbb{F}_q$ by $\mathcal{V}$ and not known to $\mathcal{P}$, the fingerprints will not match with high probability if the p_i's are *not* as claimed, when the polynomials are evaluated over a field of size polynomial in $(b+c)$.

To analyze the vcost, we observe that $\mathcal{V}$ can compute all fingerprints in $O(c_v)$ space. As $\mathcal{P}$ provides each polynomial $p_i(x)$ in turn, $\mathcal{V}$ can incrementally compute z^* and check that this matches $\sum_{y=1}^{c_v} z_y g(r,y)$. At the same time, $\mathcal{V}$ also computes $\sum_{i=1}^{b}\sum_{x=1}^{c_a} p_i(x)$, as the value of $A\mathbf{x}$. Note that if each p_i is sent one after another, $\mathcal{V}$ can forget each previous o_i after the required fingerprints and evaluations have been made; and if c_a is larger than c_v, $\mathcal{V}$ does not even need to keep p_i in memory, but can instead evaluate it term by term in parallel for each value of x. Thus the total space needed by $\mathcal{V}$ is dominated by the c_v fingerprints and check values.

The total size of the information sent by $\mathcal{P}$ is dominated by the b polynomials of degree c_a.

To prove the lower bound, we give a simple reduction of INDEX to matrix-vector multiplication. Suppose we have an instance (x,k) of INDEX where $x \in \{0,1\}^{n^2}$, $k \in [n^2]$. Alice constructs an $n \times n$ matrix A from x alone, in which $A_{i,j} = 1$ if $x_{f(i,j)=1}$, where f is a 1-1 correspondence $[n] \times [n] \to [n^2]$, and $A_{i,j} = 0$ otherwise. Assume $f(i,j) = k$. Bob then constructs a vector $\mathbf{x} \in \mathbb{R}^n$ such that $\mathbf{x}_i = 1$ and all other entries of $\mathbf{x}$ are 0. Then the j'th entry of $A\mathbf{x}$ is 1 if and only if $x_{f(i,j)=1}$, and therefore the value of $x_{f(i,j)}$ can be extracted from the vector $A\mathbf{x}$. Therefore, if we had a (c_a, c_v) protocol for verifying matrix-vector multiplication given an $n \times n$ matrix A (even for a stream in which all entries of A come before all entries

of $\mathbf{x}$), we would obtain a $(\sqrt{c_a}, \sqrt{c_v})$ protocol for INDEX. The lower bound for matrix-vector multiplication thus holds by a lower bound for INDEX given in Theorem 3.3.2. □

3.10 Graph Problems

In this section we consider computing properties of graphs with n vertices, determined by a stream of m edges [48, 64] (possibly with deletions of edges). That is, each stream update specifies an edge in $[n] \times [n]$ to be inserted or deleted. We present tight results for connectivity of sparse graphs, bipartiteness, determining if a bipartite graph has a perfect matching, counting triangles, shortest s-t path, and minimum weight bipartite perfect matching. Our (standard) bipartite perfect matching result achieves optimal tradeoffs up to logarithmic factors, as does our shortest s-t path result for small-diameter graphs.

3.10.1 Counting Triangles via Matrix Multiplication

Estimating the number of triangles in a graph has received significant attention because of its relevance to database query optimization—knowing the degree of transitivity of a relation is useful when estimating the cost of evaluation plans for certain relational queries—and investigating structural properties of the web-graph and social graphs [11, 25, 67]. In the absence of annotation, any single-pass algorithm to determine if there is a non-zero number of triangles requires $\Omega(n^2)$ bits of space, where n is the number of vertices in the graph [11]. In contrast, we show that the exact number of triangles can be verified in logarithmic space, with the help of $O(n^2 \log n)$ bits of annotation. The following theorem, proved using ideas from Bar-Yossef et al. [11] coupled with Theorem 3.5.5, shows that this amount of annotation is nearly optimal, for a log-space verifier.

Theorem 3.10.1. *Any (c_a, c_v)-scheme for counting triangles on graphs with n vertices must*

have $c_a \cdot c_v = \Omega(n^2)$ for graphs with $\Omega(n^2)$ edges, even in the insert-only update model.

Proof. We show a reduction from $\text{DISJ}_{(n^2/9),2}$. We represent an instance of DISJ as a pair of $(n/3) \times (n/3)$ Boolean matrices X, Y in the natural way. We proceed to construct a graph that has a triangle if and only if $X_{ij} = Y_{ij} = 1$ for some $i, j \in [n/3]$. The nodes are partitioned into sets U, V, W so that $|U| = |V| = |W| = n/3$. Insert edges $\{(u_i, w_i) : i \in [n/3]\} \cup \{(u_i, v_j) : X_{ij} = 1\} \cup \{(w_i, v_j) : Y_{ij} = 1\}$. There is a triangle (u_i, v_j, w_i) iff $X_{ij} = Y_{ij} = 1$, and there is no other way to form a triangle. The result follows from Theorem 3.5.5. □

We now outline an online scheme with vcost $= O(\log n)$ and hcost $= O(n^2 \log n)$. A major subroutine of our algorithm is the verification of (integer) matrix multiplication in our model. That is, given $n \times n$ matrices A, B and C with integer entries, verify that $AB = C$. Our technique extends the classic result of Frievalds [53] by showing that if the prover presents the results in an appropriate order, the verifier needs only $O(\log n)$ bits to check the claim. Note that this much annotation is necessary if the prover is to provide C in his stream.

Theorem 3.10.2. *There exists an online $(n^2 \log n, \log n)$-scheme for multiplying two $n \times n$ integer matrices.*

Proof. Let q be a prime larger than $2n \cdot o^2 + 1$, where o is an *a priori* upper bound on the absolute values of all entries of A and B. By the result of Kimbrel and Sinha [68], the verifier can check $AB = C$ by picking r uniformly from $\mathbb{F}_q$ and checking that $A(B\mathbf{r}^\mathsf{T}) = C\mathbf{r}^\mathsf{T}$, in the field $\mathbb{F}_q$, for vector $\mathbf{r} = (r^0, r^1, \ldots, r^{n-1})$. This fails to identify an incorrect product with probability at most n/q. Rather than computing $A(B\mathbf{r}^\mathsf{T})$ and $C\mathbf{r}^\mathsf{T}$ explicitly, the verifier will compare fingerprints of $C\mathbf{r}^\mathsf{T}$ and $AB\mathbf{r}^\mathsf{T}$. These are computed as $\mathbf{y}C\mathbf{r}^\mathsf{T}$ and $\mathbf{y}AB\mathbf{r}^\mathsf{T}$, for a vector $\mathbf{y} = (y^0, y^1, \ldots, y^{n-1})$ where y is picked uniformly from $\mathbb{F}_q$. This fingerprinting fails to distinguish distinct vectors with probability at most n/q.

We observe that (1) $\mathbf{y}C\mathbf{r}^{\mathsf{T}} = \sum_{i,j} s^i r^j C_{ij}$ can be computed easily whatever order the entries of C are presented in. (2) $\mathbf{y}AB\mathbf{r}^{\mathsf{T}} = (\mathbf{y}A)(B\mathbf{r}^{\mathsf{T}})$ is the inner product of two n-dimensional vectors, and that $(\mathbf{y}A)_i = \sum_j y^j A_{ij}$ and $(B\mathbf{r}^{\mathsf{T}})_i = \sum_j r^j B_{ji}$. Therefore, if the prover presents the ith column of A followed by the ith row of B for each i in turn, the verifier can easily compute $\mathbf{s}AB\mathbf{r}^{\mathsf{T}}$ in $O(\log q)$ space. Picking $q \geq 6n$ ensures that the verifier is fooled with probability at most $1/3$, and the total space used by the verifier to store r, y and intermediate values is $O(\log n)$. □

With this primitive, arbitrary matrix products $A_\ell A_{\ell-1} \cdots A_2 A_1$ are verified with $O(\ell n^2 \log n)$ annotation by verifying $\underline{A}^{(2)} := A_2 A_1$, then $\underline{A}^{(3)} := A_3 \underline{A}^{(2)}$, etc. Matrix powers A^ℓ are verified with $O(n^2 \log \ell \log n)$ annotation, using repeated squaring. Here, we assume that the entries computed do not grow too large, and so can be represented within $O(\log n)$ bits.

Theorem 3.10.3. *There is an online $(n^2 \log n, \log n)$-scheme for counting triangles in a graph with n nodes.*

Proof. Denote the graph adjacency matrix by A, with $A_{ii} := 0$. The prover lists A_{vw} and A^2_{vw} for all pairs (v, w) in some canonical order. The verifier computes $\sum_{v,w} A_{vw} A^2_{vw}$ as the number of triangles. The verifier uses fingerprints to check that A matches the original set of edges, and the scheme in Theorem 3.10.2 to ensure that A^2 is as claimed. □

We also show that it is possible to trade off the computation with the prover in a "smooth" manner. The approach is based on the following observation of Bar-Yossef et al. [11].

From the given stream of edges of a graph, we can create a *derived* stream, of length $m(n-2)$, by replacing each edge (u, v) with the set of triples $\{(u, v, w) : w \notin \{u, v\}\}$. The frequency moments of this derived stream can be expressed in terms of the numbers of triples of nodes with exactly zero, one, two and three edges between them. It follows that the number of triangles can be expressed in terms of the frequency moments of this derived

stream, as $(F_3 - 3F_2 + 2F_1)/6$. By using the scheme of Theorem 3.5.3, we obtain the following theorem.

Theorem 3.10.4. *For every pair of positive integers c_a, c_v such that $c_a \cdot c_v \geq n^3$, there is an online $(c_a \log n, c_v \log n)$-scheme for counting triangles in a graph with n nodes.*

3.10.2 Bipartite Perfect Matching

We present an online scheme for determining whether a bipartite graph on n nodes has a perfect matching. Our scheme achieves *optimal* tradeoffs between hcost and vcost for dense graphs (i.e., graphs with $\Omega(n^2)$ edges), up to logarithmic factors. Graph matchings have been considered in the stream model [48, 108] and it can be shown that any single-pass algorithm for determining the exact size of the maximum matching requires $\Omega(n^2)$ space. We show that for any positive integers c_a, c_v satisfying $c_a c_v \geq n$ we can off-load this computation to the prover such that, with only $O(nc_a \log n)$ annotation, the answer can be verified in $O(c_v \log n)$ space. This is shown to be best possible by combining a reduction from [48] coupled with Theorem 3.3.2. Later (see Theorem 3.11.1), we also present an online $(m \log n, \log n)$ scheme for this problem. Here, m denotes the number of edges in the graph – this corresponds to the sparsity of the stream of edges, and so this notation is consistent with the use of m to denote stream sparsity throughout this chapter and the next.

Theorem 3.10.5. *Let c_a, c_v be positive integers such that $c_a c_v \geq n$. There exists an online $(nc_a \log n, c_v \log n)$-scheme for bipartite perfect matching on graphs with n nodes. Any online (c_a, c_v)-scheme for bipartite perfect matching requires $c_a \cdot c_v = \Omega(n^2)$ for graphs with $\Omega(n^2)$ edges, even in the insert-only update model.*

Proof. We begin with the upper bound. Our scheme follows the following outline: if G has a perfect matching, the prover provides the matching, while if G has no perfect matching, the

prover demonstrates this via Hall's Theorem. The details follow.

If there is a perfect matching $\mathcal{M}$, the annotation lists all edges in $\mathcal{M}$, followed by a proof that $\mathcal{M} \subseteq E$. More specifically, for any $c_a \cdot c_v \geq n^2$, Theorem 3.8.2 describes how to obtain an online $(c_a \log n, c_v \log n)$-scheme for showing $\mathcal{M} \subseteq E$, assuming no duplicate edges. This can be extended to a $(kc_a \log n, c_v \log n)$-scheme if edges may be duplicated up to k times by Theorem 3.8.3. The prover uses this scheme to demonstrate $\mathcal{M} \subseteq E$, and the verifier checks that $\mathcal{M}$ is a matching by comparing a fingerprint of M to one of the set $\{1, 2, \ldots, n\}$.

If the graph does not have a perfect matching, let (L, R) be a bipartition, and let $L' \subseteq L$ be such that $|L'| > |\Gamma(L')|$. We will use the online $(nc_a \log n, c_v \log n)$-scheme for integer $n \times n$ matrix-vector multiplication given in Theorem 3.9.1. The verifier must check that (1) L is a bipartition of n; (2) $L' \subseteq L$; and (3) $|L'| > |\Gamma(L')|$. Let $\mathbf{x} \in \{0, 1\}^n$ be the indicator vector of L, and let A be the adjacency matrix of G, i.e., $A_{ij} = 1$ if there is an edge between i and j in G and $A_{ij} = 0$ otherwise. Condition (1) is equivalent to $\mathbf{x}^\mathsf{T} A\mathbf{x} = 0$, which can be checked using integer matrix-vector multiplication to verify $A\mathbf{x}$, followed by an inner-product scheme to verify $\mathbf{x}^\mathsf{T} A\mathbf{x}$. Condition (2) can be checked trivially while the prover specifies L by requiring the nodes of L' to be marked. To check (3), notice that $|\Gamma(L')|$ is equal to the number of non-zero entries in the vector $A\mathbf{x}$. This can be computed while the verifier checks (1), and that $|\Gamma(L')| < |L'|$.

The result is an online $(knc_a \log n, c_v \log n)$-scheme, where k is an *a priori* upper bound on the number of times each edge may be duplicated. □

3.10.3 Bipartiteness

The problem of determining if a graph is bipartite was considered in the standard stream model [48, 49], and it can be shown that any one-pass algorithm without annotations needs $\Omega(n)$ bits of space. We prove later (see Theorem 3.11.1) that in our model, the prover can

convince a verifier with $O(\log n)$ space whether a graph is bipartite, using only $O(m \log n)$ annotation, and we show that this is essentially the best possible for sparse graphs where $m = O(n)$ using a reduction from $\text{DISJ}_{n,2}$ to bipartiteness. Here, we achieve tradeoffs between hcost and vcost for dense graphs, obtaining an online $(nc_a \log n, c_v \log n)$-scheme for every pair of positive integers c_a, c_v such that $c_a c_v \geq n$.

Theorem 3.10.6. *Let c_a, c_v be positive integers such that $c_a \cdot c_v \geq n$. There exists an online $(nc_a \log n, c_v \log n)$-scheme for bipartiteness on graphs with n nodes. Any (c_a, c_v)-scheme (online or prescient) for bipartiteness requires $c_a \cdot c_v = \Omega(n)$ even when $m = O(n)$, where m is the number of edges.*

Proof. In our scheme, the prover proves that a graph is *non*-bipartite by providing an odd cycle C. The verifier must check that the number of edges in C is odd, that C is a cycle, and that $C \subseteq E$. The verifier can easily perform the first two checks in logarithmic space. The verifier checks that $C \subseteq E$ using Theorem 3.8.2.

The prover proves that a graph *is* bipartite by specifying all nodes L in the left set of a bipartition. Checking that L is indeed a bipartition of G can be done exactly as in Theorem 3.10.5.

For the lower bound, we reduce an instance $(x, y) \in \{0,1\}^n \times \{0,1\}^n$ of $\text{DISJ}_{n,2}$ to an instance of bipartiteness on a graph with $O(n)$ edges over nodes $(v_{ij})_{i \in [3], j \in [n]}$. For each $j \in [n]$, create edges (v_{1j}, v_{2j}); if $x_j = 1$, add the edge (v_{1j}, v_{3j}), and if $y_j = 1$, add the edge (v_{2j}, v_{3j}). The resulting graph contains an odd cycle if and only if x and y are not disjoint. □

3.10.4 Connectivity

The problem of determining if a graph is connected was considered in the standard stream model [48, 64] and the multi-pass W-stream model [46]. In both models, it can be

shown that any constant-pass algorithm without annotations needs $\Omega(n)$ bits of space. Later (see Theorem 3.11.1), we show that we can off-load this computation to the prover such that, with only $O(m \log n)$ annotation, the answer can be verified in $O(\log n)$ space. This is essentially the best possible for sparse graphs where $m = O(n)$ by combining a reduction from [48] with Theorem 3.3.2. Here, we achieve tradeoffs between hcost and vcost for dense graphs, obtaining an online $(nc_a \log n, c_v \log n)$-scheme for every pair of positive integers c_a, c_v such that $c_a c_v \geq n$.

Theorem 3.10.7. *Let c_a, c_v be positive integers such that $c_a \cdot c_v \geq n$. There exists an online $(nc_a \log n, c_v \log n)$-scheme for connectivity on graphs with n nodes. Any (c_a, c_v)-scheme (online or prescient) for connectivity in the insert-only update model requires $c_a \cdot c_v = \Omega(n)$ even when $m = O(n)$, where m is the number of edges.*

Proof. Our scheme follows the following conceptual outline: if G is connected, the prover demonstrates this by providing a spanning tree; if G is disconnected, the prover identifies a connected component of the graph. In the first case, the prover provides a set of edges T claimed to be a spanning tree, and the verifier must check that (1) T is spanning and that (2) $T \subseteq E$. Checking (1) is accomplished by appropriate labeling of the $O(n)$ edges, with $O(n)$ annotation as follows. T can be chosen to be directed towards the root, such that there is an injective labeling of the nodes $\ell : V \to [n]$ such that each non-root node with label j is linked to exactly one node with label greater than j. The prover outputs such a function ℓ, and the verifier ensures that it is an injection. Then each (directed) edge (u, v) in T and its labels $\ell(u) < \ell(v)$ is presented in decreasing order of $\ell(u)$. The verifier checks this order, and ensures that it is consistent with ℓ via fingerprinting (as per Lemma 3.8.1). By Theorem 3.8.2, condition (2) can be checked with space $O(c_v \log n)$ and annotation $O(nc_a \log n)$.

If G is disconnected, the prover presents a set $L \subset V$, $L \neq V$, and claims that L is disconnected from $V \setminus L$. Let A be the adjacency matrix of G, and let $\mathbf{x} \in \{0, 1\}^n$ be the

indicator vector of L. To check that L is as claimed, it suffices for the verifier to compute $A\mathbf{x}$, and check that the each non-zero entry of $A\mathbf{x}$ corresponds to vertices in L (intuitively, this means the set L' of vertices at distance one from L is contained in L). The first step uses the integer matrix-vector multiplication scheme of Theorem 3.9.1. This allows the verifier to ensure that the set $\{i : (A\mathbf{x})_i \neq 0\}$ matches L, via fingerprints.

For the lower bound, we reduce an instance of $\text{DISJ}_{n,2}$ to connectivity of a graph with $O(n)$ edges over nodes $v_{0,0} \ldots v_{3,n}$: create edges $(v_{j,0}, v_{j,i})$ for $j \in \{0, 2, 3\}$ and $i \in [n]$. Then if $x_i = 1$, add edge $(v_{0,i}, v_{1,i})$, else add edge $(v_{1,i}, v_{2,i})$; and if $y_i = 1$, add edge $(v_{1,i}, v_{3,i})$ else add edge $(v_{2,i}, v_{3,i})$. The resulting graph is connected only if x and y are not disjoint. The result follows from Theorem 3.5.5. □

3.10.5 Totally Unimodular Integer Programs

Consider a linear program of the form $\max\{\mathbf{c}^T\mathbf{x} \mid A\mathbf{x} \leq \mathbf{b},\}$. where $\mathbf{b}$ is a b-dimensional vector and $\mathbf{c}$ is a c-dimensional vector. The dual of this program is of the form
$\{\min \mathbf{b}^T\mathbf{y} | A^T\mathbf{y} = \mathbf{c}, \mathbf{y} \geq 0\}$.

Weak LP duality implies that any feasible solution $\mathbf{y}$ to the dual linear program yields an upper bound on the value of the primal linear program. Strong LP duality implies that given an optimal solution $\mathbf{x}$ to the primal linear program, there is in fact a dual solution whose value *equals* that of $\mathbf{x}$, thereby witnessing the optimality of $\mathbf{x}$. This suggests a natural approach to developing a scheme for linear programs: $\mathcal{P}$ tells $\mathcal{V}$ a (claimed) optimal solution $\mathbf{x}$ to the linear program, as well as a (claimed) optimal solution $\mathbf{y}$ for the dual program, and then proves that $\mathbf{x}$ and $\mathbf{y}$ are primal and dual feasible respectively, and that their values are equal. Of course, it is necessary (and often non-trivial) to ensure that optimal solutions $\mathbf{x}$ and $\mathbf{y}$ can be specified succinctly, and that $\mathcal{V}$ can check their feasibility and values with minimal space and annotation.

Definition 3.10.8. An integer matrix A is *totally unimodular* if all sub-determinants of A have absolute value 1.

Consider an integer program of the form $\max\{\mathbf{c}^T\mathbf{x} \mid A\mathbf{x} \leq \mathbf{b}, \mathbf{b} \in \mathbb{Z}^c\}$. It is well-known [87, Corollary 19.1a] that if the constraint matrix A of an *integer* program is totally unimodular, then any extreme point solution $\mathbf{y}$ to the dual of the linear programming relaxation of the integer program is in fact integral. Hence, the suggested approach to developing schemes for linear programs in fact applies to totally unimodular integer programs: $\mathcal{P}$ can specify an optimal solution $\mathbf{x}$ to the primal program and prove its optimality by specifying an integral optimal solution $\mathbf{y}$ to the LP relaxation of the dual. We execute this approach below to develop schemes for two important graph problems that can be formulated as totally unimodular integer programs: shortest s-t path, and minimum weight bipartite perfect matching (MWBPM).

Shortest s-t Path. We start with a lower bound on the space/annotation tradeoffs achievable by any online scheme that even approximates the shortest s-t path problem up to a factor 4/3.

Theorem 3.10.9. *For any nodes s and t, any online (c_a, c_v)-scheme that approximates the length of the shortest s-t path to within a factor 4/3 requires $c_a \cdot c_v \geq n^2$ for graphs on n nodes with $\Omega(n^2)$ edges, even in the insert-only update model.*

Proof. The lower bound follows from a straightforward reduction from INDEX, for which a lower bound linear in hcost $\cdot$ vcost was proven in 3.3.2.

Given an instance (x, k) of INDEX where $x \in \{0,1\}^{\binom{n}{2}}$, $k \in [\binom{n}{2}]$, we construct a graph G, with $V_G = [n+2]$, and $E_G = E_A \cup E_B$. The edge set $E_A = \{(i,j) : x_{(i,j)} = 1\}$ is created from x alone, where, without loss of generality, we assume that x is indexed by edges (i,j) with $1 \leq i < j \leq n$. Then E_B is created from k alone, as $E_B = \{(n+1, i), (j, n+2)\}$

using $(i, j) = k$. Note that E_A and E_B can be created by $\mathcal{V}$ incrementally as the stream is seen, using $O(\log n)$ bits of memory, to generate an implicit edge stream. The shortest path between nodes $n+1$ and $n+2$ is 3 if $x_k = 1$ and is 4 or more otherwise. Hence, solving the s-t path problem with $\text{hcost} \cdot \text{vcost} = o(n^2)$ would also solve the INDEX problem with this bound, contradicting the linear (in the length of x) bound from Theorem 3.3.2. This also implies that any approximation within $\sqrt{4/3}$ requires $\text{hcost} \cdot \text{vcost} = \Omega(n^2)$. □

We now give an online scheme achieving space/annotation tradeoffs that are essentially optimal for small-diameter graphs. A key insight is that for the relevant totally unimodular integer program, any dual solution is quite compact, requiring $O(n \log n)$ bits to specify, while a primal optimal solution can be succinctly specified by directly demonstrating a path which obtains the claimed length.

Theorem 3.10.10. *Given a graph G with n vertices specified as a stream of weighted directed edges such that each edge appears at most once, let $d(s, w)$ denote the shortest-path distance from s to w in G, and let C_s be the set of nodes reachable from s. Let $d = \max_{w \in C_s} d(s, w)$ be the maximum distance from s to any node reachable from s. For any c_a, c_v such that $c_a c_v \geq dn^2$ and $c_a \geq dn$, there is an online $(c_a \log n, c_v \log n)$ scheme for shortest s-t path on directed graphs with non-negative integer edge weights.*

Proof. Our protocol handles graphs with non-negative integer edge weights; notice however that the lower bound of $c_a \cdot c_v = \Omega(n^2)$ from Corollary 3.10.9 applies even to unweighted graphs with constant diameter, so our protocol is optimal in this regime. We assume an upper bound on d is known in advance, and later show how to remove this assumption at the cost of a logarithmic factor in space, and no asymptotic increase in annotation.

To aid in the computation, $\mathcal{V}$ tracks properties of an (implicit) derived matrix X. Before observing the stream, $\mathcal{V}$ conceptually sets all entries of X to d, where d is the (assumed)

upper bound on the distances. Then $\mathcal{V}$ sees the set of weights w_{ij} while observing the stream and treats each as an addition of $w_{ij} - d$ to entry (i, j) of X; this has the effect of setting $X_{ij} = w_{ij}$. This requires the assumption that each edge (i, j) appears at most once in the stream. At the end of the stream, $X_{ij} = w_{ij}$ if $(i, j) \in E$, and $X_{ij} = d$ otherwise. We note that it is straightforward for $\mathcal{V}$ to check in parallel that each edge appears at most once by tracking the matrix Y which counts the number of times each edge (i, j) is seen, and verifying that the squared Frobenius norm of Y, $\|Y\|_F^2 = \sum_{i,j} Y_{ij}^2$, satisfies $\|Y\|_F^2 = m$, using the F_2 protocol described above.

First we handle the case where an s-t path exists.

Upper bound on path length. To prove an upper bound on the value of the shortest path, $\mathcal{P}$ lists the edges in a valid s-t path $\mathcal{P}$.

To compute the cost of $\mathcal{P}$, the inner-product protocol that follows from Theorem 3.5.2 is used to compute $P \cdot X$, where we treat $\mathcal{P}$ as an indicator matrix, i.e. $P_{ij} = 1$ iff (i, j) is an edge in $\mathcal{P}$, and 0 otherwise. If $\mathcal{P}$ includes any edges (i, j) not present in E, then $X_{ij} = d$ and so these are charged at a cost of d. That is, the cost of $\mathcal{P}$ is made higher than the bound on distances, so it is easily detected if $\mathcal{P}$ contains edges not in E. This protocol requires $O(c_a)$ annotation and $O(c_v)$ space for any $c_a \cdot c_v \geq n^2$.

Lower bound on path length. To prove a lower bound on the value of the shortest path, we leverage the total unimodularity of the integer program for the problem. The dual integer program for shortest s-t path has a variable y_i for every $i \in V$ and a constraint for every edge $(i, j) \in E$:

$$\text{maximize } \mathbf{y}_t - \mathbf{y}_s \text{ subject to } \mathbf{y}_j - \mathbf{y}_i \leq w_{ij} \text{ for all } (i, j) \in E.$$

At a high level, $\mathcal{P}$ will prove a lower bound on the value of the shortest s-t path by

presenting a feasible solution $\mathbf{y}$ to the above linear program. Importantly, we note that the solution is *compact*: $\mathbf{y}$ has only n variables. We present a carefully posed protocol allowing $\mathcal{V}$ to check that $\mathbf{y}$ satisfies all m constraints using sublinear annotation.

First, we show that there exists an optimal solution to the dual such that $\mathbf{y}_s = 0$ and all $\mathbf{y}$ are non-negative integers with $\mathbf{y}_i \leq d$ for all i, where $d = \max_{v \in C_s} d(s, v)$ and C_s is the set of nodes reachable from s. Let $\mathbf{y}_v = d$ for all v not reachable from s, and let $\mathbf{y}_v = d(s, v)$ if v is reachable from s. All dual constraints are satisfied by $\mathbf{y}$: if not, suppose $\mathbf{y}_j - \mathbf{y}_i > w(i, j)$ for some edge (i, j). Then clearly $\mathbf{y}_i < d$ since edge weights are non-negative and $\mathbf{y}_j \leq d$, and hence i is reachable from s. But then j is reachable from s as well, and this contradicts that $\mathbf{y}_j = d(s, j)$, as there is a path from s to j of cost $\mathbf{y}_i + w(i, j) < \mathbf{y}_j$.

Specifying $\mathbf{y}$ requires $O(n \log n)$ bits of annotation since there are n dual variables (more precisely, it requires $n' \log n$ bits where $n' = |C_s|$, since there are only n' variables not set to d). $\mathcal{V}$ immediately outputs $\perp$ if any variable $\mathbf{y}_i$ in the solution is non-integral, if $\mathbf{y}_s \neq 0$, or if $\mathbf{y}_v > d$ for any v.

Given the dual assignment $\mathbf{y}$, let $W \in \mathbb{Z}^{n^2}$ be the matrix defined by

$$W_{ij} = w_{ij} - \mathbf{y}_j + \mathbf{y}_i \text{ if } (i, j) \in E \text{ and } W_{ij} = d - \mathbf{y}_j + \mathbf{y}_i \text{ if } (i, j) \notin E.$$

It is clear that the $\mathbf{y}_i$'s constitute a feasible assignment to the dual if and only if $W_{ij} \geq 0$ for all $(i, j) \in E$: if $(i, j) \in E$, $W_{ij} \geq 0$ only if the constraint corresponding to edge (i, j) is satisfied, and if $(i, j) \notin E$, the addition of d to W_{ij} ensures $X_{ij} \geq 0$, which corresponds to "no constraint". We also observe that $W_{ij} = X_{ij} - y_j + y_i$ for X as described above.

Let $w_{\max}$ be the heaviest edge in G. We can assume $w_{\max} \leq d$ since $\mathcal{V}$ can filter away any edges with $w_{\max} > d$, as these edges will not effect the value of the shortest $s - t$ path.

We apply the online scheme of Theorem 3.5.2 to W, using the lowest-degree polynomial g over $\mathbb{F}_p$ such that $g(x) = 0$ for $x \in \{1, \ldots, d + w_{\max}\}$ and $g(x) = 1$ for $x \in \{-d, \ldots, 0\}$. g has degree $w_{\max} + 2d = O(d)$, and clearly $\sum_{i,j \in [n]} g(w_{i,j}) = 0$ if and only if $\mathbf{y}$ is feasible for

the dual LP, as for all (i, j), $-d \leq W_{ij} \leq w_{\max} + d$. The cost of the scheme of Theorem 3.5.2 l is thus $O(dc_a)$ annotation and $O(c_v)$ space for any $c_a \cdot c_v \geq n^2$. Lastly, note that $\mathcal{V}$ can apply the scheme of Theorem 3.5.2 on the matrix X derived from the stream, and updated by performing the necessary additions and subtractions of y_i and y_j values to all affected coordinates.

We now remove the assumption that d is known in advance, at the cost of a logarithmic increase in space. At a high level, while observing the stream $\mathcal{V}$ can keep logarithmically many "guesses" for the value of d, and after the stream is seen, $\mathcal{P}$ can tell $\mathcal{V}$ which guess is the tightest upper bound on the value of all variables in the optimal solution to the dual LP. Then $\mathcal{V}$ can forget about the other guesses and simply complete the execution of the protocol corresponding to the best guess.

More formally, it suffices for $\mathcal{V}$, while observing the stream, to run $O(\log n)$ instances of the above protocol in parallel, with the i'th instance run with parameter $d = 2^i$. This ensures that one instance will be run with parameter $d \leq \max_i \mathbf{y}_i < 2d$. We require $\mathcal{P}$ to prepend the annotation with the value $d^* = \min\{2^i : \mathbf{y}_j \leq 2^i \text{ for all } j\}$. $\mathcal{V}$ then needs only to continue the instance of the protocol run with parameter $d = d^*$. If d^* is not as claimed, $\mathcal{V}$ will detect this when a dual variable $\mathbf{y}_j$ is presented with $\mathbf{y}_j > d^*$, and output $\perp$. Thus, the protocol is valid. The space cost increases by a logarithmic factor compared to when the true value of d is known in advance, since $\mathcal{V}$ must run $O(\log n)$ instances of the protocol while observing the stream. The annotation cost does not increase asymptotically, since the only instance of the protocol $\mathcal{V}$ continues to run after the stream has been observed satisfies $2d \leq \max_i \mathbf{y}_i$ i.e. was run with a "guess" for d that was within a factor of two of the true value of d.

No path from s to t. If the shortest s-t path is infinite (there is no s-t path), let $C_t \subseteq V$ be the connected component of t. Then the dual assignment with $\mathbf{y}_i = 1$ for $i \in C_t$ and $\mathbf{y}_i = 0$

for all other i satisfies $\mathbf{y}_i - \mathbf{y}_j = 0$ for all $(i,j) \in E$, and the value of the dual objective function $\mathbf{y}_t - \mathbf{y}_s$ is positive. By Farkas' Lemma, this serves as a witness to the fact that the primal is infeasible. $\mathcal{V}$ can check $\mathbf{y}$ is as claimed by running the sum-check scheme of Theorem 3.5.2 on the vector Y with $Y_{ij} = \mathbf{y}_i - \mathbf{y}_j$ if $(i,j) \in E$, and $Y_{ij} = \mathbf{y}_i - \mathbf{y}_j - 3$ if $(i,j) \notin E$ and using the degree-5 polynomial g over $\mathbb{F}_p$ such that $g(0) = g(-4) = g(-3) = g(-2) = 0$ and $g(-1) = g(1) = 1$. We can construct the derived stream in the same manner as X and W above. It is clear that if $(i,j) \notin E$ then $Y_{ij} \in \{-4,-3,-2\}$, if $(i,j) \in E$ with $\mathbf{y}_i - \mathbf{y}_j = 0$ then $Y_{ij} = 0$, and otherwise $Y_{ij} \in \{-1,1\}$. Thus, $\sum_{i,j} g(Y_{ij}) = 0$ if and only if $\mathbf{y}$ is as claimed; this instance of the sum-check scheme of Theorem 3.5.2 requires annotation $O(c_a)$ and space $O(c_v)$ for any $c_a \cdot c_v \geq n^2$. □

Remark 2. If d is not known in advance, then neither $\mathcal{P}$ nor $\mathcal{V}$ knows the annotation cost of the protocol of Theorem 3.10.10 until after observing the stream. Only the space usage c_v can be fixed in advance in this case, and the annotation cost will be $O(n^2 d/c_v)$.

We note that the protocol of Theorem 3.10.10, as well as Theorem 3.10.11 below, does not handle edge weights that are specified incrementally. The reason is that $\mathcal{V}$ must be able to derive a stream specifying the matrices X and W, which we know how to do only in the absence of duplicate edges. This is in contrast to schemes such as the matrix vector multiplication scheme Theorem 3.9.1, which works even when the entries of the matrix and the vector are specified incrementally.

Minimum Weight Bipartite Perfect Matching.

Theorem 3.10.11. *Let w_{max} be the heaviest edge in a graph G on n vertices. For any c_a, c_v such that $c_a \cdot c_v \geq n^3 w_{max}$ and $c_a \geq n w_{max}$, there is an online $(c_a \log n, c_v \log n)$ scheme for MWBPM with non-negative integer edge weights.*

In particular, Theorem 3.10.11 implies that both the annotation length and space us-

age can be chosen sublinear in the stream length if the number of edges m satisfies $m = n^{3/2+\delta} w_{\max}^{1/2}$ for some $\delta > 0$.

Proof. Upper bound. To prove an upper bound on the value of the minimum weight perfect matching, $\mathcal{P}$ sends the edges in a valid perfect matching $\mathcal{M}$. It is straightforward for $\mathcal{V}$ to store $\mathcal{M}$ and verify that it is perfect matching over n nodes in $O(n)$ space. As in the previous protocol, $\mathcal{V}$ can compute the cost of this matching as an inner product $\mathcal{M} \cdot X$, where X_{ij} is set to w_{ij} if (i, j) is an edge, or $2nw_{\max}$ otherwise. Hence, if M includes edges not present in E, it will have excessively high cost, and can be rejected. $\mathcal{V}$ can check $\mathcal{M}$ is a perfect matching by comparing a fingerprint of the set $\{1, \ldots, n\}$ to that of the (multi)set of nodes incident to an edge in $\mathcal{M}$. If the fingerprints match, then with high probability, each node in n is incident to exactly one edge in $\mathcal{M}$.

Lower bound. A lower bound on the cost of the optimal matching is proven via a feasible solution to the dual linear program. The dual is given by:

$$\begin{aligned} &\text{maximize} \sum_{i \in A} \mathbf{y}_i + \sum_{j \in B} \mathbf{y}_j \\ &\text{subject to } \mathbf{y}_i + \mathbf{y}_j \leq w_{ij} \text{ for all } (i, j) \in E, \end{aligned}$$

where A and B are the two sides of the bipartition of G, and w_{ij} is the cost of edge (i, j). Given a dual solution $\mathbf{y} \in \mathbb{Z}^n$, let $X \in \{0, 1\}^{n^2}$ be the vector with $X_{ij} = \mathbf{y}_i + \mathbf{y}_j - w_{ij}$ if $(i, j) \in E$ and $X_{ij} = \mathbf{y}_i + \mathbf{y}_j - 2d'$ otherwise, where d' is an upper bound on the value of any variable $\mathbf{y}$. We show below $d' = nw_{\max}$ is sufficient. $\mathbf{y}$ is a feasible solution to the dual if and only if all entries of X are less than or equal to zero.

The scheme now proceeds essentially identically to that of Theorem 3.10.10, although here we can only guarantee the existence of a dual-optimal assignment $\mathbf{y}$ with $|\mathbf{y}_i| \leq nw_{\max}$ for all i. This results in increased annotation requirements compared to those of Theorem

3.10.10. We remark that this bound is tight, in that there are graphs for which any dual-optimal solution $\mathbf{y}$ has $|\mathbf{y}_i| = \Omega(nw_{\max})$ for some i; one such example is a simple path on n vertices, with $w_{i,i+1} = w_{\max}$ if i is odd and $w_{i,i+1} = 0$ if i is even.

To argue that there always exists a dual-optimal $\mathbf{y}$ with $|\mathbf{y}_i| \leq nw_{\max}$ for all i, note that Cramer's Rule implies any extreme point solution $\mathbf{y}$ to the dual LP can be written $\mathbf{y}_i = \frac{\det(\widetilde{U}_i)}{\det(U)}$ for some submatrix U of the constraint matrix of the dual, where $\widetilde{U}_i$ obtained from U by replacing the i'th column with the vector $\mathbf{w}$ of edge weights. Furthermore, it follows from the total unimodularity of the dual program that $\det(U) = \pm 1$, and hence for any extreme point $|\mathbf{y}_i| \leq |\det(\widetilde{U}_i)| \leq nw_{\max}$, where the last inequality can be seen by performing cofactor expansion along the i'th column of $\widetilde{U}_i$. Since there is always a dual optimal solution that is an extreme point, there is always an integral dual optimal solution $\mathbf{y}$ for which $|\mathbf{y}_i| \leq nw_{\max}$.

To conclude, we apply the sum-check scheme of Theorem 3.5.2 to the vector X with $X_{ij} = \mathbf{y}_i + \mathbf{y}_j - w_{ij}$ if $(i,j) \in E$ and $X_{ij} = \mathbf{y}_i + \mathbf{y}_j - 2nw_{\max}$ otherwise. $\mathcal{V}$ can construct a *derived* stream defining X just as in Theorem 3.10.10, and the sum-check scheme of Theorem 3.5.2 is applied using a polynomial g such that $g(x) = 1$ for $x \in \{1, \ldots, 2nw_{\max}\}$ and $g(x) = 0$ for $x \in \{-4nw_{\max}, \ldots, 0\}$. g has degree $O(nw_{\max})$, and $\sum_{i,j \in [n]} g(w_{i,j}) = 0$ if and only if $\mathbf{y}$ is feasible for the dual LP. This scheme has hcost $= O(c_a nw_{\max})$ and vcost $= O(c_v)$ for for any $c_a \cdot c_v = \Omega(n^2)$ and $c_a \geq nw_{\max}$.

If no perfect matching exists, Farkas' Lemma implies this can be proven by demonstrating a dual solution $\mathbf{y}$ such that $\mathbf{y}_i + \mathbf{y}_j = 0$ for all $(i,j) \in E$, and $\sum_{i \in A} \mathbf{y}_i + \sum_{j \in B} \mathbf{y}_j > 0$. $\mathcal{V}$ can check this similarly to the scheme of Theorem 3.10.10 when no s-t path exists. □

3.11 Simulating Non-Streaming Algorithms

Next, we give protocols by appealing to known non-streaming algorithms for graph problems. At a high level, we can imagine the prover running an algorithm on the graph, and presenting a "transcript" of operations carried out by the algorithm as the proof to $\mathcal{V}$ that the final result is correct. Equivalently, we can imagine that $\mathcal{V}$ runs the algorithm, but since the data structures are large, they are stored by $\mathcal{P}$, who provides the contents of memory needed for each step. There may be many choices of the algorithm to simulate and the implementation details of the algorithm: our aim is to choose ones that result in smaller annotations.

To make this concrete, consider the case of requiring the graph to be presented in a particular order, such as depth first order. Starting from a given node, the exploration retrieves nodes in order, based on the pattern of edges. Assuming an adjacency list representation, a natural implementation of the search in the traditional model of computation maintains a stack of edges representing the current path being explored. Edges incident on the current node being explored are pushed, and pops occur whenever all nodes connected to the current node have already been visited. $\mathcal{P}$ can allow $\mathcal{V}$ to recreate this exploration by providing at each step the next node to push, or the new head of the stack when a pop occurs, and so on. To ensure the correctness of the protocol, additional checking information can be provided, such as pointers to the location in the stack when a node is visited that has already been encountered.

With care, this idea of "augmenting a transcript" of a traditional algorithm can be made to work on an algorithm-by-algorithm basis. However, while the resulting protocols are lightweight, it rapidly becomes tedious to provide appropriate protocols for other computations based on this idea. Instead, we introduce a more general approach, which argues that any (deterministic) algorithm to solve a given problem can be converted into a protocol

in our model. The running time of the algorithm in the RAM model becomes the size of the proof in our setting.

Our main technical tool is the off-line memory checker of Blum et al. [18], which we use to efficiently verify a sequence of accesses to a large memory. Consider a *memory transcript* of a sequence of read and write operations to this memory (initialized to all zeros). Such a transcript is *valid* if each read of address i returns the last value written to that address. The protocol of Blum et al. requires each read to be accompanied by the timestamp of the last write to that address; and to treat each operation (read or write) as a read of the old value followed by the write of a new value. Then to ensure validity of the transcript, it suffices to check that a fingerprint of all write operations (augmented with timestamps) matches a fingerprint of all read operations (using the provided timestamps), along with some simple local checks on timestamps. Consequently, any valid (timestamp-augmented) transcript is accepted by $\mathcal{V}$, while any invalid transcript is rejected by $\mathcal{V}$ with high probability.

We use this memory checker to obtain the following general simulation result.

Theorem 3.11.1. *Suppose $\mathcal{P}$ is a graph problem possessing a deterministic algorithm A in the random-access memory model that, when given $G = (V, E)$ in adjacency list or adjacency matrix form, outputs $P(G)$ in time $t(m, n)$, where $m = |E|$ and $n = |V|$. Then there is an online $(m \log n + t(m, n) \log n, \log n)$ scheme for $\mathcal{P}$.*

Proof. $\mathcal{P}$ first repeats (the non-zero locations of) a valid adjacency list or matrix representation G, as writes to the memory (which is checked by $\mathcal{V}$); $\mathcal{V}$ uses fingerprints to ensure the edges included in the representation precisely correspond to those that appeared in the stream, and can use local checks to ensure the representation is otherwise valid. This requires $O(m)$ annotation and effectively initializes memory for the subsequent simulation. Thereafter, $\mathcal{P}$ provides a valid augmented transcript T' of the read and write operations performed by algorithm A, which is checked by $\mathcal{V}$ using the memory checking protocol of [18]. $\mathcal{V}$ rejects

if the memory checking protocol rejects T', or if any read or write operation executed in T' does not agree with the prescribed action of A. As only one read or write operation is performed by A in each timestep, the length of T' is $O(t(m,n))$, resulting in an $(m+t(m,n),1)$ protocol for $\mathcal{P}$. □

Although Theorem 3.11.1 only allows the simulation of deterministic algorithms, $\mathcal{P}$ can non-deterministically "guess" an optimal solution S and prove optimality by invoking Theorem 3.11.1 on a (deterministic) algorithm that merely checks whether S is optimal. Unsurprisingly, it is often the case that the best-known algorithms for verifying optimality are more efficient than those finding a solution from scratch (see e.g. the MST protocol below), and this gives the simulation theorem considerable power. We specify some easy corollaries below. These are our first results that improve over the schemes of previous sections for sparse graphs – these schemes allow the product of the annotation length and the space usage to be $\tilde{O}(m)$, while all previous schemes required the product to be $\Omega(n^2)$, even if $m = o(n^2)$. However, the algorithm simulation approach does not allow for *both* space usage and annotation length to be sublinear in m. We will achieve this for a wide range of problems in Chapter 4.

Theorem 3.11.2. *Let G be a graph with n vertices and m edges. There is an online $(m \log n, \log n)$ scheme to find a minimum cost spanning tree in G; online $(m \log n, \log n)$ schemes for connectivity, bipartiteness, and bipartite perfect matching in G; an online $(m \log n + n \log^2 n, \log^2 n)$ scheme to verify single-source shortest paths in G; and online $(m \log n, \log n)$ schemes for max-flow and min-cut in G.*

Proof. We first prove the bound for MST. Given a spanning tree T, there exists a linear-time algorithm A for verifying that T is minimum (see e.g. [69]). Let A' be the linear-time algorithm that, given G and a subset of edges T in adjacency matrix form, first checks that

T is a spanning tree by ensuring $|T| = n - 1$ and T is connected (by using e.g. breadth-first search), and then executes A to ensure T is minimum. We obtain an online $(m \log n, \log n)$ scheme for MST by having $\mathcal{P}$ provide a minimum spanning tree T and using Theorem 3.11.1 to simulate algorithm A'.

The upper bounds for connectivity and bipartiteness follow from the fact the breadth-first such runs in $O(m)$ time, and the upper bound for bipartite perfect matching follows from the fact that there exist simple $O(m)$-time algorithms for checking that a (claimed) matching is valid, and for checking that $|N(S)| \leq |S|$ for a set S whose existence is guaranteed by Hall's Theorem if a bipartite graph G lacks a perfect matching.

The upper bound for single-source shortest path follows from Theorem 3.11.1 and the fact that there exist implementations of Djikstra's algorithm that run in time $m + n \log n$.

The upper bound for max-flow and min-cut follow from the fact that, given a flow and a cut can be specified with $O(m \log n)$ annotation, and the feasibility and value of both the max-flow and the min-cut can be checked by a RAM algorithm in time $O(m)$. The max-flow, min-cut theorem (which is a special case of strong LP duality) implies that if the values of the flow and the cut are equal, then the flow is a max-flow and the cut is a min-cut. □

3.12 Improving the Runtime of the Prover and Experimental Results.

3.12.1 Fast Proofs via Fast Fourier Transforms

In this section, we describe how to drastically speed up $\mathcal{P}$'s computation for essentially all sum-check based schemes developed in this chapter. In these schemes, $\mathcal{P}$ often needs to evaluate a low-degree extension (i.e. the polynomial $\tilde{f}$ within the scheme of Theorem 3.5.2)

at a large number of locations, which can be the bottleneck. Here, we show how to reduce the cost of this step to near linear, via Fast Fourier Transform (FFT) methods.

For concreteness, we describe the approach in the context of the F_2 scheme of Theorem 3.5.3. Our initial experiments with a naive implementation of this scheme that did not use FFTs identified the prover's runtime as the principal bottleneck in the protocol. In this naive implementation, $\mathcal{P}$ required $\Theta(n^{3/2})$ time, and consequently the implementation failed to scale for $n > 10^7$. Here, we show that FFT techniques can dramatically speed up the prover, leading to a protocol that easily scales to streams consisting of billions of items.

We stress that while our experiments focus on the F_2 and matrix-vector multiplication problems, our FFT techniques apply to *all* of the sum-check based schemes in this thesis, including graph problems such as testing connectivity and identifying perfect matchings. Thus, by developing a scalable, practical implementation for F_2 and matrix-vector multiplication, we also achieve big improvements in protocols for a host of important (and seemingly unrelated) problems.

Non-interactive F_2 and matrix-vector multiplication Protocols. We first briefly recall the F_2 scheme from 3.5.3. This construction yields a $(c_a \log n, c_v \log n)$ protocol for positive integers c_a, c_v satisfying $c_a c_v \geq n$, i.e., it allows a tradeoff between the amount of communication and space used by $\mathcal{V}$; for brevity we describe the protocol when $c_a = c_v = \sqrt{n}$.

Assume for simplicity that n is a perfect square. We treat the n dimensional vector f as a $\sqrt{n} \times \sqrt{n}$ array. This implies a two-variate polynomial $\tilde{f}$ over a suitably large finite field $\mathbb{F}_q$, such that

$$\forall (x, y) \in [\sqrt{n}] \times [\sqrt{n}] : \tilde{f}(x, y) = f_{x,y}.$$

To compute F_2, we wish to compute

$$\sum_{x\in[\sqrt{n}],y\in[\sqrt{n}]} f_{x,y}^2 = \sum_{x\in[\sqrt{n}],y\in[\sqrt{n}]} \tilde{f}^2(x,y).$$

The low-degree extension $\tilde{f}$ can also be evaluated at locations outside $[\sqrt{n}] \times [\sqrt{n}]$. In the scheme, the verifier $\mathcal{V}$ picks a random position $r \in \mathbb{F}_q$, and evaluates $\tilde{f}(r,y)$ for every $y \in [\sqrt{n}]$. The proof given by $\mathcal{P}$ is in the form of a degree $2(\sqrt{n}-1)$ polynomial $p(X)$ which is claimed to be $\sum_{y\in[\sqrt{n}]} \tilde{f}(X,y)^2$. $\mathcal{V}$ uses the values of $\tilde{f}(r,y)$ to check that $p(r) = \sum_{y\in[\sqrt{n}]} \tilde{f}(r,y)^2$, and if so accepts $\sum_{x\in[\sqrt{n}]} p(x)$ as the correct answer.

3.12.2 Breaking the Bottleneck

Since $p(X)$ has degree at most $2\sqrt{n}-1$ it is uniquely specified by its values at any $2\sqrt{n}$ locations. We show how $\mathcal{P}$ can quickly evaluate all values in the set

$$S := \{(x, p(x)) : x \in [2\sqrt{n}]\}.$$

Since $p(X) = \sum_{y\in[\sqrt{n}]} \tilde{f}(X,y)^2$, given all values in the set

$$T := \{(x, y, \tilde{f}(x,y)) : x \in [2\sqrt{n}], y \in [\sqrt{n}]\},$$

all values in S can be computed in time linear in n. A naive implementation calculates each value in T independently. This requires $\Theta(n^{1/2})$ time for each value $(x,y) \in ([2\sqrt{n}] \setminus [\sqrt{n}]) \times [\sqrt{n}]$, and hence $\Theta(n^{3/2})$ time overall. Indeed, for each $x \in [2\sqrt{n}] \setminus [\sqrt{n}]$, and each $y \in [\sqrt{n}]$, computing $\tilde{f}(x,y)$ requires computing $S(x)$, where S is the univariate polynomial of degree at most $\sqrt{n}-1$ defined via interpolation as follows: on input $x' \in [\sqrt{n}]$, $S(x')$ is satisfies the identity $S(x') = f(x',y)$. As $S(x')$ is defined via its values on $\sqrt{n}$ points, computing $S(x)$ via polynomial interpolation requires $\Theta(\sqrt{n})$ time.

We show how FFT techniques allow us to calculate T much faster. The task of computing T boils down to multi-point evaluation of the polynomial f. It is known how to perform fast

multi-point evaluation of univariate degree t polynomials in time $O(t \log t)$, and bivariate polynomials in subquadratic time, if the polynomial is specified by its coefficients [81]. However, there is substantial overhead in converting $\tilde{f}$ to a coefficient representation. It is more efficient for us to directly work with and exchange polynomials in an implicit representation, by specifying their values at sufficiently many points.

Representing as a convolution. We are given the values of $\tilde{f}$ at all points located on the $[\sqrt{n}] \times [\sqrt{n}]$ "grid". We leverage this fact to compute T efficiently in nearly linear time by a direct application of the Fast Fourier Transform. For $(x, y) \in [\sqrt{n}] \times [\sqrt{n}]$, $\tilde{f}(x, y)$ is just $f_{x,y}$, which $\mathcal{P}$ can store explicitly while processing the stream. It remains to calculate $(x, y, \tilde{f}(x, y))$ for $\sqrt{n} \leq x < 2\sqrt{n}$. For fixed $y \in [\sqrt{n}]$, we may write $\tilde{f}(X, y)$ explicitly as

$$\tilde{f}(X, y) = \sum_{i \in [\sqrt{n}]} f_{i,y} \chi_i(X),$$

where χ_i is the Lagrange polynomial – the unique polynomial of degree $\sqrt{n}$ such that $\chi_i(i) = 1$, while for all $j \in [\sqrt{n}]$ such that $j \neq i$, it holds that $\chi_i(j) = 0$. Here, the inverse is the multiplicative inverse within the field.

$$\chi_i(j) = \prod_{x \in [\sqrt{n}] \setminus \{i\}} (x - j)(x - i)^{-1}$$

If $j \notin [\sqrt{n}]$, then we may write

$$\tilde{f}(j, y) = \sum_{i \in [\sqrt{n}]} h(j) b_y(i) g(j - i) \tag{3.4}$$

$$\text{where} \quad b_y(i) = f_{i,y} \prod_{x \in [\sqrt{n}] \setminus \{i\}} (x - i)^{-1},$$

$$h(j) = \prod_{k=(j+1-\sqrt{n})}^{j} k,$$

$$\text{and} \quad g(j - i) = (j - i)^{-1}.$$

As a result $\tilde{f}(j, y)$ can be computed as a circular convolution of b_y and g, scaled by $h(j)$. That is, for a fixed y, all values in the set $T_y := \{(x, y, \tilde{f}(x, y)) : x \in [2\sqrt{n}]\}$ can be found by computing the convolution in Equation 3.4, then scaling each entry by the appropriate value of $h(j)$.

Computing the Convolution. We represent b_y and g by vectors of length $2\sqrt{n}$ over a suitable field, and take the Discrete Fourier Transform (DFT) of each. The convolution is the inverse transform of the inner product of the two transforms [73, Chapter 5]. There is some freedom to choose the field over which to perform the transform. We can compute the DFT of $\tilde{f}_y$ and g over the complex field $\mathbb{C}$ using $O(\sqrt{n} \log n)$ arithmetic operations via standard techniques such as the Cooley-Tukey algorithm [36], and simply reduce the final result modulo q, rounded to the nearest integer. Logarithmically many bits of precision past the decimal point suffice to obtain a sufficiently accurate result. Since we compute $O(\sqrt{n})$ such convolutions, we obtain the following result:

Theorem 3.12.1. *The honest prover in the F_2 protocol of 3.5.3 requires $O(n \log n)$ arithmetic operations on numbers of bit-complexity $O(\log n)$.*

In practice, however, working over $\mathbb{C}$ can be slow and requires us to deal with precision issues. Since the original data resides in some finite field $\mathbb{F}_q$, and can be represented as fixed-precision integers, it is preferable to also compute the DFT over the same field. Here, we exploit the fact that in designing our protocol, we can choose to work over *any* sufficiently large finite field $\mathbb{F}_q$.

There are two issues to address: we need that there *exists* a DFT for sequences of length $2\sqrt{n}$ (or thereabouts) in $\mathbb{F}_q$, and further that this DFT has a corresponding (*fast*) Fourier Transform algorithm. We can resolve both issues with the *Prime Factor Algorithm* (PFA) for the DFT in $\mathbb{F}_q$ [26]. The "textbook" Cooley-Turkey FFT algorithm operates on sequences

whose length is a power of two. Instead, the PFA works on sequences of length $N = N_1 \times N_2 \times \ldots \times N_k$, where the N_i's are pairwise coprime. The time cost of the transform is $O((\sum_i N_i)N)$. The algorithm is typically applied over the complex numbers, but also applies over $\mathbb{F}_q$: it works by breaking the large DFT up into a sequence of smaller DFTs, each of size N_i for some i. These base DFTs for sequences of length N_i exist for $\mathbb{F}_q$ whenever there exists a primitive N_i'th root of unity in F_q. This is the case whenever N_i is a divisor of $q-1$. So we are in good shape so long as $q-1$ has many distinct prime factors.

Here, we use our freedom to fix q, and choose $q = 2^{61} - 1$.[1] Notice that

$$2^{61} - 2 = 2 \times 3^2 \times 5^2 \times 7 \times 11 \times 13 \times 31 \times 41 \times 61 \times 151 \times 331 \times 1321,$$

and so there are many such divisors N_i to choose from when working over $\mathbb{F}_q$. If $2\sqrt{n}$ is not equal to a factor of $q-1$, we can simply pad the vectors f_y and g such that their lengths are factors of $2^{61} - 2$. Since $2^{61} - 2$ has many *small* factors, we never have to use too much padding: we calculated that we never need to pad any sequence of length $100 \leq N \leq 10^9$ (good for n up to 10^{18}) by more than 16% of its length. This is better than the Cooley-Tukey method, where padding can double the length of the sequence.

As an example, we can work with the length $N = 2 \times 5 \times 7 \times 9 \times 11 \times 13 = 90090$, sufficient for inputs of size $n = (N/2)^2$, which is over 10^9. The cost scales as $(2+5+7+9+11+13)N = 47N$. Therefore, the PFA approach offers a substantial improvement over naive convolution in $\mathbb{F}_q$, which takes time $\Theta(N^2)$.

Parallelization. This protocol is highly amenable to parallelization. Observe that $\mathcal{P}$ performs $O(\sqrt{n})$ independent convolutions of each of length $O(\sqrt{n})$ (one for each column y of the matrix $f_{x,y}$), followed by computing $\sum_y f_{x,y}^2$ for each row x of the result. The convolutions can be done in parallel, and once complete, the sum of squares of each row can also

[1] Arithmetic in this field can also be done quickly, see Section 7.4.1.

be parallelized. This protocol also possesses a simple two-round MapReduce protocol. In the first round, we assign each column y of the matrix $f_{x,y}$ a unique key, and have each reducer perform the convolution for the corresponding column. In the second round, we assign each row x a unique key, and have each reducer compute $\sum_y f_{x,y}^2$ for its row x.

3.12.3 Implications

As we experimentally demonstrate, the results of this section make practical the fundamental building block for all the sum-check based protocols of this thesis. Indeed, by combining Theorem 3.12.1 with schemes developed earlier in this chapter we obtain the following immediate corollaries. We stress that while we invoke Theorem 3.12.1 to obtain asymptotic results, in practice it is desirable to work over a finite field and use the Prime Factor Algorithm as described above. All runtimes below are stated assuming that addition multiplication on $O(\log n)$ bit numbers can be done in $O(1)$ machine operations.

Corollary 3.12.2. *1. (Extending Theorem 3.5.3) For any $c_a \cdot c_v \geq n$, there is an online $(c_a \log n, c_v \log n)$ scheme for computing the inner product and Hamming distance of two n-dimensional vectors, where $\mathcal{V}$ runs in time $O(n)$ and $\mathcal{P}$ runs in time $O(n \log n)$.*

2. (Extending Theorem 3.9.1) For any $c_a \cdot c_v \geq c$, there is an online $(b \cdot c_a \log n, c_v \log n)$ scheme for $b \times c$ integer matrix-vector multiplication, where $\mathcal{V}$ runs in time $O(bc)$ and $\mathcal{P}$ runs in time $O(bc \log n)$.

3. (Extending Theorem 3.10.4) For any $c_a \cdot c_v \geq n^3$, there is an online $(c_a \log n, c_v \log n)$ scheme for counting the number of triangles in a graph with n vertices and m edges, where $\mathcal{V}$ runs in time $O(mn)$ and $\mathcal{P}$ runs in time $O(n^3 \log n)$.

4. (Extending Theorem 3.10.7) For any $c_a \cdot c_v \geq n^2$, $c_a \geq n$, there is an online $(c_a \log n, c_v \log n)$ scheme for graph connectivity on graphs with n nodes and m edges,

where $\mathcal{P}$ runs in time $O(n^2 \log n)$ and $\mathcal{V}$ runs in time $O(m)$.

5. *(Extending Theorem 3.10.5) For any $c_a \cdot c_v \geq n^2$, $c_a \geq n$, there is an online $(c_a \log n, c_v \log n)$ scheme for bipartite perfect matching on graphs with n nodes and m edges, where $\mathcal{V}$ runs in time $O(m)$ and $\mathcal{P}$ runs in time $O(t(n)+n^2 \log n)$, where $t(n)$ is the time required to find a perfect matching if one exists, or to find a counter-example (via Hall's Theorem) otherwise.*

3.12.4 Experiments

We now describe our experiments with the schemes developed in this chapter. To focus our study, we concentrate on two problems in particular: F_2 and matrix-vector multiplication. As both of these schemes are based on sum-check techniques, the results are representative for all sum-check based schemes developed in this chapter.

We evaluated the schemes on a multi-core machine with 64-bit AMD Opteron processors and 32 GB of memory available. Our scalability results here use a single core, though we also briefly describe results for parallel operation. The large amount of memory allowed us to experiment with universes of size several billion, with the prover able to store the full data in memory.

Summary of Experimental Results

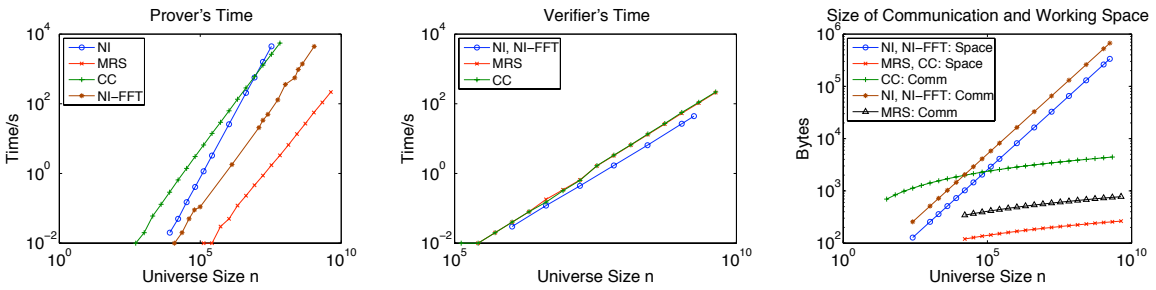
The high-level conclusion we draw from our experiments in this section is similar to the conclusions we will draw for the interactive protocols we develop later in Chapters 5-8. Specifically, the costs of our protocols are attractive from the verifier's perspective – the space and communication costs of our schemes, as well as the verifier's runtime, are all reasonable. The bottleneck in our protocols is the runtime of the prover. We show that while a naive implementation of our prover fails to scale to large inputs, the FFT techniques of Section

3.12.1 yield non-interactive protocols that easily scale to streams with billions of updates, improving over a naive implementation by three orders of magnitude. Detailed results follow.

F_2: We compare our non-interactive F_2 scheme developed in this chapter to two interactive protocols developed later in this thesis. The first multi-round protocol is a specialized interactive protocol for F_2 given in Chapter 6 (MRS). The second multi-round protocol is based on the general-purpose circuit-checking approach of Chapter 7 based on the GKR protocol (CC). We compared these multi-round protocols against two implementations of our non-interactive F_2 scheme from this chapter: the first involves a naive implementation of the prover (NI), while the second prover implementation utilizes our FFT techniques developed in Section 3.12.1 (NI-FFT). Figures 3.3(b) and 3.3(c) illustrate the verifier's time and space costs for all four protocols, while Figure 3.3(a) illustrates the prover's runtime for these protocols. Note that in the case of NI and NI-FFT, the verifier behaves *identically*: the prover computes the same messages in both cases, but more quickly using FFT.

The main observation from Figures 3.3(b) and 3.3(c) is that the verifier's costs are extremely low for all four protocols. $\mathcal{V}$ processed over 20 million items/s across all stream lengths for all protocols. The space usage and communication cost for both interactive protocols (CC and MRS) is less than 1 kilobyte across all stream lengths tested, while the space usage for the non-interactive case is much larger but still reasonable (comfortably under a megabyte even for stream lengths in excess of 1 billion).

Figure 3.3(a) shows a clear separation between the four methods in $\mathcal{P}$'s effort in generating the proof. For large streams, it is clear that NI is not scalable, with $\mathcal{P}$'s runtime growing like $n^{3/2}$; this implementation failed to process streams larger than about 40 million updates. In contrast, the FFT-based implementation of the non-interactive protocol processed between $350,000$ and $750,000$ items per second for all tested values of n, even for values of n well into the billions. Thus, the FFT techniques of Section 3.12.1 speed up $\mathcal{P}$'s computation



(a) $\mathcal{P}$'s time for F_2 protocols (b) Verifier's time for F_2 protocols (c) Space and communication costs for F_2

Figure 3.3: Experimental Results for both multi-round and non-interactive F_2 protocols.

by several orders of magnitude compared to the naive implementation, and allowed the protocol to easily scale to streams with billions of items. As mentioned in Section 3.12.1, a wide variety of more complicated protocols use this protocol as a subroutine, and therefore these non-interactive techniques are as powerful as they are general.

For the multi-round protocols, circuit checking (CC) eventually outpaces NI, and scales linearly: the CC prover processed about 20,000 items per second across all stream lengths. Finally, the MRS prover processed 20-21 million items per second.

Matrix-Vector Multiplication: Figure 3.4 shows the behavior of our FFT-based implementation of the non-interactive protocol for matrix-vector multiplication given in Theorem 3.9.1. Recall that for an $n \times n$ matrix, Theorem 3.9.1 yielded an online (nc_a, c_v) for any positive integers c_a, c_v satisfying $c_a c_v \geq n$. In the following, we write $c_a = n^{\alpha}$. A convenient feature of this protocol is that when $\alpha = 0$, the honest prover's message consists simply of the correct answer i.e. the vector Ax. Consequently, we obtain an (n, n) protocol for which the prover can handle enormous throughputs: 30-50 million items/second as evidenced in Figure 3.4(b). In outsourcing settings where one can tolerate space usage $O(n)$ for the verifier, this protocol is truly ideal, as the prover need do nothing more than solve the problem, and the verifier's

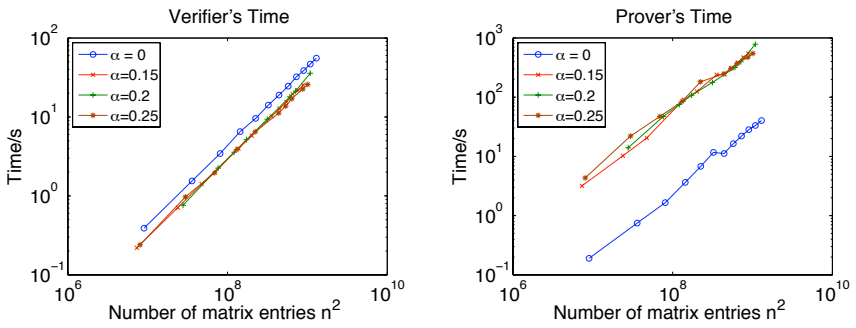
Table 3.1: Experimental results for the matrix-vector multiplication scheme of Theorem 3.9.1.

α	Space	Annotation Length	$\mathcal{P}$ time (s)	$\mathcal{V}$ time (s)
0	78.1 KB	78.1 KB	1.6	4.3
.15	19.9 KB	468.8 KB	33.9	3.0
.20	12.8 KB	937.5 KB	58.9	2.8
.25	7.8 KB	1.52 MB	61.5	2.6

Results are for matrices of size 10,000 $\times$ 10,000 (763 MBs of data). Here, α is a parameter that controls the tradeoff between space and annotation length.

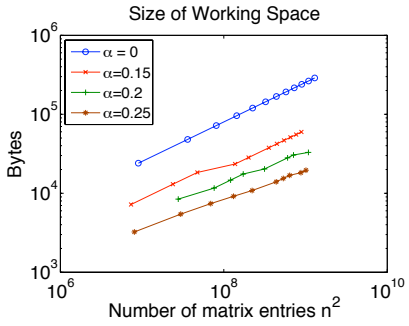
computation consists only of maintaining n fingerprints. That is, this (n, n) protocol allows the user to obtain a strong security guarantee on the integrity of the query, almost for free. Note that for this problem, the size of the input is $O(n^2)$ for an $n \times n$ matrix, so $O(n)$ space at the verifier is still much smaller than the full input size.

The behavior becomes more interesting when we set $\alpha > 0$—in this case, in addition to providing the correct answer, the prover has to do non-trivial computation to prove correctness. Because higher values of α mean less space but more communication (see Figure 3.4(c)), setting $\alpha > 0$ may be needed when the verifier is severely space-limited. It may also be necessary when the matrix is very wide: in full generality the protocol has communication and space cost $(mn^{\alpha}, n^{1-\alpha})$ for an $m \times n$ matrix. We show how different costs vary as a function of α: $\mathcal{V}$'s time to process the input (Figure 3.4(a)), $\mathcal{P}$'s time (Figure 3.4(b)), the communication cost (Figure 3.4(d)), and the space used by $\mathcal{V}$ (Figure 3.4(c)). Across all values of α, $\mathcal{P}$ can process in excess of 1 million items per second using our FFT techniques. The verifier runs over the stream slightly faster for higher values of α, because $\mathcal{V}$ maintains fewer fingerprints for larger α's. When $\alpha = 0$, $\mathcal{V}$ processed about 20 million items per second, and when $\alpha = .25$, $\mathcal{V}$ processed in excess of 30 million items per second. For concreteness, Table 3.1 displays the costs of the protocol when run on matrices of size 10,000 $\times$ 10,000.

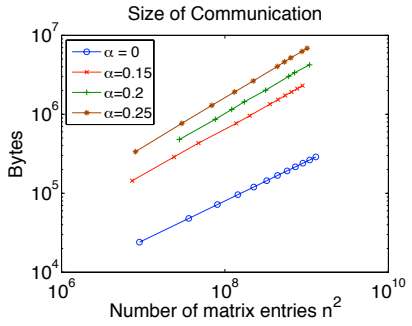


(a) Verifier's time

(b) Prover's time



(c) Space cost



(d) Communication cost

Figure 3.4: Experiments on non-interactive matrix-vector multiplication protocols.

Parallel Implementations

The prover's computations in all of the non-interactive protocols studied here are highly parallelizable, as noted previously. Indeed, using just three OpenMP[2] statements, we were able to achieve more than a 7-fold speedup over the sequential implementation of the FFT protocol, by using all 8 cores of the multi-core machine our experiments were run on. Consequently, with 8 processors, the ratio between the speed of the MR and NI-FFT protocols for F_2 drops from 20-60 to 3-8. In theory, the interactive F_2 protocol is just as easy to parallelize as the non-interactive protocol; however, we did not find this to be the case in practice. The prover's computations in the multi-round protocol are so light-weight (as evidenced by its

[2] `http://www.openmp.org`

very high throughput) that memory access forms the principle bottleneck.

3.13 Related Work

On MA Communication Complexity. Babai, Frankl, and Simon [10] introduced the communication complexity analogs of many fundamental Turing Machine complexity classes. Most relevant to us, they introduced the model of Merlin-Arthur (MA) communication complexity. Aaronson and Wigderson [3] gave a beautiful MA communication protocol for DISJ using algebraic techniques analogous to those in the famous sum-check protocol of Lund et al. [77]. Theorem 3.5.2 is based on Aaronson and Wigderson's MA protocol. Their protocol is nearly optimal, essentially matching a lower bound of Klauck [70]. Theorem 3.5.4, in turn, generalizes the lower bound of Klauck to multiparty MA communication complexity. This generalization is critical to showing that higher frequency moments are hard even to approximate in the annotations model. Aaronson [2] studied the hardness of the INDEX problem in a restricted version of the MA communication model, as well as in a quantum variant of this model. His classical model is similar to the online MA communication model that we consider.

Other existing results have primarily focused on quantum or multiparty variants of MA-communication complexity. Klauck [71] proved an $\Omega(n^{1/3})$ lower bound on the quantum Merlin-Arthur communication complexity of two-party set disjointness. Raz and Shpilka [83] gave a problem whose quantum communication complexity is exponentially smaller than its MA-communication complexity. Gavinsky and Sherstov [54] proved a separation between co-NP and MA in the setting of number-on-the-forehead multiparty communication complexity, and their lower bound was further refined by Sherstov in [94, 95].

As can be seen from the above summary, only a handful of researchers have studied

MA communication complexity, probably because the model was considered somewhat esoteric [71]. However, the relatively recent work of Aaronson and Wigderson [3] gave a new motivation for studying the communication complexity analog of any Turing Machine complexity class, and MA in particular: they showed that if a communication complexity class $\mathcal{C}$ is not contained within another communication complexity class $\mathcal{D}$, then a certain class of proof techniques (called *algebraizing* techniques) will not suffice to show that the Turing Machine analog of $\mathcal{C}$ is contained within the Turing Machine analog of $\mathcal{D}$. We view the connection between MA communication complexity and annotated data stream models as another significant motivation for studying MA communication complexity.

Proof Systems for Space-Bounded Verifiers. Early work on interactive proof systems studied the power of space-bounded verifiers (the survey by Condon [35] provides a comprehensive overview), but many of the protocols developed in this line of work require the verifier to store the input, and therefore do not work in the annotations model, where the verifier must be streaming. An exception is work by Lipton [76], who relied on using fingerprinting techniques to allow a log-space streaming verifier to ensure that the prover correctly plays back the transcript of an algorithm in an appropriate computational model. This approach does not lead to protocols with sublinear annotation length. More recently, Das Sarma et al. studied the "best order streaming model," which can be thought of as the annotations model where the annotation is restricted to be a permutation of the input [43].

Other Work. There has also been more applied work which implicitly defines annotation schemes. Tucker et al. [102] considered *stream punctuations*, which, in our terminology, are simple prescient annotations, indicating facts such as that there are no more tuples relevant to timestamp t in the remainder of the stream. Yi et al. [106], in their work on stream outsourcing, study the problem of verifying that a claimed "grouping" corresponds to the input

data. They solve exact and approximate versions of the problem by using a linear amount of annotation. Lastly, the work of Li et al. [74] on *proof infused streams* answers various selection and aggregation queries (e.g. range sum) over sliding windows with logarithmic space and linear annotation. However, a critical difference is that Li et al. require that the helper and verifier agree on a one-way (cryptographic) hash function, for which it is assumed the helper cannot find collisions. Our results are in a stronger model without this assumption.

Independent work subsequent to this chapter. Two recent works have considered variants of the annotated data stream model. Klauck and Prakash [72] study a restricted version of the annotations model in which the annotation must essentially end by the final stream update. Gur and Raz [61] give protocols for a class of problems in a model that is similar to annotated data streams, but more powerful in that the verifier has access to both public and private randomness. This corresponds to the AMA communication and streaming models. We briefly gave a scheme for approximate F_2 in this model, and return to it again in Section 4.7.2 of Chapter 4.

3.14 Open Problems

Up to logarithmic factors, we have resolved the cost of annotated data streaming protocols for a number of fundamental streaming problems, including exact computation of INDEX, frequency moments, bipartite perfect matching, and shortest s-t path in small-diameter graphs. However, we have also presented several non-trivial annotated data streaming protocols whose optimality we cannot yet establish. In particular, we highlight the following open problems.

- Theorem 3.5.4 proves that any (c_a, c_v) scheme (online or prescient) that approximates kth frequency moment, F_k, up to a constant factor requires $c_a c_v = \Omega(n^{1-5/k})$. It is not

possible to prove a lower bound better than $c_a c_v = \tilde{\Omega}(n^{1-2/k})$ since there exist standard (Merlin-less) streaming algorithms for computing F_k that use $\tilde{O}(n^{1-2/k})$ space [7, 21]. (We clarify that it is not known how to achieve a (c_a, c_v)-scheme for all $c_a \cdot c_v = n^{1-2/k}$, only for $c_a = O(1)$, $c_v = \tilde{\Theta}(n^{1-2/k})$). It would be interesting to close the gap between the $\tilde{O}(n^{1-2/k})$ upper bound and the $\Omega(n^{1-5/k})$ lower bound.

- Assume $m = O(n)$. Determine whether there is a (c_a, c_v) scheme (online or prescient) for exactly computing F_0 for which c_a and c_v are both $O(n^{2/3-\delta})$ for some constant $\delta > 0$. A negative answer to this question would prove the optimality of Theorem 3.7.1. Note Gur and Raz have recently given a $(\sqrt{n} \cdot \operatorname{polylog} n, \sqrt{n} \cdot \operatorname{polylog} n)$ AMA scheme for this problem, but it remains open whether there exists a scheme of similar costs that does not use public randomness.

- Determine whether there is an (online or standard) MA communication protocol of cost $O(n^{3/2-\delta})$ for some constant $\delta > 0$ (cf. [1]). A negative answer to this question would prove the optimality of Theorem 3.10.4 and resolve the (online or standard) MA communication complexity of counting triangles.

- Determine whether there is an (online or standard) MA communication protocol for connectivity or bipartiteness of cost $n^{1-\delta}$ for some constant $\delta > 0$. A negative answer to this question would resolve the (online or standard) MA communication complexity of connectivity and bipartiteness. Such a protocol would be interesting even if it only works under the promise that the graph is *sparse*, that is, that the graph contains $m = O(n)$ edges.

Chapter 4

Annotations for Sparse Data Streams

4.1 Introduction

Chapter 3 provided considerable understanding of the power of annotated data streams, revealing a surprisingly rich theory. A number of fundamental problems that possess no non-trivial algorithms in the standard streaming model do have efficient schemes when the data stream may be annotated by a prover. By exploiting powerful algebraic techniques originally developed in the literature on interactive proofs, the results of Chapter 3 achieved essentially optimal tradeoffs between annotation size and the space usage of the verifier for problems ranging from frequency moments to bipartite perfect matching.

However, these schemes are only optimal for streams for which the total number of updates is large relative to the size of the *data universe*. In contrast, many real-world data sets are *sparse*: for example, many real-world graphs, though large, contain much fewer than the maximum possible number $\binom{n}{2}$ of edges, and IP traffic streams contain much fewer than the total number of possible IP addresses, 2^{128} in IPv6.

In this chapter, we give the first schemes in the annotations model that allow both the annotation size and space usage to be *sublinear in the number of items with non-zero*

frequency in the data stream, rather than the size of the data universe n. On the negative side, we also give a new lower bound that for the first time rules out smooth tradeoffs between annotation size and space usage for a specific problem. The latter result is derived from a new lower bound in the Merlin–Arthur (MA) communication model that may be of independent interest.

4.2 Overview of Results and Techniques

We give an informal overview of our results and the techniques we use to obtain them. Recall that n denotes the size of the data universe and m the number of items with non-zero frequency at the end of a data stream (we refer to m as the "sparsity" of the stream).

Section 4.3 contains our first set of results. We begin by precisely characterizing the complexity of the sparse PointQuery problem—a natural variant of the well-known index problem from communication complexity—giving an $(x \log n, y \log n)$-scheme whenever $xy \geq m$. We give similar upper bounds for the related problems Selection and Heavy-Hitters. We also prove a lower bound showing that *any* (c_a, c_v)-scheme for these problems requires $c_a c_v = \Omega(m \log(n/m))$, improving by a $\log(n/m)$ factor over lower bounds that follow from prior work on "dense" streams. By a dense stream we mean one where n is not much larger than m. This $\log(n/m)$ factor may seem minor, but we show that it has the following striking consequence: the (very) sparse index problem—where Alice's n-bit string has Hamming weight $O(\log n)$—has one-way randomized communication complexity that is within a logarithmic factor of its online MA communication complexity. This implies that no non-trivial tradeoffs between Merlin's and Alice's message sizes are possible for this problem. That is, it is not possible for Alice's message to be more than a logarithmic factor smaller than in the best Merlin-less protocol, unless Merlin's message is very long (at least as long,

up to a logarithmic factor, as the cost of the best Merlin-less protocol). To our knowledge this is the first problem that provably exhibits this phenomenon.

Our scheme for sparse PointQuery relies on universe reduction: the prover succinctly describes a mapping $h : [n] \to [r]$ that maps the input stream, which is defined over the huge data universe $[n]$, down to a derived stream defined over a smaller universe $[r]$. By design, if the prover is honest and the mapping h does not cause "too many collisions," then the answer on the original stream can be determined from the answer on the derived stream. We then efficiently apply known schemes for dense streams to the derived stream.

For our lower bound in Section 4.3, we give a novel reduction from the standard (dense) index problem to sparse index that is tailored to the MA communication model. We then apply known lower bounds for dense index. Our technique also gives what is to our knowledge the first polynomial separation between the online MA and AMA communication complexities of a specific (and natural) problem.

For clarity, the remainder of this overview omits factors logarithmic in n and m when stating the costs of schemes. Though these factors are important for Section 4.3 (the consequences of our lower bound being most significant when $n = m^{\omega(1)}$), we anticipate that in practice n and m will usually be polynomially related.

Sections 4.4 and 4.5 contain the most interesting and technically involved results of the chapter, namely, efficient schemes for size-m-set-disjointness (henceforth, m-disj) and kth Frequency Moments (henceforth, F_k). The schemes here are substantially more complex than those in Section 4.3 and represent the main technical contributions of this chapter.

Section 4.4 gives $(m^{2/3}, m^{2/3})$-schemes for both problems, but the schemes rely on "prescient" annotation, i.e., annotation provided at the start of the stream that depends on the stream itself. The even more complex schemes of Section 4.5 eliminate the need for prescient annotation and also achieve much more general tradeoffs between annotation length and

space usage. Specifically, Section 4.5 gives $(mc_v^{-1/2}, c_v)$-schemes for m-DISJ and F_k for any $c_v < m$. Notice that one recovers the costs achieved in Section 4.4 by setting $c_v = m^{2/3}$, and thus the results of Section 4.5 almost subsume those of Section 4.4.[1] However, Section 4.4 allows us to present a self-contained and easier proof of a weaker set of results that paves the way for the results of Section 4.5.

These schemes are the first for these problems that allow both the annotation length and space usage to be sublinear in m. At a very high level, there are three interlocking ideas that allow us to achieve this.

1. The first idea is a careful application of universe reduction. We are able to use a simple version of this idea to derive the upper bound for the POINTQUERY problem in Section 4.3, but in the case of DISJ and F_k the universe-reduction mapping $h : [n] \to [r]$ specified by the prover is more complicated, requiring refinement in the form of the additional ideas described below.

2. The second idea is addressed to ensuring that the prover performed the universe-reduction step in an honest manner, in the sense that the answer on the original stream can indeed be determined from the answer on the derived stream. The difficulty of ensuring $\mathcal{P}$ is honest varies depending on the structure of the problem at hand. For F_k, the verifier has to make sure that the universe-reduction mapping h is injective on the items appearing in the data stream. This requires developing an efficient way for $\mathcal{V}$ to detect collisions under h, even though $\mathcal{V}$ does not have the space to store all of the values $h(x_i)$ for stream updates x_i. For m-DISJ, a notion weaker than injectiveness is sufficient.

[1] Two advantages of the schemes of Section 4.4 are that the schemes of Section 4.4 satisfy perfect completeness, while those of Section 4.5 do not, and that the schemes of Section 4.4 are more efficient by polylogarithmic factors.

3. The third idea pertains to allowing $\mathcal{P}$ to specify the universe-reduction mapping h *online*. That is, for many problems it would be much simpler if $\mathcal{P}$ could determine the mapping h in advance i.e., if $\mathcal{P}$ could be prescient and send h to $\mathcal{V}$ at the start of the stream so that $\mathcal{V}$ can determine the derived "mapped-down" stream on her own (this is the approach taken in Section 4.4). When $\mathcal{P}$ must specify h in an online fashion, additional insight is required. At a high level, our approach is to have $\mathcal{P}$ specify a "guess" as to the right hash function at the beginning of the steam and retroactively modify the hash function after the stream has been observed. The challenging aspect of this approach is to ensure that $\mathcal{P}$'s retroactive modification of the hash function is consistent with the observed data stream, even though $\mathcal{V}$ cannot refer back to the stream to enforce this.

   We exploit similar ideas to allow $\mathcal{V}$ to avoid storing the universe-reduction mapping h herself; this is the key to achieving general tradeoffs between annotation length and space usage in Section 4.5. In some schemes, storing this mapping h would be the bottleneck in $\mathcal{V}$'s space usage. We show how $\mathcal{V}$ can store only a *partial* description of h, and ask $\mathcal{P}$ to fill in the remainder of the description at the end of the stream.

Section 4.6 exploits all of these results, applying them to several graph problems, including counting triangles and demonstrating a perfect matching. Our schemes have costs that depend on the number of edges in the graph, rather than the total number of possible edges and demonstrate that the ideas underlying our m-DISJ and F_k schemes are broadly applicable. We state clearly how our schemes improve over prior work throughout.

Section 4.7 considers the general turnstile stream update model, which allows items to have negative frequencies (i.e., provides correct answers for problems such as frequency moments and inner products even when the entries of the "frequency vector" may be negative).

These negative frequencies potentially break the "collision detection" sub-protocol used in the previous sections, so we show how to exploit a source of public randomness to allow these protocols to be carried out. Essentially, the public randomness specifies a remapping of the input, so that the prover is highly unlikely to be able to use negative frequencies to "hide" collisions. Because the protocols of Section 4.7 require public randomness, they work in the AMA communication and streaming models, as opposed to the MA models in which all of our other protocols operate.

4.3 Point Queries, Selection, and Heavy Hitters

4.3.1 Upper Bounds

Our first result is an efficient online annotation scheme for the PointQuery problem, a generalization of the familiar index problem.

Definition 4.3.1. In the PointQuery problem, the data stream $\mathbf{x}$ consists of a sequence of updates of the form (i, δ), followed by an index ι, and the goal is to determine the frequency $f_\iota(\mathbf{x}) = \sum_{(j_k,\delta_k)\in\mathbf{x}:j_k=\iota} \delta_k$.

A prescient $(\log n, \log n)$-scheme for this problem is trivial as $\mathcal{P}$ can just tell $\mathcal{V}$ the index ι at the start of the stream, and $\mathcal{V}$ can track the frequency of ι while observing the stream. The vcost can be improved to $O(\log m)$ if $\mathcal{V}$ retains a hashed value of ι and tracks the frequency of matching updates. The first scheme has perfect completeness, while the second has completeness error polynomially small in m.

The costs of the scheme below are in terms of the stream sparsity m and not the stream length N or the stream footprint M; this is significant if $m \ll M$, which is the case, e.g., for the well-known straggler and set-reconciliation problems that have been studied in traditional

streaming and communication models [47, 79]. Our lower bound in Theorem 4.3.8 shows our scheme is essentially optimal for moderate universe sizes, i.e., when the universe size n is sub-exponential in the sparsity m.

Theorem 4.3.2. *For any pair of positive integers c_a, c_v such that $c_a \cdot c_v \geq m$, there is an online $(c_a \log n, c_v \log n)$-scheme in the non-strict turnstile update model for the* PointQuery *problem with imperfect completeness. Any online (c_a, c_v) scheme with $c_a \geq \log n$ for this problem requires $c_a \cdot c_v = \Omega(m \log(n/m))$.*

Proof. $\mathcal{V}$ requires $\mathcal{P}$ to specify at the start of the stream a hash function $h : [n] \to [c_v]$. $\mathcal{V}$ requires h to have description length $O(c_a)$, rejecting if this is not the case. We define the derived streams $\mathbf{x}^j \in \mathcal{U}^N$ based on h: we set $\mathbf{x}^j_k = x_k$ iff $h(x_k) = j$, and 0 otherwise. Intuitively, the hash function h partitions the stream updates in $\mathbf{x}$ into c_v disjoint buckets, and the vector $\mathbf{x}^j$ describes the contents of the jth bucket. $\mathcal{V}$ maintains fingerprints over a field of size poly(n) of each of the c_v different $\mathbf{x}^j$ vectors.

At the end of the stream, given the desired index ι, $\mathcal{P}$ provides a description of the (claimed) frequency vector in the $h(\iota)$th derived stream, $f(\mathbf{x}^{h(\iota)})$. $\mathcal{V}$ computes a fingerprint of the claimed frequency vector and compares it to the fingerprint she computed from the data stream, accepting if and only if the fingerprints match. Since each $\mathbf{x}^j$ is sparse in expectation, the cost of this description can be low: provided h does not map more than $O(c_a)$ items with non-zero frequency to $h(\iota)$, $\mathcal{P}$ can just specify the item id and frequency of the items with non-zero frequency in $f(\mathbf{x}^{h(\iota)})$. In this case, the annotation size is just $O(c_a \log n)$. If $\mathcal{P}$ exceeds this amount of annotation, $\mathcal{V}$ will halt and reject (output $\perp$).

Soundness follows from the fingerprinting guarantee: if $\mathcal{P}$ does not honestly provide $\mathbf{x}^{h(\iota)}$, $\mathcal{V}$'s fingerprint of $\mathbf{x}^{h(\iota)}$ computed from the data stream will not match her fingerprint of the claimed vector of frequencies.

To show (imperfect) completeness, we study the probability that the output of an hon-

est prover is rejected. This happens only if $m(\mathbf{x}^{h(\iota)})$, the number of non-zero entries in $\mathbf{x}^{h(\iota)}$, is much larger than its expectation. By the pairwise independence of h, $\mathbb{E}[m(\mathbf{x}^{h(\iota)})] = m(\mathbf{x})/c_v = c_a$. Thus, by Markov's inequality, $\Pr[m(\mathbf{x}^{h(\iota)}) > 10c_a] < 1/10$. So by specifying a hash function chosen at random from a pairwise independent hash family and then honestly playing back the items that map to the same region as ι, $\mathcal{P}$ can convince $\mathcal{V}$ to accept with probability 9/10.

Notice that $\mathcal{V}$ does not need to *enforce* that $\mathcal{P}$ picks the hash function h at random from a pairwise-wise independent hash family, as $\mathcal{P}$ has no incentive not to pick the hash functions in this way. That is, since $\mathcal{V}$ will reject if too many items map to the same region as ι, it is *sufficient* for $\mathcal{P}$ to pick h at random from a pairwise independent hash family in order to convince $\mathcal{V}$ to accept with constant probability. But it is equally acceptable if $\mathcal{P}$ wants to pick h another way; if he does so, $\mathcal{P}$ just risks that $\mathcal{V}$ will reject with a higher probability.

The lower bound follows from Theorem 4.3.8, which we prove in Section 4.3.2. □

The scheme of Theorem 4.3.2 yields nearly optimal schemes for the HeavyHitters and Selection problems, described below. Table 4.1 summarizes these results and compares to prior work.

Selection. For the reader's convenience, we reproduce the definition of the Selection problem original defined in Definition 3.3.1. Our definition assumes all frequencies $f_i := \sum_{(j_k,\delta_k):j_k=i} \delta_k$ are non-negative, so this definition is valid only for the strict turnstile update model.

Definition 4.3.3. The Selection problem is defined in terms of the quantity $N' = \sum_{i\in[n]} f_i$, the sum of all the frequencies. Given a desired rank $\rho \in [N']$, output an item j from the stream $\mathbf{x} = \langle (j_1, \delta_1), \ldots, (j_N, \delta_N)\rangle$, such that $\sum_{(j_k,\delta_k):j_k<j} \delta_k < \rho$ and $\sum_{(j_k,\delta_k):j_k>j} \delta_k \geq N' - \rho$.

Table 4.1: Comparison among PointQuery, Selection, and ϕ-HeavyHitters schemes.

Problem	Scheme Costs	Completeness	Prescience	Source
PointQuery	$(\log n, \log n)$	Perfect	Prescient	Ch. 3
PointQuery	$(m \log n, \log n)$	Perfect	Online	Thm 3.11.1
PointQuery	$(c_a \log n, c_v \log n)$: $c_a c_v \geq n$	Perfect	Online	Thm. 3.3.2
PointQuery	$(c_a \log n, c_v \log n)$: $c_a c_v \geq m$	Imperfect	Online	Thm 4.3.2
Selection	$(\log n, \log n)$	Perfect	Prescient	Ch. 3
Selection	$(c_a \log n, c_v \log n)$: $c_a c_v \geq n$	Perfect	Online	Thm 3.3.3
Selection	$(m \log n, \log n)$	Perfect	Online	Thm 3.11.1
Selection	$(c_a \log^2 n, c_v \log n)$: $c_a c_v \geq m \log n$	Imperfect	Online	Cor 4.3.4
ϕ-HeavyHitters	$(\frac{\log n}{\phi}, \frac{\log n}{\phi})$	Perfect	Prescient	Thm 3.6.1
ϕ-HeavyHitters	$(\frac{\log^2 n}{\phi} + c_a \log n, c_v \log n)$: $c_a c_v \geq n$	Perfect	Online	Thm 3.6.1
ϕ-HeavyHitters	$(m \log n, \log n)$	Perfect	Online	Thm 3.11.1
ϕ-HeavyHitters	$(\frac{\log n}{\phi} c_a, c_v \log n)$: $c_a c_v \geq m \log n$	Imperfect	Online	Cor 4.3.6
ϕ-HeavyHitters	$(\frac{\log n}{\phi} + c_a \log n, c_v \log n)$: $c_a c_v \geq m \log n$	Imperfect	Online	Cor 4.5.6

For all three problems, the schemes of this chapter are the first online schemes to achieve both annotation and space usage sublinear in the stream sparsity m when $m \ll \sqrt{n}$, and strictly improve over the online MA communication cost of prior schemes whenever $m = o(n)$. For brevity, we omit factors of $\log_{c_v}(m)$ from the statement of costs of the ϕ-HeavyHitters scheme due to Corollary 4.5.6.

Corollary 4.3.4. *For any pair of positive integers c_a, c_v such that $c_a c_v \geq m \log n$, there is an online $(c_a \log^2 n, c_v \log n)$-scheme for* Selection *in the strict turnstile update model.*

The corollary follows from a standard observation to reduce Selection to answering prefix sum queries and hence to multiple instances of the PointQuery problem. $\mathcal{V}$ treats each stream update (i, δ) in the stream $\mathbf{x}$ as an update to $O(\log n)$ dyadic ranges, where a dyadic range is a range of the form $[j2^k, (j+1)2^k - 1]$ for some j and k. Thus, we can view the set of dyadic range updates implied by $\mathbf{x}$ as a derived stream of sparsity $m \log n$. Notice we are using the fact that this transformation from the original stream of sparsity m

results in a derived stream of sparsity at most $m \log n$; a different derived stream was used in Theorem 3.3.3 to address the SELECTION problem, but the sparsity of that derived stream could be substantially larger than the sparsity of the original stream.

For any i, the quantity $T_i := \sum_{(j,\delta):j \leq i} \delta$ can be written as the sum of the counts of $O(\log n)$ dyadic ranges. Thus, at the end of the stream $\mathcal{P}$ can convince $\mathcal{V}$ that item i has the desired T_i value by running $\log n$ POINTQUERY protocols as in Theorem 4.3.2 in parallel on the derived stream of sparsity $m \log n$. The verifier's space usage is the same as for a single POINTQUERY instance on this stream: $\mathcal{V}$ fingerprints each of the derived streams $\mathbf{x}^j$ defined in the proof of Theorem 4.3.2 and uses these fingerprints in all $\log n$ instances of the POINTQUERY scheme. The annotation length is $\log n$ times larger than that required for a single POINTQUERY instance because $\mathcal{P}$ may have to describe the frequency vectors of up to $\log n$ derived streams.

Thus, we get an online $(c_a \log^2 n, c_v \log n)$-scheme as long as $c_a c_v = \Omega(m \log n)$.

Frequent Items. We also reproduce the definition of he ϕ-HEAVYHITTERS problem for the reader's convenience. Here, we assume all frequencies $f_i := \sum_{(j_k,\delta_k):j_k=i} \delta_k$ are non-negative, and so this definition is only valid for the strict turnstile update model.

Definition 4.3.5. The ϕ-HEAVYHITTERS problem (also known as frequent items) is to list those items i such that $f_i \geq \phi N'$ where $N' = \sum_{i \in [n]} f_i$, i.e., to list the items i whose frequency of occurrence exceeds a ϕ fraction of the total count N'.

We give a preliminary result for the ϕ-HEAVYHITTERS problem in Corollary 4.3.6 below. We give a substantially improved scheme in Section 4.5 using the ideas underlying our online scheme for frequency moments.

Corollary 4.3.6. *For all positive integers c_a, c_v such that $c_a c_v \geq m \log n$, there is an online $(c_a \phi^{-1} \log^2 n, c_v \log n)$-scheme for solving ϕ-HEAVYHITTERS in the strict turnstile update*

model.

Corollary 4.3.6 follows from the following analysis. Theorem 3.4.1 describes how to reduce ϕ-HeavyHitters to demonstrating the frequencies of $O(\phi^{-1} \log n)$ items in a derived stream. Moreover, the derived stream has sparsity $O(m \log n)$ if the original stream has sparsity m. We use the PointQuery scheme of Theorem 4.3.2. As in Corollary 4.3.4, the annotation length blows up by a factor $\phi^{-1} \log n$ relative to a single PointQuery, but the space usage of $\mathcal{V}$ can remain the same as in a single PointQuery instance. Hence, we obtain an online $(c_a \phi^{-1} \log^2 n, c_v \log n)$-scheme for any $c_a c_v \geq m \log n$.

4.3.2 Lower Bound

In this section, we prove a new lower bound on the online MA communication complexity of the (m, n)-Sparse index problem.

Definition 4.3.7. In the (m, n)-Sparse index problem, Alice is given a vector $x \in \{0, 1\}^n$ of Hamming weight at most m, and Bob is given an index ι. Their goal is to output the value x_ι.

We prove our lower bound by reducing the (dense) index problem (i.e., the (m, n)-Sparse index problem with $m = \Theta(n)$) in the MA communication model to the (m, n)-Sparse index problem for small m. The idea is to replace Alice's dense input with a sparser input over a bigger universe and then take advantage of our sparse PointQuery protocol. A lower bound on the online MA communication complexity of the dense index problem was proven in Theorem 3.3.2. There, it was shown that any online MA communication protocol $\mathcal{Q}$ requires $\text{hcost}(\mathcal{Q}) \, \text{vcost}(\mathcal{Q}) \geq n$. Combining this with our reduction of the dense index problem to the sparse version, we conclude that any protocol for sparse index must be costly.

Theorem 4.3.8. *Any online MA communication protocol $\mathcal{Q}$ for the (m,n)-Sparse* INDEX *problem for which* $\text{hcost}(\mathcal{Q}) \geq \log n$ *must have* $\text{hcost}(\mathcal{Q})\,\text{vcost}(\mathcal{Q}) = \Omega(m \log(n/m))$.

Proof. Assume we have an online MA communication protocol $\mathcal{Q}$ for (m,n)-sparse INDEX. We describe how to use this online MA protocol for the sparse INDEX problem to design one for the dense INDEX problem on vectors of length $n' = m\log(n/m)$.

Let $k = \log(n/m)$. Given an input x to the dense INDEX problem, Alice partitions x into n'/k blocks of length k, and constructs a 0-1 vector y of Hamming weight n'/k over the universe $\{0,1\}^{(n'/k)\cdot 2^k} = \{0,1\}^n$ as follows. She replaces each block B_i with a 1-sparse vector $v_i \in \{0,1\}^{2^k}$, where each entry of v_i corresponds to one of the 2^k possible values of block B_i. That is, if block B_i of x equals the binary representation of the number $j \in [2^k]$, then Alice replaces block B_i with the vector $e_j \in \{0,1\}^{2^k}$, where e_j denotes the vector with a 1 in coordinate j and 0s elsewhere.

Alice now has an $n'/k = m$-sparse derived input y over the universe $\{0,1\}^n$. Merlin looks at Bob's input to identify the index ι of the dense vector x in which Bob is interested. Merlin then tells Bob the index ℓ such that $\ell = 2^k(i-1) + j$, where B_i is the block in which ι is located, and block B_i of Alice's input x equals the binary representation of the number $j \in [2^k]$. Notice that Merlin can specify ℓ using $\log n$ bits. If Bob is convinced that $y_\ell = 1$, then Bob can deduce the value of *all* the bits in block B_i of the original dense vector x, and in particular, the value of x_ι.

The parties then run the assumed online MA protocol for (m,n)-Sparse INDEX. The total hcost of this protocol is $\text{hcost}(\mathcal{Q}) + \log n = O(\text{hcost}(\mathcal{Q}))$, and the total vcost is $\text{vcost}(\mathcal{Q})$. Thus, by Theorem 3.3.2, $\text{hcost}(\mathcal{Q})\,\text{vcost}(\mathcal{Q}) = \Omega(n') = \Omega(m\log(n/m))$ as claimed. □

Theorem 4.3.8 should be contrasted with the following well-known upper bound.

Theorem 4.3.9. *Assume $n < m^m$. Then the one-way randomized communication complexity*

of the (m, n)*-Sparse* INDEX *Problem is* $O(m \log m)$.

Proof. Alice chooses a hash function $h : [n] \to [m^3]$ at random from a pairwise independent family and uses h to perform "universe reduction". That is, she sends h along with the set S of m values $\{h(j) : x_j = 1\}$. Notice h can be specified with $O(\log n) = O(m \log m)$ bits, and S can be specified with $O(m \log m)$ bits. Bob outputs 1 if $h(\iota) \in S$, and 0 otherwise. The correctness of the protocol follows from the pairwise independence property of h: if $x_\iota = 0$, then with high probability ι will not collide under h with any j such that $x_j = 1$. The total cost of this protocol is $O(m \log m)$. □

4.3.3 Implications of the Lower Bound

Our lower bound in Theorem 4.3.8 has interesting consequences when it is combined with the upper bound in Theorem 4.3.9. Consider in particular the (m, n)-Sparse INDEX Problem, where $n = 2^m$. Theorem 4.3.9 implies that the one-way randomized communication complexity of this problem is $O(m \log m)$; that is, without any need of Merlin, Alice and Bob can solve the problem with $O(m \log m)$ communication.

Meanwhile, Theorem 4.3.8 implies that even if Merlin's message to Bob has length $\Omega(\log n) = \Omega(m)$, Alice's message to Bob must have length $\Omega(m \log(n/m)/m) = \Omega(m)$. Indeed, Theorem 4.3.8 shows that for any protocol $\mathcal{Q}$, if $\text{hcost}(\mathcal{Q}) \geq \log n = m$, then we must have $\text{hcost}(\mathcal{Q})\,\text{vcost}(\mathcal{Q}) = \Omega(m \log(n/m)) = \Omega(m^2)$. In particular, this means that if $\text{hcost}(\mathcal{Q}) = m$, $\text{vcost}(\mathcal{Q})$ must be $\Omega(m)$. This trivially implies that for any protocol $\mathcal{Q}$ with $\text{hcost}(\mathcal{Q})$ *less* than m, $\text{vcost}(\mathcal{Q})$ must still be $\Omega(m)$; otherwise we could achieve a protocol with $\text{hcost}(\mathcal{Q}) = m$ and $\text{vcost}(\mathcal{Q}) = o(m)$ simply by running $\mathcal{Q}$ and adding in extraneous bits to the proof to bring the proof length up to m.

Consequently, the online MA communication complexity of this problem is at least $\Omega(m)$, which is at most a logarithmic factor smaller than the one-way randomized communication

complexity. To our knowledge, this is the first problem that provably exhibits this behavior. Specifically, this rules out smooth tradeoffs between annotation size and space usage in any annotated streaming protocol for the $(m, 2^m)$-Sparse INDEX Problem: there is an online $(1, m \log m)$-scheme for this problem, but in order to reduce the space cost even to $o(m)$, the annotation length must be $\Omega(m)$.

Corollary 4.3.10. *The one-way randomized communication complexity of the $(m, 2^m)$-Sparse* INDEX *Problem is $O(m \log m)$. The online Merlin-Arthur communication complexity is $\Omega(m)$.*

Other Sparse Problems. A number of lower bounds in Chapter 3 are proved via reductions from INDEX that preserve stream length up to logarithmic factors. This holds for SELECTION and HEAVYHITTERS, as well as for the problem of determining the existence of a triangle in a graph. For all such problems, the lower bound of Theorem 4.3.8 implies corresponding new lower bounds for sparse streams, i.e., streams for which $m = o(n)$. We omit the details for brevity.

Separating Online MA and AMA Communication Complexity. Another implication of Theorem 4.3.8 is a polynomial separation between online MA communication complexity and online AMA communication complexity. Indeed, there is an online AMA protocol of cost $\tilde{O}(\sqrt{m})$ for the $(m, 2^{\sqrt{m}})$-Sparse INDEX Problem, where the $\tilde{O}$ notation hides factors polylogarithmic in m: the first message, which consists of public random coins, is used to specify a hash function $h : [n] \to [m^3]$ from a pairwise independent hash family; this message has length $O(\log n) = O(\sqrt{m})$. With high probability, h is injective on the set $\{j : x_j = 1\}$. The parties then run the online MA communication protocol of Theorem 4.3.2 on the inputs $h(\mathbf{x})$ and $h(\iota)$ and output the result. The total cost of this protocol is $\tilde{O}(\sqrt{m})$ as claimed.

Meanwhile, the lower bound of Theorem 4.3.8 implies that the online MA communication complexity of this problem is $\Omega(m^{3/4})$. Indeed, if we have a protocol $\mathcal{Q}$ with $\text{hcost}(\mathcal{Q}) = m^{3/4} > \log n$, Theorem 4.3.8 implies that $\text{hcost}(\mathcal{Q})\,\text{vcost}(\mathcal{Q}) = \Omega(m \log(n/m)) = \Omega(m^{3/2})$, and hence $\text{vcost}(\mathcal{Q}) > m^{3/4}$.

To our knowledge, this is the first such separation between online AMA and online MA communication complexity (we remark that polynomial separations between online MA and MAMA communication complexity were already known for problems including INDEX and DISJ [3, 30]). Indeed, all previous lower bound methods that apply to online MA communication complexity, such as the proof of Theorem 3.3.2 and the methods of Klauck and Prakash [72], in fact yield equivalent AMA lower bounds. At a high level, the reason is that these methods work via round reduction – they remove the need for Merlin's message. They therefore turn any online MA protocol for a function F into an online "A" protocol for F, which is really just a one-way randomized protocol without a prover, allowing one to invoke a known lower bound on the one-way randomized communication complexity of F. Similarly, they turn an online AMA protocol for F into an online AA protocol, which is also just a one-way randomized protocol for F.

The reason Theorem 4.3.8 is capable of separating online AMA from MA communication complexity is that the reduction in the proof of Theorem 4.3.8 turns an online MA protocol for the (m, n)-Sparse INDEX Problem into an online MA protocol for the (dense) INDEX Problem with related costs. However, the natural variant of the reduction applied to an online AMA protocol for the (m, n)-Sparse INDEX Problem yields an online MAMA protocol for the dense INDEX Problem, *not* an online AMA protocol. And the dense INDEX Problem has an online MAMA protocol that is polynomially more efficient than any online AMA protocol (see e.g., [3, 42]).

Table 4.2: Comparison of m-DISJ schemes.

Scheme Costs	Completeness	Prescience	Source
$((m \log m)^{2/3},\ (m \log m)^{2/3})$: $m = \Omega(\log n)$	Perfect	Prescient	Thm 4.4.2
$(c_a \log n,\ c_v \log n)$: $c_a c_v \geq n$	Perfect	Online	Thm 3.5.2
$(m \log n,\ \log n)$	Perfect	Online	Thm 3.11.1
$(c_a \log n \log_{c_v} m,\ c_v \log n \log_v m)$: $c_a = m c_v^{-1/2}$	Imperfect	Online	Thm 4.5.1

The m-DISJ schemes of this chapter are the first to achieve annotation length and space usage that are both sublinear in m for $m \ll \sqrt{n}$, and strictly improve over the MA communication cost (online or prescient) of prior schemes whenever $m = o(n)$.

4.4 Prescient Schemes for Sparse Disjointness and Frequency Moments

In this section and the next, we describe schemes for the m-Disjointness (m-DISJ) and Frequency Moment (F_k) problems. These schemes contain the main ideas of the chapter.

4.4.1 A Prescient Scheme for Sparse Disjointness

An important special case of the communication problem DISJ is when Alice's and Bob's input sets are promised to be small, i.e., have size at most $m \ll n$. These should be thought of as *sparse* instances. The sparsity parameter m has typically been denoted by the letter k in the communication complexity literature, and the problem has typically been referred to as k-DISJ rather than m-DISJ; we use m rather than k for consistency with our notation in the rest of the thesis (where m denotes the sparsity of a data stream).

Among the original motivations for studying this variant is its relation to the clique-vs.-independent-set problem introduced by Yannakakis [105] to study linear programming

formulations for combinatorial optimization problems. More recent motivations include connections to property testing [17]. A clever protocol of Håstad and Wigderson [63] gives an optimal $O(m)$ communication protocol for m-DISJ, improving upon the trivial $O(m \log n)$ and the easy $O(m \log m)$ bounds. This protocol requires considerable interaction between Alice and Bob, a feature that turns out to be necessary. Recent results of Buhrman et al. [24] and Dasgupta et al. [44] give tight $\Theta(m \log m)$ bounds for m-DISJ in the one-way model. Very recently, Brody et al. [23] and Sağlam and Tardos [86] have given tight rounds-vs.-communication tradeoffs for m-DISJ.

Here we obtain the first nontrivial bounds for m-DISJ in the annotated streams model and thus also in the online MA communication model.

Definition 4.4.1. In the m-DISJ problem, the data stream specifies two multi-sets $S, T \subseteq [n]$, with $\|S\|_0, \|T\|_0 \leq m$, where $\|S\|_0$ denotes the number of distinct items in S. An update of the form $((0, i), \delta)$ is interpreted as an insertion of δ copies of item i into set S, and an update of the form $((1, i), \delta)$ is interpreted as an insertion of δ copies of item i into T. The goal is to determine whether or not S and T are disjoint.

Notice Definition 4.4.1 allows S and T to be multi-sets, but assumes the strict turnstile update model, where the frequency of each item is non-negative (recall that the turnstile model allows for updates to contains negative δ values, thereby modeling deletions, but the strict turnstile model requires all frequencies to be non-negative at the end of the stream).

Theorem 4.4.2. *Assume $m > \log n$. There is a prescient $((m \log m)^{2/3}, (m \log m)^{2/3})$-scheme for m-DISJ with perfect completeness in the strict turnstile update model. In particular, the MA-communication complexity of m-DISJ is $O((m \log m)^{2/3})$. Any prescient (c_a, c_v) protocol requires $c_a c_v = \Omega(m)$.*

Proof. Obviously if S and T are not disjoint, the prescient prover can provide an item

$i \in S \cap T$ at the start of the stream and the verifier can check that i indeed appears in both S and T. The total space usage and annotation length is just $O(\log n)$ in this case.

Suppose now that S and T are disjoint. We first recall that a $(\sqrt{n}\log n, \sqrt{n}\log n)$-scheme for DISJ follows from Theorem 3.5.2, with $f^{(1)}$ and $f^{(2)}$ set to the indicator vectors of S and T respectively, and g equal to the product function. We refer to this as the dense DISJ scheme, because its cost does not improve if $|S|$ and $|T|$ are both $o(n)$.

Our prescient scheme for m-DISJ works as follows. At the start of the stream, the prover describes a hash function $h : [n] \to [r]$, for some smaller universe $[r]$, with the property that h is injective on $S \cup T$. We will write $h(S)$ to denote the result of applying h to every member of S. The parties can now run the dense DISJ scheme whereby $\mathcal{P}$ convinces $\mathcal{V}$ that $h(S)$ and $h(T)$ are disjoint. Given the existence of an injective function h, perfect completeness follows from the fact that if S and T are disjoint, so are $h(S)$ and $h(T)$, combined with the perfect completeness of the dense DISJ scheme. Soundness follows from the fact that if $i \in S \cap T$, then $h(i) \in h(S) \cap h(T)$ i.e., if S and T are not disjoint, then the same holds trivially for $h(S)$ and $h(T)$.

The dense DISJ scheme run on $h(S)$ and $h(T)$ requires annotation length and space usage $O(\sqrt{r}\log r)$. We now show that, for a suitable choice of r, $\mathcal{P}$'s description of h is also limited to $O(\sqrt{r}\log r)$ communication, balancing out the cost of the rest of the scheme.

A family of functions $\mathcal{F} \subseteq [r]^{[n]}$ is said to be κ-perfect if, for all $S \subseteq [n]$ with $|S| \leq \kappa$, there exists a function $h \in \mathcal{F}$ that is injective when restricted to S. Fredman and Komlós [52] have shown that for all $n \geq r \geq \kappa$, there exists a κ-perfect family $\mathcal{F}$, with

$$|\mathcal{F}| \;\leq\; (1 + o(1)) \left(\frac{\kappa \log n}{-\log(1 - t(r,\kappa))} \right) ,$$

where

$$t(r,\kappa) \;:=\; \prod_{j=1}^{\kappa-1} \left(1 - \frac{j}{r}\right) .$$

For $r \geq 2\kappa$, we can use the crude approximation

$$-\log(1 - t(r,\kappa)) \;\geq\; t(r,\kappa) \;\geq\; \left(1 - \frac{\kappa}{r}\right)^{\kappa} \;\geq\; e^{-2\kappa^2/r}$$

to obtain the bound $|\mathcal{F}| = O(\kappa e^{2\kappa^2/r} \log n)$, which implies

$$\log |\mathcal{F}| \;=\; O(\kappa^2/r)\,,$$

for $\kappa^2/r = \Omega(\log \kappa)$ and $\kappa = \Omega(\log n)$.

Let us pick a family $\mathcal{F}$ that is $(2m)$-perfect. Once $\mathcal{P}$ and $\mathcal{V}$ agree upon such a family $\mathcal{F}$, the prover, upon seeing the input sets S and T, can pick $h \in \mathcal{F}$ that is injective on $S \cup T$. Describing h requires $O(m^2/r)$ bits; $\mathcal{P}$ sends this to $\mathcal{V}$ before the stream is seen, and $\mathcal{V}$ stores it while observing the stream in order to run the dense DISJ scheme on $h(S)$ and $h(T)$. To balance out this communication with the $O(\sqrt{r}\log r)$ cost of running the dense DISJ scheme on $h(S)$ and $h(T)$, we choose r so that

$$\frac{m^2}{r} \;=\; \Theta(\sqrt{r}\log r)\,.$$

This is achieved by setting $r = m^{4/3}/\log^{2/3} m$. The resulting upper bound is that both the annotation length and verifier's space usage are $O\left((m \log m)^{2/3}\right)$.

The lower bound follows from the lower bound for "dense" DISJ proved in Theorem 3.5.5. □

4.4.2 A Prescient Scheme for Frequency Moments

We now present prescient schemes for the kth Frequency Moment problem, F_k, defined in Definition 3.5.1.

The idea behind the scheme, as in the case of m-DISJ, is that $\mathcal{P}$ is supposed to specify a "hash function" h to reduce the universe size in a way that does not introduce false collisions. However, for F_k it is essential that $\mathcal{V}$ ensure h is truly injective on the items appearing in

Table 4.3: Comparison of F_k schemes.

Scheme Costs	Completeness	Prescience	Source
$(k^2 c_a \log n,\ kc_v \log n)$: $c_a c_v \geq n$	Perfect	Online	Thm 3.5.2
$(m \log n,\ \log n)$	Perfect	Online	Thm 3.11.1
$(k^2 m^{2/3} \log n,\ km^{2/3} \log n)$	Perfect	Prescient	Thm 4.4.3
$(k^2 m\, c_v^{-1/2} \log n \log_{c_v} m,\ kc_v \log n \log_{c_v} m)$: $c_v > 1$	Imperfect	Online	Thm 4.5.1

The F_k schemes of this chapter are the first to achieve annotation length and space usage that are both sublinear in m for $m \ll \sqrt{n}$, and strictly improve over the MA communication cost of prior protocols (online or prescient) whenever $m = o(n)$.

the data stream. This is in contrast to m-DISJ, where a weaker notion than injectiveness was sufficient to guarantee soundness. The fundamental difference between the two problems is that for m-DISJ, collisions only "hurt the prover's claim" that the two sets are disjoint, whereas for F_k the prover could try to use collisions to convince the verifier that the answer to the query is higher or lower than the true answer.

Theorem 4.4.3. *There is a prescient $(k^2 m^{2/3} \log n, km^{2/3} \log n)$-scheme for computing F_k over a data stream of sparsity m in the strict turnstile update model. This scheme has perfect completeness. Any prescient (c_a, c_v) protocol requires $c_a c_v = \Omega(m)$.*

Proof. The idea is to have the prover specify for the verifier a perfect hash function $h : [n] \to [r]$, where r is to be determined later, i.e., $\mathcal{P}$ specifies a hash function h such that for all $x \neq y$ appearing in at least one update in the data stream, $h(x) \neq h(y)$. The verifier stores the description of h, and while observing the stream runs the dense F_k scheme of Theorem 3.5.2 on the derived stream in which each update (i, δ) is replaced with the update $(h(i), \delta)$.

As discussed above, it is essential that $\mathcal{V}$ ensure h is injective on the set of items that have non-zero frequency, as otherwise $\mathcal{P}$ could try to introduce collisions to try to trick the

verifier. To deal with this, we introduce a mechanism by which $\mathcal{V}$ can "detect" collisions.

Definition 4.4.4. Define the problem INJECTION as follows. We observe a stream of tuples $t_i = ((x_i, b_i), \delta_i)$. Each t_i indicates that δ_i copies of item x_i are placed in bucket $b_i \in [r]$. We allow δ_i to be negative, modeling deletions, and refer to the quantity $f_{(j,b)} = \sum_{i:(x_i,b_i)=(j,b)} \delta_i$ as the *count* of pair (j, b). We assume the strict turnstile model, so that for all pairs (j, b) we have $f_{(j,b)} \geq 0$.

We say that the stream is an injection if for every two pairs (j, b) and (j', b) with positive counts, it holds that $j = j'$. Define the output as 1 if the stream defines an injection, and 0 otherwise.

Lemma 4.4.5. *For any $c_a c_v \geq r$, there is an online $(c_a \log r, c_v \log r)$-scheme for determining whether a stream in the strict turnstile model is an injection.*

Proof. Say that bucket b is *pure* if there is at most one $j \in [n]$ such that $f_{(j,b)} > 0$. The stream defines an injection if and only if every bucket b is pure.

Notice that a bucket b is pure if and only if the variance of the item identifiers mapping to the bucket with positive count is zero. Intuitively, our scheme will compute the sum of the these variances across all buckets b; this sum will be zero if and only if the stream defines an injection. Details follow.

Define three r-dimensional vectors u, v, w as follows:

$$u_b = \sum_{j \in [n]} f_{(j,b)},$$

$$v_b = \sum_{j \in [n]} f_{(j,b)} j,$$

$$w_b = \sum_{j \in [n]} f_{(j,b)} j^2.$$

It is easy to see that if bucket b is pure then $v_b^2 = u_b \cdot w_b$. Moreover, if bucket b is impure then $v_b^2 < u_b w_b$; this holds by the Cauchy-Schwarz inequality applied to the n-dimensional

vectors whose jth entries are $\sqrt{f_{(j,b)}}$ and $\sqrt{f_{(j,b)}} \cdot j$ respectively (the strict inequality holds because for an impure bucket b, the vector given by $\sqrt{f_{(j,b)}} \cdot j$ is not a scalar multiple of the vector given by $\sqrt{f_{(j,b)}}$). Here, we are exploiting the assumption that $f_{(j,b)} \geq 0$ for all pairs (i, b), as this allows us to conclude that all $\sqrt{f_{(j,b)}}$ values are real numbers.

It follows that $\sum_{b \in [r]} v_b^2 = \sum_{b \in [r]} u_b \cdot w_b$ if and only if the stream defined an injection. Both quantities can be computed using the "dense" scheme of Theorem 3.5.2. Notice that each update $t_i = ((x_i, b_i), \delta_i)$ contributes independently to each of the vectors u, v, and w, and hence it is possible for $\mathcal{V}$ to run the scheme of Theorem 3.5.2 on these vectors as required. This yields an online $(c_a \log r, c_v \log r)$-scheme for the injection problem for any $c_a c_v \geq r$ as claimed. □

Returning to our F_k scheme, $\mathcal{P}$ specifies a hash function h claimed to be one-to-one on the set of items that appear in one or more updates of the stream $\mathbf{x}$. $\mathcal{V}$ verifies that h is injective using the scheme of Lemma 4.4.5. If this claim is true, then $F_k(\mathbf{x}) = F_k(h(\mathbf{x}))$, the frequency moment of the mapped-down stream, and $\mathcal{P}$ can prove this by running the scheme of [30, Theorem 4.1] on the derived stream $h(\mathbf{x})$.

Perfect completeness follows from $\mathcal{P}$'s ability to find a perfect hash function just as in Theorem 4.4.2. Soundness follows from the soundness of the INJECTION scheme of Lemma 4.4.5, in addition to the soundness property of the F_k scheme of [30, Theorem 4.1].

To analyze the costs, note that by using the hash family of Fredman and Komlós [52], the annotation length and space cost due to specifying and storing the hash function h is $O(m^2 \log n/r)$. The annotation length and space cost of the dense F_k scheme of Theorem 3.5.2 are $O(k^2 c_a \log r)$ and $O(k c_v \log r)$ for any $c_a c_v \geq r$. The annotation length and space cost of the INJECTION scheme can be set to $O(c_a \log r)$ and $O(c_v \log r)$ respectively. Setting $r = m^{4/3}$ and $c_a = c_v = m^{2/3}$ yields the desired costs. □

4.5 An Online Scheme for Frequency Moments

We now give an online version of the F_k scheme of Theorem 4.4.3. A simple modification of this scheme yields the scheme for m-DISJ with analogous costs as claimed in Row 4 of Table 4.2. In addition to avoiding the use of prescience, our online scheme avoids requiring $\mathcal{V}$ to explicitly store the hash function sent by $\mathcal{P}$, allowing us to achieve a much wider range of tradeoffs between annotation size and space usage relative to Theorems 4.4.2 and 4.4.3.

Theorem 4.5.1. *For any $c_v > 1$, there is an online $(k^2 m c_v^{-1/2} \log n \log_{c_v} m, k c_v \log n \log_{c_v} m)$-scheme for F_k in the strict turnstile model for a stream of sparsity m over a universe of size n. Any online (c_a, c_v)-scheme for this problem with $c_a \geq \log n$ requires $c_a c_v = \Omega(m \log(n/m))$.*

Notice that the annotation length is less than $m \log n$ for any $c_v = m^{\Omega(1)}$, and therefore this protocol is not subsumed by the simple "sparse" scheme (second row of Table 4.3) in which $\mathcal{P}$ just replays the entire stream in a sorted order, and $\mathcal{V}$ checks this is done correctly using fingerprints. Notice also that the product of the space usage and annotation length is $k^3 m c_v^{1/2} \log^2 n \log_{c_v}^2 m$, which is in $o(n)$ for many interesting parameter settings. This improves upon the dense sum-check scheme (first row of Table 4.3) in such cases.

4.5.1 An Overview of the Scheme

In order to achieve an online scheme, we examine how to construct perfect hash functions such as those used in the prescient F_k scheme of Theorem 4.4.3. Let S be the set of m items with non-zero frequency at the end of the stream: we want the hash function to be one-to-one on S. Choose a hash function h at random from pairwise independent hash family mapping $[n]$ to $[r]$, for r to be specified later – this requires just $O(\log n)$ bits to specify. We only expect $O(m^2/r)$ pairs to collide under h, which means that with constant probability there will be $O(m^2/r)$ collisions if h is chosen as specified. The final hash function h^* is specified by

writing down h (which takes only $O(\log n)$ bits), followed by the items involved in a collision and some special locations for them. The total (expected) bit length to specify this hash function is $O(m^2 \log(n)/r)$.

In our online F_k scheme, $\mathcal{P}$ will send such an h at the start of the stream. Notice h does not depend on the stream itself – it is just a random pairwise independent hash function – so $\mathcal{P}$ is not using prescience. $\mathcal{P}$ also has no incentive not to choose h at random from a pairwise independent hash family, since the only purpose of choosing h in this manner is to minimize the number of collisions under h. If $\mathcal{P}$ chooses h in a different way, $\mathcal{P}$ simply risks that there are too many collisions under h, causing $\mathcal{V}$ to reject.

Now while $\mathcal{V}$ observes the stream, she runs the online sum-check scheme for F_k given in Theorem 3.5.2 on the mapped-down universe of size r, using h as the mapping-down function. At the end of the stream, $\mathcal{P}$ is asked to retroactively specify a hash function h^* that is one-to-one on S as follows. $\mathcal{P}$ provides a list L_0 of all items in S that were involved in a collision under h, accompanied by their frequencies. Assuming that these items and their frequencies are honestly specified by $\mathcal{P}$, $\mathcal{V}$ can compute their contribution to F_k and *remove them* from the stream. By design, h^* is then (claimed to be) injective on the remaining items. $\mathcal{V}$ can confirm this tentatively using the Injection scheme of Lemma 4.4.5.

The remainder of the scheme is devoted to making the correctness a certainty by ensuring that the items in L_0 and their frequencies are as claimed (we stress that while our exposition of the scheme is modular, all parts of the scheme are executed in parallel, with no communication ever occurring from $\mathcal{V}$ to $\mathcal{P}$). A naive approach to checking the frequencies of the items in L_0 would be to run $|L_0|$ independent PointQuery schemes, one for each item in L; however there are too many items in L_0 for this to be cost-effective. Instead, we check all of the frequencies as a batch, with a (sub-)scheme whose cost is roughly equal to that of a single Injection query.

This (sub-)scheme can be understood as proceeding in stages, with each stage i using a different pairwise independent hash function h_i to map down the full original input. Say that an item j is *isolated* by h_i if j is not involved in a collision under h_i with any other item with non-zero frequency in the original data stream $\mathbf{x}$. The goal of stage i is to isolate a large fraction of items that were not isolated by any previous stage.

A key technical insight is that at each stage i, it is possible for $\mathcal{V}$ to "ignore" all items that are not isolated at that stage. This enables $\mathcal{V}$ to check that the frequencies of all items that *are* isolated at stage i are as claimed. We bound the number of stages that are required to isolate all items if $\mathcal{P}$ behaves as prescribed – if $\mathcal{P}$ reaches an excessive number of stages, then $\mathcal{V}$ will simply reject.

4.5.2 Details of the Scheme

Proof of Theorem 4.5.1: Let $r = mc_v^{1/2}$. $\mathcal{P}$ sends a hash function $h : [n] \to [r]$ at the start of the stream, claimed to be chosen at random from a pairwise independent hash family. While observing the stream, $\mathcal{V}$ runs the dense online sum-check scheme for F_k given in Theorem 3.5.2 on the mapped-down universe $[r]$. Let S be the set of items with non-zero frequency at the end of the stream. After the stream is observed, $\mathcal{P}$ is asked to provide a list L_0 of all items with nonzero frequency that were involved in a collision, followed by a claimed frequency f_i^* for each $i \in L_0$.

Assuming that these items and their frequencies are honestly specified in L_0 by $\mathcal{P}$, $\mathcal{V}$ can compute their contribution $C_0 = \sum_{i \in L_0} f_i^*$ to F_k and then remove them from the stream by processing updates $U = \{(i, -f_i^*) : i \in L_0\}$ within the dense F_k scheme. h is injective on the remaining items. $\mathcal{V}$ can confirm this using the INJECTION scheme of Lemma 4.4.5 (conditioned on the assumed correctness of L_0). Thus the dense F_k scheme will output $C_1 = \sum_{i \notin L_0} f_i^k$. Assuming all of $\mathcal{V}$'s checks within the dense F_k scheme pass, $\mathcal{V}$ outputs

$C_0 + C_1$ as the answer.

The remainder of the scheme is directed towards determining that the frequency of items in L_0 are correctly reported. We abstract this goal as the following problem.

Definition 4.5.2. Define the ℓ-MULTIINDEX problem as follows. Consider a data stream $\mathbf{x} \circ L_0$, where $\circ$ denotes concatenation. $\mathbf{x}$ is a usual data stream in the strict turnstile model, while L_0 is a list of ℓ pairs (i, f_i^*). Let f be the frequency vector of $\mathbf{x}$. The desired output is 1 if $f_i = f_i^*$ for all $i \in L_0$, and 0 otherwise.

We defer our solution to the ℓ-MULTIINDEX problem to Section 4.5.3. For now, we state our main result about the problem in the following lemma.

Lemma 4.5.3. *For all $c_v > 1$, the ℓ-MULTIINDEX problem has an online $(mc_v^{-1/2} \log n \log_{c_v} \ell,\ c_v \log n \log_{c_v} \ell)$-scheme in the strict turnstile update model.*

Analysis of Costs. Let S be the set of items with non-zero frequency when the stream ends. First, we argue that if r is the size of the mapped-down universe, and $\mathcal{P}$ chooses the hash function h at random from a pairwise independent hash family, then with probability $9/10$, there will be at most $10m^2/r$ items in S that collide under g. Indeed, by a union bound, the probability any item i with non-zero count is involved in a collision is at most m/r, and hence by linearity of expectation, the expected number of items involved in a collision is at most m^2/r.

So by Markov's inequality, with probability at least $9/10$, the total number of items involved in a collision will be at most $10m^2/r = O(mc_v^{-1/2})$ under the setting $r = mc_v^{1/2}$. Conditioned on this event, $\mathcal{P}$ can specify the list L_0 and the associated frequencies with annotation length $O(mc_v^{-1/2} \log n)$, and $\mathcal{V}$ can use the MULTIINDEX scheme of Lemma 4.5.3 with $\ell = O(mc_v^{-1/2})$ to verify the frequencies of the items in L_0 are as claimed. For any $c_v > 1$, Lemma 4.5.3 under this setting of ℓ yields an $(mc_v^{-1} \log n \cdot \log_{c_v} \ell, c_v \log n \cdot \log_{c_v} \ell)$-scheme.

Running all of the sum-check schemes (i.e., the INJECTION scheme and the F_k scheme itself) on the mapped-down universe requires annotation $O(k^2 r c_v^{-1} \log r)$ and space $O(k c_v \log r)$ for $\mathcal{V}$; in total, this provides an online $(m^2 \log n/r + k^2 r \log n/c_v + k m c_v^{-1} \log n \cdot \log_{c_v} m, c_v \log n \cdot \log_{c_v} M)$-scheme.

Since we set $r = m c_v^{1/2}$, we obtain a online $(k^2 m c_v^{-1/2} \log_{c_v}(m), k c_v \log n \log_v(m))$-scheme for any $c_v > 1$.

The lower bound stated in Theorem 4.5.1 follows from Theorem 4.3.8 and an easy reduction from the (m, n)-sparse INDEX problem. □

4.5.3 A Scheme for MultiIndex: Proof of Lemma 4.5.3

Before presenting an efficient online scheme for the ℓ-MULTIINDEX Problem, we define two "sub"problems, which apply a function to only a subset of the desired input.

Definition 4.5.4. Define the problem SUBINJECTION as follows. We observe a stream of tuples $t_i = (x_i, b_i, \delta_i)$, followed by a vector $z \in \{0,1\}^r$. As in the INJECTION problem, each t_i indicates that δ_i copies of item x_i are placed in bucket $b_i \in [r]$.

We say that the stream defines a *subinjection* based on z if for every b such that $z_b \geq 1$, for every two pairs (x, b) and (y, b) with positive counts, it holds that $x = y$. The SUBINJECTION problem is to decide whether the stream defines a subinjection based on z.

Notice that the INJECTION problem is a special case of the SUBINJECTION problem with $z_i = 1$ for all i.

Lemma 4.5.5. *For any $c_a c_v \geq r$, there is an online $(c_a \log r, c_v \log r)$-scheme for* SUBINJECTION *in the strict turnstile update model. Moreover, for any constant $c > 0$, this scheme can be instantiated to have soundness error $1/r^c$.*

Proof. Define vectors u, v, and w exactly as in the proof of Lemma 4.4.5, and observe that the stream defines a sub-injection if and only if $\sum_{b\in[r]} z_b v_b^2 = \sum_{b\in[r]} z_b u_b w_b$. $\mathcal{V}$ can compute both quantities using the dense scheme of Theorem 3.5.2, with the same asymptotic costs as the scheme of Lemma 4.4.5. The soundness error can be made smaller than $1/r^c$ for any constant c by running the scheme of Theorem 3.5.2 over a finite field of size $\text{poly}(r)$, for a sufficiently fast-growing polynomial in r. □

We similarly define the problem SubF$_2$ over a data universe of size n based on a vector $z \in \{0,1\}^n$ as $\sum_{i\in[n]} z_i f_i^2$, the sum of squared frequencies of items indicated by z. This too is a low-degree polynomial function of the input values, and so Theorem 3.5.2 implies SubF$_2$ can be computed by an online $(c_a \log r, c_v \log r)$-scheme in the general turnstile update model for any c_a, c_v such that $c_a c_v \geq r$ (and the soundness error in this protocol can be made smaller than $1/r^c$ for any desired constant c).

Online scheme for ℓ-MultiIndex. The scheme can be thought of as proceeding in t stages (t will be specified later), although these stages merely serve to partition the annotation: there is no communication from $\mathcal{V}$ to $\mathcal{P}$ during these stages. Each stage j makes use of a corresponding hash function $h_j : [n] \to [r]$ for $r = mc_v^{1/2}$. The t hash functions are provided by $\mathcal{P}$ at the start of the stream, so that $\mathcal{V}$ has access to them throughout the stream. Each h_j is claimed to be chosen at random from a pairwise independent hash family: if they are, then there are unlikely to be too many collisions, so $\mathcal{P}$ has no incentive not to choose h_j at random. Let f denote the vector of frequencies defined by the input stream, and let $f^{(0)}$ denote the vector satisfying $f_i^{(0)} = f_i$ for $i \in L_0$, and $f_i^{(0)} = 0$ for $i \notin L_0$.

Stage j begins with a list L_{j-1} of items. We will refer to these items as "exceptions". $\mathcal{P}$ provides a new list $L_j \subseteq L_{j-1}$ of items that remain exceptions in stage j; $\mathcal{P}$ implicitly claims that no items in $L_{j-1} \setminus L_j$ collide with some other input items under hash function h_j. Let

$z^{(j)}$ denote the indicator vector of the list of buckets corresponding to $L_{j-1} \setminus L_j$, i.e. $z^{(j)}_{h_j(i)} = 1$ if $i \in L_{j-1} \setminus L_j$, and $z^{(j)}$ entries are 0 otherwise. To check that no items in $L_{j-1} \setminus L_j$ collide under h_j, $\mathcal{V}$ will use the SubInjection scheme based on the indicator vector $z^{(j)}$ over the full original input f as mapped by the hash function h_j. Note that since the original input stream is in the strict turnstile update model, so is the stream on which the SubInjection scheme is run (as the SubInjection scheme is simply run on the original input stream as mapped by the hash function h_j, based on the vector $z^{(j)}$). Note also that L_{j-1} and L_j are provided explicitly, so $\mathcal{V}$ can compute $z^{(j)}$ easily.[2]

Having established that the items in $L_{j-1} \setminus L_j$ are no longer exceptions, $\mathcal{V}$ also wants to ensure that the frequencies of these items were reported correctly in L_0. To do so, $\mathcal{V}$ run the SubF_2 scheme over the vector $f - f^*$ as mapped by h_j to r buckets, based on the $z^{(j)}$ indicator vector. The result is zero if and only if $f_i = f_i^{(j)}$ for all i where $z_i^{(j)} = 1$.

The stages continue until $L_j = \emptyset$, and there are no more exceptions. Provided all schemes conclude correctly, and the number of stages to reach $L_j = \emptyset$ is at most t, $\mathcal{V}$ can accept the result, and output 1 for the answer to the MultiIndex decision problem.

Lastly, note that $\mathcal{V}$ does not need to explicitly store any of the lists L_j. In fact, $\mathcal{P}$ can implicitly specify all of the lists L_j while playing the list L_0: for each item $i \in L_0$, he provides a number j, thereby implicitly claiming that $i \in L_{j'}$ for $j' \leq j$, and $i \notin L_{j'}$ for $j' > j$.

Analysis of costs. If h_j is chosen at random from a pairwise independent hash family, the probability an item i in L_{j-1} is involved in a collision with the original stream f under h_j is $O(m/r) = O(c_v^{-1/2})$. Consider the probability that any item i survives as an exception to stage t. The probability of this is $O(c_v^{-t/2})$, and summed over all ℓ items, the expected number is $O(\ell c_v^{-t/2})$. Invoking Markov's inequality, with constant probability it suffices to

[2]For example, $\mathcal{V}$ can add one to the corresponding entry of $z^{(j)}$ for each item that is marked as an exception. This will cause $z^{(j)}$ to count the number of exceptions in each bucket, rather than indicate them, but this does not affect the correctness.

set $t = O(\log_{c_v} \ell)$ to ensure that we need at most t stages before no more exceptions need to be reported.

In stage j, the SubInjection and SubF_2 schemes cost $(mc_v^{-1/2} \log n, c_v \log n)$. Summing over the t stages, we achieve for any $c_v > 1$ an $(mc_v^{-1/2} \log(n) \cdot \log_{c_v}(m), c_v \log(n) \cdot \log_{c_v}(m))$-scheme as claimed in the statement of Lemma 4.5.3.

Formal Proof of Soundness. The soundness error of the protocol can be bounded by the probability that any invocation of the SubInjection scheme or the SubF_2 scheme returns an incorrect answer. The soundness errors of both the SubInjection scheme and the SubF_2 scheme can be made smaller than $\frac{1}{r^c}$ for any constant $c > 0$, and therefore a union bound over all $t = O(\log_{c_v} \ell)$ invocations of each protocol implies that with high probability, no invocation of either scheme returns an incorrect answer.

4.5.4 Implications of the Online Scheme for Frequency Moments

Our online scheme for F_k in Theorem 4.5.1 has a number of important consequences.

Inner Product and Hamming Distance. The discussion following Theorem 3.5.3 showed that computing inner products and Hamming Distance can be directly reduced to (exact) computation of the second Frequency Moment F_2, so Theorems 4.4.3 and 4.5.1 immediately yield schemes for these problems of identical cost.

An improved scheme for ϕ-HeavyHitters. We can use Lemma 4.5.3 to yield an online scheme for the ϕ-HeavyHitters problem.

Corollary 4.5.6. *For all c_a, c_v such that $c_a c_v \geq m \log n$, there is an online $(c_a \log n \cdot \log_{c_v}(m) + \phi^{-1} \log n,\ c_v \log n \log_{c_v}(m))$-scheme for solving ϕ-HeavyHitters in the strict turnstile update model.*

Corollary 4.5.6 follows from a similar analysis to Corollary 4.3.6. Theorem 3.6.1 describes

how to reduce ϕ-HeavyHitters to demonstrating the frequencies of $O(\phi^{-1} \log n)$ items in a derived stream. Moreover, the derived stream has sparsity $O(m \log n)$ if the original stream has sparsity m. We use the MultiIndex scheme of Lemma 4.5.3 to verify these claimed frequencies.

Frequency-based functions. Theorem 3.7.1 shows how to extend the sum-check scheme of Theorem 3.5.2 to efficiently compute arbitrary *frequency-based functions*, which are functions of the form $F(\mathbf{x}) = \sum_{i \in [n]} g(f_i(\mathbf{x}))$ for an arbitrary $g : (-[N] \cup [N]) \to \mathbb{Z}$. A similar but more involved extension applies in the sparse setting considered in this chapter. We spell out the details below, restricting ourselves to the prescient case for brevity; an online scheme with essentially identical costs follows by using the ideas underlying Theorem 4.5.1.

Corollary 4.5.7. *Let $F(\mathbf{x}) = \sum_{i \in [n]} g(f_i(\mathbf{x}))$ be a frequency-based function. Then there is a prescient $(N^{3/4} \log n,\ N^{3/4} \log n)$-scheme for computing $F(\mathbf{x})$ in the strict unit-update turnstile model. This scheme satisfies perfect completeness.*

Proof. We use a natural modification of the frequency-based functions scheme of Theorem 3.7.1. $\mathcal{P}$ specifies a hash function h at the start of the stream mapping the universe $[n]$ into $[N^{5/4}]$; $\mathcal{P}$ chooses h to be injective on the set of items that have non-zero frequency at the end of the stream. Using the perfect hash functions of Fredman and Komlós [52], h can be represented with $O(N^2/r \log n) = O(N^{3/4} \log n)$ bits. $\mathcal{V}$ stores h explicitly. After the stream is observed, $\mathcal{P}$ and $\mathcal{V}$ run the ϕ-HeavyHitters scheme of Corollary 4.5.6, with $\phi = N^{-1/4}$. Using the fact that $\sum_i f_i < N$, by setting the parameters of Corollary 4.5.6 appropriately we can ensure that this part of the scheme requires annotation length $O(N^{3/4} \log n)$ and has space cost $O(N^{3/4} \log n)$. This scheme also allows $\mathcal{V}$ to determine the exact frequencies of the items in H, allowing $\mathcal{V}$ to compute $\text{cont}(H) := \sum_{i \in H} g(f_i(\mathbf{x}))$, which gives the contribution of the items in H to the output $F(\mathbf{x})$. Moreover, whenever $\mathcal{V}$ learns the frequency f_i of an

item in $i \in H$, $\mathcal{V}$ treats this as a deletion of f_i occurrences of item i, thereby obtaining a derived stream $\mathbf{z}$ in which all frequencies have absolute value at most $N^{1/4}$.

$\mathcal{P}$ and $\mathcal{V}$ now run the sum-check scheme of Theorem 3.5.2 on the "mapped-down" input $h(\mathbf{z})$ over the universe $[N^{5/4}]$. For any pair of positive integers c_a, c_v satisfying $c_a c_v \geq r$, this scheme can achieve cost $(F_\infty(\mathbf{z})c_a \log n, c_v \log n)$, where $F_\infty(\mathbf{z})$ denotes $\max_i |f_i(\mathbf{z})|$, the largest frequency in absolute value of any item (that is, the sum-check scheme of Theorem 3.5.2 is run using a polynomial $\tilde{g}$ of degree F_∞ such that $\tilde{g}(x) = g(x)$ for all $x \in \{0, F_\infty\}$.) Setting $c_v = N^{3/4}$ and $c_a = N^{1/4}$, we obtain a prescient $(N^{3/4} \log n, N^{3/4} \log n)$-scheme as claimed. $\mathcal{V}$ computes the final answer as $F(\mathbf{x}) = \text{cont}(H) + F(h(\mathbf{z})) - |H|g(0)$.

The final issue is that $\mathcal{V}$ needs to verify that h is actually injective over the items that appear in $\mathbf{x}$. $\mathcal{V}$ can accomplish this using the Injection scheme of Lemma 4.4.5. This does not affect the asymptotic costs of our scheme, as the Injection scheme can support annotation cost $c_a \log r$ and space cost $c_v \log r$ for any $c_a c_v = \Omega(N^{5/4})$. □

Finally, we provide one additional corollary, which describes a protocol that will be useful in the next section when building graph schemes.

Theorem 4.5.8. *Let $X, Y \subseteq [n]$ be sets with $|X| \leq |Y| \leq m$. Then given a stream in the strict turnstile update model with elements of X and Y arbitrarily interleaved, there is an online $(mc_v^{-1/2} \cdot \log(n) \cdot \log_{c_v}(m), c_v \cdot \log(n) \cdot \log_{c_v}(m))$-scheme for determining whether $X \subseteq Y$ for any $c_v > 1$.*

Proof. If $X \not\subseteq Y$, $\mathcal{P}$ can specify an $x \in X \setminus Y$ and prove that x is indeed in X and not Y with two point queries using the scheme of Theorem 4.3.2. For the other case, Theorem 3.8.2 shows how to directly reduce the case $X \subseteq Y$ to computation of frequency moments. The claimed costs follow from Theorem 4.5.1. □

Table 4.4 provides a comparison of the Subset schemes given in this thesis.

Table 4.4: Comparison of SUBSET schemes.

Scheme Costs	Completeness	Prescience	Source
$(\lvert X\rvert \log n,\, \log n)$	Perfect	Prescient	Lemma 3.8.1
$(c_a \log n,\, c_v \log n)$: $c_a c_v \geq n$	Perfect	Online	Thm 3.8.2
$(m \log n,\, \log n)$	Perfect	Online	Thm 3.11.1
$(mc_v^{-1/2} \log_{c_v}(m) \log n,\, c_v \log n \log_{c_v} m)$: $c_v > 1$	Imperfect	Online	Thm 4.5.8

The SUBSET scheme of this chapter is the first online SUBSET scheme to achieve annotation length and space usage that are both sublinear in m for $m \ll \sqrt{n}$, and strictly improves over the online MA communication cost of prior protocols whenever $m = o(n)$.

4.6 Graph Problems

We now describe some applications of the techniques developed above to graph problems. The main purpose of this section is to demonstrate that the techniques developed within the F_k and m-DISJ schemes are broadly applicable to a range of settings.

We begin with several non-trivial graph schemes that are direct consequences of the Subset scheme of Theorem 4.5.8. Recall that our definition of a scheme for a function F requires a convincing proof of the value of $F(\mathbf{x})$ *for all values* $F(\mathbf{x})$. This is stricter than the traditional definition of interactive proofs for decision problems, which just require that if $F(\mathbf{x}) = 1$ then there is some prover that will cause the verifier to accept with high probability, and if $F(\mathbf{x}) = 0$ there is no such prover. Here, we consider a relaxed definition of schemes that is in the spirit of the traditional definition. We require only that a scheme $\mathcal{A} = (\mathfrak{h}, \mathcal{V})$ satisfy:

1. For all $\mathbf{x}$ s.t. $F(\mathbf{x}) = 1$, we have $\Pr_{r_\mathcal{P}, r_\mathcal{V}}[\text{out}^\mathcal{V}(\mathbf{x}^{\mathfrak{h}, r_P}, r_\mathcal{V}) \neq 1] \leq 1/3$.

2. For all $\mathbf{x}$ s.t. $F(\mathbf{x}) = 0$, $\mathfrak{h}' = (\mathfrak{h}'_1, \mathfrak{h}'_2, \ldots, \mathfrak{h}'_N) \in (\{0,1\}^*)^N$, we have $\Pr_{r_\mathcal{V}}[\text{out}^\mathcal{V}(\mathbf{x}^{\mathfrak{h}'}, r_\mathcal{V}) = 1] \leq 1/3$.

Theorem 4.6.1. *Under the above relaxed definition of a scheme, each of the problems* PERFECT-MATCHING, CONNECTIVITY, *and* NON-BIPARTITENESS *has an* $(n \log n + mc_v^{-1/2} \log n \log_{c_v} m,\ c_v \log n \log_{c_v} m)$*-scheme on graphs with* n *vertices and* m *edges for all* $c_v > 1$*. All three schemes work in the strict turnstile update model and improve over prior work if* $c_v = \omega(\log^2 m)$ *and* $c_v = o(m)$*.*

Proof. In the case of perfect matching, the prover can prove a perfect matching exists by sending a matching $\mathcal{M}$, which requires $n \log n$ bits of annotation. In order to prove $\mathcal{M}$ is a valid perfect matching, $\mathcal{P}$ needs to prove that every node appears in exactly one edge of $\mathcal{M}$, and that $\mathcal{M} \subseteq E$, where E is the set of edges appearing in the stream. $\mathcal{V}$ can check the first condition by comparing a fingerprint of the nodes in $\mathcal{M}$ to a fingerprint of the set $\{1, \ldots, n\}$. $\mathcal{V}$ can check that $\mathcal{M} \subseteq E$ using Theorem 4.5.8.

In the case of connectivity, the prover demonstrates the graph is connected by specifying a spanning tree T. $\mathcal{V}$ needs to check T is spanning, which can be done as in Theorem 3.10.7, and needs to check that $T \subseteq E$, which can be done using Theorem 4.5.8.

In the case of non-bipartiteness, $\mathcal{P}$ demonstrates an odd cycle C. $\mathcal{V}$ needs to check C is a cycle, C has an odd number of edges, and that $C \subseteq E$. The first condition can be checked by requiring $\mathcal{P}$ to play the edges of C in the natural order. The second condition can be checked by counting. The third condition can be checked using Theorem 4.5.8. □

Counting Triangles. Returning to our strict definition of a scheme, we give an online scheme for counting the number of triangles in a graph.

Theorem 4.6.2. *For any* $c_v > 1$*, there is an online* $(c_a \log n \log m,\ c_v \log n \log m)$*-scheme, with imperfect completeness, for counting the number of triangles in a graph on* n *nodes and* m *edges, where* $c_a = mnc_v^{-1/2}$*. The scheme is valid in the strict turnstile update model.*

Proof. Theorem 3.10.4 showed how to reduce counting the number of triangles in a graph

Table 4.5: Comparison of schemes for counting triangles.

Scheme Costs	Completeness	Prescience	Source
$(c_a \log n, c_v \log n)$: $c_a c_v \geq n^3$	Perfect	Online	Thm 3.10.4
$(n^2 \log n, \log n)$	Perfect	Online	Thm 3.10.3
$(c_a \log^2 n, c_v \log^2 n)$: $c_a = mnc_v^{-1/2}$	Imperfect	Online	Thm 4.6.2

Results are stated for graphs with n nodes and m edges. For concreteness, notice that by setting $c_v = n$, Theorem 4.6.2 achieves a $(mn^{1/2} \log^2 n, n \log^2 n)$-scheme, which improves over prior work as long as $m \ll n^{3/2}$.

to computing the first three frequency moments of a derived stream. The derived stream has sparsity $m(n-2)$. Using the online scheme of Theorem 4.5.1 to compute the relevant frequency moments of the derived stream yields the claimed bounds. □

The scheme of Theorem 4.6.2 should be compared to the $(n^2, \log n)$-scheme from Theorem 3.10.3 based on matrix multiplication, referenced in Row 2 of Table 4.5 and the (c_a, c_v)-scheme for any $c_a c_v \geq n^3$ from Theorem 3.10.4, referenced in Row 1 of Table 4.5. To compare to the former, notice that Theorem 4.6.2 yields a
$(c_a \log^2 n, c_v \log^2 n)$-scheme with $c_a < n^2$ as long as $m < n\sqrt{c_v}$. To compare to the latter, note that in our new scheme, $c_a c_v = mnc_v^{1/2}$, which is less than n^3 as long as $c_v^{1/2} < \frac{n^2}{m}$. In particular, if we set $c_v = n$, then Theorem 4.6.2 improves over both old schemes as long as $m < n^{3/2}$.

Unfortunately, Theorem 4.6.2 does not yield a non-trivial MA-protocol for showing no triangle exists. Indeed, equalizing annotation length and space usage in our new protocol occurs by setting both quantities to $(mn)^{2/3}$. But $\Omega\left((mn)^{2/3}\right) < m$ only when $m > n^2$, which is to say that the MA communication complexity of this protocol is always larger than m, a cost that can be achieved by the trivial MA protocol where Merlin is ignored and Alice

just sends her whole input to Bob. That is, the interest in the new protocol is that it can lower the space usage of $\mathcal{V}$ to less than m without drastically blowing up the message length of $\mathcal{P}$ to n^2 as in the matrix-multiplication based protocol from Theorem 3.10.3.

4.7 Non-strict Turnstile Update Model

All schemes in Sections 4.4 and 4.5 work in the strict turnstile update model. The reason these schemes require this update model is that they use the INJECTION and SUBINJECTION schemes of Lemmata 4.4.5 and 4.5.5 as sub-routines, and these sub-routines assume the strict turnstile update model.

In this section, we consider two ways to circumvent this issue. To focus the discussion, we concentrate on the online F_k protocol of Theorem 4.5.1.

4.7.1 An Online Scheme

One simple method for handling streams in the non-strict turnstile update model is the following. We use the scheme of Theorem 4.5.1, but within the SUBINJECTION sub-routine, we treat deletions of items in the input stream as *insertions* of items into the derived stream of (x_i, b_i, δ_i) updates. This ensures that the INJECTION and SUBINJECTION schemes correctly output 1 if the derived stream is a subinjection (and the remainder of the scheme computes the correct answer on the original stream). However it increases the expected number of collisions under the universe-reduction mappings h_i, from $m \cdot |L_{i-1}|/r$ to $M \cdot |L_{i-1}|/r$. The result is that we achieve the same costs as Theorem 4.5.1, except the costs depend on to the stream footprint M rather than the stream sparsity m (see Section 3.2.4).

Corollary 4.7.1. *For any $c_v > 1$, there is a $(k^2 M c_v^{-1/2} \cdot \log(n) \cdot \log_{c_v}(M), k c_v \cdot \log(n) \cdot \log_{c_v}(M))$ online scheme for F_k in the non-strict turnstile update model over a stream with footprint*

M over a universe of size n.

4.7.2 An Online AMA Scheme

In this section, we describe an AMA scheme for the INJECTION problem that works in the non-strict turnstile stream update model i.e., the input may define a frequency vector where some elements end with negative frequency. The scheme for INJECTION of Lemma 4.4.5 breaks down here, since there may be some cases where the checks performed by the protocol indicate that a bucket is pure, when this is not the case: cancellations of item weights in the bucket may give the appearance of purity. To address this, we use public randomness, thereby yielding an AMA scheme. In essence, the verifier asks the prover to demonstrate the purity of each of the r buckets via fingerprints of the bucket contents. However, if we allow the prover to choose the fingerprint function, $\mathcal{P}$ could pick a function that leads to false conclusions. Instead, $\mathcal{V}$ chooses the fingerprinting function using public randomness. The players then execute a new INJECTION protocol using the data remapped under the fingerprint function, which is intended to convince $\mathcal{V}$ of the purity of the buckets. This then allows us to construct protocols with costs that depend on the stream sparsity m rather than the footprint M as in Corollary 4.7.1.

In detail, the new AMA scheme proceeds as follows. Consider the INJECTION problem as defined in Definition 4.4.4, but generalized to allow items with arbitrary integer counts. Consider again a bucket b, and for $1 \leq j \leq \log n$ define $b^{j=\ell}$ to be the frequency vector of the subset of stream updates (x_k, b, δ_k) placing items into bucket b, subject to the restriction that the j'th bit of x_k is equal to ℓ. We observe the following property: if bucket b is pure, then one of $b^{j=0}$ and $b^{j=1}$ must be the zero vector $\mathbf{0}$, for each j. Moreover, if b is not pure, then there exists a j such that both $b^{j=0}$ and $b^{j=1}$ are not the zero vector.

A natural way to compactly test whether these vectors are equal to zero (probabilistically)

is to use fingerprinting (discussed in Section 3.2.4). The verifier $\mathcal{V}$ could do this unaided for a single bucket, but we wish to run this test in parallel for r buckets. At a high level, we achieve this as follows. Given a stream of updates (x_k, b, δ_k), we define two vectors z and o of length $r \log n$, such that each coordinate of z and o corresponds to a (bucket, coordinate) pair $(b, j) \in [r] \times [\log n]$. In more detail, we will define z and o such that for each bucket b and coordinate $j \in [\log n]$, the (b, j)th entry of z is a fingerprint of the vector $b^{j=0}$, and the (b, j)th entry of o is a fingerprint of the vector $b^{j=1}$.

We choose the fingerprinting functions to satisfy two properties.

1. The fingerprint of the all-zeros vector $\mathbf{0}$ is always 0. This ensures that if all buckets are pure, then the inner product of z and o is 0, as $z_{b,j} \cdot o_{b,j}$ is 0 for all pairs $(b, j) \in [r] \times [\log n]$.

2. If there is an impure bucket, then the inner product of z and o will be non-zero with high probability over the choice of fingerprint functions.

Therefore, in order to determine whether the stream defines an injection, it suffices to compute $\sum_{(b,j)\in[r]\times[\log n]} z_{b,j} \cdot o_{b,j}$, which can be computed using Theorem 3.5.2 with annotation length $c_a \log n$ and space cost $c_v \log n$ for any $c_a \cdot c_v \geq r \log n$.

The idea allowing us to achieve the second property is as follows. If bucket b is impure, then there is at least one coordinate $j \in [\log n]$ such that $b^{j=0}$ and $b^{j=1}$ are both not equal to the all-zeros vector $\mathbf{0}$. By basic properties of fingerprints, this ensures that both $z_{b,j}$ and $o_{b,j}$ are non-zero with high probability over the choice of fingerprint functions. Moreover, we choose the fingerprinting functions in such a way that non-zero terms in the sum $\sum_{(b,j)\in[r]\times[\log n]} z_{b,j} \cdot o_{b,j}$ are unlikely to "cancel out" to zero.

Consequently, we can state an analog of Lemma 4.4.5.

Lemma 4.7.2. *For any $c_a c_v \geq r \log n$, there is an online $(c_a \log n, c_v \log n)$-scheme for determining whether a stream in the non-strict turnstile model is an injection.*

Proof. Let $\mathbb{F}_q$ be a finite field of size $q = \text{poly}(n)$, where the subsequent analysis determines the required magnitude of q. $\mathcal{V}$ uses public randomness to choose two field elements α, and β uniformly at random from $\mathbb{F}_q$. For each bucket $b \in [r]$, and each coordinate $j \in [\log n]$, we define two "fingerprinting" functions $g_{b,j,\alpha}$ and $g_{b,j,\beta}$ mapping an n-dimensional vector $\mathbf{y}$ as follows:

$$g_{b,j,\alpha}(\mathbf{y}) = \alpha^{n(b \cdot \log n + j)} \sum_{\ell \in [n]} \mathbf{y}_\ell \alpha^\ell,$$

and

$$g_{b,j,\beta}(\mathbf{y}) = \beta^{n(b \cdot \log n + j)} \sum_{\ell \in [n]} \mathbf{y}_\ell \beta^\ell,$$

where each entry $\mathbf{y}_\ell$ of $\mathbf{y}$ is treated as an element of $\mathbb{F}_q$ in the natural manner.

We now (conceptually) construct two vectors z and o of dimension $r \log n$, where for each $(b, j) \in [r] \times [\log n]$, $z_{b,j} = g_{b,j,\alpha}(b^{j=0})$ and $o_{b,j} = g_{b,j}(b_i^{j=1})$. That is, the (b, j)th entry of z equals the fingerprint of the frequency vector of items mapping to bucket b with a 0 in the jth bit of their binary representation. Observe that $g_{b,j,\alpha}(\mathbf{0}) = g_{b,j,\beta}(\mathbf{0}) = 0$ for all $(b, j) \in [r] \times [\log n]$, as required by Property 1 above.

We now show that Property 2 holds, i.e., if there is an impure bucket, then the inner product of z and o will be non-zero with high probability over the choice of α and β. In the following, for an item $\ell \in [n]$ and bucket $b \in [r]$, we let $f_\ell(b)$ denote the frequency with which item ℓ is mapped to bucket b, and we let ℓ_j denote the j'th bit in the binary representation

of ℓ. We can write the inner product of z and o as

$$\begin{aligned}
&\sum_{(b,j)\in[r]\times[\log n]} g_{b,j,\alpha}(b^{j=0}) g_{b,j,\beta}(b^{j=1}) \\
&= \sum_{(b,j)\in[r]\times[\log n]} \alpha^{n(b\cdot\log n+j)}\beta^{n(b\cdot\log n+j)} \left(\sum_{\ell\in[n],\ell_j=0} f_\ell(b)\alpha^\ell\right)\left(\sum_{\ell\in[n],\ell_j=1} f_\ell(b)\beta^\ell\right) \\
&= \sum_{(b,j)\in[r]\times[\log n]} \alpha^{n(b\cdot\log n+j)}\beta^{n(b\cdot\log n+j)} \sum_{(\ell,\ell'):\ell_j=0,\ell'_j=1} f_\ell(b) f_{\ell'}(b)\alpha^\ell\beta^{\ell'}
\end{aligned}$$

We therefore see that the inner product of z and o is a polynomial in α and β of total degree $n^2 r \log n$ in each variable. Moreover, the coefficient of the term $\alpha^{n(b\cdot\log n+j)+\ell}\beta^{n(b\cdot\log n+j)+\ell'}$ is precisely $f_\ell(b)\cdot f_{\ell'}(b)$ if $\ell_j = 0$ and $\ell'_j = 1$, and is 0 otherwise.

Recall that if bucket b is not pure, then there is at least one coordinate $j \in [\log n]$, and items $\ell, \ell' \in [n]$ with $\ell_j = 0$ and $\ell'_j = 1$, such that $f_\ell(b) \neq 0$ and $f_{\ell'}(b) \neq 0$. The above analysis implies that $z \cdot o$ is a *non-zero* polynomial in α and β, as the coefficient of $\alpha^{n(b\cdot\log n+j)+\ell}\beta^{n(b\cdot\log n+j)+\ell'}$ is non-zero. Hence, by the Schwartz-Zippel lemma, the probability over a random choice of α and β that $z \cdot o = 0$ is at most $n^2 r \log n/q$. Setting q to be polynomial in n, there is only negligible probability (over the choice of α and β) that $z \cdot o$ is zero if the stream is not an injection.

Finally, notice that the verifier can apply the scheme of Theorem 3.5.2 to compute $\sum_{(b,j)\in[r]\times[\log n]} z_{b,j} \cdot o_{b,j}$, as each stream update (x_k, b, δ_k) can be treated as $\log n$ updates to the vectors z and o. For example, if the jth bit of x_k is 0, then update (x_k, b, δ_k) causes $z_{b,j}$ to be incremented by $\delta_k \cdot \alpha^{n(b\cdot\log n+j)+x_k}$. □

Applications. We can apply this online scheme to compute Frequency Moments (and Inner Product, Hamming Distance, Heavy Hitters etc.) over sparse data in the non-strict turnstile update model. In particular, this lets us compute frequency moments even when the frequency vectors may have negative entries, and it lets us compute the inner product of two

vectors even when they may have negative entries. The costs of the resulting online AMA scheme are similar to the costs of the online schemes for the same problems developed in previous sections. The only difference is that we have scaled m up by a $\log n$ factor, to account for the fact that within the new AMA sub-scheme for INJECTION, we must run the dense protocol of Theorem 3.5.2 on vectors z and o of length $r \log n$, rather than on vectors of length r as in prior sections, and substitute the bounds from Lemma 4.7.2. For example, the analog of Theorem 4.5.1 is that for any $c_v > 1$, there is a $(k^2 m c_v^{-1/2} \cdot \log^2 n \cdot \log_{c_v} m, kc_v \cdot \log n \cdot \log_{c_v} m)$ online AMA scheme for F_k in the non-strict turnstile model.

4.8 Discussion and Open Problems

In this chapter, we have presented a number of protocols in the annotated data streaming model that for the first time allows both the annotation length and the space usage of the verifier to be sublinear in the stream sparsity, rather than just the size of the data universe. Our protocols substantially improve on the applicability of prior work in natural settings where data streams are defined over very large universes, such as IP packet flows and sparse graph data.

A number of interesting questions remain for future work. The biggest open question is to determine the precise dependence on the stream sparsity in problems such as m-DISJ and frequency moments. When setting the annotation length and the space usage of the verifier to be equal, our protocols have cost roughly $m^{2/3}$, where m is the sparsity of the data stream. The best known lower bound is roughly $m^{1/2}$. We conjecture that our upper bound is tight up to logarithmic factors, but proving any Merlin-Arthur communication lower bound larger than $m^{1/2}$ will require new lower bound techniques in communication complexity. Another interesting open question is to give improved protocols for multiplying an $n \times n$ matrix

A by a vector x, when A is sparse (i.e., has $o(n^2)$ non-zero entries), but x may be dense. Achieving this would yield improved protocols for proving disconnectedness, bipartiteness, or the non-existence of a perfect matching in a bipartite graph. Currently we do not know of any protocols for these problems that leverage graph sparsity in any way.

Chapter 5

The GKR Protocol

With the sum-check protocol described in Chapter 2 in hand, we are ready to present Goldwasser, Kalai, and Rothblum's interactive proof system (henceforth GKR protocol) [57]. We begin with a high-level overview, before describing the technical details. The GKR protocol is framed in the context of *circuit evaluation*. Given a layered arithmetic circuit C of fan-in 2, the GKR protocol allows a prover to evaluate C, while providing a guarantee that the output is correct.

5.1 The GKR Protocol From 10,000 Feet

In the GKR protocol, $\mathcal{P}$ and $\mathcal{V}$ first agree on an arithmetic circuit C of fan-in 2 over a finite field $\mathbb{F}$ computing the function of interest (C may have multiple outputs). Each gate of C performs an addition or multiplication over $\mathbb{F}$. C is assumed to be in layered form, meaning that the circuit can be decomposed into layers, and wires only connect gates in adjacent layers. Suppose the circuit has depth d; we will number the layers from 1 to d with layer d referring to the input layer, and layer 1 referring to the output layer. The reason we use this numbering system is that the GKR protocol starts by examining the output layer

of the circuit, and works its way one layer at a time toward the input layer.

In the first message, $\mathcal{P}$ tells $\mathcal{V}$ the (claimed) output of the circuit. The protocol then works its way in iterations towards the input layer, with one iteration devoted to each layer. The purpose of iteration i is to reduce a claim about the values of the gates at layer i to a claim about the values of the gates at layer $i+1$, in the sense that it is safe for $\mathcal{V}$ to assume that the first claim is true as long as the second claim is true. This reduction is accomplished by applying the sum-check protocol to a certain polynomial.

More concretely, the GKR protocol starts with a claim about the values of the output gates of the circuit, but $\mathcal{V}$ cannot check this claim without evaluating the circuit herself, which is precisely what she wants to avoid. So the first iteration uses a sum-check protocol to reduce this claim about the outputs of the circuit to a claim about the gate values at layer 2 (more specifically, to a claim about an evaluation of the *multilinear extension* of the gate values at layer 2; multilinear extensions are formally defined in the next section). Once again, $\mathcal{V}$ cannot check this claim herself, so the second iteration uses another sum-check protocol to reduce the latter claim to a claim about the gate values at layer 3, and so on. Eventually, $\mathcal{V}$ is left with a claim about the inputs to the circuit, and $\mathcal{V}$ can check this claim on her own.

In summary, the GKR protocol uses a sum-check protocol at each level of the circuit to enable $\mathcal{V}$ to go from verifying a randomly chosen evaluation of the multilinear extension of the gate values at layer i to verifying a (different) evaluation of the multilinear extension of the gate values at layer $i+1$. Importantly, apart from the input layer and output layer, $\mathcal{V}$ does not ever see all of the gate values at a layer (in particular, $\mathcal{P}$ does not send these values in full). Instead, $\mathcal{V}$ relies on $\mathcal{P}$ to do the hard work of actually evaluating the circuit, and uses the power of the sum-check protocol as the main tool to force $\mathcal{P}$ to be consistent and truthful over the course of the protocol.

5.2 Notation

The notation introduced in this section will be used throughout Chapters 5-8.

For any d-variate polynomial $p(x_1, \ldots, x_d) : \mathbb{F}^d \to \mathbb{F}$, we use $\deg_i(p)$ to denote the degree of p in variable i. We say p is said to be *multilinear* if $\deg_i(p) \leq 1$ for all $i \in [d]$. Given a function $V : \{0,1\}^d \to \{0,1\}$ whose domain is the d-dimensional Boolean hypercube, the *multilinear extension* (MLE) of V over $\mathbb{F}$, denoted $\tilde{V}$, is the unique multilinear polynomial $\mathbb{F}^d \to \mathbb{F}$ that agrees with V on all Boolean-valued inputs. That is, $\tilde{V}$ is the unique multilinear polynomial over $\mathbb{F}$ satisfying $\tilde{V}(x) = V(x)$ for all $x \in \{0,1\}^d$.

Suppose we are given a layered arithmetic circuit C of size $S(n)$, depth $d(n)$, and fan-in two. For the purposes of this thesis, C will always be defined over a finite field $\mathbb{F}$ of prime order. Let S_i denote the number of gates at layer i of the circuit C. Assume S_i is a power of 2 and let $S_i = 2^{s_i}$. In order to explain how each iteration of the GKR protocol proceeds, we need to introduce several functions, each of which encodes certain information about the circuit.

To this end, number the gates at layer i from 0 to $S_i - 1$, and let $V_i : \{0,1\}^{s_i} \to \mathbb{F}$ denote the function that takes as input a binary gate label and outputs the corresponding gate's value at layer i. The GKR protocol makes use of the multilinear extension $\tilde{V}_i$ of the function V_i.

The GKR protocol also makes use of the notion of a "wiring predicate" that encodes which pairs of wires from layer $i+1$ are connected to a given gate at layer i in C. We define two functions, add_i and mult_i mapping $\{0,1\}^{s_i+2s_{i+1}}$ to $\{0,1\}$, which together constitute the wiring predicate of layer i of C. Specifically, these functions take as input three gate labels (j_1, j_2, j_3), and return 1 if gate j_1 at layer i is the addition (respectively, multiplication) of gates j_2 and j_3 at layer $i+1$, and return 0 otherwise. Let $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ denote the multilinear extensions of add_i and mult_i respectively.

Finally, let $\beta_{s_i}(z,p)$ denote the function

$$\beta_{s_i}(z,p) = \prod_{j=1}^{s_i} ((1-z_j)(1-p_j) + z_j p_j)\,.$$

It is straightforward to check that β_{s_i} is the multilinear extension of the function $B(x,y) : \{0,1\}^{s_i} \times \{0,1\}^{s_i} \to \{0,1\}$ that evaluates to 1 if $x = y$, and evaluates to 0 otherwise.

5.2.1 Technical Outline of the GKR Protocol

The GKR protocol consists of $d(n)$ iterations, one for each layer of the circuit. Each iteration starts with $\mathcal{P}$ claiming a value for $\tilde{V}_i(z)$ for some field element $z \in \mathbb{F}^{s_i}$. In the case of iteration one and circuits with a single output gate, $z = 0$ and $\tilde{V}_1(0)$ corresponds to the output value of the circuit.

In the case of iteration one and circuits with many output gates, Vu et al. [104] observe the following: $\mathcal{P}$ may simply send $\mathcal{V}$ the (claimed) values of all output gates, thereby specifying a function $V_1' : \{0,1\}^{s_1} \to \mathbb{F}$ claimed to equal V_1. $\mathcal{V}$ can pick a random point $z \in \mathbb{F}^{s_1}$ and evaluate $\tilde{V}_1'(z)$ on her own in $O(S_1)$ time. The Schwartz-Zippel Lemma (Lemma 2.2.1) implies that if $\tilde{V}' \neq \tilde{V}$, then $\tilde{V}_1'(z) \neq \tilde{V}(z)$ with probability at least $1 - d/|\mathbb{F}|$ over the random choice of z. Thus, it is safe for $\mathcal{V}$ to believe that V_1' indeed equals V_1 as claimed, as long as $\tilde{V}_1(z) = \tilde{V}_1'(z)$ (which will be checked in the remainder of the protocol).

The purpose of iteration i is to reduce the claim about the value of $\tilde{V}_i(z)$ to a claim about $\tilde{V}_{i+1}(\omega)$ for some $\omega \in \mathbb{F}^{s_{i+1}}$, in the sense that it is safe for $\mathcal{V}$ to assume that the first claim is true as long as the second claim is true. To accomplish this, the iteration applies the sum-check protocol to a specific polynomial derived from $\tilde{V}_{i+1}$, $\tilde{\text{add}}_i$, and $\tilde{\text{mult}}_i$, and β_{s_i}.

Details for Each Iteration

Applying the Sum-Check Protocol. It can be shown (see Lemma 8.4.1) that for any $z \in \mathbb{F}^{s_i}$,

$$\tilde{V}_i(z) = \sum_{(p,\omega_1,\omega_2)\in\{0,1\}^{s_i+2s_{i+1}}} f_z^{(i)}(p,\omega_1,\omega_2),$$

where

$$f_z^{(i)}(p,\omega_1,\omega_2) =$$
$$\beta_{s_i}(z,p)\cdot\left(\tilde{\text{add}}_i(p,\omega_1,\omega_2)(\tilde{V}_{i+1}(\omega_1)+\tilde{V}_{i+1}(\omega_2)) + \tilde{\text{mult}}_i(p,\omega_1,\omega_2)\tilde{V}_{i+1}(\omega_1)\cdot\tilde{V}_{i+1}(\omega_2)\right). \quad (5.1)$$

Remark 3. Equation 5.1 is valid using any extensions of add and mult, not just the multilinear extensions. In fact, Goldwasser, Kalai, and Rothblum [57] use higher-degree extensions derived from a circuit computing the functions add_i and mult_i. We present the protocol as using the multilinear extensions $\tilde{\text{add}}$ and $\tilde{\text{mult}}$ because their use is critical to our results in Chapter 7.

Iteration i therefore applies the sum-check protocol to the polynomial $f_z^{(i)}$. There remains the issue that $\mathcal{V}$ can only execute her part of the sum-check protocol if she can evaluate the polynomial $f_z^{(i)}$ at a random point $f_z^{(i)}(r_1,\ldots,r_{s_i+2s_{i+1}})$. This is handled as follows.

Let p^* denote the first s_i entries of the vector $(r_1,\ldots,r_{s_i+2s_{i+1}})$, ω_1^* the next s_{i+1} entries, and ω_2^* the last s_{i+1} entries. Evaluating $f_z^{(i)}(p^*,\omega_1^*,\omega_2^*)$ requires evaluating $\beta(z,p^*)$, $\tilde{\text{add}}_i(p^*,\omega_1^*,\omega_2^*)$, $\tilde{\text{mult}}_i(p^*,\omega_1^*,\omega_2^*)$, $\tilde{V}_{i+1}(\omega_1^*)$, and $\tilde{V}_{i+1}(\omega_2^*)$.

$\mathcal{V}$ can easily evaluate $\beta(z,p^*)$ in $O(s_i)$ time. For many circuits, particularly those with "regular" wiring patterns, $\mathcal{V}$ can evaluate $\tilde{\text{add}}_i(p^*,\omega_1^*,\omega_2^*)$ and $\tilde{\text{mult}}_i(p^*,\omega_1^*,\omega_2^*)$ on her own in $\text{poly}(s_i,s_{i+1})$ time as well.[1]

[1]Various suggestions have been put forth for what to do if this is not the case. For example, we note in Chapter 7 that these computations can always be done by $\mathcal{V}$ in $O(\log S(n))$ *space* as long as the circuit is log-space uniform, which is sufficient in streaming applications where the space usage of the verifier is paramount. Moreover, these computations can be done offline before the input is even observed, because

$\mathcal{V}$ cannot however evaluate $\tilde{V}_{i+1}(\omega_2^*)$, and $\tilde{V}_{i+1}(\omega_1^*)$ on her own without evaluating the circuit. Instead, $\mathcal{V}$ asks $\mathcal{P}$ to simply tell her these two values, and uses iteration $i+1$ to *verify* that these values are as claimed. However, one complication remains: the precondition for iteration $i+1$ is that $\mathcal{P}$ claims a value for $\tilde{V}_i(z)$ for a single $z \in \mathbb{F}^{s_i}$. So $\mathcal{V}$ needs to reduce verifying both $\tilde{V}_{i+1}(\omega_2^*)$ and $\tilde{V}_{i+1}(\omega_1^*)$ to verifying $\tilde{V}_{i+1}(\omega^*)$ at a single point $\omega^* \in \mathbb{F}^{s_{i+1}}$, in the sense that it is safe for $\mathcal{V}$ to accept the claimed values of $\tilde{V}_{i+1}(\omega_1^*)$ and $\tilde{V}_{i+1}(\omega_2^*)$ as long as the value of $\tilde{V}_{i+1}(\omega^*)$ is as claimed. This is done as follows.

Reducing to Verification of a Single Point. Let $\ell : \mathbb{F} \to \mathbb{F}^{s_{i+1}}$ be some canonical line passing through ω_1^* and ω_2^*. For example, we can let ℓ be the unique line such that $\ell(0) = \omega_1^*$ and $\ell(1) = \omega_2^*$. $\mathcal{P}$ sends a degree-s_{i+1} polynomial h claimed to be $\tilde{V}_{i+1} \circ \ell$, the restriction of $\tilde{V}_{i+1}$ to the line ℓ. $\mathcal{V}$ checks that $h(0) = \omega_1^*$ and $h(1) = \omega_2^*$ (rejecting if this is not the case), picks a random point $r^* \in \mathbb{F}$, and asks $\mathcal{P}$ to prove that $\tilde{V}_{i+1}(\ell(r^*)) = h(r^*)$. By the Schwartz-Zippel Lemma (Lemma 2.2.1), as long as $\mathcal{V}$ is convinced that $\tilde{V}_{i+1}(\ell(r^*)) = h(r^*)$, it is safe for $\mathcal{V}$ to believe that the values of $\tilde{V}_{i+1}(\omega_1^*)$ and $\tilde{V}_{i+1}(\omega_2^*)$ are as claimed by $\mathcal{P}$. This completes iteration i; $\mathcal{P}$ and $\mathcal{V}$ then move on to the iteration for layer $i+1$ of the circuit, whose purpose is to verify that $\tilde{V}_{i+1}(\ell(r^*))$ has the claimed value.

The Final Iteration. Finally, at the final iteration d, $\mathcal{V}$ must evaluate $\tilde{V}_d(\omega^*)$ on her own. But the vector of gate values at layer d of C is simply the input x to C. We observe in the next chapter that $\mathcal{V}$ can compute $\tilde{V}_d(\omega^*)$ on her own in $O(n \log n)$ time, with a single streaming pass over the input.

---

they only depend on the wiring of the circuit, and not on the input [39,57]. Finally, [104] notes that the cost of this computation can be effectively amortized in a batching model, where many identical computations on different inputs are verified simultaneously. See Chapter 8 for further discussion, and a protocol that mitigates this issue in the context of data parallel computation.

Discussion of Costs.

Observe that the polynomial $f_z^{(i)}$ defined in Equation (5.1) is an $(s_i + 2s_{i+1})$-variate polynomial of degree at most 2 in each variable, and so the invocation of the sum-check protocol at iteration i requires $s_i + 2s_{i+1}$ rounds, with three field elements transmitted per round. Thus, the total communication cost is $O(d(n) \log S(n))$ field elements, where $d(n)$ is the depth of the circuit C. We show in Chapter 7 that the verifier can be implemented in a streaming manner, and moreover that the time cost to $\mathcal{V}$ is $O(n \log n + d(n) \log S(n))$ (recall that here and throughout this thesis, an addition or multiplication within a finite field is assumed to require a single time step). The $n \log n$ term is due to the time required to evaluate $\tilde{V}_d(\omega^*)$ (see Lemma 6.3.1), and the $d(n) \log S(n)$ term is the time required for $\mathcal{V}$ to send messages to $\mathcal{P}$ and process and check the messages from $\mathcal{P}$.

As for $\mathcal{P}$'s runtime, for any iteration i of the GKR protocol, a naive implementation of the prover in the corresponding instance of the sum-check protocol would require time $\Omega(2^{s_i+2s_{i+1}})$, as the sum defining each of $\mathcal{P}$'s messages is over as many as $2^{s_i+2s_{i+1}}$ terms. This cost can be $\Omega(S(n)^3)$, which is prohibitively large in practice. However, we show in Chapter 7 that each gate at layers i and $i+1$ of C contributes to only a *single* term of sum, and exploit this to bring the runtime of the $\mathcal{P}$ down to $O(S(n) \log S(n))$.

Chapter 6

Streaming Interactive Proofs

In this chapter, we introduce the model of streaming interactive proofs, which extends the annotated data streams model to allow for multiple rounds of interaction between the prover and verifier. We then present some preliminary results on streaming interactive proofs: we first observe that the GKR protocol can be made to work with a streaming verifier,[1] before presenting improved protocols for specific problems of fundamental importance in streaming and database processing. Our results in this chapter reveal an exponential separation between the annotated data streams model and the streaming interactive proofs model.

6.1 The Model

Definition 6.1.1 (Streaming Interactive Proofs)**.** Consider a prover $\mathcal{P}$ and verifier $\mathcal{V}$ who both observe a stream $\mathbf{x}$ of length N over a data universe of size n and wish to compute a function $F(\mathbf{x})$. We assume $\mathcal{V}$ has access to a private random string $r_{\mathcal{V}}$, and one-way access to the data stream $\mathbf{x}$. The stream is also observed by the prover $\mathcal{P}$. After the stream is observed, $\mathcal{P}$ and $\mathcal{V}$ exchange a sequence of messages. Denote the output of $\mathcal{V}$ on input $\mathbf{x}$, given prover

[1]This observation is due to Guy Rothblum; we present the details for completeness.

$\mathcal{P}$ and $\mathcal{V}$'s random bits $r_\mathcal{V}$, by $\text{out}(\mathcal{V}, \mathbf{x}, r_\mathcal{V}, \mathcal{P})$. $\mathcal{V}$ can output $\perp$ if $\mathcal{V}$ is not convinced that $\mathcal{P}$'s claim is valid.

$\mathcal{P}$ is a *valid prover* with respect to $\mathcal{V}$ if for all data streams $\mathbf{x}$, $\Pr_{r_\mathcal{V}}[\text{out}(\mathcal{V}, \mathbf{x}, r_\mathcal{V}, \mathcal{P}) = F(\mathbf{x})] = 1$. We say $\mathcal{V}$ is a *valid verifier* for F with *soundness error* δ_s if there is at least one valid prover $\mathcal{P}$ with respect to $\mathcal{V}$, and for all provers $\mathcal{P}'$ and all inputs x, $\Pr[\text{out}(\mathcal{V}, \mathbf{x}, r_\mathcal{V}, \mathcal{P}') \notin \{F(\mathbf{x}), \perp\}] \leq \delta_s$. We say a prover-verifier pair $(\mathcal{P}, \mathcal{V})$ is a *valid streaming interactive proof protocol* for F if $\mathcal{V}$ is a valid verifier for F with soundness error $1/3$, and $\mathcal{P}$ is a valid prover with respect to $\mathcal{V}$.

Notice that the streaming interactive proofs model permits multiple rounds of interaction between the prover and verifier. This is in contrast to the annotated data streams model of Chapters 3 and 4, where the prover had to send a single message to the verifier with no communication allowed in the reverse direction. As such, streaming interactive proofs can be thought of as a generalization of annotated data streams to allow for multiple rounds of interaction between the prover and verifier. Alternatively, the streaming interactive proofs model can be thought of as a restriction of the standard interactive proofs model defined in Chapter 2, requiring the verifier be streaming.

We remark that the constant $1/3$ used for the soundness error in Definition 6.1.1 is chosen for consistency with the interactive proofs literature, where $1/3$ is used by convention. As in the annotated data stream protocols of Chapters 3 and 4, the soundness error in all of our interactive proof protocols can be set arbitrarily small by appropriate choice of a parameter, specifically the size of the finite field used. The space and communication costs of the protocol will be proportional only to the *logarithm* of the field size, while the soundness error will decrease *linearly* in the field size.

Notice for simplicity we require valid streaming interactive proof protocols to have completeness error 0 (i.e., *perfect completeness*), rather than completeness error at most $1/3$ as

in Definition 2.1.1. All of the methods developed in the remainder of this thesis naturally satisfy the perfect completeness property.

As our first concern in a streaming setting is the space requirements of the verifier as well as the communication cost for the streaming interactive proof protocol, we make the following definition.

Definition 6.1.2. We say F possesses an r-message (c_a, c_v) streaming interactive proof protocol, if there exists a valid verifier $\mathcal{V}$ for F such that:

1. $\mathcal{V}$ has access to only $O(v)$ bits of working memory, both while processing the stream, and while interacting with the prover.

2. There is a valid prover $\mathcal{P}$ for $\mathcal{V}$ such that $\mathcal{P}$ and $\mathcal{V}$ exchange at most r messages in total, and the sum of the lengths of all messages is $O(c_a)$ bits.

We say an r-message streaming interactive proof protocol has $\lceil r/2 \rceil$ *rounds*.

6.2 Notation

For a vector $\mathbf{b} \in \{0,1\}^{\log n}$, let $\chi_{\mathbf{b}}(x_1, \ldots, x_{\log n}) = \prod_{k=1}^{\log n} \chi_{\mathbf{b}_k}(x_k)$, where $\chi_0(x_k) = 1 - x_k$ and $\chi_1(x_k) = x_k$. Notice that $\chi_{\mathbf{b}}$ is the unique multilinear polynomial that takes $\mathbf{b} \in \{0,1\}^{\log n}$ to 1 and all other values in $\{0,1\}^{\log n}$ to 0, i.e., it is the multilinear extension of the indicator function for boolean vector $\mathbf{b}$.

6.3 A General Theorem

Recall from Chapter 5 that the GKR protocol is defined in the context of circuit evaluation: it allows a prover to evaluate a layered arithmetic circuit C, while providing a guarantee that the output is correct. In this section, we show that the GKR protocol can be made to

work with a streaming verifier. Before doing so, we must precisely state how the data stream specifies the input to the circuit C.

Input Representation. Each element of the stream is a tuple (i, δ), where $i \in [n]$ and δ is an integer. The δ values may be negative, thereby modeling deletions. The data stream implicitly defines a frequency vector f, where f_i is the sum of all δ values associated with i in the stream. When checking the evaluation of a circuit C, we consider the inputs to C to be the entries of the frequency vector f, interpreted as elements of the field $\mathbb{F}$ in the natural way. Here, we assume that $\mathbb{F}$ is a field of prime order, and that $|\mathbb{F}| > 2\|f\|_\infty$. We emphasize that the results below imply that, in order to run the GKR protocol on a circuit C operating on a vector f, a streaming verifier only needs to see the raw stream and not the aggregated frequency vector f (see Lemma 6.3.1 for details). Notice that we may interpret the frequency vector f as an object other than a vector, such as a matrix or a string. For example, in the MatMult problem that we consider prominently in Chapters 7 and 8, the data stream defines two matrices to be multiplied.

Implementing the GKR Protocol with a Streaming Verifier. Within the GKR protocol, the verifier chooses a random vector $\omega^* \in \mathbb{F}^{\log n}$ and evaluates $\tilde{V}_d(\omega^*)$. Recall from Chapter 5 that $\tilde{V}_d$ denotes the multilinear extension of V_d, where $V_d : \{0,1\}^{\log n} \to \mathbb{F}$ is the function that takes as input the binary representation of an integer i between 1 and n, and outputs f_i, the value of the ith input gate. This is the *only* information the verifier must extract from the input in order to run the GKR protocol, and we show in Lemma 6.3.1 that a streaming verifier can compute $\tilde{V}_d(\omega^*)$ using $O(\log |\mathbb{F}| \cdot \log n)$ bits of space. Notice that in the presentation of Chapter 5, the verifier only chose ω^* in the final iteration of the interactive proof protocol. Here, $\mathcal{V}$ will choose ω^* before observing the stream, store it explicitly in memory, and keep it private from the prover until the final iteration of GKR protocol.

Lemma 6.3.1. *Assume n is a power of two. Given a data stream $\mathbf{x} \in \mathbb{F}^n$ and a vector*

$\omega^* \in \mathbb{F}^{\log n}$, $\mathcal{V}$ can compute $\tilde{V}_d(\omega^*)$ with a single pass over the data stream $\mathbf{x}$ while storing $O(\log n)$ field elements. $\mathcal{V}$ must perform $O(\log n)$ field operations per stream update, where $\tilde{V}_d$ is the multilinear extension of the function that maps $i \in \{0,1\}^{\log n}$ to the value of the ith entry of x.

Proof. We exploit the following explicit expression for $\tilde{V}_d$:

$$\tilde{V}_d(p_1, \ldots, p_{\log n}) = \sum_{\mathbf{b} \in \{0,1\}^{\log n}} V_d(\mathbf{b}) \chi_{\mathbf{b}}(p_1, \ldots p_{\log n}) \tag{6.1}$$

Indeed, it is easy to check that the right hand side of Equation (6.1) is a multilinear polynomial, and that it agrees with V_d on all Boolean inputs. Hence, the right hand side must equal the multilinear extension of V_d.

In particular, by letting $(p_1, \ldots, p_{\log n}) = \omega^*$ in Equation (6.1), we see that

$$\tilde{V}_d(\omega^*) = \sum_{\mathbf{b} \in \{0,1\}^{\log n}} V_d(\mathbf{b}) \chi_{\mathbf{b}}(\omega^*). \tag{6.2}$$

Given any stream update (i, δ), let $(i_1, \ldots, i_{\log n})$ denote the binary representation of i. Notice that update (i, δ) has the effect of increasing $V_d(i_1, \ldots, i_{\log n})$ by δ, and does not affect $V_d(x_1, \ldots x_{\log n})$ for any $(x_1, \ldots, x_{\log n}) \neq (i_1, \ldots, i_{\log n})$. Thus, $\mathcal{V}$ can compute $\tilde{V}_d(\omega^*)$ incrementally from the raw stream by initializing $\tilde{V}_d(\omega^*) \leftarrow 0$, and processing each update (i, δ) via:

$$\tilde{V}_d(\omega^*) \leftarrow \tilde{V}_d(\omega^*) + \delta \cdot \chi_{(i_1, \ldots, i_{\log n})}(\omega^*).$$

$\mathcal{V}$ only needs to store $\tilde{V}_d(\omega^*)$ and ω^*, which is $O(\log n)$ field elements in total. Moreover, for any i, $\chi_{(i_1, \ldots, i_{\log n})}(\omega^*)$ can be computed in $O(\log n)$ field operations, and thus $\mathcal{V}$ can compute $\tilde{V}_d(\omega^*)$ with one pass over the raw stream, using $O(\log n)$ words of space and $O(\log n)$ field operations per update. □

Recall that the the total communication of the GKR protocol is $O(d(n) \log S(n))$ field elements, where $d(n)$ is the depth of the circuit C. Thus, Lemma 6.3.1 implies the following

theorem.

Theorem 6.3.2. *There are* $(\text{polylog}\, n, \text{polylog}\, n)$ *streaming interactive proof protocols for every function in log-space uniform* NC.

Here, NC is the class of all problems decidable by Boolean circuits of polynomial size and polylogarithmic depth. Any problem in this class also possesses an *arithmetic* circuit of polynomial size and polylogarithmic depth, over a field, and the GKR protocol can be applied to C. NC includes, for example, frequency moments, many fundamental matrix problems (e.g., determinant, product, inverse), and graph problems (e.g., minimum spanning tree, shortest paths) (see [8, Chapter 6]).

We showed in Chapter 3 that any (c_a, c_v) annotated data streaming scheme for the second frequency moment problem requires $c_a \cdot c_v \geq \Omega(n)$, even if the scheme is allowed to be prescient. Thus, Theorem 6.3.2 yields an exponential separation between the streaming interactive proofs model and the (prescient) annotated data streaming model.

Despite its powerful generality, the protocol implied by Theorem 6.3.2 is not optimal for many important functions in streaming and database applications. The remainder of this chapter obtains improved, practical protocols for two fundamental streaming problems: the second frequency moment problem and the heavy hitters problem. These examples are meant to be illustrative, as every one of our *sum-check schemes* developed in Chapter 3 has an interactive variant obtained by using the full-fledged sum-check protocol as presented in Chapter 2 in place of the non-interactive techniques of Chapter 3.

6.4 Second Frequency Moment

Let $\mathbf{x}$ be a stream of length N over a universe of size n, and let f be the frequency vector of $\mathbf{x}$. Recall that the second frequency moment of $\mathbf{x}$, denoted $F_2(\mathbf{x})$, is defined to be

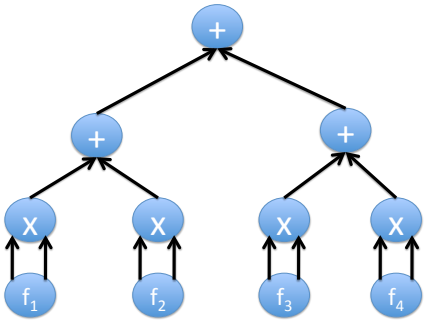


Figure 6.1: A circuit for F_2 on 4 inputs.

$F_2(\mathbf{x}) = \sum_{i \in [n]} f_i^2$.

Let q be a prime larger than N^2, and let $\mathbb{F}_q$ denote the finite field with cardinality q. The function $F_2(\mathbf{x})$ is computed by a natural arithmetic circuit over $\mathbb{F}_q$ of size $O(n)$ and depth $\log n$. This circuit is depicted in Figure 6.1 in the case $n = 4$. Naively applying the GKR protocol to this circuit yields a $(\log^3 n, \log^2 n)$ streaming interactive proof protocol requiring $\Theta(\log^2 n)$ rounds. Using the results of Chapter 7, the honest prover in this protocol could be made to run time $O(n \log n)$.

We give here a much simpler $(\log^2 n, \log^2 n)$ streaming interactive proof protocol for F_2 requiring $\log n$ rounds, based on a direct application of the sum-check protocol to a carefully chosen polynomial. This protocol is illustrative of two techniques that we will use later in this thesis in more general contexts. First, our F_2 protocol makes use of a *quadratic* extension, $\tilde{V}_d^2$, of the function V_d. In contrast, the GKR protocol applied to a circuit C only ever uses *multilinear* extensions of the set of gate values at any layer of C. We crucially exploit the use of quadratic extensions again in Chapter 8 (Theorem 8.7.2).

Second, in Theorem 6.4.2 we show how to implement the prover in our F_2 protocol in time $O(n)$. To accomplish this, we develop a method that allows the prover to reuse work across rounds of the sum-check protocol. We will exploit this technique in a much more

general context in Chapter 8.

Theorem 6.4.1. *There is a* $(\log^2 n, \log^2 n)$ *streaming interactive proof protocol for* F_2 *requiring* $\log n$ *rounds, where* n *is the size of the data universe. The verifier runs in time* $O(N \log n)$*, where* N *is the stream length, assuming all operations in a field of size* $\text{poly}(n)$ *require a unit time.*

Proof. Assume for simplicity that n is a power of 2. Let $\tilde{V}_d$ be as in Lemma 6.3.1. Then $F_2(\mathbf{x}) = \sum_{(i_1,\ldots,i_{\log n}) \in \{0,1\}^{\log n}} \tilde{V}_d^2(i_1, \ldots, i_{\log n})$. This equality holds because $\tilde{V}_d^2(i_1, \ldots, i_{\log n}) = f_i^2$ for all $(i_1, \ldots, i_{\log n}) \in \{0,1\}^{\log n}$, where i is the element of the data universe with binary representation equal to $(i_1, \ldots, i_{\log n})$. Hence, $F_2(\mathbf{x})$ can be computed by applying the sum-check protocol of Proposition 2.3.1 to the polynomial $\tilde{V}_d^2$. In order to perform the final check in this protocol, $\mathcal{V}$ needs to evaluate the polynomial $\tilde{V}_d^2$ at a random point $\mathbf{r} \in \mathbb{F}^{\log n}$. Lemma 6.3.1 implies that $\mathcal{V}$ can evaluate $\tilde{V}_d(\mathbf{r})$ in with a single pass over the data stream $\mathbf{x}$ while storing $O(\log n)$ field elements. $\mathcal{V}$ can then compute $\tilde{V}_d^2(\mathbf{r})$ via the identity $\tilde{V}_d^2(\mathbf{r}) = \left(\tilde{V}_d(\mathbf{r})\right)^2$.

In total, there are $\log n$ rounds of the sum-check protocol, each requiring $\mathcal{P}$ to transmit $O(1)$ field elements. Thus, the total communication cost of the protocol is $O(\log^2 n)$ bits and the total space usage is $O(\log^2 n)$ bits. The verifier's runtime follows from Lemma 6.3.1. □

Theorem 6.4.2. *Given a stream of length* N *over a data universe of size* n*, the honest prover in the streaming interactive proof protocol of Theorem 6.4.1 can be implemented to run in time* $O(\min\{N \log n, n\})$*, assuming all operations in a field of size* $\text{poly}(n)$ *require a unit time.*

Proof. In the jth round of the sum-check protocol applied to $\tilde{V}_d^2$, $\mathcal{P}$ must specify a polynomial of the form

$$g_j(X_j) = \sum_{(x_{j+1},\ldots,x_{\log n}) \in \{0,1\}^{\log n - j}} g(r_1, \ldots, r_{j-1}, X_j, x_{j+1}, \ldots, x_{\log n}),$$

where $(r_1 \dots r_{j-1}) \in \mathbb{F}^{j-1}$ are the values to which variables 1 through $j-1$ were bound in the first $j-1$ rounds observe the protocol. Observe that $g_j(X_j)$ is a polynomial of degree 2, so it is sufficient for $\mathcal{P}$ to evaluate $g_j(x_j)$ at three locations, say at $x_j = 0, 1, 2$, to determine $g_j(x_j)$. For a location $x_j = c$, we rewrite:

$$\begin{aligned}
g_j(c) &= \sum_{(x_{j+1},\dots,x_{\log n})\in\{0,1\}^{\log n-j}} \tilde{V}_d^2(r_1,\dots,r_{j-1},c,x_{j+1},\dots x_{\log n}) \\
&= \sum_{(x_{j+1},\dots,x_{\log n})\in\{0,1\}^{\log n-j}} \left(\sum_{\mathbf{b}\in\{0,1\}^{\log n}} V_d(\mathbf{b})\chi_{\mathbf{b}}(r_1,\dots,r_{j-1},c,x_{j+1},\dots x_{\log n})\right)^2 \\
&= \sum_{(x_{j+1},\dots,x_{\log n})\in\{0,1\}^{\log n-j}} \sum_{(\mathbf{b}_1,\mathbf{b}_2)\in\{0,1\}^{\log n}\times\{0,1\}^{\log n}} V_d(\mathbf{b}_1)V_d(\mathbf{b}_2) \\
&\qquad \cdot \chi_{\mathbf{b}_1}(r_1,\dots,r_{j-1},c,x_{j+1},\dots x_{\log n})\cdot\chi_{\mathbf{b}_2}(r_1,\dots,r_{j-1},c,x_{j+1},\dots x_{\log n}) \\
&= \sum_{(\mathbf{b}_1,\mathbf{b}_2)\in\{0,1\}^{\log n}\times\{0,1\}^{\log n}} \Bigg(V_d(\mathbf{b}_1)V_d(\mathbf{b}_2)\left(\prod_{k=1}^{j-1}\chi_{\mathbf{b}_{1,k}}(r_k)\right)\cdot\chi_{\mathbf{b}_{1,j}}(c)\cdot\left(\prod_{k=1}^{j-1}\chi_{\mathbf{b}_{2,k}}(r_k)\right)\cdot \\
&\qquad \chi_{\mathbf{b}_{2,j}}(c)\cdot\sum_{(x_{j+1}\dots x_{\log n})\in\{0,1\}^{\log n-j}}\Big(\prod_{k=j+1}^{d}\chi_{\mathbf{b}_{1,k}}(x_k)\chi_{\mathbf{b}_{2,k}}(x_k)\Big)\Bigg).
\end{aligned}$$

Note that for $x_k \in \{0,1\}$, $\chi_{\mathbf{b}_k}(x_k) = 1$ if $x_k = \mathbf{b}_k$ and 0 otherwise. Thus, for any pair $(\mathbf{b}_1, \mathbf{b}_2) \in \{0,1\}^{\log n} \times \{0,1\}^{\log n}$, we have

$$\sum_{(x_{j+1},\dots,x_{\log n})\in\{0,1\}^{\log n-j}} \left(\prod_{k=j+1}^{d}\chi_{\mathbf{b}_{1,k}}(x_k)\chi_{\mathbf{b}_{2,k}}(x_k)\right) = 1$$

if and only if $\forall j+1 \le k \le d : \mathbf{b}_{1,k} = \mathbf{b}_{2,k}$, and 0 otherwise. Thus,

$$\begin{aligned}
g_j(c) &= \sum_{(\mathbf{b}_1,\mathbf{b}_2)\in\{0,1\}^{\log n}\times\{0,1\}^{\log n},\forall j+1\le k\le d:\mathbf{b}_{1,k}=\mathbf{b}_{2,k}} \Big(V_d(\mathbf{b}_1)V_d(\mathbf{b}_2)\prod_{k=1}^{j-1}\chi_{\mathbf{b}_{1,k}}(r_k) \\
&\qquad \cdot\chi_{\mathbf{b}_{1,j}}(c)\prod_{k=1}^{j-1}\chi_{\mathbf{b}_{2,k}}(r_k)\chi_{\mathbf{b}_{2,j}}(c)\Big) \\
&= \sum_{(\mathbf{b}_{j+1},\dots,\mathbf{b}_{\log n})\in\{0,1\}^{\log n-j}} \left(\sum_{(\mathbf{b}_1,\dots,\mathbf{b}_j)\in\{0,1\}^j} V_d(\mathbf{b})\chi_{\mathbf{b}_j}(c)\prod_{k=1}^{j-1}\chi_{\mathbf{b}_k}(r_k)\right)^2.
\end{aligned}$$

$\mathcal{P}$ can maintain $V_d(\mathbf{b})\prod_{k=1}^{j-1}\chi_{\mathbf{b}_k}(r_k)$ for each nonzero $V_d(b)$, updating with the new r_k in each round as it is revealed in constant time. Thus the total time spent by the prover for the

verification process can be bounded via $O(m \log n) = O(N \log n)$, where m is the number of nonzero $V_d(\mathbf{b})$'s. However, we also want to bound $\mathcal{P}$'s runtime by $O(n)$, and to accomplish this we observe that it is possible for $\mathcal{P}$ to "reuse work" between rounds.

Indeed, at the heart of $\mathcal{P}$'s computation in round j a summation over $\{0,1\}^j$ for each $(\mathbf{b}_{j+1}, \ldots, \mathbf{b}_{\log n}) \in \{0,1\}^{\log n - j}$. Notice that for each $(\mathbf{b}_{j+1}, \ldots, \mathbf{b}_{\log n}) \in \{0,1\}^{\log n - j}$,

$$\sum_{(\mathbf{b}_1,\ldots,\mathbf{b}_j)\in\{0,1\}^j} \left(V_d(\mathbf{b}) \chi_{\mathbf{b}_j}(c) \prod_{k=1}^{j-1} \chi_{\mathbf{b}_k}(r_k) \right)$$
$$= \sum_{\mathbf{b}_j=0}^{1} \left(\chi_{\mathbf{b}_j}(c) \cdot \sum_{\mathbf{b}_1,\ldots,\mathbf{b}_{j-1}\in\{0,1\}^{j-1}} \left(V_d(\mathbf{b}) \prod_{k=1}^{j-1} \chi_{\mathbf{b}_k}(r_k) \right) \right)$$

And for each $(\mathbf{b}_j, \ldots, \mathbf{b}_{\log n}) \in \{0,1\}^{\log n - j + 1}$, we can decompose

$$\sum_{(\mathbf{b}_1,\ldots,\mathbf{b}_{j-1})\in\{0,1\}^{j-1}} \left(V_d(\mathbf{b}) \prod_{k=1}^{j-1} \chi_{\mathbf{b}_k}(r_k) \right)$$
$$= \sum_{\mathbf{b}_{j-1}=0}^{1} \left(\chi_{\mathbf{b}_{j-1}}(r_{j-1}) \sum_{(\mathbf{b}_1,\ldots,\mathbf{b}_{j-2})\in\{0,1\}^{j-2}} \left(V_d(\mathbf{b}) \prod_{k=1}^{j-2} \chi_{\mathbf{b}_k}(r_k) \right) \right).$$

By storing $A_j[\mathbf{b}_j \ldots \mathbf{b}_{\log n}] = \sum_{(\mathbf{b}_1 \ldots \mathbf{b}_{j-1})\in\{0,1\}^{j-1}} \left(\mathcal{V}_d(\mathbf{b}) \prod_{k=1}^{j-1} \chi_{\mathbf{b}_k}(r_k) \right)$, $\mathcal{P}$ computes

$$A_{j+1}[\mathbf{b}_{j+1} \ldots \mathbf{b}_{\log n}] = \chi_0(r_j) A_j[0, \mathbf{b}_{j+1} \ldots \mathbf{b}_{\log n}] + \chi_1(r_j) A_j[1, \mathbf{b}_{j+1}, \ldots \mathbf{b}_{\log n}]$$

in time $O(n/2^j)$. Thus, over all $\log n$ rounds of the sum-check protocol, $\mathcal{P}$'s total runtime can be bounded by $O(\sum_{j=1}^{\log n} n/2^j) = O(n)$ as desired. □

Experimental Results. We performed a brief experimental study to evaluate the practical efficiency of the F_2 protocol of Theorem 6.4.2 and more generally of the sum-check techniques on which it is based. We compared the multi-round protocol of Theorem 6.4.2 to the non-interactive protocol given in Theorem 3.5.2. We built a prototype implementation in C++: it simulated the computations of both parties, and measured the resources consumed by the protocols. All programs were compiled with g++ using the -O3 optimization flag. For the

data, we generated synthetic streams where the number of occurrences of each item was picked uniformly in the range $[0, 1000]$. Note that the choice of data does not affect the efficiency of the protocols, as these costs only depend on the universe size n and the size of the field over which we work. The computations were made over the field of size $q = 2^{61} - 1$, giving a probability of $2 \log n/q$ of the verifier being fooled by a dishonest prover, which is less than 10^{-17} for all parameter values experimented upon. These computations were executed using native 64-bit arithmetic. The failure probability could be reduced further to, e.g., $127/(2^{127} - 1)$, at the cost of using 128 bit arithmetic (using a larger field size would also be necessary to avoid "overflow" if the second frequency moment of the data stream can be larger than $2^{61} - 1$, as when operating over a field of cardinality q for prime q, the protocol effectively computes the second frequency moment modulo q).

We evaluated the protocols on a single core of a multi-core machine with 64-bit AMD Opteron processors and 32 GB of memory available. The large amount of memory allowed us to experiment with data universes containing several billion items, with the prover able to store the entire frequency vector in memory. We measured the time for $\mathcal{V}$ to compute the check information from the stream, for $\mathcal{P}$ to generate the proof, and for $\mathcal{V}$ to verify this proof. We also measured the space required by $\mathcal{V}$, and the size of the proof provided by $\mathcal{P}$.

Figures 6.2, 6.3, and 6.4 show the behavior of the protocols as the size of the domain n varies. First, Figure 6.2 shows the time for $\mathcal{V}$ to process the stream to evaluate $\tilde{V}_d(\mathbf{r})$ as the domain size increases. Both show a linear trend (here, plotted on log-log scale). Moreover, both take roughly the same time (within a factor of two), with the multi-round verifier processing about 21 million updates per second, and the single round $\mathcal{V}$ processing 35 million. The similarity is not surprising: both methods are taking each element of the stream and computing the product of the frequency with a function of the element's index i and the random parameter $\mathbf{r}$. The effort in computing this function is roughly similar in

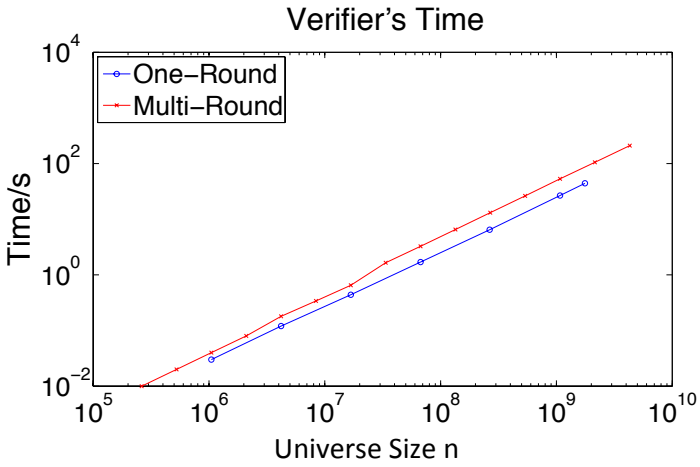


Figure 6.2: Verifier's time.

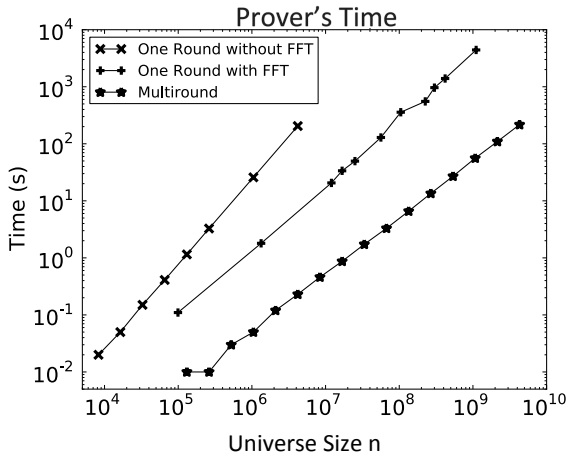


Figure 6.3: Prover's time.

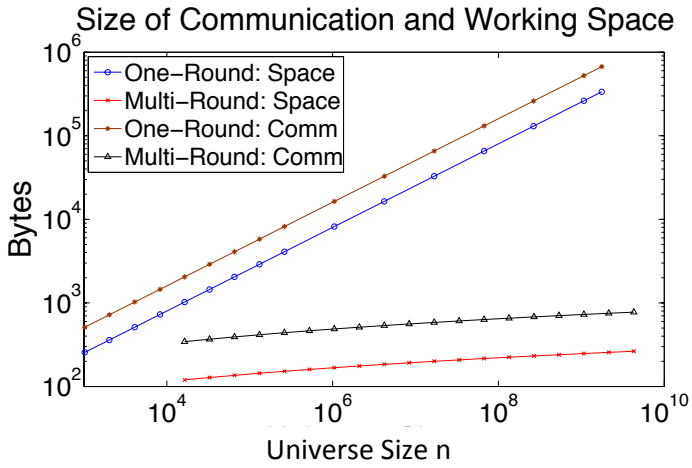


Figure 6.4: Space and Communication complexity

both cases. The single round $\mathcal{V}$ has a slight advantage, since it can compute and use lookup tables within the $O(\sqrt{n})$ space bound, while the multi-round verifier, limited to logarithmic space, must recompute some values multiple times. The time to check the proof is essentially negligible: less than a millisecond across all data sizes. Hence, we do not consider this a significant cost.

Figure 6.3 shows a clear separation between the two methods in $\mathcal{P}$'s effort in generating the proof. Here, we measure total time across all rounds in the multi-round case, and the time to generate the single round proof. For the single-round proof, we plot both the cost of a naive prover implementation that does not utilize the FFT techniques of Corollary 3.12.2 and an implementation that does use the FFT techniques of Corollary 3.12.2. For large streams, it is clear that the the prover in the non-interactive protocol is only scalable if FFT techniques are used, as in the naive implementation $\mathcal{P}$'s runtime growing like $n^{3/2}$; this implementation failed to process streams over a universe of size larger than 40 million. The FFT-based implementation of the non-interactive protocol processed between $350,000$

and $750,000$ items per second for all tested values of n, even for values of n well into the billions. Still, the cost in the multi-round case is dramatically lower: our multi-round prover implementation based on Theorem 6.4.2 processed 20-21 million universe items per second.

The trend is similar for the space resources required to execute the protocol. In the single round case, both the verifier's space and size of the proof grow proportional to $\sqrt{n}$. This is not impossibly large: Figure 6.4 shows that for u of the order of 1 billion, both these quantities are comfortably under a megabyte. Nevertheless, it is still orders of magnitude larger than the sizes seen in the multi-round protocol: there, the space required and proof size are never more than 1KB even when handling gigabytes of data.

In summary, the methods we have developed in this chapter are applicable to genuinely large data sets, defined over a domain of size in the billions. Our implementation is capable of processing such datasets within a matter of seconds or minutes.

6.5 Heavy Hitters

We close this chapter by presenting a highly efficient protocol for the fundamental problem of computing the ϕ-heavy hitters of a data stream. Recall that in this problem, the goal is to list those items i such that $f_i \geq T = \phi N'$, i.e. whose frequency of occurrence exceeds a ϕ fraction of the total count $N' = \sum_{i \in [n]} f_i$. The protocol we give is the natural interactive variant of the online annotated data streaming scheme of Theorem 3.6.1.

Theorem 6.5.1. *There is a $(\phi^{-1} \log^2 n + \log^2 n, \log^2 n)$ streaming interactive proof protocol requiring $O(\log n)$ rounds for identifying the ϕ-heavy hitters in a data stream over a data universe of size n. The verifier runs in time $O(N \log n)$ and the prover runs in time $O(\min\{N \log n, n\})$, and N is the stream length.*

Proof. Let W be the witness set from Theorems 3.4.1 and 3.6.1. Recall that W is a subset of

the nodes of a binary tree $\mathcal{T}$ imposed over the data universe, and in the schemes of Theorems 3.4.1 and 3.6.1, $\mathcal{P}$ sends to $\mathcal{V}$ a claimed value for $\hat{f}(w) = \sum_{i \in L(w)} f_i$, where $L(w)$ denotes the set of all leaves in the subtree rooted at W. The verifier must check that W, $\hat{f}(\ell) > T$ for all leaves $\ell \in W$, and that $\hat{f}(u) \leq T$ for all all non-leaf nodes $u \in W$.

We show how $\mathcal{V}$ can check that $\hat{f}(w)$ is as claimed for *all* items $w \in W$, at the same cost as a single point query. Let z denote the $2n-1$-dimensional vector such that $z_w = 1$ if $w \in W$, and $z_w = 0$ otherwise. Let f^* denote the $2n-1$-dimensional vector such that f^*_w equals the claimed value of $\hat{f}(w)$ if $w \in W$, and $f^*_w = 0$ otherwise. Abusing notation, we will also think of $\hat{f}$ itself as a $2n-1$-dimensional vector such that $\hat{f}_w = \hat{f}(w)$. As in the protocol of Theorem 3.6.1, we observe that $\hat{f}(w) = f^*_w$ for all $w \in W$ if and only if $0 = \sum_{j \in [2n-1]} z_j (\hat{f}_w - f^*_w)^2$.

Let $\mathbb{F}$ be a field of cardinality q for some prime $q > 2(N')^2$. We now view the vectors z, f^*, and $\hat{f}$ as functions mapping $\{0,1\}^{\log n+1}$ to $\mathbb{F}$: each of these functions take as input a Boolean vector $(i_1, \ldots, i_{\log n+1})$, interprets this vector as an integer i between 0 and $2n-1$, and outputs the ith entry of the corresponding vector. Let $\tilde{z}$, $\tilde{f^*}$, and $\tilde{\hat{f}}$ respectively denote the multilinear extension of each of these functions. Then in order to compute $\sum_{j \in [2n-1]} z_j (\hat{f}_w - f^*_w)^2$, it suffices to apply the sum-check protocol to the polynomial $g = \tilde{z} \cdot (\tilde{\hat{f}} - \tilde{f^*})^2$. The total number of rounds in the sum-check protocol applied to g is $\log n + 1$. Since g has degree 3 in each variable, each of $\mathcal{P}$'s messages consists of 4 field elements, and so the total communication cost of the sum-check protocol applied to G is $O(\log^2 n)$ bits. Thus, the total communication of the protocol as a whole is $O(\phi^{-1} \log^2 n)$ bits to specify the elements w of the witness set W as well as the claimed values for $\hat{f}(w)$, and another $O(\log^2 n)$ bits to apply the sum-check protocol to the polynomial g.

To perform the required check in the final round of the sum-check protocol, $\mathcal{V}$ must evaluate g at a random point $\mathbf{r} \in \mathbb{F}^{\log n+1}$. $\mathcal{V}$ can do this by evaluating $\tilde{z}(\mathbf{r})$, $\tilde{\hat{f}}(\mathbf{r})$, and $\tilde{f^*}(\mathbf{r})^2$.

As long as $\mathcal{V}$ stores the random vector $\mathbf{r}$, which requires $O(\log|\mathbb{F}|\log n)$ bits of space, each of these three quantities can be computed by $\mathcal{V}$ with a single streaming pass over the input using the observations of Lemma 6.3.1. For example, given any update (i,δ) to the ith entry of the vector $\hat{f}$, let $(i_1,\ldots,i_{\log n+1})$ denote the binary representation of i. $\mathcal{V}$ can compute $\tilde{\hat{f}}(\mathbf{r})$ incrementally from the stream of updates to $\hat{f}$ by initializing $\tilde{\hat{f}}(\mathbf{r}) \leftarrow 0$, and processing each update (i,δ) via:

$$\tilde{\hat{f}}(\mathbf{r}) \leftarrow \tilde{\hat{f}}(\mathbf{r}) + \delta \cdot \chi_{(i_1,\ldots,i_{\log n+1})}(\mathbf{r}).$$

The bound on the verifier's runtime follows from Lemma 6.3.1, while the bound on the prover's runtime can be derived using the techniques of Theorem 6.4.2. □

6.6 Discussion

In this chapter, we introduced the model of streaming interactive proofs and demonstrated that streaming interactive proofs require exponentially less space and communication than annotated data streaming protocols for a large class of problems. We also observed that the GKR protocol can be made to work with a streaming verifier and presented improved protocols for two specific problems of fundamental importance in streaming and database processing: F_2 and heavy hitters. These protocols are meant to be illustrative, and although they were tailored to specific problems, they introduced important techniques that we will utilize repeatedly as we develop general-purpose protocols in Chapters 7 and 8.

Chapter 7

Practical Verified Computation with Streaming Interactive Proofs

In this chapter, we revisit the GKR protocol and show how to reduce the runtime of the prover from $\Omega(S^3)$ in a naive implementation down to $O(S \log S)$, where S is the size of an arithmetic circuit computing the function of interest. We also describe a full implementation of the protocol, demonstrating much greater scalability than one might have expected. Finally, we describe a parallel implementation of the protocol that leverages Graphics Processing Units (GPUs) and experimentally demonstrate the GKR protocol's substantial amenability to parallelization.

7.1 Overview and Statement of Results

Recall from Chapter 5 that in the GKR protocol, $\mathcal{P}$ and $\mathcal{V}$ first agree on an arithmetic circuit C of size S and fan-in 2 over a finite field $\mathbb{F}$ computing the function of interest. The protocol proceeds in iterations, with one iteration per layer of C. In the ith iteration, the

sum-check protocol is applied to the polynomial $f_z^{(i)} : \mathbb{F}^{s_i+2s_{i+1}} \to \mathbb{F}$ defined via:

$$f_z^{(i)}(p, \omega_1, \omega_2) =$$
$$\beta_{s_i}(z,p) \cdot \left(\tilde{\text{add}}_i(p, \omega_1, \omega_2)(\tilde{V}_{i+1}(\omega_1) + \tilde{V}_{i+1}(\omega_2)) + \tilde{\text{mult}}_i(p, \omega_1, \omega_2)\tilde{V}_{i+1}(\omega_1) \cdot \tilde{V}_{i+1}(\omega_2) \right), \tag{7.1}$$

where $\beta_{s_i}, \tilde{\text{add}}_i, \tilde{\text{mult}}_i$, and $\tilde{V}_{i+1}$ are as defined in Chapter 5. In the j'th round of this sum-check protocol, $\mathcal{P}$ is required to send the univariate polynomial

$$g_j(X_j) = \sum_{(x_{j+1}, \ldots, x_{s_i+2s_{i+1}}) \in \{0,1\}^{s_i+2s_{i+1}-j}} f_z^{(i)}(r_1^{(i)}, \ldots, r_{j-1}^{(i)}, X_j, x_{j+1}, \ldots, x_{s_i+2s_{i+1}}).$$

The sum defining g_j involves as many as S^3 terms, and thus a naive implementation of $\mathcal{P}$ would require $\Omega(S^3)$ time per iteration of the protocol. However, we show that by exploiting the *multilinearity* of the low-degree extensions $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ that we use in the definition of f_i, each gate at layer i contributes to exactly one term in the sum defining g_j, as does each gate at layer $i+1$.[1] Thus, the polynomial g_j can be computed with a single pass over the gates at layer i, and a single pass over the gates at layer $i+1$. As the sum-check protocol requires $O(s_i + s_{i+1}) = O(\log S(n))$ messages for each layer of the circuit, $\mathcal{P}$ requires logarithmically many passes over each layer of the circuit in total.

A complication in applying the above observation is that $\mathcal{V}$ must process the circuit in order to pull out information about its structure necessary to check the validity of $\mathcal{P}$'s messages. Specifically, each application of the sum-check protocol requires $\mathcal{V}$ to evaluate $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ at a random point. Theorem 7.1.1 below follows from the fact that for any log-space uniform circuit, $\mathcal{V}$ can evaluate the multilinear extension of the wiring predicates at any point using $O(\log S(n) \log |\mathbb{F}|)$ bits of space. We present detailed proofs and discussions of the following theorems in Section 7.2.

[1] In order to obtain an interactive proof protocol for log-space uniform NC in which the prover runs in polynomial time and the verifier runs in quasi-linear time, Goldwasser, Kalai, and Rothblum used *polylogarithmic* degree extensions of add_i and mult_i. In contrast, we use the multilinear extensions $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$. Our use of multilinear extensions is essential in lowering the prover's runtime from $O(S^3)$ to $O(S \log S)$.

Theorem 7.1.1. *For any log-space uniform circuit C of size $S(n)$ over finite field $\mathbb{F}$, $\mathcal{P}$ can run in $O(S(n)\log S(n))$ time over the entire execution of the GKR protocol applied to C, and $\mathcal{V}$ can make a single streaming pass over the input, using $O(\log S(n)\log|\mathbb{F}|)$ bits of space over the entire execution of the protocol.*

Moreover, we can strengthen Theorem 7.1.1 as follows. Because the circuit's wiring predicate is independent of the input, we can separate $\mathcal{V}$'s computation into an offline non-interactive preprocessing phase, which occurs before the data stream is seen, and an online interactive phase, which occurs after both $\mathcal{P}$ and $\mathcal{V}$ have seen the input. This is similar to [57, Theorem 4] and ensures that $\mathcal{V}$ is space-efficient (but may require time $O(S(n))$ during the offline phase), and that $\mathcal{P}$ is *both* time- and space-efficient in the online interactive phase. In order to determine which circuit to use, $\mathcal{V}$ does need to know (an upper bound on) the length of the input during the preprocessing phase.

Theorem 7.1.2. *For any log-space uniform circuit C of size $S(n)$ and depth $d(n)$ over finite field $\mathbb{F}$, $\mathcal{P}$ can run in $O(S(n)\log S(n))$ total time over the entire execution of the GKR protocol applied to C. $\mathcal{V}$ can make a single streaming pass over the input, using $O(d(n)\log S(n)\log|\mathbb{F}|)$ bits of space over the entire execution of the protocol. $\mathcal{V}$ can run in time $O(S(n))$ using space $O(d(n)\log S(n)\log|\mathbb{F}|)$ in a non-interactive, data-independent preprocessing phase, and run in time $O(n\log n + d(n)\log S(n))$ using $O(d(n)\log S(n)\log|\mathbb{F}|)$ bits of space in an online interactive phase, where the $O(n\log n)$ term is due to the time required to evaluate the low-degree extension of the input at a point.*

Finally, Theorem 7.1.3 follows by assuming $\mathcal{P}$ can evaluate the multilinear extension of the wiring predicate quickly. We believe that the hypothesis of Theorem 7.1.3 is extremely mild, and we discuss this point at length in Section 7.3, identifying a diverse array of circuits to which Theorem 7.1.3 applies. Moreover, the solutions we adopt in our circuit-checking

experiments of Section 7.4 correspond to Theorem 7.1.3, and are both space- and time-efficient for the verifier.

Theorem 7.1.3. *Let C be any log-space uniform circuit of size $S(n)$ and depth $d(n)$ over finite field $\mathbb{F}$, and assume that for all $i \in \{1, \ldots, d(n)\}$, there exists a $O(\log S(n) \log |\mathbb{F}|)$-space, $\text{poly}(\log S(n))$-time algorithm for evaluating $\tilde{add}_i$ and $\tilde{mult}_i$ at a point. Then in order to to implement the GKR protocol applied to C, $\mathcal{P}$ requires $O(S(n) \log S(n))$ time, and $\mathcal{V}$ requires $O(\log S(n) \log |\mathbb{F}|)$ bits of space and time $O(n \log n + d(n)\text{poly}(\log S(n)))$, where the $O(n \log n)$ term is due to the time required to evaluate the low-degree extension of the input at a point.*

7.2 Methodology and Proofs

7.2.1 Overview

In the j'th round of the sum-check protocol applied to $f_z^{(i)}$, $\mathcal{P}$ is required to send the univariate polynomial

$$g_j(X_j) = \sum_{(x_{j+1}, \ldots x_{s_i+2s_{i+1}}) \in \{0,1\}^{s_i+2s_{i+1}-j}} f_z^{(i)}(r_1^{(i)}, \ldots, r_{j-1}^{(i)}, X_j, x_{j+1}, \ldots, x_{s_i+2s_{i+1}}).$$

Theorems 7.1.1, 7.1.2, and 7.1.3 rely on the observation that, when $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ are multilinear extensions, rather than arbitrary low-degree extensions, then each gate at layers i and $i+1$ contributes to exactly one term in the sum.

More specifically, the key observation is that the multilinear extension of the wiring predicate acts as a sum of *variable-wise* indicator functions on boolean-valued variables, with one indicator function for each gate at the layer of interest. At any round j of the sum-check protocol, the "unbound" variables (i.e., those appearing in the sum defining g_j) still only range over values in $\{0, 1\}$, and thus each gate $\mathbf{b}$ at the current layer of the circuit

still contributes to only one term in the sum in intermediate rounds. Namely, $\mathbf{b}$ contributes to the unique term of the sum that agrees with the trailing bits in the binary representation of $\mathbf{b}$, despite the fact that "bound" variables may take values outside of $\{0,1\}$.

7.2.2 Decomposing $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ as Sums of Variable-Wise Indicator Functions

Since $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ are the multilinear extensions of the wiring predicate, we can write them explicitly as follows.

As in Chapter 6, for $\mathbf{b} \in \{0,1\}^{s_i+2s_{i+1}}$ let $\chi_{\mathbf{b}}(x_1,\ldots,x_{s_i+2s_{i+1}}) = \prod_{k=1}^{s_i+2s_{i+1}} \chi_{\mathbf{b}_k}(x_k)$, where $\chi_0(x_k) = 1 - x_k$ and $\chi_1(x_k) = x_k$. $\chi_{\mathbf{b}}$ is the unique multilinear polynomial that takes $\mathbf{b} \in \{0,1\}^{s_i+2s_{i+1}}$ to 1 and all other values in $\{0,1\}^{s_i+2s_{i+1}}$ to 0, i.e., it is the multilinear extension of the indicator function for boolean vector $\mathbf{b}$.

Notice that if $(x_{j+1},\ldots,x_{s_i+2s_{i+1}}) \in \{0,1\}^{s_i+2s_{i+1}-j}$, then for any $(r_1,\ldots r_j) \in \mathbb{F}^j$,

$$\chi_{\mathbf{b}}(r_1,\ldots,r_j,x_{j+1},\ldots,x_{s_i+2s_{i+1}}) = \tag{7.2}$$
$$\begin{cases} \prod_{l=1}^{j} \chi_{\mathbf{b}_l}(r_l), & \text{if } x_k = \mathbf{b}_k \text{ for all } k \geq j+1. \\ 0, & \text{otherwise.} \end{cases}$$

Informally, Equation (7.2) implies that one may think of $\chi_{\mathbf{b}}$ acting as a *variable-wise* indicator function on boolean-valued variables. Since $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ are multilinear extensions, they can be written as a sum of these $\chi_{\mathbf{b}}$ functions, where each gate at layer $i+1$ contributes a term $\chi_{\mathbf{b}}$ to the sum. Details follow.

Recall that we numbered the layers of the circuit so that the in-neighbors of a gate at layer i are at layer $i+1$. We will refer to the triple $\mathbf{b} = (\mathbf{b}_1,\mathbf{b}_2,\mathbf{b}_3) \in \{0,1\}^{s_i+2s_{i+1}}$ as a *gate triple* at layer i if the first and second in-neighbors of the gate with label $\mathbf{b}_1$ at layer i are gates $\mathbf{b}_2$ and $\mathbf{b}_3$ at layer $i+1$ respectively. We will say $\mathbf{b} = (\mathbf{b}_1,\mathbf{b}_2,\mathbf{b}_3)$ is an *add triple*

(respectively, *mult triple*) if gate $\mathbf{b}_1$ at layer i is an addition (respectively, multiplication) gate. With this terminology in hand, we may write:

$$\tilde{\text{add}}_i(x_1, \ldots, x_{s_i+2s_{i+1}}) = \sum_{\text{add triples } \mathbf{b} \text{ at layer } i} \chi_{\mathbf{b}}(x_1, \ldots, x_{s_i+2s_{i+1}}) \tag{7.3}$$

and

$$\tilde{\text{mult}}_i(x_1,\ldots,x_{s_i+2s_{i+1}}) = \sum_{\text{mult triples } \mathbf{b} \text{ at layer } i+1} \chi_{\mathbf{b}}(x_1, \ldots, x_{s_i+2s_{i+1}}). \tag{7.4}$$

It is straightforward to observe the expressions on the right hand sides of Equations (7.3) and (7.4) are multilinear polynomials that agree with add_i and mult_i on boolean-valued inputs, and hence the right hand sides are equal to the unique multilinear extensions of add_i and mult_i respectively.

For any vector $\mathbf{x} = (x_{j+1}, \ldots x_{s_i+2s_{i+1}}) \in \{0,1\}^{s_i+2s_{i+1}-j}$, and for any $(r_1, \ldots r_j) \in \mathbb{F}^j$, let $\mathbf{x}^*$ denote the vector

$$\mathbf{x}^* := (r_1, \ldots, r_j, x_{j+1}, \ldots, x_{s_i+2s_{i+1}}) \in \mathbb{F}^{s_i+2s_{i+1}},$$

and let $S_{\mathbf{x}}$ denote the set of gate triples at layer i given by $\{\mathbf{b} \in \{0,1\}^{s_i+2s_{i+1}} : \mathbf{b}_k = x_k \text{ for all } k \geq j+1\}$. Equations 7.3 and 7.4 imply that

$$\tilde{\text{add}}_i(\mathbf{x}^*) = \sum_{\text{add triples } \mathbf{b} \in S_{\mathbf{x}}} \left(\prod_{l=1}^{j} \chi_{\mathbf{b}_l}(r_l) \right), \tag{7.5}$$

and similarly

$$\tilde{\text{mult}}_i(\mathbf{x}^*) = \sum_{\text{mult triples } \mathbf{b} \in S_{\mathbf{x}}} \left(\prod_{l=1}^{j} \chi_{\mathbf{b}_l}(r_l) \right) \tag{7.6}$$

Completing the Calculation. At round j of this sum-check protocol, the prover must compute the message

$$g_j(X_j) = \sum_{x_{j+1}\ldots x_{s_i+2s_{i+1}} \in \{0,1\}^{s_i+2s_{i+1}-j}} f_z^{(i)}(r_1^{(i)}, \ldots, r_{j-1}^{(i)}, X_j, x_{j+1} \ldots x_{s_i+2s_{i+1}}).$$

Since g_j has degree three if we are using multilinear extensions, it suffices for the prover to send $g_j(r_j)$ for $r_j \in \{0,1,2\}$, as these evaluations uniquely define g_j.

Using Equations (7.5) and (7.6), we can now easily observe that each gate at layer i contributes to exactly one term in the sum. Specifically, for any term $\mathbf{x} = (x_{j+1} \dots x_{s_i+2s_{i+1}}) \in \{0,1\}^{s_i+2s_{i+1}-j}$ in the sum, let $\mathbf{x}^*$ denote the vector

$$\mathbf{x}^* := (r_1^{(i)}, \dots, r_j^{(i)}, x_{j+1}, \dots, x_{s_i+2s_{i+1}}) \in \mathbb{F}^{s_i+2s_{i+1}}$$

as before, and let $\mathbf{p}^* \in \mathbb{F}^{s_i}$ be the first s_i entries of this vector, $\omega_1^* \in \mathbb{F}^{s_{i+1}}$ the middle s_{i+1} entries, and $\omega_2^* \in \mathbb{F}^{s_{i+1}}$ the final s_{i+1} entries. Then combining Equations (7.5) and (7.6) with (7.1), we see

$$f_z^{(i)}(\mathbf{x}^*) = \beta_{s_i}(z, p^*) \cdot \left(\left(\sum_{\text{add triples } \mathbf{b} \in S_{\mathbf{x}}} \left(\prod_{l=1}^{j} \chi_{\mathbf{b}_l}(r_l) \right) \right) \left(\tilde{V}_{i+1}(\omega_1^*) + \tilde{V}_i(\omega_2^*) \right) \right.$$
$$\left. + \left(\sum_{\text{mult triples } \mathbf{b} \in S_{\mathbf{x}}} \left(\prod_{l=1}^{j} \chi_{\mathbf{b}_l}(r_l) \right) \right) \cdot \tilde{V}_{i+1}(\omega_1^*) \cdot \tilde{V}_{i+1}(\omega_2^*) \right). \tag{7.7}$$

Each gate triple $\mathbf{b}$ at layer i is in $S_{\mathbf{x}}$ for exactly one $\mathbf{x} \in \{0,1\}^{s_i+2s_{i+1}-j}$. Namely, $\mathbf{x}$ is the boolean vector equal to the last $s_i + 2s_{i+1} - j$ bits of the binary representation of $\mathbf{b}$. Denote this vector by $\mathbf{x}(\mathbf{b})$, and similarly let $\mathbf{x}^*(\mathbf{b})$, $\mathbf{p}^*(\mathbf{b})$, $\omega_1^*(\mathbf{b})$ and $\omega_2^*(\mathbf{b})$ denote the corresponding vectors implied by $\mathbf{x}(\mathbf{b})$.

Equation (7.7) implies that $\mathbf{b}$ contributes only to the term $\mathbf{x}(\mathbf{b})$ of the sum defining $g_j(r_j)$ for $r_j \in \{0,1,2\}$. That is, we may write

$$g_j(r_j) =$$
$$\sum_{\text{add triples } \mathbf{b} \text{ at layer } i} \beta_{s_i}(z, p^*(\mathbf{b})) \left(\prod_{l=1}^{j} \chi_{\mathbf{b}_l}(r_l) \right) (\tilde{V}_{i+1}(\omega_1^*(\mathbf{b})) + \tilde{V}_{i+1}(\omega_2^*(\mathbf{b})))$$
$$+ \sum_{\text{mult triples } \mathbf{b} \text{ at layer } i} \beta_{s_i}(z, p^*(\mathbf{b})) \left(\prod_{l=1}^{j} \chi_{\mathbf{b}_l}(r_l) \right) \cdot \tilde{V}_{i+1}(\omega_1^*(\mathbf{b})) \cdot \tilde{V}_{i+1}(\omega_2^*(\mathbf{b})).$$

Thus, the prover can compute $g_j(0)$, $g_j(1)$, and $g_j(2)$ with a single pass over the gates at layer i. By a similar calculation, all necessary $\tilde{V}_{i+1}(\omega_1)$ and $\tilde{V}_{i+1}(\omega_2)$ for each message of the prover can be computed with a single pass over the gates at layer $i+1$. In conclusion, as long as we use the multilinear extension of the circuit's wiring predicate, the prover can compute each message within the sum-check protocol at layer i with a single pass over the gates at layer i and a single pass over the gates at layer $i+1$, performing a constant number of field operations for each gate. Thus, the prover requires at most $O((S_i + S_{i+1}) \log(S_i + S_{i+1}))$ time during the sum-check protocol as applied to the ith layer of the circuit. It follows that, over the entire execution of the GKR protocol, the prover requires at most $O(S(n) \log S(n))$ time to compute his messages within the various invocations of the sum-check protocol.

7.2.3 Reducing to Verification of a Single Point

Recall from Chapter 5 that in the ith iteration of the GKR protocol, after executing the sum-check protocol, $\mathcal{V}$ is left with the task of verifying both $\tilde{V}_{i+1}(\omega_1^*)$ and $\tilde{V}_{i+1}(\omega_2^*)$. $\mathcal{V}$ reduces these two tasks to the task of verifying $\tilde{V}_{i+1}(\omega^*)$ at a single point $\omega^* \in \mathbb{F}^{s_{i+1}}$. $\mathcal{V}$ does this by asking $\mathcal{P}$ to send a degree-s_{i+1} polynomial claimed to be $\tilde{V}_{i+1} \circ \ell$, the restriction of $\tilde{V}_{i+1}$ to the line ℓ, where ℓ be the unique line $\mathbb{F} \to \mathbb{F}^{s_{i+1}}$ such that $\ell(0) = \omega_1^*$ and $\ell(1) = \omega_2^*$. Thus, to complete our the proof that the prover in the GKR protocol can run in time $O(S(n) \log S(n))$, we must show that $\mathcal{P}$ can compute $\tilde{V}_{i+1} \circ \ell$ in time $O(S_{i+1} \cdot s_{i+1})$, as this will imply that $\mathcal{P}$ spends $O(\sum_i S_i \cdot s_i) = O(S(n) \log S(n))$ time on this calculation across all layers of the circuit.

Since $\tilde{V}_{i+1} \circ \ell$ is a univariate polynomial of degree s_{i+1}, it can be specified by computing its value at $s_{i+1} + 1$ points, say the points in the set $T = \{0, \ldots, s_{i+1}\}$. Thus, it suffices for $\mathcal{P}$ to evaluate $\tilde{V}_{i+1}$ at the s_{i+1} points $\ell(j) : j \in T$.

For any point $p \in \mathbb{F}^{s_{i+1}}$, we may write:

$$\tilde{V}_{i+1}(p) = \sum_{\mathbf{b} \in \{0,1\}^{s_{i+1}}} V_{i+1}(\mathbf{b}) \chi_{\mathbf{b}}(p_1, \ldots p_{s_{i+1}}) \tag{7.8}$$

We will show that $\mathcal{P}$ can compute a table containing $\chi_{\mathbf{b}}(p_1, \ldots p_{s_{i+1}})$ for *all* $\mathbf{b} \in \{0,1\}^{s_{i+1}}$ in $O(S_{i+1})$ time. This fact, combined with Equation 7.8, shows that once the circuit C has been evaluated (thereby yielding the $V_{i+1}(\mathbf{b})$ values), $\mathcal{P}$ can compute $\tilde{V}_{i+1}(p)$ in $O(2^{s_{i+1}}) = O(S_{i+1})$ time. Hence $\mathcal{P}$ can evaluate $\tilde{V}_{i+1}$ at all s_{i+1} points $\ell(j) : j \in T$ in $O(S_{i+1} \cdot s_{i+1})$ time as desired.

To compute all the $\chi_{\mathbf{b}}(p_1, \ldots p_{s_{i+1}})$ values, we use a simple memoization procedure. This procedure will arise again in Lemma 8.3.1. The memoization procedure consists of s_{i+1} stages, where stage j constructs a table $A^{(j)}$ of size 2^j, such that for any $(b_1, \ldots, b_j) \in \{0,1\}^j$, $A^{(j)}[(b_1, \ldots, b_j)] = \prod_{i=1}^{j} \chi_{b_i}(p_i)$. Notice $A^{(j)}[(b_1, \ldots, b_j)] = A^{(j-1)}[(b_1, \ldots, b_{j-1})] \cdot \chi_{b_j}(p_j)$, and so the jth stage of the memoization procedure requires time $O(2^j)$. The total time across all s_{i+1} stages is therefore $O(\sum_{j=1}^{s_{i+1}} 2^j) = O(2^{s_{i+1}}) = O(S_{i+1})$ as desired.

7.2.4 Finishing the Proofs of Theorems 7.1.1 and 7.1.2

We have demonstrated that if the GKR protocol is instantiated with the multilinear extensions of the circuits wiring predicate and gate value function, then $\mathcal{P}$ can be made to run in time $O(S(n) \log S(n))$. All that remains in proving Theorem 7.1.1 is to show that for any log-space uniform circuit, the verifier can evaluate $\tilde{\text{add}}_i(p, \omega_1, \omega_2)$ and $\tilde{\text{mult}}_i(p, \omega_1, \omega_2)$ using $O(\log S(n) \log |\mathbb{F}|)$ bits of space. This holds because the verifier can make an "implicit" pass over each layer of the circuit and compute the contribution of each gate to $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$. That is, the verifier considers each gate triple $\mathbf{b}$ in turn, and computes $\mathbf{b}$'s contribution to $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ using Equations (7.3) and (7.4). The space requirement is $O(\log S(n))$ bits to enumerate the triples and $O(\log S(n) \log |\mathbb{F}|)$ to store the $O(\log S(n))$ field elements comprising (p, ω_1, ω_2). This requires $O(S(n))$ time in total, but only $O(\log S(n) \log |\mathbb{F}|)$ bits

of space, since the verifier never needs to store an explicit representation of the circuit. Theorem 7.1.1 follows.

Theorem 7.1.2 follows from the additional observation that $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ do not depend on the input, nor do the random coins of the verifier, and these coins uniquely determine the points at which $\mathcal{V}$ must evaluate $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$. Thus, $\mathcal{V}$ can toss all her coins in the pre-processing phase and compute the necessary evaluations of $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$. $\mathcal{V}$ stores the answers and the random coins for use in the online phase. In the online phase, $\mathcal{V}$ only needs to spend $O(1)$ time per round of the protocol to check $\mathcal{P}$'s messages for consistency, and thus $\mathcal{V}$ takes time $O(d(n) \log S(n))$ in the online phase.

In streaming contexts, where $\mathcal{V}$ is more space-constrained than time-constrained, this may be acceptable. However, the solutions we adopt in our experimental implementation correspond to the stronger Theorem 7.1.3, which further reduces the space and time costs for the verifier.

7.3 Discussion and Applicability

Theorem 7.1.3 makes the assumption that that the multilinear polynomials $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ can be evaluated quickly by a small-space algorithm. In return for this assumption, we avoid the need for an offline pre-processing phase for the verifier as required in Theorem 7.1.2, which can be cumbersome in practice. We believe this assumption is mild in both theory and practice, and we devote this section to discussing the applicability of Theorem 7.1.3 by demonstrating several natural examples where it applies. In contrast, other work in this area does require preprocessing in all cases – see e.g., [56, 82, 90–92, 104]. We view the ability of our implementation to avoid an offline pre-processing phase under many circumstances as an important advantage of our work.

7.3.1 Motivating Problems

To focus our discussion and experimental study, we describe three key problems that capture different aspects of computation, including data aggregation and linear algebra.

F_2: Given a stream of N elements from $[n]$, compute $\sum_{i \in [n]} f_i^2$ where f_i is the number of occurrences of i in the stream. As discussed in prior chapters, this is also known as the *second frequency moment*.

DISTINCT: Given a stream of N elements from $[n]$, compute the number of *distinct* elements, i.e., the number of i with $f_i \neq 0$, where again f_i is the number of occurrences of i in the stream.

MATMULT: Given a stream defining two $n \times n$ integer matrices A and B, compute the product $A \cdot B$.

For simplicity, we will assume through this chapter that the stream length N and the universe size n are on the same order of magnitude i.e., $m = \Theta(n)$.

All three problems require linear space in the streaming model to solve exactly (although can be solved by space-efficient approximation algorithms [80]).

We demonstrate the applicability of Theorem 7.1.3 by showing that the three motivating problems defined above possess succinct circuits to which Theorem 7.1.3 applies. In [39], we further identify several other important circuits from the algorithmic literature to which Theorem 7.1.3 also applies, but we omit this analysis from this thesis for brevity. In essence, Theorem 7.1.3 applies to any circuit with a "highly regular" wiring pattern; this explains why it applies to such a wide array of circuits. The details in the remainder of the section grow lengthy at times, but the main idea remains clear: Theorem 7.1.3 applies to most circuits that arise in both practical applications and theoretical constructions.

7.3.2 Circuits For Motivating Problems

In this section, we describe natural arithmetic circuits computing our motivating problems. In Section 7.4, we report the performance of our implementation of the GKR protocol when applied to these circuits.

Circuit for F_2: The arithmetic circuit for F_2 appeared in Chapter 6 and is quite straightforward: the inputs are entries of the frequency vector f, and the the first level of the circuit computes the square of the input values, then subsequent levels sum these up pairwise to obtain the sum of all squared values. The total depth d is $O(\log n)$. The GKR protocol applied to this circuit yields a $O(\log^2 n)$ message $(\log^2 n, \log^2 n)$ protocol (as per Definition 6.1.2).

Circuit for Distinct : We describe a succinct arithmetic circuit over $\mathbb{F}_q$, the field of cardinality q, that computes DISTINCT. When q is a prime larger than n, Fermat's Little Theorem (FLT) implies that for $x \in \mathbb{F}_q$, $x^{q-1} = 1$ if and only if $x \neq 0$. Consider the circuit that, for each coordinate i of the input vector f, computes each f_i^{q-1} via $O(\log q)$ multiplications, and then sums the results. This circuit has total size $O(n \log q)$ and depth $O(\log q)$. Applying our implementation of the GKR protocol to this circuit, we obtain a $(\log n \log q, \log n)$ protocol where $\mathcal{P}$ runs in time $O(n \log n \log q)$.

Circuit for MatMult : We describe an arithmetic circuit C of size $O(n^3)$ for multiplying two $n \times n$ matrices A and B – this circuit essentially performs naive matrix multiplication. Let the input gate labelled $(0, i, j) \in \{0,1\} \times \{0,1\}^{\log n} \times \{0,1\}^{\log n}$ correspond to A_{ij}, and the input labelled $(1, i, j) \in \{0,1\} \times \{0,1\}^{\log n} \times \{0,1\}^{\log n}$ correspond to B_{ij}. The layer of C adjacent to the input consists of n^3 gates, where the gate labeled $(i, j, k) \in \{0,1\}^{\log n} \times \{0,1\}^{\log n} \times \{0,1\}^{\log n}$ computes $A_{ik} \cdot B_{kj}$. All subsequent layers constitute a binary tree of addition gates summing up the results and thereby computing $\sum_k A_{ik} B_{kj}$ for all $(i, j) \in [n] \times [n]$.

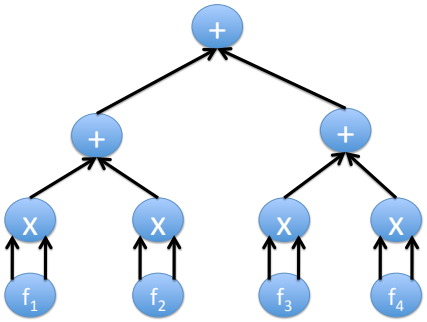


Figure 7.1: A circuit for F_2 on 4 inputs.

Wiring Predicates for F_2, Distinct, and MatMult

We demonstrate that Theorem 7.1.3 applies to natural circuits computing all three of our motivating problems.

1. F_2**:** Recall that the circuit for F_2 had a layer of multiplication gates used for computing the square of each input, and then subsequent levels formed a binary tree of addition gates used to sum up the results. A visual depiction of this circuit on $n = 4$ inputs was provided in Figure 6.1, and is reproduced in Figure 7.1 for the reader's convenience.

   First, consider layer $d-1$ immediately above the input gates, which consists of multiplication gates used to square each input; both the in-neighbors of gate i at layer $d-1$ are equal to the i'th input gate. Therefore, if $p = (p_1, \ldots, p_{\log n}) \in \{0,1\}^{\log n}$ denotes the boolean representation of a gate at layer $d-1$, and $\omega_1 = (\omega_{1,1}, \ldots, \omega_{1,\log n}) \in \{0,1\}^{\log n}$ and $\omega_2 = (\omega_{2,1}, \ldots, \omega_{2,\log n}) \in \{0,1\}^{\log n}$ denote the boolean representation of two gates at the input layer, then mult_d evaluates to true if and only if $p = \omega_1 = \omega_2$, while add_d is identically zero. It is easily seen that the multilinear extension of mult_d is the polynomial

$$\tilde{\text{mult}}_d(p, \omega_1, \omega_2) = \prod_{j=1}^{v} \big(p_j \omega_{1,j} \omega_{2,j} + (1 - p_j)(1 - \omega_{1j})(1 - \omega_{2,j})\big),$$

while the multilinear extension of add_d is the zero polynomial. Clearly, $\tilde{\text{mult}}_d$ can be evaluated at any point in $\mathbb{F}^{3\log n}$ in time and space $O(\log n)$.

The rest of the circuit for F_2 consists of a binary tree of addition gates, which is used to sum up the squared item frequencies. Thus, $\tilde{\text{mult}}_i$ is the zero polynomial for all $i < d$. Meanwhile, for $i < d$ the predicate $\text{add}_i(p_1, \omega_1, \omega_2)$ is as follows. Label the gates at layers i and $i+1$ in the natural way, so that the first input to the gate labelled $p = (p_1, \ldots, p_{s_i}) \in \{0,1\}^{s_i}$ at layer i is the gate with label $(p, 0)$ at layer $i+1$, and the second input to gate p has label $(p, 1)$. Here and throughout, $(p, 0)$ denotes the (s_i+1)-dimensional vector obtained by concatenating the entry 0 to the end of the vector p. Interpreting $p = (p_1, \ldots, p_{s_i}) \in \{0,1\}^{s_i}$ as an integer between 0 and $2^{s_i} - 1$ with p_1 as the high-order bit and p_{s_i} as the low-order bit, this says that the first in-neighbor of p is $2p$ and the second is $2p+1$, when interpreting the binary vector p as an integer. Thus, we may write:

$$\tilde{\text{add}}_i(p, \omega_1, \omega_2) = (1 - \omega_{1,s_{i+1}})\omega_{2,s_{i+1}} \cdot \prod_{j=1}^{s_i - 1} (p_j \omega_{1,j} \omega_{2,j} + (1-p_j)(1-\omega_{1,j})(1-\omega_{2,j}))\,.$$

Conceptually, the leading factor $(1 - \omega_{1,s_{i+1}})\omega_{2,s_{i+1}}$ ensures that ω_1 is even (i.e., its first bit is 0) and ω_2 is odd (i.e., its first bit is 1), while the expression

$$\prod_{j=1}^{s_{i+1}-1} (p_j \omega_{1,j} \omega_{2,j} + (1-p_j)(1-\omega_{1,j})(1-\omega_{2,j}))$$

ensures that the high-order s_i bits of ω_1 and ω_2 agree with the bits of p. $\tilde{\text{add}}_i$ is therefore the unique multilinear polynomial evaluating to 1 on boolean inputs (p, ω_1, ω_2) if $\omega_1 = 2p$ and $\omega_2 = 2p+1$, and evaluating to 0 otherwise. Clearly $\tilde{\text{add}}_i$ can be evaluated at any point in time and space $O(s_i + s_{i+1}) = O(\log n)$. This completes the description of $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ for all layers of the circuit for F_2.

2. DISTINCT: Recall that for each of the n inputs f_j, the circuit for DISTINCT from Section 7.2 computes f_j^{q-1} via $O(\log q)$ multiplications, and then sums the results via a binary

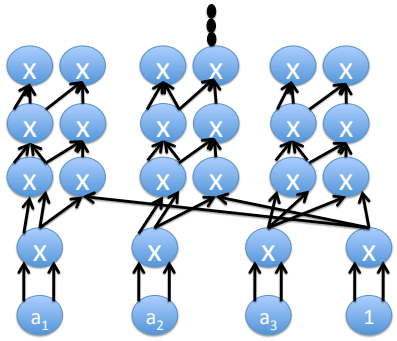


Figure 7.2: The first several layers of a circuit for DISTINCT on three inputs (in place of a fourth input is a "constant" gate with value one) over the field $\mathbb{F}_q$ with $q = 2^{61} - 1$. The first layer from the bottom computes f_j^2 for all j. The second layer from the bottom computes f_j^4 and f_j^2 for all j. The third layer computes f_j^8 and $f_j^6 = f_j^4 \times f_j^2$ for all j, while the fourth layer computes f_j^{16} and $f_j^{14} = f_j^8 \times f_j^6$ for all j. The remaining layers (not shown) have structure identical to the third and fourth layers until the value f_j^{q-1} is computed for all j, and the circuit culminates in a binary tree of addition gates.

tree of addition gates. We have already seen the wiring predicate for binary trees, so here we only sketch the wiring predicate for the f_j^{q-1} computation, omitting some details for brevity. We do so for the special case of $q = 2^{61} - 1$, which is the value of q used in our experiments, as this happens to have a particularly "regular" circuit for computing f_j^{q-1}; the calculation would be similar but less symmetric for other values of q.

We may write $q - 1 = 2^{61} - 2$, whose binary representation is 60 1s followed by a 0. Thus, $f_j^{q-1} = \prod_{k=1}^{60} f_j^{2^k}$. The circuit computing f_j^{q-1} repeatedly squares f_j, and multiplies together the results "as it goes". In more detail, for $i > 1$ there are two multiplication gates at each layer $d - i$ of the circuit for computing f_j^{q-1}; the first computes $f_j^{2^i}$ by squaring the corresponding gate at layer $d - i + 1$, and the second computes $\prod_{k=1}^{i-1} f_j^{2^{k-1}}$. See Figure 7.2 for a visual depiction of the first few layers of the DISTINCT circuit.

At a high level then, the wiring predicate $\tilde{\text{mult}}_i(p, \omega_1, \omega_2)$ tests equality of ω_1 and ω_2 with two strings that depend on the parity of p, as even values of p correspond to gates computing $f_i^{2^j}$ while odd values correspond to gates computing $\prod_{k=1}^{i-1} f_j^{2^{k-1}}$. Thus, we may write

$$\tilde{\text{mult}}_i(p, \omega_1, \omega_2) = (1 - p_1)\chi_{\text{even}}(p, \omega_1, \omega_2) + p_1\chi_{\text{odd}}(p, \omega_1, \omega_2),$$

where χ_{odd} and χ_{even} are multilinear extensions of the appropriate equality predicates, which do not depend on p_1 (we omit a precise definition of χ_{odd} and χ_{even} for brevity, as this circuit is studied in considerable detail in Section 8.4.3). This can clearly be evaluated in $O(\log n)$ time and space.

3. MATMULT: Recall that the circuit for MATMULT consists of a layer (layer $\ell = \log n + 1$) of multiplication gates in which the gate with label $(i, j, k) \in \{0, 1\}^{\log n} \times \{0, 1\}^{\log n} \times$

$\{0,1\}^{\log n}$ is connected to the input gates with labels $(0,i,k) \in \{0,1\} \times \{0,1\}^{\log n} \times \{0,1\}^{\log n}$ and $(1,k,j) \in \{0,1\} \times \{0,1\}^{\log n} \times \{0,1\}^{\log n}$, followed by a binary of tree of addition gates of depth $\log n$. We have already analyzed the wiring predicate for a binary tree of addition gates in the context of F_2, so here we focus on the wiring predicate of the remaining layer, layer ℓ. Since layer ℓ consists only of multiplication gates, $\tilde{\text{add}}_\ell$ is the zero polynomial. Meanwhile, we can express $\tilde{\text{mult}}_\ell$ as follows.

$$\tilde{\text{mult}}(p,\omega_1,\omega_2) = (1-\omega_{1,1})\omega_{2,1}\left(\prod_{k=1}^{\log n}(p_k\omega_{1,k+1} + (1-p_k)(1-\omega_{1,k+1}))\right)$$
$$\cdot\left(\prod_{k=1}^{\log n}(p_{\log n+k}\omega_{2,\log n+k+1} + (1-p_{\log n+k})(1-\omega_{2,\log n+k+1}))\right)\cdot$$
$$\left(\prod_{k=1}^{\log n}(p_{2\log n+k}\omega_{1,\log n+k+1}\omega_{2,k+1} + (1-p_{2\log n+k})(1-\omega_{1,\log n+k+1})(1-\omega_{2,k+1}))\right)$$

Intuitively, the factor $(1-\omega_{1,1})\omega_{2,1}$ in the above equation ensures that the first bit of ω_1 is 0 and the first bit of ω_2 is 1. The second factor ensures that the first $\log n$ bits in p equals the row-index in ω_1, the third ensures that the second $\log n$ bits in p equals the column-index in ω_2, and the final factor ensures that the final $\log n$ bits in p equals both the column index in ω_1 and the row-index in ω_2. Using the above expression for $\tilde{\text{mult}}$, it is clear that $\tilde{\text{mult}}$ can be evaluated by $\mathcal{V}$ at a point in $O(\log n)$ time.

7.3.3 Circuit Design Issues

So far we have described the GKR protocol in the context of arithmetic circuits with addition ($+$) and multiplication gates ($\times$). This is sufficient to prove the power of this system, since any efficiently computable boolean function on boolean inputs can be computed by an (asymptotically) small arithmetic circuit. Typically such arithmetic circuits are obtained by constructing a *boolean* circuit (with AND, OR, and NOT gates) for the function, and then "arithmetizing" the circuit [8, Chapter 8]. However, we strive not just for asymptotic efficiency, but genuine practicality, and the factors involved can grow quite quickly: every layer of (arithmetic) gates in the circuit adds $s_i + 2s_{i+1}$ rounds of interaction to the protocol.

Hence, we further explore optimizations and implementation issues.

Extended Gates. The GKR circuit checking protocol [57] can be extended with any gates that compute low-degree polynomial functions of their inputs. If g is a polynomial of degree j, we can use gates computing $g(x)$; this increases the communication complexity in each round of the protocol by at most $j-2$ words, as $\mathcal{P}$ must send a degree-j polynomial, rather than a degree-2 polynomial.

The low-depth circuits we use to compute functions of interest such as DISTINCT make use of the function $f(x) = x^{q-1}$. Using only $+$ and $\times$ gates, they require depth about $\log_2 q$. If we also use gates computing $g(x,y) = x^j y^j$ for a small j, we can reduce the depth of the circuits to about $\log_{2j} q$; as the number of rounds in the protocol depends linearly on the depth of the circuit, this reduces the number of rounds by a factor of about $\log_2 q / \log_{2j} q = \log_2 2j = 1 + \log_2 j$. At the same time this increases the communication cost of each round by a factor of (at most) $j-2$. We can optimize the choice of j. In our experiments, we use $j = 4$ (so $g(x,x)$ is x^8) and $j = 8$ ($g(x,x) = x^{16}$) to simultaneously reduce the number of messages by a factor of about 3 and the communication cost and prover runtime by significant factors as well.

Another optimization is possible. The circuits we use for F_2 and DISTINCT eventually compute the sum of a large number of values. Let g be the low-degree extension of the values being summed. For functions of this form, $\mathcal{V}$ can use a single *sum-check protocol* [8, Chapter 8] to reduce the computation of the sum to computing $g(\mathbf{r})$ for a random point $\mathbf{r}$. $\mathcal{V}$ can then use the GKR protocol to delegate computation of $g(\mathbf{r})$ to $\mathcal{P}$. Conceptually, this optimization corresponds to replacing a binary tree of addition gates in an arithmetic circuit C with a single $\oplus$ gate with large fan-in, which sums all its inputs. This optimization can reduce the communication cost and the number of messages required by the protocol.

General Circuit Design. The circuit checking approach can be combined with existing

compilers, such as that in the Fairplay system [78], that take as input a program in a high-level programming language and output a corresponding boolean circuit. This boolean circuit can then be arithmetized and "verified" by our implementation; this yields a full-fledged system implementing statistically-secure verifiable computation. However, this system is likely to remain impractical even though the prover $\mathcal{P}$ can be made to run in time linear in the size of the arithmetic circuit. For example, in most hardware, one can compute the sum of two 32-bit integers x and y with a single instruction. However, when encoding this operation into a boolean circuit, it is unclear how to do this with depth less than 32. At $3 \log n$ rounds per circuit layer, for reasonable parameters, single additions can turn into thousands of rounds.

The circuits we described above avoid this overhead by avoiding boolean circuits entirely and instead view the input directly as elements over $\mathbb{F}_q$, the finite field with q elements, for a large prime q. For example, if the input is an array of 32-bit integers, then we view each element of the array as a value of $\mathbb{F}_q$ in the natural way, and calculating the sum of two integers requires a single depth-1 addition gate, rather than a depth-32 boolean circuit. However, this approach seems to severely limit the functionality that can be implemented. For instance, we know of no compact arithmetic circuit to test whether $x > y$ for two integers x and y.

This polylogarithmic blowup in circuit depth compared to input size appears inherent in any construction that encodes computations as arithmetic circuits. Therefore, the development of general purpose protocols that avoid this representation remains an important direction for future work.

7.4 Implementation and Experiments

7.4.1 Implementation Details

Our experimental setup was identical to that of our F_2 implementation described in Chapter 6. Specifically, we built our implementations in C++ and simulated the computations of both parties and measured the time and resources consumed by the protocols. All computations are over the field of size $q = 2^{61} - 1$, implying a very low probability of the verifier being fooled by a dishonest prover. Our source code for the experiments of this section is available online at [38].

For the experiments of this section, we evaluated the protocols on a multi-core machine with 64-bit AMD Opteron processors and 32 GB of memory available. Our scalability results here use a single core. The large amount of memory allowed us to experiment with universes of size several billion, with the prover able to store the full data in memory. We measured the time for $\mathcal{V}$ to compute the check information from the stream, for $\mathcal{P}$ to generate the proof, and for $\mathcal{V}$ to verify the proof. We also measured the space required by $\mathcal{V}$, and the size of the proof provided by $\mathcal{P}$.

Choice of Field Size. While all the protocols we implemented work over arbitrary finite fields, our choice of $\mathbb{F}_q$ with $q = 2^{61} - 1$ proves ideal for engineering practical protocols. First, the field size is large enough to provide a minuscule probability of error, but small enough that any field element can be represented with a single 64-bit data type. By using native types, we achieve a speedup of several factors. Second, reducing modulo q can be done with a bit shift, a bit-wise AND operation, and an addition [100] The same observation applies to any field whose size equals a Mersenne Prime, including $2^{89} - 1$, $2^{107} - 1$, and $2^{127} - 1$. $2^{127} - 1$ in particular is an excellent choice if one needs to deal with larger integer values, as elements of this field can be represented as 2 64-bit words. We experienced a speedup of

nearly an order of magnitude by switching to this specialized "mod" operation rather than using "% q' operation in C++.

The main potential issue with our choice of field size is that "overflow" can occur for problems such as matrix multiplication if the entries of the input matrices are very large. For example, with 512×512 matrix multiplication, if the entries of the input matrices A, B are larger than 2^{26}, an entry in the product matrix AB can be as large as 2^{61}, which is larger than our field size. If this is a concern, a larger field size is appropriate. (Notice that for a problem such as DISTINCT, there is no danger of overflow issues as long as the length of the stream is smaller than $2^{61} - 2$, which is larger than any stream encountered in practice). A second reason to use larger field sizes is to handle floating-point or rational arithmetic as proposed by Setty et al. in [91].

All of our protocols can be instantiated over fields with more than $2^{61} - 1$ elements, with an implementation using these fields experiencing a slowdown proportional to the increased cost of arithmetic over these fields.

7.4.2 Experimental Results

In this chapter, we have put significant effort into optimizing the runtime of the prover within the GKR protocol, achieving an implementation for which $\mathcal{P}$ takes time nearly linear in the size of the circuit. Nonetheless, this cost remains the chief limitation of the implementation.

We experimented with our implementation on circuits for our three motivating problems: F_2, DISTINCT, and MATMULT. Results are summarized in Table 7.1. Throughout, when we refer to $\mathcal{P}$'s runtime in an interactive protocol, we are referring to the total time over all rounds of the protocol. The speed per gate can be very high: $\mathcal{P}$ processed circuits with tens of millions of gates in a matter of minutes. For example, our basic implementation processed a

Table 7.1: Experimental results with our implementation of the GKR protocol.

Problem	Gates	Size (gates)	$\mathcal{P}$ time (s)	Rounds	Communication (KBs)	$\mathcal{V}$ time (s)
F_2	$+, \times$	0.4M	8.5	986	11.5	.01
F_2	$+, \times, \oplus$	0.2M	6.5	118	2.5	.01
DISTINCT	$+, \times$	16M	552.6	3730	87.4	.01
DISTINCT	$+, \times,$ ^8	8.4M	462.2	1684	60.0	.01
DISTINCT	$+, \times,$ ^16	6.4M	457.4	1399	65.8	.01
DISTINCT	$+, \times, \oplus$	15.8M	546.4	3355	78.4	.01
DISTINCT	$+, \times,$ ^8, $\oplus$	8.2M	432.6	1310	51.0	.01
DISTINCT	$+, \times,$ ^16, $\oplus$	6.2M	441.2	1024	56.8	.01
MATMULT	$+, \times$	300.0M	9759.1	767	17.9	.10

The F_2 and DISTINCT rows report on input vectors of length $n = 2^{17}$, while the MATMULT row reports on 512×512 matrix multiplication. For MATMULT, the Communication column does not count the cost of specifying the answer, only the additional communication required to prove correctness.

circuit for DISTINCT with close to 16 million gates in under 9 minutes, or close to 30,000 gates per second. However, since the circuit's size was more than 100 times larger than the universe over which the input is drawn, this translated to only about 300 items per second. The other costs incurred are very low. The verifier's space usage and the communication cost are never more than a few dozen kilobytes, and the verifier processes close to thirty million updates per second across all stream lengths. Indeed, for problems that require superlinear time to solve, the verifier's runtime in our protocol is less than the time required to solve the problem locally, without the help of a prover. For example, for 512×512 matrix multiplication, our verifier implementation requires .10 seconds to process the input stream, while a C++ program performing naive matrix multiplication with floating point arithmetic required 1.53 seconds. We stress that the savings for the verifier would be larger for more complicated

problems and for larger input sizes, as the verifier's runtime grows quasi-linearly with the size of the input. The time for $\mathcal{V}$ to check the prover's messages for consistency in our implementation is negligible compared to the (already low) time to compute the required low-degree extension of the input.

As we previously discussed in Section 7.3.3, adding additional gate types can reduce the cost of circuit checking. We demonstrate experimentally that adding gates that compute the 8th power (ˆ8) or the 16th power (ˆ16) of their inputs achieves substantial reductions in the size of the circuits needed (see Table 7.1). For DISTINCT, this reduced the number of rounds by nearly a factor of three, the prover time by close to 20%, and the overall communication cost by close to 30%. We also discuss in Section 7.3.3 how to (conceptually) replace a binary tree of addition gates with a single $\oplus$ gate of very large fan-in which sums all its inputs. For DISTINCT, this optimization further reduced both communication and number of rounds by 10-20%. The effect of $\oplus$ gates was much more pronounced for F_2, where we saw an order of magnitude reduction in the number of rounds, and 5-fold reduction in communication cost. The change was larger here because the addition gates represent a much larger fraction of the gates in F_2 circuits than in DISTINCT circuits.

Summary. We conclude that the costs to the verifier of our implementation of the GKR protocol are quite attractive. Our implementation saves the verifier time and space even for small problem instances, and the communication cost is never more than a few dozen kilobytes. The bottleneck in our implementation is the runtime of the prover. While our methods achieve a prover runtime that is very close to linear in the size of the circuit, and our prover implementation demonstrates significantly better scalability than one might expect *a priori*, the prover runtime remains the bottleneck in the protocol. This bottleneck is mitigated further in Chapter 8.

7.5 GPU Implementation

In this section, we describe a parallel implementation of our protocol using graphics processing units (GPUs). The primary purpose of this section is to demonstrate the GKR protocol's substantial amenability to parallelization, which can substantially mitigate the bottlenecks of the sequential implementation already described. GPUs are not by any means the only parallel platform suitable for our task, but the GPU is becoming a standard parallel platform, and it provides a good testing ground for implementation. While some of the bottlenecks we identify in our parallel implementation are GPU-specific, several of our design choices should be broadly applicable. For example, in this section we describe how to structure the implementation to ensure memory coalescing, as well as how to minimize the number of memory transfers when working within a memory hierarchy.

7.5.1 Overview

We begin by explaining the insights necessary to parallelize the computation of both the prover and the verifier for the implementation of the GKR protocol described in this chapter.

Parallelizing $\mathcal{P}$'s computation

In every one of $\mathcal{P}$'s responses in the GKR protocol, the prescribed message from $\mathcal{P}$ is defined via a large sum over roughly S^3 terms, where S is the size of the circuit, and so computing this sum naively would take $\Omega(S^3)$ time. Roughly speaking, We observed in Section 7.1 that each gate of the circuit contributes to only a single term of this sum, and thus this sum can be computed via a single pass over the relevant gates. The contribution of each gate to the sum can be computed in constant time, and each gate contributes to logarithmically many messages over the course of the protocol. Using these observations

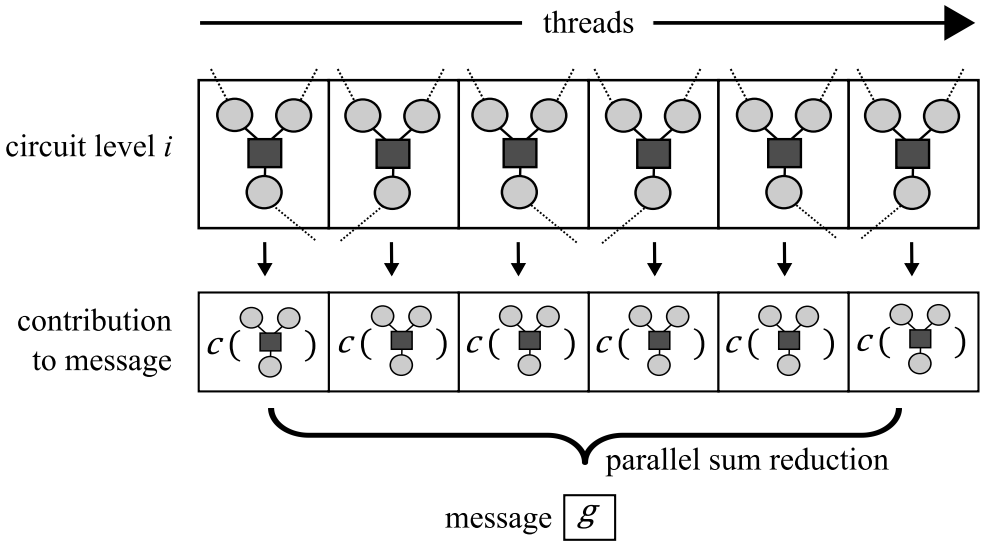


Figure 7.3: Illustration of parallel computation of the server's message to the client in the GKR protocol.

carefully reduces $\mathcal{P}$'s runtime from $\Omega(S^3)$, to $O(S \log S)$, where again S is the circuit size.

The same observation reveals that $\mathcal{P}$'s computation can be parallelized: each gate contributes *independently* to the sum in $\mathcal{P}$'s prescribed response. Therefore, $\mathcal{P}$ can compute the contribution of many gates in parallel, save the results in a temporary array, and use a parallel reduction to sum the results. We stress that all arithmetic is done within the finite field $\mathbb{F}$, rather than over the integers. Figure 7.3 illustrates this process.

Parallelizing $\mathcal{V}$'s computation

The bulk of $\mathcal{V}$'s computation (by far) consists of computing $\tilde{V}_d(\mathbf{r})$, an evaluation of the multilinear extension of the input at a random point $\mathbf{r} \in \mathbb{F}^{\log n}$. As observed in Lemma 6.3.1, each input symbol contributes *independently* to $\tilde{V}_d(\mathbf{r})$. Thus, $\mathcal{V}$ can compute the contribution of many input symbols in parallel, and sum the results via a parallel reduction, just as in the parallel implementation of $\mathcal{P}$'s computation. This speedup is perhaps of secondary importance, as $\mathcal{V}$ runs extremely quickly even in our sequential implementation. However,

parallelizing $\mathcal{V}$'s computation is still an appealing goal, especially as GPUs are becoming more common on personal computers and mobile devices.

7.5.2 Architectural considerations

The primary issue with any GPU-based implementation of the prover in the GKR protocol is that the computation is extremely memory-intensive: for a circuit of size S (which corresponds to S arithmetic operations in an unverifiable algorithm), the prover in the GKR protocol has to store all S gates explicitly, because she needs the values of these gates to compute her prescribed messages. We investigate three alternative strategies for managing the memory overhead of the GKR protocol, which we refer to as the no-copying approach, the copy-once-per-layer approach, and the copy-every-message approach.

The no-copying approach

The simplest approach is to store the entire circuit explicitly on the GPU. We call this the *no-copying approach*. However, this means that the entire circuit must fit in device memory, a requirement which is violated even for relatively small circuits, consisting of roughly tens of million of gates.

The copy-once-per-layer approach

Another approach is to keep the circuit in host memory and copy information to the device only when it is needed. This is possible because at any point in the protocol the prover operates on only two layers of the circuit at a time, so only two layers of the circuit need to reside in device memory. We refer to this as the *copy-once-per-layer approach*. This is the approach we use the experiments in Section 7.5.3.

Care must be taken with this approach to prevent host-to-device copying from becoming

a bottleneck. Fortunately, in the protocol for each layer there are several dozen messages to be computed before the prover moves on to the next layer, and this ensures that the copying from host to device makes up a very small portion of the runtime.

This method is sufficient to scale to very large circuits for circuits in which no single layer is significantly larger than the problem input itself. However, this method remains problematic for circuits that have (one or several) layers that are particularly wide, as an explicit representation of all the gates within a *single* wide layer may still be too large to fit in device memory.

The copy-every-message approach

In the event that there are individual layers that are too large to reside in device memory, a third approach is to copy *part* of a layer at a time from the host to the device, and compute the contribution of each gate in the part to the prover's message before swapping the part back to host memory and bringing in the next part. We call this the copy-every-message approach. This approach is viable, but it raises a significant issue, alluded to in its name. Namely, this approach requires host-to-device copying for every message, rather than just once per layer of the circuit. That is, in any iteration i of the protocol, $\mathcal{P}$ cannot compute her jth message until after the $(j-1)$th challenge from $\mathcal{V}$ is received. Thus, for *each message* j, the entirety of the ith layer must be loaded piece-by-piece into device memory, swapping each piece back to host memory after the piece has been processed. In contrast, the copy-once-per-layer approach allows $\mathcal{P}$ to copy an entire layer i to the device and leave the entire layer in device memory for the entirety of iteration i (which will consist of several dozen messages). Thus, the slowdown inherent in the copy-every-message approach is not just that $\mathcal{P}$ has to break each layer into parts, but that $\mathcal{P}$ has to do host-to-device and device-to-host copying for each message, instead of copying an entire layer and computing several messages

from that layer.

We did not perform a careful evaluation of the copy-every-message approach, but preliminary experiments suggest that this approach is viable in practice, resulting in less than a 3× slowdown compared to the copy-once-per-layer approach. Notice that even after paying this slowdown, our GPU-based implementation would still achieve a 10-40× speedup compared to the sequential implementation described earlier in this chapter.

Memory access

Recall that for each message in the ith iteration of the GKR protocol, we assign a thread to each gate g at the ith layer of the circuit, as each gate contributes independently to the prescribed message of the prover. The contribution of gate g depends only on the index of g, the indices of the two gates feeding into g, and the *values* of the two gates feeding into g.

Given this data, the contribution of gate g to the prescribed message can be computed using roughly 10-20 additions and multiplications within the finite field $\mathbb{F}$ (the precise number of arithmetic operations required varies over the course of the iteration). As described in Section 7.3.3, we choose to work over a field that allows for extremely efficient arithmetic; for example, multiplying two field elements requires three machine multiplications of 64-bit data types, and a handful of additions and bit shifts.

In all of the circuits we consider, the *indices* of g's in-neighbors can be determined with very little arithmetic and no global memory accesses. For example, if the wiring pattern of the circuit forms a binary tree, then the first in-neighbor of g has index $2 \cdot \text{index}(g)$, and the second in-neighbor of g has index $2 \cdot \text{index}(g) + 1$. For each message, the thread assigned to g can compute this information from scratch without incurring any memory accesses.

In contrast, obtaining the *values* of g's in-neighbors requires fetching 8 bytes per in-neighbor from global memory. Memory accesses are necessary because it is infeasible to

compute the value of each gate's in-neighbors from scratch each message, so we store these values explicitly. As these global memory accesses can be a bottleneck in the protocol, we strive to arrange the data in memory to ensure that adjacent threads access adjacent memory locations. To this end, for each layer i we maintain two separate arrays, with the j'th entry of the first (respectively, second) array storing the first (respectively, second) in-neighbor of the j'th gate at layer i. During iteration i, the thread assigned to the jth gate accesses location j of the first and second array to retrieve the value of its first and second in-neighbors respectively. This ensures that adjacent threads access adjacent memory locations.

For all layers, the corresponding arrays are populated with in-neighbor values when we evaluate the circuit at the start of the protocol (we store each layer i's arrays on the host until the i'th iteration of the protocol, at which point we transfer the array from host memory to device memory as describe in Section 7.3.3). Notice this methodology sometimes requires data duplication: if many gates at layer i share the same in-neighbor g_1, then g_1's value will appear many times in layer i's arrays. We feel that slightly increased space usage is a reasonable price to pay to ensure memory coalescing.

7.5.3 GPU Evaluation

Implementation details

Except where noted, we performed our experiments on an Intel Xeon 3 GHz workstation with 16 GB of host memory. Our workstation also has an NVIDIA GeForce GTX 480 GPU with 1.5 GB of device memory. We implemented all our GPU code in CUDA and Thrust [65] with all compiler optimizations turned on. Our source code is available online at [98]. We remark that no floating point operations were necessary in any of our implementations, because all arithmetic is done over finite fields. Finally, we stress that in all reported costs below, we do count the time taken to copy data between the host and the device, and all

reported speedups relative to sequential processing take this cost into account. We do not count the time to allocate memory for scratch space because this can be done in advance.

We ran our GPU-based implementation of the GKR protocol on four separate circuits, which together capture several different aspects of computation, from data aggregation to search, to linear algebra. The first three problems we consider, F_2, DISTINCT, and MATMULT, were described and evaluated earlier in the chapter, while the fourth problem, PM, is a classic search problem, and is motivated, for example, by clients wishing to store (and search) their email on the cloud. Specifically, in the pattern matching problem, the input is a stream representing text $T = (t_0, \ldots, t_{n-1}) \in [n]^n$ and a pattern $P = (p_0, \ldots, p_{q-1}) \in [n]^q$, and the pattern P is said to occur at location i in t if, for every position j in P, $p_j = t_{i+j}$. The pattern-matching problem is to determine the number of locations at which P occurs in T.

Description of circuits

We briefly review the circuits for our benchmark problems.

The circuit for F_2 is by far the simplest and was described earlier in the Chapter (see Figure 6.1 for an illustration). This circuit computes the square of each input wire using a layer of multiplication gates, and then sums the results using a single sum-gate of very large fan-in.

The circuit for F_0 was also described earlier: it exploits Fermat's Little Theorem, which says that for prime q, $a^{q-1} \equiv 1 \mod q$ if and only if $a \not\equiv 0$. Thus, this circuit computes the $q-1$'th power of each input wire (taking all non-zero inputs to 1, and leaving all 0-inputs at 0), and sums the results via a single sum-gate of high fan-in.

The circuit for PM is similar to that for F_0: essentially, for each possible location of the pattern, it computes a value that is 0 if the pattern is at the location, and non-zero otherwise. It then computes the $(q-1)$th power of each such value and sums the results (i.e., it uses

the F_0 circuit as a subroutine) to determine the number of locations where the pattern does (not) appear in the input.

For our GPU experiments, the MatMult circuit we experimented on in this section differs slightly from the one used in Section 7.3.2. The circuit we use here is slightly more complicated – the reason is that the GPU experiments reported on here were performed prior to the matrix multiplication experiments reported on in Section 7.4, and it was not until later that we realized it was possible to use the simpler circuit of Section 7.3.2. While the matrix multiplication circuit we used for the experiments of this section is slightly more complicated than necessary, our results on this problem nonetheless demonstrate the amenability of the GKR protocol to parallelization.

Specifically, the circuit we use here contains a *single* output gate, rather than n^2 output gates as in Section 7.3.2. This circuit computes the n^2 entries in $C = A \cdot B$ via naive matrix multiplication just like the circuit from Section 7.3.2, but then it subtracts the corresponding entry of C from each. It then computes the number of non-zero values using the F_0 circuit as a subroutine. The final output of the circuit is zero if and only if $C = AB$.

Scaling to large inputs

As described in Section 7.5.2, the memory-intensive nature of the GKR protocol made it challenging to scale to large inputs, especially given the limited amount of device memory. Indeed, with the no-copying approach (where we simply keep the entire circuit in device memory), we were only able to scale to inputs of size roughly $150,000$ for the F_0 problem, and to 32×32 matrices for the MatMult problem on a machine with 1 GB of device memory. Using the copy-once-per-layer approach, we were able to scale to inputs with over 2 million entries for the F_0 problem, and 128×128 matrices for the MatMult problem. By running on a NVIDIA Tesla C2070 GPU with 6 GBs of device memory, we were able to

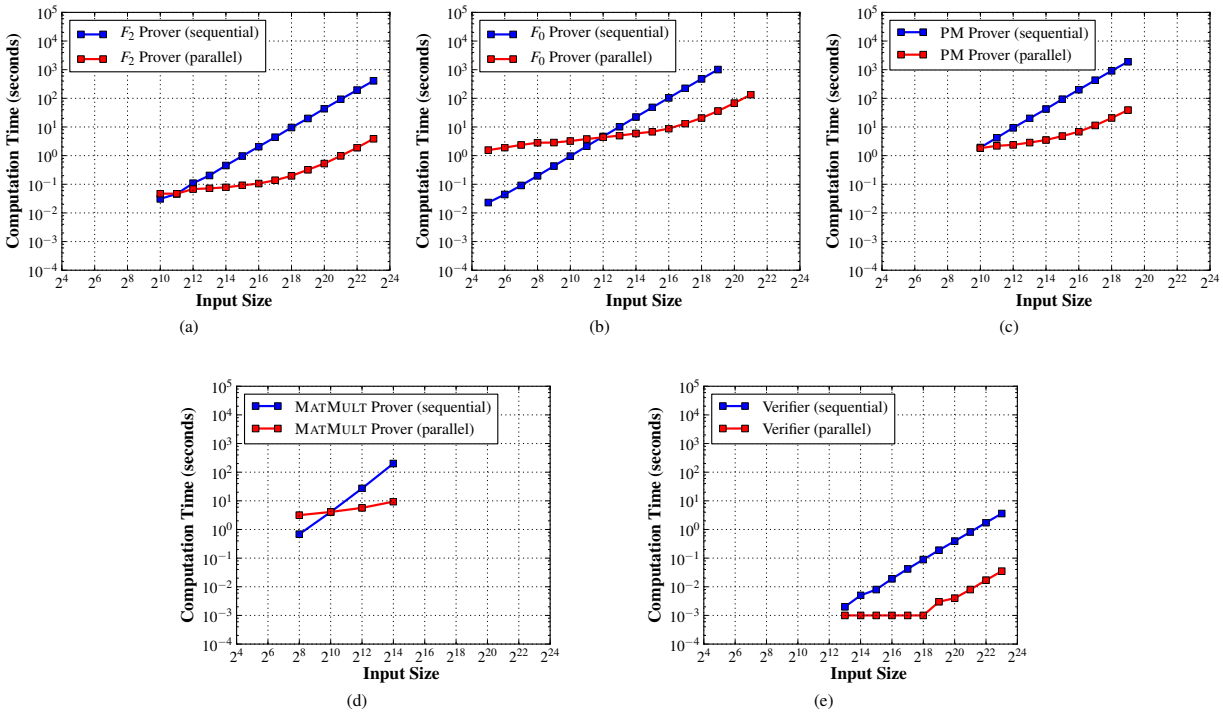


Figure 7.4: Comparison of prover and verifier runtimes between our sequential implementation of the GKR protocol and our GPU-based implementation. Note that all plots are on a log-log scale. Plots (a), (b), (c), and (d) depict the prover runtimes for F_0, F_2, Pm, MatMult respectively. Plot (e) depicts the verifier runtimes for the GKR protocol. We include only one plot for the verifier, since its dominant cost in the GKR protocol is problem-independent.

push to 256×256 matrices for the MatMult problem; the data from this experiment is reported in Table 7.2.

Experimental Results

Figure 7.4 demonstrates the performance of our GPU-based implementation of the GKR protocol. Table 7.2 also gives a succinct summary of our results, showing the costs for the largest instance of each problem we ran on. We consider the main takeaways of our experiments to be the following.

Table 7.2: Experimental results for our GPU implementation of the GKR protocol.

Problem	Input Size (number of entries)	Circuit Size (number of gates)	GPU $\mathcal{P}$ Time (s)	Sequential $\mathcal{P}$ Time (s)	Circuit Eval Time (s)	GPU $\mathcal{V}$ Time (s)	Sequential $\mathcal{V}$ Time (s)	Unverified Alg. Time (s)
F_2	8.4 million	25.2 million	3.7	424.6	0.1	0.035	3.600	0.028
DISTINCT	2.1 million	255.8 million	128.5	8,268.0	4.2	0.009	0.826	0.005
PM	524,288	76.0 million	38.9	1,893.1	1.2	0.004	0.124	0.006
MATMULT	65,536	42.3 million	39.6	1,658.0	0.9	0.003	0.045	0.080

Server-side speedup obtained by GPU computing. Compared to the our sequential implementation described earlier in the chapter, our GPU-based server implementation ran close to 115× faster for the F_2 circuit, about 60× faster for the F_0 circuit, 45× faster for PM, and about 40× faster for MATMULT (see Figure 7.4).

Notice that for the first three problems, we need to look at large inputs to see the asymptotic behavior of the curve corresponding to the parallel prover's runtime. Due to the log-log scale in Figure 7.4, the curves for both the sequential and parallel implementations are asymptotically linear, and the 45-120× speedup obtained by our GPU-based implementation is manifested as an additive gap between the two curves. The explanation for this is simple: there is considerable overhead relative to the total computation time in parallelizing the computation at small inputs, but this overhead is more effectively amortized as the input size grows.

In contrast, notice that for MATMULT the slope of the curve for the parallel prover remains significantly smaller than that of the sequential prover throughout the entire plot. This is because our GPU-based implementation ran out of device memory well before the overhead in parallelizing the prover's computation became negligible. We therefore believe the speedup for MATMULT would be somewhat higher than the 40× speedup observed if we

were able to run on larger inputs.

Server-side slowdown relative to unverifiable sequential algorithms. For F_2, the total slowdown for the prover was roughly 130× (3.7 seconds compared to 0.028 seconds for the unverifiable algorithm, which simply iterates over all entries of the frequency vector and computes the sum of the squares of each entry). We stress that it is likely that we overestimate the slowdown resulting from our protocol, because we did not count the time it takes for the unverifiable implementation to compute the number of occurrences of each item i, that is, to *aggregate* the stream into its frequency vector representation $(f_1, \ldots, f_n)$. Instead, we simply generated the vector of frequencies at random (we did not count the generation time), and calculated the time to compute the sum of their squares. In practice, this aggregation step may take much longer than the time required to compute the sum of the squared frequencies once the stream is in aggregated form.

For DISTINCT, our GPU-based server implementation ran roughly 25,000× slower than the obvious unverifiable algorithm which simply counts the number of non-zero items in a vector. The larger slowdown compared to the F_2 problem is unsurprising. Since DISTINCT is a less arithmetic problem than F_2, its circuit representation is much larger. Once again, it is likely that we overestimate the slowdowns for this problem, as we did not count the time for an unverifiable algorithm to aggregate the stream into its frequency-vector representation. Despite the substantial slow-down incurred for F_0 compared to a naive unverifiable algorithm, it remains valuable as a primitive for use in heavier-duty computations like PM and MATMULT.

For PM, the bulk of the circuit consists of a DISTINCT sub-routine, and so the runtime of our GPU-based implementation was similar to those for DISTINCT. However, the sequential unverifiable algorithm for PM takes longer than that for DISTINCT. Thus, our GPU-based server implementation ran roughly 6,500× slower than the naive unverifiable algorithm,

which exhaustively searches all possible locations for occurrences of the pattern.

For MatMult, our GPU-based server implementation ran roughly 500× slower than naive matrix-multiplication for 256×256 matrices. Moreover, this number is likely inflated due to cache effects from which the naive unverifiable algorithm benefited. That is, the naive unverifiable algorithm takes only 0.09 seconds for 256×256 matrices, but takes 7.1 seconds for 512×512 matrices, likely because the algorithm experiences very few cache misses on the smaller matrix. We therefore expect the slowdown of our implementation to fall to under 100× if we were to scale to larger matrices. Furthermore, the GKR protocol is capable of verifying matrix-multiplication over the finite field $\mathbb{F}_q$ rather than over the integers at no additional cost. Naive matrix-multiplication over this field is between 2-3× slower than matrix multiplication over the integers (even using the fast arithmetic operations available for this field). Thus, if our goal was to work over this finite field rather than the integers, our slowdown would fall by another 2-3×. It is therefore possible that our server-side slowdown may be less than 50× at larger inputs compared to naive matrix multiplication over $\mathbb{F}_q$.

Client-side speedup obtained by GPU computing. The bulk of $\mathcal{V}$'s computation consists of evaluating the multilinear extension of the input at a point; this computation is independent of the circuit being verified. For reasonably large inputs (see the row for F_2 in Table 7.2), our GPU-based client implementation performed this computation over 100× faster than the sequential implementation described earlier in this Chapter. For smaller inputs the speedup was unsurprisingly smaller due to increased overhead relative to total computation time. Still, we obtained a 15× speedup even for an input of length 65,536 (256×256 matrix multiplication).

Client-side speedup relative to unverifiable sequential algorithms. Our results on matrix multiplication clearly demonstrate that for problems requiring super-linear time to solve, even the sequential implementation of described earlier in the chapter will save the

client time compared to doing the computation locally. Indeed, the runtime of the client is dominated by the cost of evaluating the multilinear extension of the input at a single point, and this cost grows quasi-linearly with the input size. Even for relatively small matrices of size 256×256, the client in our sequential implementation saved time. For matrices with tens of millions of entries, our results demonstrate that the client will still take just a few seconds, while performing the matrix multiplication computation would require orders of magnitude more time. Our results demonstrate that GPU computing can be used to reduce the verifier's computation time by another $100\times$.

Chapter 8

Time-Optimal Interactive Proofs for Circuit Evaluation

8.1 Introduction

In Chapter 7, we showed how to implement the prover in the GKR protocol in time $O(S \log S)$, where S is the size of an arithmetic circuit computing the function of interest. In this chapter, we describe further refinements to the GKR protocol, which apply to circuits with sufficiently "regular" wiring patterns; for these circuits, we bring the runtime of the prover down to $O(S)$. That is, our prover can evaluate the circuit with a guarantee of correctness, with only a constant-factor blowup in work compared to evaluating the circuit with no guarantee. The $\log S$ factor we save over the implementation of Chapter 7 is significant – easily a factor of 20 even for circuits with a just a few million gates – and our approach also leads to significant reductions in the communication costs and the number of rounds of interaction required by the protocol.

We argue that our refinements capture a large class of circuits, and we complement our theoretical results with experiments on problems such as matrix multiplication and deter-

mining the number of distinct elements in a data stream. Experimentally, our refinements yield a 200x speedup for the prover over the implementation of Chapter 7, and our prover is less than 10x slower than a C++ program that simply evaluates the circuit. Moreover, a parallel implementation of our prover using a graphics processing unit (GPU) is roughly 30x faster than our serial implementation, and therefore takes less time than that required to evaluate the circuit in serial.

We also make progress toward addressing another issue of existing interactive proof implementations: their applicability. The GKR protocol applies in principle to any problem computed by a small-depth arithmetic circuit, but this is not the case when more fine-grained considerations of prover and verifier efficiency are taken into account. In brief, existing implementations of interactive proof protocols for circuit evaluation such as that of Chapter 7 as well as that by Vu et al. [104] all require that the circuit have a highly regular wiring pattern. If this is not the case, then these implementations require the verifier to perform an expensive (though data-independent) preprocessing phase to pull out information about the wiring of the circuit, and they require a substantial factor blowup (logarithmic in the circuit size) in runtime for the prover relative to evaluating the circuit without a guarantee of correctness. Developing a protocol that avoids these pitfalls and applies to more general computations remains an important open question.

Our approach is the following. We do not have a magic bullet for dealing with irregular wiring patterns; if we want to avoid an expensive pre-processing phase for the verifier and minimize the blowup in runtime for the prover, we do need to make an assumption about the structure of the circuit we are verifying. Acknowledging this, we ask whether there is some general structure in real-world computations that we can leverage for efficiency gains.

To this end, we design a protocol that is highly efficient for data parallel computation. By data parallel computation, we mean any setting in which one applies the same computa-

tion independently to many pieces of data. Many outsourced computations are data parallel, with Amazon Elastic MapReduce[1] being one prominent example of a cloud computing service targeted specifically at data parallel computations. Crucially, we do not want to make significant assumptions on the sub-computation that is being applied, and in particular we want to handle sub-computations computed by circuits with highly irregular wiring patterns.

The verifier in our protocol still has to perform an offline phase to pull out information about the wiring of the circuit, but the cost of this phase is proportional to the size of a single instance of the sub-computation, avoiding any dependence on the number of pieces of data to which the sub-computation is applied. Similarly, the blowup in runtime suffered by the prover is the same as it would be if the prover had run the basic GKR protocol on a single instance of the sub-computation.

Our final contribution is to describe a new protocol specific to matrix multiplication that we believe to be of interest in its own right. This protocol is formalized in Theorem 8.7.2. Given any *unverifiable* algorithm for $n \times n$ matrix multiplication that requires time $T(n)$ using space $s(n)$, Theorem 8.7.2 allows the prover to run in time $T(n) + O(n^2)$ using space $s(n)+o(n^2)$. Note that Theorem 8.7.2 (which is specific to matrix multiplication) is much less general than Theorem 8.4.10 (which applies to any circuit with a sufficiently regular wiring pattern). However, Theorem 8.7.2 achieves optimal runtime and space usage for the prover up to leading constants, assuming there is no $O(n^2)$ time algorithm for matrix multiplication. While these properties are also satisfied by a classic protocol due to Freivalds [53], the protocol of Theorem 8.7.2 is significantly more amenable for use as a primitive when verifying computations that repeatedly invoke matrix multiplication. For example, using the protocol of Theorem 8.7.2 as a primitive, we give a natural protocol for computing the diameter of an unweighted directed graph G. $\mathcal{V}$'s runtime in this protocol is $O(m \log n)$, where m is the

[1] `http://aws.amazon.com/elasticmapreduce/`

number of edges in G, $\mathcal{P}$'s runtime matches the best known unverifiable diameter algorithm up to a low-order additive term [89,107], and the total communication is just polylog (n). We know of no other protocol achieving this. We complement Theorem 8.7.2 with experimental results demonstrating its efficiency.

8.1.1 Roadmap for Chapter

We give a high-level overview of the ideas underlying our main results in Section 8.2. Section 8.3 describes technical background. Section 8.4 contains the details of our time-optimal protocol for circuit evaluation as formalized in Theorem 8.4.10. Section 8.5 describes our experimental cases studies of the protocol described in Theorem 8.4.10. Section 8.6 describes our protocol for arbitrary data parallel computation. Section 8.7 describes some additional optimizations that apply to specific important wiring patterns. In particular, this section describes our special-purpose protocol for MatMult that achieves optimal prover efficiency up to leading constants. Section 8.8 concludes.

8.2 Overview of the Ideas

8.2.1 Achieving Optimal Prover Runtime for Regular Circuits

In Theorem 8.4.10, we describe an interactive proof protocol for circuit evaluation that brings $\mathcal{P}$'s runtime down to $O(S(n))$ for a large class of circuits, while maintaining the same verifier runtime as in prior implementations of the GKR protocol. Informally, Theorem 8.4.10 applies to any circuit whose wiring pattern is sufficiently "regular".

This protocol follows the same general outline as the GKR protocol, in that we proceed in iterations from the output layer of the circuit to the input layer, using a sum-check protocol at iteration i to reduce a claim about the gate values at layer i to a claim about the gate

values at layer $i+1$. However, at each iteration i we apply the sum-check protocol to a carefully chosen polynomial that differs from the one used by GKR. In each round j of the sum-check protocol, our choice of polynomial allows $\mathcal{P}$ to reuse work from prior rounds in order to compute the prescribed message for round j, allowing us to shave a $\log S(n)$ factor from the runtime of $\mathcal{P}$ relative to the $O(S(n) \log S(n))$-time implementation of Chapter 7.

Specifically, at iteration i, the GKR protocol uses a polynomial $f_z^{(i)}$ defined over $\log S_i + 2 \log S_{i+1}$ variables, where S_i is the number of gates at layer i. The "truth table" of $f_z^{(i)}$ is sparse on the Boolean hypercube, in the sense that $f_z^{(i)}(x)$ is non-zero for at most S_i of the $S_i \cdot S_{i+1}^2$ inputs $x \in \{0,1\}^{\log S_i + 2 \log S_{i+1}}$. In Chapter 7, we leveraged this sparsity to bring the runtime of $\mathcal{P}$ in iteration i down to $O(S_i \log S_i)$ from a naive bound of $\Omega(S_i \cdot S_{i+1}^2)$. The $\log S_i$ factor was due to the fact that $\mathcal{P}$ needed to make a pass over all of the gates at layer i for each of the $\log S_i$ rounds of the sum-check protocol. However, this same sparsity prevents $\mathcal{P}$ from reusing work from prior iterations as we seek to do.

In contrast, we use a polynomial $g_z^{(i)}$ defined over only $\log S_i$ variables rather than $\log S_i + 2 \log S_{i+1}$ variables. Moreover, the truth table of $g_z^{(i)}$ is dense on the Boolean hypercube, in the sense that $g_z^{(i)}(x)$ may be non-zero for all of the S_i Boolean inputs $x \in \{0,1\}^{\log S_i}$. This density allows $\mathcal{P}$ to reuse work from prior iterations in order to speed up her computation in round i of the sum-check protocol.

In more detail, in each round j of the sum-check protocol, the prover's prescribed message is defined via a sum over a large number of terms, where the number of terms falls geometrically fast with the round number j. Moreover, it can be shown that in each round j, each gate at layer $i+1$ contributes to exactly one term of this sum. Essentially, what we do is group the gates at layer $i+1$ by the term of the sum to which they contribute. Each such group can be treated as a single unit, ensuring that in any round of the sum-check protocol, the amount of work $\mathcal{P}$ needs to do is proportional to the number of terms in the sum rather

than the number of gates S_i at layer i.

We remark that a similar "reuse of work" technique was used in Theorem 6.4.2, which gave an efficient special-purpose protocol for the the second frequency moment problem. This frequency moment protocol was the direct inspiration for our refinements, though we require additional insights to apply the reuse of work technique in the context of evaluating general arithmetic circuits.

It is worth clarifying why our methods do not yield savings when applied to the polynomial $f_z^{(i)}$ used in the basic GKR protocol. The reason is that, since $f_z^{(i)}$ is defined over $\log S_i + 2\log S_{i+1}$ variables instead of just $\log S_i$ variables, the sum defining $\mathcal{P}$'s message in round j is over a much larger number of terms when using $f_z^{(i)}$. It is still the case that each gate contributes to only one term of the sum, but until the number of terms in the sum falls below S_i (which does not happen until round $j = \log S_i + \log S_{i+1}$ of the sum-check protocol), it is possible for each gate to contribute to a different term. Before this point, grouping gates by the term of the sum to which they contribute is not useful, since each group can have size 1.

8.2.2 Verifying General Data Parallel Computations

Theorem 8.4.10 below only applies to circuits with regular wiring patterns, as do other existing implementations of interactive proof protocols for circuit evaluation [39, 104]. For circuits with irregular wiring patterns, these implementations require the verifier to perform an expensive preprocessing phase (requiring time proportional to the size of the circuit) to pull out information about the wiring of the circuit, and they require a substantial factor blowup (logarithmic in the circuit size) in runtime for the prover relative to evaluating the circuit without a guarantee of correctness.

To address these bottlenecks, we do need to make an assumption about the structure

of the circuit we are verifying. Ideally our assumption will be satisfied by many real-world computations. To this end, Theorem 8.6.1 will describe a protocol that is highly efficient for any data parallel computation, by which we mean any setting in which one applies the same computation independently to many pieces of data. See Figure 8.2 in Section 8.6 for a schematic of a data parallel computation.

The idea behind Theorem 8.6.1 is as follows. Let C be a circuit of size S with an arbitrary wiring pattern, and let C^* be a "super-circuit" that applies C independently to B different inputs before possibly aggregating the results in some fashion. If one naively applied the basic GKR protocol to the super-circuit C^*, $\mathcal{V}$ might have to perform a pre-processing phase that requires time proportional to the size of C^*, which is $\Omega(B \cdot S)$. Moreover, when applying the basic GKR protocol to C^*, $\mathcal{P}$ would require time $\Theta(B \cdot S \cdot \log(B \cdot S))$.

In order to improve on this, the key observation is that although each sub-computation C can have a complicated wiring pattern, the circuit is "maximally regular" between sub-computations, as the sub-computations do not interact at all. Therefore, each time the basic GKR protocol would apply the sum-check protocol to a polynomial derived from the wiring predicate of C^*, we instead use a simpler polynomial derived only from the wiring predicate of C. This immediately brings the time required by $\mathcal{V}$ in the pre-processing phase down to $O(S)$, which is proportional to the cost of executing a single instance of the sub-computation. By using the reuse of work technique underlying Theorem 8.4.10, we are also able to bring $\mathcal{P}$'s runtime down from $\Theta(B \cdot S \cdot \log(B \cdot S))$ to $\Theta(B \cdot S \cdot \log S)$, i.e., $\mathcal{P}$'s requires a factor of $O(\log S)$ more time to evaluate the circuit with a guarantee of correctness, compared to evaluating the circuit without such a guarantee. This $O(\log S)$ factor overhead does not depend on the batch size B.

Our improvements are most significant when $B \gg S$, i.e., when a (relatively) small but potentially complicated sub-computation is applied to a very large number of pieces of data.

For example, given any very large database, one may ask "How many people in the database satisfy Property P?" Our protocol allows one to verifiably outsource such *counting* queries with overhead that depends minimally on the size of the database, but that necessarily depends on the complexity of the property P.

8.2.3 A Special-Purpose Protocol for MatMult

We describe a special-purpose protocol for $n \times n$ MatMult in Theorem 8.7.2. The idea behind this protocol is as follows. The GKR protocol, as well the protocols of Theorems 8.4.10 and 8.6.1, only make use of the multilinear extension $\tilde{V}_i$ of the function V_i mapping gate labels at layer i of the circuit to their values. In some cases, there is something to be gained by using a higher-degree extension of V_i, and this is precisely what we exploit here.

In more detail, our special-purpose protocol can be viewed as an extension of our circuit-checking techniques applied to a circuit C performing naive matrix multiplication, but using a quadratic extension of the gate values in this circuit. This allows us to verify the computation using a single invocation of the sum-check protocol. More importantly, $\mathcal{P}$ can evaluate this higher-degree extension at the necessary points without explicitly materializing all of the gate values of C, which would not be possible if we had used the multilinear extension of the gate values of C.

In the protocol of Theorem 8.7.2, $\mathcal{P}$ just needs to compute the correct output (possibly using an algorithm that is much more sophisticated than naive matrix multiplication), and then perform $O(n^2)$ additional work to prove the output is correct. Since $\mathcal{P}$ does not have to evaluate C in full, this protocol is perhaps best viewed outside the lens of circuit evaluation. Still, the idea underlying Theorem 8.7.2 can be thought of as a refinement of our circuit evaluation protocols, and we believe that similar ideas may yield further improvements to general-purpose protocols in the future.

8.3 Technical Background: Making $\mathcal{V}$ Fast vs. Making $\mathcal{V}$ Streaming

Prior work has identified two methods for implementing the verifier in the GKR protocol. We described the first method in Lemma 6.3.1. If the circuit to which the protocol is applied is defined over the finite field $\mathbb{F}$, and addition and multiplication in $\mathbb{F}$ require a constant number of machine operations, then the method of Lemma 6.3.1 requires $O(n \log n)$ time and allows $\mathcal{V}$ to make a single streaming pass over the input using $O(\log n \log |\mathbb{F}|)$ bits of space. The second method is due to Vu et al. [104]. It enables $\mathcal{V}$ to compute $\tilde{V}_d(\omega^*)$ in $O(n)$ time, but requires $\mathcal{V}$ to use $O(n \log |\mathbb{F}|)$ bits of space. (Recall from Chapter 5 that $\tilde{V}_d$ is the multilinear extension of the input. We use the notation introduced there throughout this chapter).

Lemma 8.3.1 (Vu et al. [104])**.** *Let $x \in \mathbb{F}^n$ be the input to an arithmetic circuit C over $\mathbb{F}$, and let $\tilde{V}_d$ be the multilinear extension of the function that maps $i \in \{0,1\}^{\log n}$ to the ith entry of x. $\mathcal{V}$ can compute $\tilde{V}_d(\omega^*)$ in $O(n)$ time while using $O(n \log |\mathbb{F}|)$ bits of space.*

Proof. We again exploit the expression for $\tilde{V}_d(\omega^*)$ in Equation (6.2). Notice the right hand side of Equation (6.2) expresses $\tilde{V}_d(\omega^*)$ as the inner product of two n-dimensional vectors, where the bth entry of the first vector is $V_d(b)$ and the bth entry of the second vector is $\chi_b(\omega^*)$. This inner product can be computed in $O(n)$ time given a table of size n whose bth entry contains the quantity $\chi_b(\omega^*)$. Vu et al. show how to build such a table in time $O(n)$ using memoization – we used this same memoization procedure in Section 7.2.3 to ensure that the prover in the GKR protocol runs quickly in the "reducing to verification of a single point" phase.

The memoization procedure consists of $\log n$ stages, where Stage j constructs a table $A^{(j)}$ of size 2^j, such that for any $(b_1, \ldots, b_j) \in \{0,1\}^j$, $A^{(j)}[(b_1, \ldots, b_j)] = \prod_{i=1}^{j} \chi_{b_i}(\omega_i^*)$.

Notice $A^{(j)}[(b_1, \ldots, b_j)] = A^{(j-1)}[(b_1, \ldots, b_{j-1})] \cdot \chi_{b_j}(\omega_j^*)$, and so the jth stage of the memoization procedure requires time $O(2^j)$. The total time across all $\log n$ stages is therefore $O(\sum_{j=1}^{\log n} 2^j) = O(2^{\log n}) = O(n)$. This completes the proof. □

Remark 4. In [103], Vu et al. further observe that if the input is presented in a specific order, then $\mathcal{V}$ can evaluate $\tilde{V}_d(\omega^*)$ using $O(\log n \log |\mathbb{F}|)$ bits of space. Compare this result to Lemma 6.3.1, which requires $O(n \log n)$ time for $\mathcal{V}$, but allows $\mathcal{V}$ to use $O(\log n \log |\mathbb{F}|)$ bits of space regardless of the order in which the input is presented.

8.4 Time-Optimal Protocols for Circuit Evaluation

8.4.1 Protocol Outline and Section Roadmap

As with the GKR protocol, our protocol consists of $d(n)$ iterations, one for each layer of the circuit. Each iteration starts with $\mathcal{P}$ claiming a value for $\tilde{V}_i(z)$ for some value $z \in \mathbb{F}^{s_i}$. The purpose of the iteration is to reduce this claim to a claim about $\tilde{V}_{i+1}(\omega)$ for some $\omega \in \mathbb{F}^{s_{i+1}}$, in the sense that it is safe for $\mathcal{V}$ to assume that the first claim is true as long as the second claim is true. As in the GKR protocol, this is done by invoking the sum-check protocol on a certain polynomial.

In order to improve on the costs of the GKR protocol implementation of Chapter 7, we replace the polynomial $f_z^{(i)}$ in Equation (5.1) with a different polynomial $g_z^{(i)}$ defined over a much smaller domain. Specifically, $g_z^{(i)}$ is defined over only s_i variables rather than $s_i + 2s_{i+1}$ variables as is the case of $f_z^{(i)}$. Using $g_z^{(i)}$ in place of $f_z^{(i)}$ allows $\mathcal{P}$ to reuse work across iterations of the sum-check protocol, thereby reducing $\mathcal{P}$'s runtime by a logarithmic factor relative to the implementation of Chapter 7, as formalized in Theorem 8.4.10 below.

The remainder of the presentation leading up to Theorem 8.4.10 proceeds as follows. After stating a preliminary lemma, we describe the polynomial $g_z^{(i)}$ that we use in the context

of three specific circuits: a binary tree of addition or multiplication gates, and a circuit computing the number of non-zero entries of an n-dimensional vector a. The purpose of this exposition is to showcase the ideas underling Theorem 8.4.10 in concrete scenarios. Second, we explain the algorithmic insights that allow $\mathcal{P}$ to reuse work across iterations of the sum-check protocol applied to $g_z^{(i)}$. Finally, we state and prove Theorem 8.4.10, which formalizes the class of circuits to which our methods apply.

8.4.2 A Preliminary Lemma

We will repeatedly invoke the following lemma, which allows us to express the value $\tilde{V}_i(z)$ in a manner amenable to verification via the sum-check protocol. This is essentially a restatement of [85, Lemma 3.2.1].

Lemma 8.4.1. *Let W be any polynomial $\mathbb{F}^{s_i} \to \mathbb{F}$ that extends V_i, in the sense that for all $p \in \{0,1\}^{s_i}$, $W(p) = V_i(p)$. Then for any $z \in \mathbb{F}^{s_i}$,*

$$\tilde{V}_i(z) = \sum_{p \in \{0,1\}^{s_i}} \beta_{s_i}(z,p) W(p). \tag{8.1}$$

Proof. It is easy to check that the right hand side of Equation (8.1) is a multilinear polynomial in z, and that it agrees with V_i on all Boolean inputs. Thus, the right hand side of Equation (8.1), viewed as a polynomial in z, must be the multilinear extension $\tilde{V}_i$ of V_i. This completes the proof. □

8.4.3 Polynomials for Specific Circuits

The Polynomial for a Binary Tree

Consider a circuit C that computes the product of all n of its inputs by multiplying them together via a binary tree. Label the gates at layers i and $i+1$ in the natural way, so that

the first input to the gate labelled $p = (p_1, \ldots, p_{s_i}) \in \{0,1\}^{s_i}$ at layer i is the gate with label $(p, 0)$ at layer $i-1$, and the second input to gate p has label $(p, 1)$. Here and throughout, $(p, 0)$ denotes the $s_i + 1$-dimensional vector obtained by concatenating the entry 0 to the end of the vector p. Interpreting $p = (p_1, \ldots, p_{s_i}) \in \{0,1\}^{s_i}$ as an integer between 0 and $2^{s_i} - 1$ with p_1 as the high-order bit and p_{s_i} as the low-order bit, this says that the first in-neighbor of p is $2p$ and the second is $2p + 1$. It follows immediately that for any gate $p \in \{0,1\}^{s_i}$ at layer i, $V_i(p) = \tilde{V}_{i+1}(p, 0) \cdot \tilde{V}_{i+1}(p, 1)$. Invoking Lemma 8.4.1, we obtain the following proposition.

Proposition 8.4.2. *Let C be a circuit consisting of a binary tree of multiplication gates. Then $\tilde{V}_i(z) = \sum_{p \in \{0,1\}^{s_i}} g_z^{(i)}(p)$, where $g_z^{(i)}(p) = \beta_{s_i}(z, p) \cdot \tilde{V}_{i+1}(p, 0) \cdot \tilde{V}_{i+1}(p, 1)$.*

Remark 5. Notice that the polynomial $g_z^{(i)}$ in Proposition 8.4.2 is a degree three polynomial in each variable of p. When applying the sum-check protocol to $g_z^{(i)}$, the prover therefore needs to send 4 field elements per round.

In the case of Proposition 8.4.2, the line $\ell : \mathbb{F} \to \mathbb{F}^{2_{i+1}}$ in the "Reducing to Verification of a Single Point" step has an especially simple expression. Let $r \in \mathbb{F}^{s_i}$ be the vector of random field elements chosen by $\mathcal{V}$ over the execution of the sum-check protocol. Notice that $\ell(0)$ must equal the point $(r, 0) \in \mathbb{F}^{s_i+1}$ i.e., the point whose first s_i coordinates equal r and whose last coordinate equals 0. Similarly, $\ell(1)$ must equal $(r, 1)$. We may therefore express the line ℓ via the equation $\ell(t) = (r, t)$. In this case, $\tilde{V}_{i+1} \circ \ell$ has degree 1 and is implicitly specified when $\mathcal{P}$ sends the claimed values of $\tilde{V}_i(r, 0)$ and $\tilde{V}_i(r, 1)$.

The case of a binary tree of addition gates is similar to the case of multiplication gates.

Proposition 8.4.3. *Let C be a circuit consisting of a binary tree of addition gates. Then $\tilde{V}_i(z) = \sum_{p \in \{0,1\}^{s_i}} g_z^{(i)}(p)$, where $g_z^{(i)}(p) = \beta_{s_i}(z, p) \left(\tilde{V}_{i+1}(p, 0) + \tilde{V}_{i+1}(p, 1) \right)$.*

Remark 6. The polynomial $g_z^{(i)}$ of Proposition 8.4.3 has degree 2 in all variables, rather than degree 3 as in Proposition 8.4.2.

The Polynomials for DISTINCT

We now describe a circuit C for computing the number of non-zero entries of a vector $f \in \mathbb{F}^n$ (this vector should be interpreted as the *frequency vector* of a data stream). A similar circuit was used in conjunction with the GKR protocol in Chapter 7 to yield an efficient protocol with a streaming verifier for DISTINCT, and we borrow heavily from the presentation there. We remark that our refinements enable us to slightly simplify the circuit used in Chapter 7 by avoiding the awkward use of a constant-valued input wire with value set to 1. This causes some gates in our circuit to have fan-in 1 rather than fan-in 2, which is easily supported by our protocol.

The circuit C is tailored for use over the field of cardinality equal to a Mersenne prime $q = 2^k - 1$ for some k. Fields of cardinality equal to a Mersenne prime can support extremely fast arithmetic, and as discussed in Section 7.3.3, there are several Mersenne primes of appropriate magnitude for use within our protocols.

The circuit C exploits Fermat's Little Theorem, computing f_i^{q-1} for each input entry f_i before summing the results. As described in Section 7.3.3, verifying the summation sub-circuit can be handled with a one invocation of the sum-check protocol, or less efficiently by running our protocol for a binary tree of addition gates described in Proposition 8.4.3.

We now turn to describing the part of the circuit computing f_i^{q-1} for each input entry f_i. We may write $q - 1 = 2^k - 2$, whose binary representation is $k - 1$ 1s followed by a 0. Thus, $f_i^{q-1} = \prod_{j=1}^{k-1} f_i^{2^j}$. To compute f_i^{q-1}, the circuit repeatedly squares f, and multiplies together the results "as it goes". In more detail, for $j > 2$ there are two multiplication gates at each layer $d(n) - j$ of the circuit for computing f_i^{q-1}; the first computes f^{2^j} by squaring

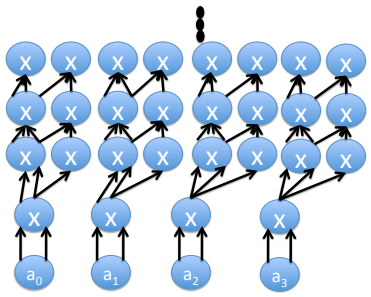


Figure 8.1: The first several layers of a circuit for F_0 on four inputs over the field $\mathbb{F}$ with $q = 2^k - 1$ elements. The first layer from the bottom computes f_i^2 for each input entry f_i. The second layer from the bottom computes f_i^4 and f_i^2 for all i. The third layer computes f_i^8 and $f_i^6 = f_i^4 \times f_i^2$, while the fourth layer computes f_i^{16} and $f_i^{14} = f_i^8 \times f_i^6$. The remaining layers (not shown) have structure identical to the third and fourth layers until the value f_i^{q-1} is computed for all i, and the circuit culminates in a binary tree of addition gates.

the corresponding gate at layer $j-1$, and the second computes $\prod_{\ell=1}^{j-1} f_i^{2^{\ell-1}}$. See Figure 8.1 for a depiction.

For our purposes there are $k+1$ relevant circuit layers, all of which consist entirely of multiplication gates. Layers 1 through $k-1$ all contain $2n$ gates. Number the gates from 0 to $2n-1$ in the natural way. In what follows, we will abuse notation and use p to refer to both a gate number as well as its binary representation.

An even-numbered gate p at layer i has both in-wires connected to gate p at layer $i+1$, while an odd-numbered gate p has one in-wire connected to gate p and another connected to gate $p-1$. Thus, the connectivity information of the circuit is a simple function of the binary representation p of each gate at layer i. If the low-order bit p_{s_i} of p is 0 (i.e., it is an even-numbered gate), then both in-neighbors at layer $i+1$ of gate p have binary representation p. If the low-order bit p_{s_i} is 1 (i.e., it is an odd-numbered gate), then the first in-neighbor of gate p has binary representation p, and the second has binary representation $(p_{-s_i}, 0)$, where p_{-s_i} denotes p with the coordinate p_{s_i} removed.

Invoking Lemma 8.4.1, the following proposition is easily verified.

Proposition 8.4.4. *Let C be the circuit described above. For layers $i \in \{1, \ldots, k-1\}$, $\tilde{V}_i(z) = \sum_{p \in \{0,1\}^{s_i}} g_z^{(i)}(p)$ where*

$$g_z^{(i)}(p) = \beta_{s_i}(z,p) \left((1 - p_{s_i}) \tilde{V}_{i+1}(p_{-s_i}, 0) \cdot \tilde{V}_{i+1}(p_{-s_i}, 0) + p_{s_i} \tilde{V}_{i+1}(p_{-s_i}, 1) \cdot \tilde{V}_{i+1}(p_{-s_i}, 0) \right),$$

where p_{-s_i} denotes p with the coordinate p_{s_i} removed.

Remark 7. To check $\mathcal{P}$'s claim in the final round of the sum-check protocol applied to $g_z^{(i)}$, $\mathcal{V}$ needs to know $\tilde{V}_{i+1}(r, 0)$ and $\tilde{V}_{i+1}(r, 1)$ for some random vector $r \in \mathbb{F}^{s_i - 1}$. This is identical to the situation in the case of a binary tree of addition or multiplication gates, where the "Reducing to Verification of a Single Point" step had an especially simple implementation.

At layer k, an even-numbered gate p has both in-wires connected to gate $p/2$ at layer $k+1$, while an odd-numbered gate p has its unique in-wire connected to gate $(p-1)/2$ at layer $k+1$. Thus, for a gate at layer $i = k$, if the the low-order bit p_{s_i} of the gate's binary representation p is 1 (i.e., it is an odd-numbered gate), then both in-neighbors at layer $i+1$ of have binary representation p_{-s_i}. If the low-order bit p_{s_i} is 0 (i.e., it is an even numbered gate), then the unique in-neighbor of p at layer $i+1$ has binary representation p_{-s_i}.

Invoking Lemma 8.4.1, the following is easily verified.

Proposition 8.4.5. *Let C be the circuit described above. For layer $i = k$,*
$\tilde{V}_i(z) = \sum_{p \in \{0,1\}^{s_i}} g_z^{(i)}(p)$ where

$$g_z^{(i)}(p) = \beta_{s_i}(z,p) \left((1 - p_{s_i}) \tilde{V}_{i+1}(p_{-s_i}) \cdot \tilde{V}_{i+1}(p_{-s_i}) + p_{s_i} \tilde{V}_{i+1}(p_{-s_i}) \right),$$

where p_{-s_i} denotes p with coordinate p_{s_i} removed.

Finally, at layer $k+1$, each gate p has both in-wires connected to gate p at layer $k+2$ (which is the input layer). Thus:

Proposition 8.4.6. *Let C be the circuit described above. For layer $i = k+1$, $\tilde{V}_i(z) = \sum_{p \in \{0,1\}^{s_i}} g_z^{(i)}(p)$ where*

$$g_z^{(i)}(p) = \beta_{s_i}(z,p)\tilde{V}_{i+1}(p) \cdot \tilde{V}_{i+1}(p).$$

8.4.4 Reusing Work

Recall that our analysis of the costs of the sum-check protocol in Section 2.3 revealed that, when applying a sum-check protocol to an s_i-variate polynomial $g_z^{(i)}$, $\mathcal{P}$ only needs to evaluate $g_z^{(i)}$ at $O(2^{s_i})$ points across all rounds of the protocol. Our goal in this section is to show how $\mathcal{P}$ can do this in time $O(2^{s_i} + 2^{s_{i+1}}) = O(S_i + S_{i+1})$ for all of the polynomials $g_z^{(i)}$ described in Section 8.4.3. This is sufficient to ensure that $\mathcal{P}$ takes $O(\sum_{i=1}^{d(n)} S_i) = O(S(n))$ time across all iterations of our circuit-checking protocol.

To this end, notice that all of the polynomials g_z described in Propositions 8.4.2-8.4.6 have the following property: for any $r \in \mathbb{F}^{s_i}$, evaluating $g_z^{(i)}(r)$ can be done in constant time given $\beta(z,r)$ and the evaluations of $\tilde{V}_{i+1}$ at a constant number of points. For example, consider the polynomial $g_z^{(i)}$ described in Proposition 8.4.4: $g_z^{(i)}(r)$ can be computed in constant time given $\beta_{s_i}(z,r)$, $\tilde{V}_{i+1}(r_{-s_i}, 0)$, and $\tilde{V}_{i+1}(r_{-s_i}, 1)$.

Moreover, the points at which $\mathcal{P}$ must evaluate $g_z^{(i)}$ within the sum-check protocol are highly structured: in round j of the sum-check protocol, the points are all of the form $(r_1, \ldots, r_{j-1}, t, b_{j+1}, \ldots, b_{s_i})$ with $t \in \{0, 1, \ldots, \deg_j(g_z^{(i)})\}$ and $(b_{j+1}, \ldots, b_{s_i}) \in \{0,1\}^{s_i - j}$.

Computing the Necessary $\beta(z,p)$ Values

Pre-processing. We begin by explaining how $\mathcal{P}$ can, in $O(2^{s_i})$ time, compute an array $C^{(0)}$ of length 2^{s_i} of all values $\beta(z,p) = \prod_{k=1}^{s_i}(p_k z_k + (1-p_k)(1-z_k))$ for $p \in \{0,1\}^{s_i}$. $\mathcal{P}$ can do this computation in preprocessing before the sum-check protocol begins, as this computation does not depend on any of $\mathcal{V}$'s messages. Naively, computing all entries of $C^{(0)}$ would require

$O(s_i 2^{s_i})$ time, as there are 2^{s_i} values to compute, and each involves $\Omega(s_i)$ multiplications. However, this can be improved using dynamic programming.

The dynamic programming algorithm proceeds in stages. In stage j, $\mathcal{P}$ computes an array $C^{(0,j)}$ of length 2^j. Abusing notation, we identify a number p in $[2^j]$ with its binary representation in $\{0,1\}^j$. $\mathcal{P}$ computes

$$C^{0,j}[p] = \prod_{k=1}^{j}(p_k z_k + (1-p_k)(1-z_k))$$

via the recurrence

$$C^{0,j}[(p_1,\ldots,p_j)] = C^{0,j-1}[(p_1,\ldots,p_{j-1})] \cdot (p_j z_j + (1-p_j)(1-z_j)).$$

Clearly $C^{(0,s_i)}$ equals the desired array $C^{(0)}$, and the total number of multiplications required over the entire procedure is $O(\sum_{j=1}^{s_i} 2^j) = O(2^{s_i})$. We remark that our dynamic programming procedure is similar to the method used by Vu et al. to reduce the verifier's runtime in the GKR protocol from $O(n \log n)$ to $O(n)$ in Lemma 8.3.1.

Overview of Online Processing. In round j of of the sum-check protocol, $\mathcal{P}$ needs to evaluate the polynomial $\beta(z,p)$ at $O(2^{s_i-j})$ points, all of which are of the form $(r_1,\ldots,r_{j-1},t,b_{j+1},\ldots,b_{s_i})$ for $t \in [\deg_j(g_z^{(i)})]$ and $(b_{j+1},\ldots,b_{s_i}) \in \{0,1\}^{s_i-j}$. $\mathcal{P}$ will do this using the help of intermediate arrays $C^{(j)}$ defined as follows.

Define $C^{(j)}$ to be the array of length 2^{s_i-j} such that for $(p_{j+1},\ldots,p_{s_i}) \in \{0,1\}^{s_i-j}$:

$$C^{(j)}[(p_{j+1},\ldots,p_{s_i})] = \left(\prod_{k=1}^{j}(r_k z_k + (1-r_k)(1-z_k))\right) \cdot \left(\prod_{k=j+1}^{s_i}(p_k z_k + (1-p_k)(1-z_k))\right),$$

Efficiently Constructing $C^{(j)}$ Arrays. Inductively, assume $\mathcal{P}$ has computed the array $C^{(j-1)}$ in the previous round. As the base case, we explained how $\mathcal{P}$ can evaluate $C^{(0)}$ in $O(2^{s_i})$ time in pre-processing. Now observe that $\mathcal{P}$ can compute $C^{(j)}$ given $C^{(j-1)}$ in $O(2^{s_i-j})$ time using the following recurrence:

$$C^{(j)}[(p_{j+1},\ldots,p_{s_i})] = z_j^{-1} C^{(j-1)}[(1,p_{j+1},\ldots,p_{s_i})] \cdot (r_j z_j + (1-r_j)(1-z_j)). \tag{8.2}$$

Remark 8. Equation (8.2) is only valid when $z_j \neq 0$. To avoid this issue, we can have $\mathcal{V}$ choose z_j at random from $\mathbb{F}^*$ rather than from $\mathbb{F}$, and this will affect the soundness probability by at most an additive $O(d(n) \cdot \log S(n)/|\mathbb{F}|)$ term.

Remark 9. Since computing multiplicative inverses in a finite field is not a constant-time operation, it is important to note that z_j^{-1} only needs to be computed once when determining the entries of $C^{(j)}$, i.e., it need not be recomputed for each entry of $C^{(j)}$. Therefore, across all s_i rounds of the sum-check protocol, only $\tilde{O}(s_i)$ time in total is required to compute these multiplicative inverses, which does not affect the asymptotic costs for $\mathcal{P}$. We discount the costs of computing z_j^{-1} for the remainder of the discussion.

Thus, at the end of round j of the sum-check protocol, when $\mathcal{V}$ sends $\mathcal{P}$ the value r_j, $\mathcal{P}$ can compute $C^{(j)}$ from $C^{(j-1)}$ using Equation (8.2) in $O(2^{s_i-j})$ time.

Using the $C^{(j)}$ Arrays. Observe that given any point of the form $p = (r_1, \ldots, r_{j-1}, t, b_{j+1}, \ldots, b_{s_i})$ with $(b_{j+1}, \ldots, b_{s_i}) \in \{0,1\}^{s_i-j}$, $\beta(z, p)$ can be evaluated in constant time using the array $C^{(j-1)}$, using the equality

$$\beta(z, p) = C^{(j-1)}[(1, p_{j+1}, \ldots, p_{s_i})] \cdot z_j^{-1} \cdot (tz_j + (1-t)(1-z_j)).$$

As above, note that z_j^{-1} can be computed just once and used for all points p, and this does not affect the asymptotic costs for $\mathcal{P}$.

Putting Things Together. In round j of the sum-check protocol, $\mathcal{P}$ uses the array $C^{(j-1)}$ to evaluate the $O(2^{s_i-j})$ required $\beta(z, p)$ values in $O(2^{s_i-j})$ time. At the end of round j, $\mathcal{V}$ sends $\mathcal{P}$ the value r_j, and $\mathcal{P}$ computes $C^{(j)}$ from $C^{(j-1)}$ in $O(2^{s_i-j})$ time. In total across all rounds of the sum-check protocol, $\mathcal{P}$ spends $O(\sum_{j=1}^{s_i} 2^{s_i-j}) = O(2^{s_i})$ time to compute the $\beta(z, p)$ values.

Computing the Necessary $\tilde{V}_{i+1}(p)$ Values

For concreteness and clarity, we restrict our presentation within this subsection to the polynomial $g_z^{(i)}$ described in Proposition 8.4.4. Theorem 8.4.10 abstracts this analysis into a general result capturing a large class of wiring patterns.

Recall that all of the polynomials $g_z^{(i)}$ described in Propositions 8.4.2-8.4.6 have the following property: for any $p \in \mathbb{F}^{s_i}$, evaluating $g_z^{(i)}(p)$ can be done in constant time given $\beta(z,p)$ and the evaluations of $\tilde{V}_{i+1}$ at a constant number of points. We have already shown how $\mathcal{P}$ can evaluate all of the necessary $\beta(z,p)$ values in $O(2^{s_i})$ time. It remains to show how $\mathcal{P}$ can evaluate all of the $\tilde{V}_{i+1}$ values in time $O(2^{s_i} + 2^{s_{i+1}})$. We remark that in the context of Proposition 8.4.4, $s_i = s_{i+1}$; however, we still distinguish between these two quantities throughout this subsection in order to ensure maximal consistency with the general derivation of Theorem 8.4.10.

Recall that the polynomial $g_z^{(i)}$ in Proposition 8.4.4 was defined as follows:

$$g_z^{(i)}(p) = \beta_{s_i}(z,p)\left((1-p_{s_i})\tilde{V}_{i+1}(p_{-s_i},0)\cdot\tilde{V}_{i+1}(p_{-s_i},0) + p_{s_i}\tilde{V}(p_{-s_i},1)\cdot\tilde{V}(p_{-s_i},0)\right).$$

In round j of the sum-check protocol, $\mathcal{P}$ needs to evaluate g_z at all points in the set

$$S^{(j)} = \{(r_1,\ldots,r_{j-1},t,b_{j+1},\ldots,b_{s_i}) : t \in \{0,\ldots,\deg_j(g_z^{(i)})\} \wedge (b_{j+1},\ldots,b_{s_i}) \in \{0,1\}^{s_i-j}\}.$$

By inspection of $g_z^{(i)}$, it suffices for $\mathcal{V}$ to evaluate $\tilde{V}_{i+1}$ at the same set of points. To show how to accomplish this efficiently, we exploit the following explicit expression for $\tilde{V}_{i+1}$. This expression was derived for the case $i+1=d$ in Equation (6.1) within Lemma 6.3.1; we re-derive it here in the general case.

For a vector $b \in \{0,1\}^{s_{i+1}}$ let $\chi_b(x_1,\ldots,x_{s_{i+1}}) = \prod_{k=1}^{s_{i+1}} \chi_{b_k}(x_k)$, where $\chi_0(x_k) = 1-x_k$ and $\chi_1(x_k) = x_k$. With this definition in hand, we may write:

$$\tilde{V}_{i+1}(p_1, \ldots, p_{s_{i+1}}) = \sum_{b \in \{0,1\}^{s_{i+1}}} V_{i+1}(b)\chi_b(p_1, \ldots p_{s_{i+1}}), \tag{8.3}$$

To see that Equation (8.3) holds, notice that the right hand side of Equation (8.3) is a multilinear polynomial in the variables $(p_1, \ldots, b_{p_{i+1}})$, and that it agrees with V_{i+1} at all points $p \in \{0,1\}^{s_{i+1}}$. Hence, it must be the unique multilinear extension of V_{i+1}.

The intuition behind our optimizations is the following. In round j of the sum-check protocol, there are $|S^{(j)}|$ points at which $\tilde{V}_{i+1}$ must be evaluated. Equation (8.3) can be exploited to show that each gate at layer $i+1$ of the circuit contributes to $\tilde{V}_{i+1}(p)$ for at most one point $p \in S^{(j)}$; namely the point p whose last $s_{i+1} - j$ coordinates agrees with those of p. This observation alone is enough to achieve an $O(S_{i+1} \log S_i)$ runtime for $\mathcal{P}$ in total across all iterations of the sum-check protocol, because there are S_{i+1} gates at layer $i+1$, and only $s_i = \log S_i$ rounds of the sum-check protocol – this is precisely how we enabled the prover in the GKR protocol to run in total time $O(S \log S)$ in Chapter 7. However, we need to go further in order to shave off the last $\log S_i$ factor from $\mathcal{P}$'s runtime. Essentially, what we do is group the gates at layer $i+1$ by the point $p \in S^{(j)}$ to which they contribute. Each such group can be treated as a single unit, ensuring that the work $\mathcal{P}$ has to do in any round of the sum-check protocol in order to evaluate $\tilde{V}_{i+1}$ at all points in $S^{(j)}$ is proportional to $|S^{(j)}|$ rather than to S_{i+1}. Since the size of $S^{(j)}$ falls geometrically with j, our desired time bounds follow.

Pre-processing. $\mathcal{P}$ will begin by computing an array $V^{(0)}$, which is simply defined to be the vector of gate values at layer $i+1$, i.e., identifying a number $0 < j < S_{i+1}$ with its binary representation in $\{0,1\}^{s_{i+1}}$, $\mathcal{P}$ sets $V^{(0)}[(j_1, \ldots, j_{s_{i+1}})] = V_{i+1}(j_1, \ldots, j_{s_{i+1}})$ for each $(j_1, \ldots, j_{s_{i+1}}) \in \{0,1\}^{s_{i+1}}$. The right hand side of this equation is simply the value of the jth gate at layer $i+1$ of C. So $\mathcal{P}$ can fill in the array $V^{(0)}$ when she evaluates the circuit C, before receiving any messages from $\mathcal{V}$.

Overview of Online Processing. In round j of of the sum-check protocol, $\mathcal{P}$ needs to evaluate the polynomial $\tilde{V}_{i+1}$ at the $O(2^{s_i-j})$ points in the set $S^{(j)}$. $\mathcal{P}$ will do this using the help of intermediate arrays $V^{(j)}$ defined as follows.

Define $V^{(j)}$ to be the length $2^{s_{i+1}-j}$ array such that for $(p_{j+1},\ldots,p_{s_{i+1}}) \in \{0,1\}^{s_{i+1}-j}$,

$$V^{(j)}[(p_{j+1},\ldots,p_{s_{i+1}})] = \sum_{(b_1,\ldots,b_j)\in\{0,1\}^j} V_{i+1}(b_1,\ldots,b_j,p_{j+1},\ldots,p_{s_{i+1}}) \cdot \prod_{k=1}^{j} \chi_{b_k}(r_k),$$

Efficiently Constructing $V^{(j)}$ Arrays. Inductively, assume $\mathcal{P}$ has computed in the previous round the array $V^{(j-1)}$ of length $2^{s_{i+1}-j+1}$.

As the base case, we explained how $\mathcal{P}$ can fill in $V^{(0)}$ in the process of evaluating the circuit C. Now observe that $\mathcal{P}$ can compute $V^{(j)}$ given $V^{(j-1)}$ in $O(2^{s_{i+1}-j})$ time using the following recurrence:

$$V^{(j)}[(p_{j+1},\ldots,p_{s_{i+1}})] = V^{(j-1)}[(0,p_{j+1},\ldots,p_{s_i})] \cdot \chi_0(r_j) + V^{(j-1)}[(1,p_{j+1},\ldots,p_{s_i})] \cdot \chi_1(r_j).$$

Thus, at the end of round j of the sum-check protocol, when $\mathcal{V}$ sends $\mathcal{P}$ the value r_j, $\mathcal{P}$ can compute $V^{(j)}$ from $V^{(j-1)}$ in $O(2^{s_{i+1}-j+1})$ time.

Using the $V^{(j)}$ Arrays. We now show how to use the array $V^{(j-1)}$ to evaluate $\tilde{V}_{i+1}(p)$ in constant time for any point of the form $p = (r_1,\ldots,r_{j-1},t,b_{j+1},\ldots,b_{s_{i+1}})$ with $(b_{j+1},\ldots,b_{s_{i+1}}) \in$

$\{0,1\}^{s_{i+1}-j}$. We exploit the following sequence of equalities:

$$
\begin{aligned}
&\tilde{V}_{i+1}(r_1, \ldots, r_{j-1}, t, b_{j+1}, \ldots, b_{s_i}) \\
&= \sum_{c \in \{0,1\}^{s_{i+1}}} V_{i+1}(c) \chi_c(r_1, \ldots, r_{j-1}, t, b_{j+1}, \ldots, b_{s_{i+1}}) \\
&= \sum_{(c_1,\ldots,c_j) \in \{0,1\}^j} \sum_{(c_{j+1},\ldots,c_{s_{i+1}}) \in \{0,1\}^{s_{i+1}-j}} V_{i+1}(c) \chi_c(r_1, \ldots, r_{j-1}, t, b_{j+1}, \ldots, b_{s_{i+1}}) \\
&= \sum_{(c_1,\ldots,c_j) \in \{0,1\}^j} \sum_{(c_{j+1},\ldots,c_{s_{i+1}}) \in \{0,1\}^{s_{i+1}-j}} V_{i+1}(c) \left(\prod_{k=1}^{j-1} \chi_{c_k}(r_k) \right) \left(\chi_{c_j}(t) \right) \left(\prod_{k=j+1}^{s_{i+1}} \chi_{c_k}(b_k) \right) \\
&= \sum_{(c_1,\ldots,c_j) \in \{0,1\}^j} V_{i+1}(c_{j+1}, \ldots, c_j, b_{j+1}, \ldots, b_{s_{i+1}}) \left(\prod_{k=1}^{j-1} \chi_{c_k}(r_k) \right) \cdot \chi_{c_j}(t) \\
&= V^{(j-1)}[(0, b_{j+1}, \ldots, b_{s_{i+1}})] \cdot \chi_0(t) + V^{(j-1)}[(1, b_{j+1}, \ldots, b_{s_{i+1}})] \cdot \chi_1(t).
\end{aligned}
$$

Here, the first equality holds by Equation (8.3). The third holds by definition of the function χ_c. The fourth holds because for Boolean values $b_k, c_k \in \{0,1\}$, $\chi_{c_k}(b_k) = 1$ if $c_k = b_k$, and $\chi_{c_k}(b_k) = 0$ otherwise. The final equality holds by definition of the array $V^{(j-1)}$.

Putting Things Together. In round j of the sum-check protocol, $\mathcal{P}$ uses the array $V^{(j-1)}$ to evaluate $\tilde{V}_{i+1}(p)$ for all $O(2^{s_i-j})$ points $p \in S^{(j)}$. This requires constant time per point, and hence $O(2^{s_i-j})$ time across all points. At the end of round j, $\mathcal{V}$ sends $\mathcal{P}$ the value r_j, and $\mathcal{P}$ computes $V^{(j)}$ from $V^{(j-1)}$ in $O(2^{s_{i+1}-j})$ time. In total across all rounds of the sum-check protocol, $\mathcal{P}$ spends $O(\sum_{j=1}^{s_i} 2^{s_i-j} + 2^{s_{i+1}-j}) = O(2^{s_i} + 2^{s_{i+1}})$ time to evaluate $\tilde{V}_{i+1}$ at the relevant points. When combined with our $O(2^{s_i})$-time algorithm for computing all the relevant $\beta(z,p)$ values, we see $\mathcal{P}$ takes $O(2^{s_i} + 2^{s_{i+1}}) = O(S_i + S_{i+1})$ time to run the entire sum-check protocol for iteration i of our circuit-checking protocol.

8.4.5 A General Theorem

In this section we formalize a large class of circuits to which our refinements yield asymptotic savings relative to prior implementations of the GKR protocol. Our protocol makes use of the following functions that capture the wiring structure of an arithmetic circuit C.

Definition 8.4.7. Let C be a layered arithmetic circuit of depth $d(n)$ and size $S(n)$ over finite field $\mathbb{F}$. For every $i \in \{1, \ldots, d-1\}$, let $\text{in}_1^{(i)} : \{0,1\}^{s_i} \to \{0,1\}^{s_{i+1}}$ and $\text{in}_2^{(i)} : \{0,1\}^{s_i} \to \{0,1\}^{s_{i+1}}$ denote the functions that take as input the binary label p of a gate at layer i of C, and output the binary label of the first and second in-neighbor of gate p respectively. Similarly, let $\text{type}^{(i)} : \{0,1\}^{s_i} \to \{0,1\}$ denote the function that takes as input the binary label p of a gate at layer i of C, and outputs 0 if p is an addition gate, and 1 if p is a multiplication gate.

Intuitively, the following definition captures functions whose outputs are simple bit-wise transformations of their inputs.

Definition 8.4.8. Let f be a function mapping $\{0,1\}^v$ to $\{0,1\}^{v'}$. Number the v input bits from 1 to v, and the v' output bits from 1 to v'. Assume that one machine word contains $\Omega(v + v')$ bits. We say that f is *regular* if f can be evaluated on any input in constant time, and there is a subset of input bits $\mathcal{S} \subseteq [v]$ with $|\mathcal{S}| = O(1)$ such that:

1. Each input bit in $[v] \setminus \mathcal{S}$ affects $O(1)$ of the output bits of f. Moreover, given input $j \in [v] \setminus \mathcal{S}$, the set $\mathcal{S}_j$ of output bits affected by x_j can be enumerated in constant time.

2. Each output bit of f depends on at most one input bit.

Our protocol applied to C proceeds in $d(n)$ iterations, where iteration i consists an application of the sum-check protocol to an appropriate polynomial derived from $\text{type}^{(i)}$, $\text{in}_1^{(i)}$, and $\text{in}_2^{(i)}$, followed by a phase for "reducing to verification of a single point". For any

layer i of C such that $\text{in}_1^{(i)}$, $\text{in}_2^{(i)}$ and $\text{type}^{(i)}$ are all regular, we can show that $\mathcal{P}$ can execute the sum-check protocol at iteration i in $O(S_i + S_{i+1})$ time. To ensure that $\mathcal{P}$ can execute the "reducing to verification of a single point" phase in $O(S_{i+1})$ time, we need to place one additional condition on $\text{in}_1^{(i)}$ and $\text{in}_2^{(i)}$.

Definition 8.4.9. We say that $\text{in}_1^{(i)}$ and $\text{in}_2^{(i)}$ are *similar* if there is a set of output bits $\mathcal{T} \subseteq [s_{i+1}]$ with $|\mathcal{T}| = O(1)$ such that for all inputs x, the jth output bit of $in_1^{(i)}$ equals the jth output bit of $in_2^{(i)}$ for all $j \in [s_{i+1}] \setminus \mathcal{T}$.

We are finally in a position to state the class of circuits to which our refinements apply.

Theorem 8.4.10. *Let C be an arithmetic circuit, and suppose that for all layers i of C, $in_1^{(i)}$, $in_2^{(i)}$, and $type^{(i)}$ are regular. Suppose moreover that $in_1^{(i)}$ is similar to $in_2^{(i)}$ for all but $O(1)$ layers i of C. Then there is a valid interactive proof protocol $(\mathcal{P}, \mathcal{V})$ for the function computed by C, with the following costs. The total communication cost is $|\mathcal{O}| + O(d(n) \log S(n))$ field elements, where $|\mathcal{O}|$ is the number of outputs of C. The time cost to $\mathcal{V}$ is $O(n \log n + d(n) \log S(n))$, and $\mathcal{V}$ can make a single streaming pass over the input, storing $O(\log(S(n)))$ field elements. The time cost to $\mathcal{P}$ is $O(S(n))$.*

The asymptotic costs of the protocol whose existence is guaranteed by Theorem 8.4.10 are identical to those of the implementation of the GKR protocol from Chapter 7, except that in Theorem 8.4.10 $\mathcal{P}$ runs in time $O(S(n))$ rather than $O(S(n) \log S(n))$ as achieved in Chapter 7. We defer the proof to Appendix A.1.

Applications

Theorem 8.4.10 applies to circuits computing functions from a wide range of applications, with the following implications.

MatMult. Consider the following circuit C of size $O(n^3)$ for multiplying two $n \times n$ matrices A and B. Let the input gate labelled $(0, i, j)$ correspond to A_{ij}, and the input labelled $(1, i, j)$ correspond to B_{ij}. The layer of C adjacent to the input consists of n^3 gates, where the gate labeled $(i, j, k) \in (\{0,1\}^{\log n})^3$ computes $A_{ik} \cdot B_{kj}$. All subsequent layers constitute a binary tree of addition gates summing up the results and thereby computing $\sum_k A_{ik}B_{kj}$ for all $(i, j) \in [n] \times [n]$.

For layers $i \in \{1, \ldots, \log n\}$ of this circuit, $\text{in}_1^{(i)}$, $\text{in}_2^{(i)}$, and $\text{type}^{(i)}$ are all regular, and moreover $\text{in}_1^{(i)}$ is similar to $\text{in}_2^{(i)}$ (see Section 8.4.3 for a careful treatment of this wiring pattern). The remaining layer of the circuit, layer $i = \log n + 1$, is regular, though $\text{in}_1^{(\log n+1)}$ and $\text{in}_2^{(\log n+1)}$ are not similar. We obtain the following immediate corollary.

Corollary 8.4.11. *There is a valid interactive proof protocol for $n \times n$ MatMult with the following costs. The total communication cost is $n^2 + O(d(n) \log n)$ field elements, where the n^2 term is required to specify the answer. The time cost to $\mathcal{V}$ is $O(n^2 \log n)$, and $\mathcal{V}$ can make a single streaming pass over the input in time $O(n^2 \log n)$ and storing $O(\log n)$ field elements. The time cost to $\mathcal{P}$ is $O(n^3)$.*

We note that the costs of Corollary 8.4.11 are subsumed by our special-purpose matrix multiplication protocol presented later in Theorem 8.7.2. We included Corollary 8.4.11 to demonstrate the applicability of Theorem 8.4.10.

Distinct. Recall the circuit C over field size $q = 2^k - 1$ described in Section 8.4.3 that takes a vector $a \in \mathbb{F}^n$ as input and outputs the number of non-zero entries of a. This circuit has $k + 1$ relevant layers and consists entirely of multiplication gates. For any layer $i \in [k - 1]$, an even-numbered gate p at layer i has both in-wires connected to gate p at layer $i + 1$, while an odd-numbered gate p at layer i has one in-wire connected to gate p at layer $i + 1$ and another connected to gate $p - 1$ (which has binary representation $(p_{-s_i}, 0)$, where p_{-s_i}

denotes the binary representation of p with the coordinate p_{s_i} removed). For these layers, $\text{in}_1^{(i)}$, $\text{in}_2^{(i)}$, and $\text{type}^{(i)}$ are all regular, and $\text{in}_1^{(i)}$ is similar to $\text{in}_2^{(i)}$.

At layer k, an even-numbered gate p is has both in-wires connected to gate $p/2$ at layer $k+1$, while an odd-numbered gate p at layer k has its unique in-wire connected to gate $(p-1)/2$ at layer $k+1$. In the former case, both in-neighbors of gate p have binary representation p_{-s_i}. In the latter case the unique in-neighbor of gate p has binary representation p_{-s_i}. It is therefore easily seen that $\text{in}_1^{(k)}$, $\text{in}_2^{(k)}$, and $\text{type}^{(k)}$ are all regular, and $\text{in}_1^{(k)}$ is similar to $\text{in}_2^{(k)}$. Finally, at layer $k+1$, both in-wires for gate p are connected to gate p at layer $k+2$. It is easily seen that $\text{in}_1^{(k+1)}, \text{in}_2^{(k+1)}$, and $\text{type}^{(k+1)}$ are all regular, and $\text{in}_1^{(k+1)}$ is similar to $\text{in}_2^{(k+1)}$. With all layers of C satisfying the requirements of Theorem 8.4.10, we obtain the following corollary.

Corollary 8.4.12. *Let $q > \max\{m, n\}$ be a Mersenne Prime. There is a valid interactive proof protocol over the field $\mathbb{F}_q$ for* DISTINCT *with the following costs. The total communication cost is $O(\log n \log q)$ field elements. The time cost to $\mathcal{V}$ is $O(m \log n)$, and $\mathcal{V}$ can make a single streaming pass over the input, storing $O(\log n)$ field elements. The time cost to $\mathcal{P}$ is $O(n \log q)$.*

To or knowledge, Corollary 8.4.12 yields the fastest known prover of any streaming interactive proof protocol for DISTINCT that also has total communication and space usage for $\mathcal{V}$ that is sublinear in both m and n. The fastest result previously was the $O\left(n \cdot \log(n) \cdot \log(p)\right)$-time prover obtained by the implementation of Chapter 7. We remark however that for a data stream with F_0 distinct items, the prover in the protocol of Chapter 7 actually can be made to run in time $O\left(n + F_0 \cdot \log(n) \cdot \log(p)\right)$, where the $O(n)$ term is due to the time required to simply observe the entire input stream. Therefore, for streams where $F_0 = o(n/\log n)$, the implementation of Chapter 7 achieves an asymptotically faster prover than implied by Corollary 8.4.12.

Remark 10. In Section 7.3.3, we described how to extend the GKR protocol to handle circuits with gates that compute more general operations than just addition and multiplication. At a high level, we showed in Section 7.3.3 that gates computing any "low-degree" operation can be handled, and they demonstrate analytically and experimentally that these more general gates can achieve cost savings for the DISTINCT problem. These same optimizations are also applicable in conjunction with our refinements. We omit further details for brevity, and did not implement these optimizations in conjunction with the refinements of this chapter.

Other Problems. In order to demonstrate its generality, we describe two other non-trivial applications of Theorem 8.4.10.

- *Pattern Matching.* In the Pattern Matching problem, the input consists of a stream of text $T = (t_0, \ldots, t_{n-1}) \in [n]^n$ and pattern $P = (p_0, \ldots, p_{m-1}) \in [n]^m$. The pattern P is said to occur at location i in T if, for every position k in P, $p_k = t_{i+k}$. The pattern-matching problem is to determine the number of locations at which P occurs in T. For example, one might want to determine the number of times a given phrase appears in a corpus of emails stored in the cloud.

  Consider the following circuit C for computing Pattern Matching over the finite field $\mathbb{F}_q$ for prime q. The circuit first computes the quantity $I_i = \sum_{j=0}^{m}(t_{i+j} - p_j)^2$ for each $i \in \{0, \ldots, n-1\}$, and then exploits Fermat's Little Theorem (FLT) by computing $M = \sum_{i=1}^{n-m} I_i^{q-1}$. The number of occurrences of the pattern equals $n - m - M$.

  Computing I_i for each i can be done in $\log m + 2$ layers: the layer closest to the input computes $t_{i+k} - p_k$ for each pair $(i, k) \in \{0, \ldots, n-1\} \times \{0, \ldots, q-1\}$, the next layer squares each of the results, and the circuit then sums the results via a depth $\log m$-binary tree of addition gates. The total size of the circuit C is $O(nm + n \log q)$, where the nm term is due to the computation of the I_i values, and the $n \log q$ term is due to

the FLT computation. The total depth of the circuit is $O(\log m + \log q) = O(\log q)$.

We have already demonstrated that Theorem 8.4.10 applies to the squaring layer, the binary tree sub-circuit, and the FLT computation. The only remaining layer of the circuit is the one that computes $t_{i+k} - p_k$ for each pair $(i, k) \in \{0, \ldots, n-1\} \times \{0, \ldots, m-1\}$. Unfortunately, Theorem 8.4.10 does *not* apply to this layer of the circuit. This is because the first in-neighbor of a gate with label $(i_1, \ldots, i_{\log n}, k_1, \ldots, k_{\log m}) \in \{0,1\}^{\log n + \log m}$ has label equal to the binary representation of the integer $i + k$, and a single bit i_j can affect many bits in the binary representation of $i + k$ (likewise, each bit in the binary representation of $i + k$ may be affected by many bits in the binary representation of i and k).

However, in Appendix A.2, we describe how to extend the ideas underlying Theorem 8.4.10 to handle this wiring pattern. The extensions in Appendix A.2 may be more broadly useful, as the wiring pattern analyzed there is an instance of a common paradigm, in that it interprets binary gate labels as a pair of integers and performs a simple arithmetic operation (namely addition) on those integers.

We also remark that, instead of going through the analysis of Appendix A.2, a more straightforward approach is to simply apply the implementation of Chapter 7 to this layer; the runtime for $\mathcal{P}$ in the corresponding sum-check protocol is $O(nm \log n)$. This does not affect the asymptotic costs of the protocol if m is constant, since in this case $nm \log n = O(n \log q)$, and the total runtime of $\mathcal{P}$ over all other layers of the circuit is $\Theta(n \log q)$.

This analysis highlights the following point: our refinements can be applied to a circuit on a layer-by-layer basis, so they can still yield speedups even if some but not all layers of a circuit are sufficiently "regular" for our refinements to apply.

A similar analysis applies to a closely related circuit that solves a more general problem known as Pattern Matching with Wildcards. We omit these details for brevity.

- *Fast Fourier Transform.* Consider the following circuit over $\mathbb{C}$ for computing the standard radix-two decimation-in-time FFT. At a high level, this circuit works as follows. It proceeds in $\log n$ stages, where for $k = (k_1, \ldots, k_n) \in \{0,1\}^n$, the kth output of stage i is recursively defined as $V_i(k_1, \ldots, k_n) = V_{i-1}(k_1, k_{i-1}, 0, k_i, \ldots, k_n) + e^{-2\pi ki/n} V_{i-1}(k_1, \ldots, k_{i-1}, 1, k_{i+1}, \ldots, k_n)$. Theorem 8.4.10 is easily seen to apply to the natural circuit executing this recurrence, and our refinements would therefore shave a logarithmic factor off the runtime of $\mathcal{P}$ applied to this circuit, relative to the implementation of Chapter 7 (since this circuit is defined over the infinite field $\mathbb{C}$, the protocol is only defined in a model where complex numbers can be communicated and operated on at unit cost).

8.5 Experimental Results

We implemented the protocols implied by Theorem 8.4.10 as applied to circuits computing MatMult and Distinct. These experiments serve as case studies to demonstrate the feasibility of Theorem 8.4.10 in practice, and to quantify the improvements over prior implementations. While Section 8.7 describes a specialized protocol for MatMult that is significantly more efficient than the protocol implied by Theorem 8.4.10, MatMult serves as an important case study for the costs of the more general protocol described in Theorem 8.4.10, and allows for direct comparison with prior implementation work that also evaluated general-purpose protocols via their performance on the MatMult problem [39, 82, 91, 92, 99, 104]. Our comparison point is the implementation of Chapter 7.

8.5.1 Summary of Results

The main takeaways of our experiments are as follows. When Theorem 8.4.10 is applicable, the prover in the resulting protocol is 200x-250x faster than the previous state of the art implementation of the GKR protocol. The communication costs and the number of rounds required by our protocols are also 2x-3x smaller than the previous state of the art. The verifier in our implementation takes essentially the same amount of time as in prior implementations of the GKR protocol; this time is much smaller than the time to perform the computation locally without a prover.

Most of the observed 200x speedup can be attributed directly to our improvements in protocol design over prior work: the circuit for 512x512 matrix multiplication is of size 2^{28}, and hence our $\log S$ factor improvement the runtime of $\mathcal{P}$ likely accounts for at least a 28x speedup. The 3x reduction in the number of rounds accounts for another 3x speedup. The remaining speedup factor of roughly 2x may be due to a more streamlined implementation relative to prior work, rather than improved protocol design per se.

We have both a serial implementation and a parallel implementation that leverages graphics processing units (GPUs). The prover in our parallel implementation runs roughly 30x faster than the prover in our serial implementation. The ability to leverage GPUs to obtain robust speedups in our setting is not unexpected, as in Section 7.5.3 we used GPUs to obtain substantial speedups using GPUs for the GKR protocol implementation from Chapter 7.

All of our code used in this chapter is available online at [96]. All of our serial code was written in C++ and all experiments were compiled with g++ using the $-$O3 compiler optimization flag and run on a workstation with a 64-bit Intel Xeon architecture and 48 GBs of RAM. We implemented all of our GPU code in CUDA and Thrust [65] with all compiler optimizations turned on, and ran our GPU implementation on an NVIDIA Tesla C2070 GPU with 6 GBs of device memory.

8.5.2 Details

Choice of Finite Field. All of our circuits work over the finite field of size $q = 2^{61} - 1$, as with the case for our implementation in Chapter 7. We use this field because it supports fast arithmetic, as reducing an integer modulo q can be done with a bit-shift, addition, and a bit-wise AND. (The same observation applies to any field whose size equals a Mersenne Prime, including $2^{89} - 1$, $2^{107} - 1$, and $2^{127} - 1$). Moreover, the field is large enough that the probability a verifier is fooled by a dishonest prover is smaller than $1/2^{45}$ for all of the problems we consider (this probability is proportional to $\frac{d(n) \log S(n)}{q}$).

As in Chapter 7, the main potential issue with our choice of field size is that "overflow" can occur for problems like matrix multiplication if the entries of the input matrices can be very large. All of our protocols can be instantiated over fields with more than $q = 2^{61} - 1$ elements, with an implementation using these fields experiencing a slowdown proportional to the increased cost of arithmetic over these fields.

Serial Implementation

MatMult. The costs of our serial MatMult implementation are displayed in Table 8.1. The prover in our matrix multiplication implementation is about 250x faster than the implementation of Chapter 7. For example, when multiplying two 512 x 512 matrices, our prover takes about 38 seconds, while our comparison implementation takes over 2.5 hours. A C++ program that simply evaluates the circuit without an integrity guarantee takes 6.07 seconds, so our prover experiences less than a 7x slowdown in order to evaluate the circuit with an integrity guarantee relative to simply evaluating the circuit without such a guarantee.

When multiplying two 512 x 512 matrices A and B, the protocol requires 236 rounds, and the total communication cost of our protocol is 5.48 KBs (plus the amount of communication

required to specify the answer AB). The implementation of Chapter 7 required 767 rounds and close to 18 KBs of communication (plus the amount of communication required to specify AB). Notice that specifying a 512x512 matrix using 8 bytes per entry requires 2 MBs, which is more than 500 times larger than the 5.48 KBs of extra communication required to verify the answer.

A serial C++ program performing 512 x 512 matrix multiplication over the integers with floating point arithmetic (without going through the circuit representation of the computation) required 1.53 seconds, so our prover runs approximately 25 times slower than a standard unverifiable matrix multiplication algorithm. A serial C++ program performing the same multiplication over the finite field of size $2^{61} - 1$ required 4.74 seconds, so our serial prover runs about 8 times slower than an unverifiable matrix multiplication algorithm over the corresponding finite field.

Our verifier takes essentially the same amount of time as in prior work, as in both implementations the bulk of the work of the verifier is spent evaluating the low-degree extension of the input at a point. This is more than an order of magnitude faster than the 1.03 seconds required by a serial C++ program performing the multiplication in an unverified manner over the integers, so the verifier is indeed saving time by using a prover (relative to doing the computation locally without a prover). We stress that the savings for the verifier would be larger at larger input sizes, as the time cost to the verifier in our implementation and the prior implementation of Chapter 7 is quasilinear in the input size, which is polynomially faster than all known matrix multiplication algorithms. Moreover, when streaming considerations are not an issue, we could apply the refinement of Vu et al. from Lemma 8.3.1 to reduce $\mathcal{V}$'s runtime from $O(n^2 \log n)$ to $O(n^2)$ and thereby further speed up the verifier.

DISTINCT. The costs of our serial DISTINCT implementation with $n = 2^{20}$ (i.e., for computing the number of non-zero entries of a vector of length $n = 2^{20}$) are displayed in Table

Table 8.1: Experimental results for $n \times n$ MatMult using the protocol of Theorem 8.4.10.

Implementation	Problem Size	$\mathcal{P}$ Time	$\mathcal{V}$ Time	Rounds	Total Communication	Circuit Eval Time
Ch. 7	256 x 256	1054 s	0.02 s	623	14.6 KBs	0.73 s
Thm. 8.4.10	256 x 256	4.37 s	0.02 s	190	4.4 KBs	0.73 s
Ch. 7	512 x 512	9759 s	0.10 s	767	17.97 KBs	6.07 s
Theorem 8.4.10	512 x 512	37.85 s	0.10 s	236	5.48 KBs	6.07 s

The Total Communication column does not count the communication required to specify the answer, only the "extra" communication required to run the verification protocol.

8.2. The comparison of our implementation with prior work is similar to the case of matrix multiplication. Our prover is roughly 200 times faster than the comparison implementation. For example, when computing the number of non-zero entries of a vector of length 2^{20}, our prover takes about 17 seconds, while our comparison implementation takes about 57 minutes. A C++ program that simply evaluates the circuit without an integrity guarantee takes 1.88 seconds, so our prover experiences roughly a 10x slowdown to provide the integrity guarantee relative to simply evaluating the circuit. Our implementation required 1361 rounds and 40.76 KBs of total communication, compared to 3916 rounds and 91.3 KBs for the implementation of Chapter 7. This is essentially a 3x reduction in the number of rounds, and a 2.25x reduction in the total amount of communication.

A C++ program that (unverifiably) computes the number of non-zero entries in a vector x with 2^{20} entries takes less than .01 seconds, and our prover implementation runs more than $1,700$ times longer than this. The reason that the slowdown for the prover relative to an unverifiable algorithm is larger for Distinct than for MatMult is that Distinct is a "less arithmetic" problem, in the sense that the size of the arithmetic circuit we use for computing Distinct is more than 100x larger than the runtime of an unverifiable serial algorithm for

Table 8.2: Experimental results for DISTINCT ($n = 2^{20}$) using the protocol of Theorem 8.4.10.

Implementation	$\mathcal{P}$ Time	$\mathcal{V}$ Time	Rounds	Total Communication	Circuit Eval Time
Ch. 7	3400.23 s	0.20 s	3916	91.3 KBs	1.88 s
Thm. 8.4.10	17.28 s	0.20 s	1361	40.76 KBs	1.88 s

the problem. We stress however that, as pointed out in [99], when solving the DISTINCT problem in practice, an unverifiable algorithm would first aggregate a data stream into its frequency-vector representation before determining the number of non-zero frequencies. In reporting a time bound of .01 seconds for unverifiably solving DISTINCT, we are not taking the aggregation time cost into account. For sufficiently long data streams, the slow-down for our prover relative to an unverifiable algorithm would be much smaller than $1,700$x if we did take aggregation time into account.

Parallel Implementation

Our serial implementation demonstrates that $\mathcal{P}$ experiences a 10x slowdown to provide the integrity guarantee relative to simply evaluating the circuit without such a guarantee. The purpose of this section is to demonstrate that parallelization can further mitigate this slowdown. To this end, we implemented a parallel version of our prover in the context of the matrix multiplication protocol of Section 8.4. Our parallel implementation uses a graphics processing unit (GPU).

The high-level idea behind our parallel implementation is the following. Each time we apply the sum-check protocol to a polynomial $g_z^{(i)}$, it suffices for $\mathcal{P}$ to evaluate $g_z^{(i)}$ at a large number of points r of the form $p = (r_1, \ldots, r_{j-1}, t, b_{j+1}, \ldots, b_{s_{i+1}})$ with $t \in \{0, \ldots, \deg_j(g_z^{(i)})\}$ and $(b_{j+1}, \ldots, b_{s_{i+1}}) \in \{0,1\}^{s_{i+1}-j}$. We can perform each of these evaluations independently.

Thus, we devote a single thread on the GPU to each value of $(b_{j+1}, \ldots, b_{s_{i+1}}) \in \{0,1\}^{s_{i+1}-j}$ and have that thread evaluate $g_z^{(i)}(r)$ at each of the $\deg_j(g_z^{(i)}) + 1$ points of the form $(r_1, \ldots, r_{j-1}, t, b_{j+1}, \ldots, b_{s_{i+1}})$ with the help of the $C^{(j-1)}$ and $V^{(j-1)}$ arrays described in Section 8.4. The one remaining issue is that after each round j of each invocation of the sum-check protocol, we need to update the arrays, i.e., we need to compute $C^{(j)}$ and $V^{(j)}$. To accomplish this, we devote a single thread to each entry of $C^{(j)}$ and $V^{(j)}$.

All steps of our parallel implementation achieve excellent memory coalescing, which likely plays a significant role in the large speedups we were able to achieve. For example, if two threads are updating adjacent entries of the array $V^{(j)}$, the only memory accesses that the threads need to perform are to adjacent entries of the array $V^{(j-1)}$.

The results are shown in Table 8.3: we obtained about a 30x speedup for the prover relative to our serial implementation. As in our GPU experiments of Chapter 7, the reported prover runtime does count the time required to copy data between the host (CPU) and the device (GPU), but does not count the time required to evaluate the circuit, which our implementation does in serial for simplicity. While our implementation evaluates the circuit serially, this step can in principle be done in parallel one layer at a time, as these circuits have only logarithmic depth. Notice that when the circuit evaluation runtime is excluded, our parallel prover implementation runs faster in the case of 512x512 matrix multiplication than the time required to evaluate the circuit sequentially.

It is possible that we would observe slightly larger speedups at larger input sizes, but our parallel implementation exhausts the memory of the GPU at inputs larger than 512x512. This memory bottleneck was also experienced by our GPU implementation of the GKR protocol described in Chapter 7 and helps motivate the importance of the improved space usage of the special purpose MatMult protocol we give later in Theorem 8.7.2. For comparison, the GPU implementation of [99] required 39.6 seconds for 256 x 256 matrix multiplication,

Table 8.3: Experimental results for $n \times n$ MatMult with our parallel prover implementation.

Implementation	Problem Size	$\mathcal{P}$ Time	Serial Circuit Eval Time
Theorem 8.4.10, Serial Implementation	256 x 256	4.37 s	0.73 s
Theorem 8.4.10, Parallel Implementation	256 x 256	0.23 s	0.73 s
Theorem 8.4.10, Serial Implementation	512 x 512	37.85 s	6.07 s
Theorem 8.4.10, Parallel Implementation	512 x 512	1.29 s	6.07 s

which is about 175x slower than our parallel implementation.

We also mention that Thaler, Roberts, Mitzenmacher, and Pfister [99] demonstrate that equally large speedups via parallelization are achievable for the (already fast) computation of the verifier. These results directly apply to our protocols as well, as the verifier's runtime in both implementations is dominated by the time required to evaluate the MLE of the input at a random point [39, 99].

8.6 Verifying General Data Parallel Computations

In this section, our goal is to extend the applicability of the GKR protocol. While the GKR protocol applies in principle to any function computed by a small-depth circuit, this is not the case when fine-grained efficiency considerations are taken into account. The implementation of Chapter 7 required the programmer to express a program as an arithmetic circuit, and moreover this circuit needed to have a regular wiring pattern, in the sense that the verifier could efficiently evaluate the polynomials $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ at a point. If this was not the case, the verifier would need to do an expensive (though data-independent) preprocessing phase to perform these evaluations. Moreover, even for circuits with regular wiring patterns, this implementation caused the prover to suffer an $O(\log(S(n)))$ factor blowup in runtime relative to evaluating the circuit without a guarantee of correctness. The results of Sections 8.4 and

8.7 asymptotically eliminate the blowup in runtime for the prover, but they also only apply when the circuit has a very regular wiring pattern.

The implementation of Vu et al. [104] allows the programmer to express a program in a high-level language, but compiles these programs into potentially irregular circuits that require the verifier to incur the expensive preprocessing phase mentioned above, in order for the verifier to evaluate the polynomials $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ at a point. They therefore propose to apply their system in a "batching" model, where multiple instances of the same sub-computation are applied independently to different pieces of data. More specifically, their system applies the GKR protocol independently to each application of the computation, and relies on the ability of the verifier to use a single $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ evaluation for all instances of the sub-computation, thereby amortizing the cost of this evaluation across the instances. To clarify, this use of a single $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ evaluation for all instances as in [104] is sound only if all of the instances are checked simultaneously. If the instances are instead verified one after the other, then $\mathcal{P}$ knows $\mathcal{V}$'s randomness in all but the first instance, and can use that knowledge to mislead $\mathcal{V}$.

The batching model of Vu et al. is identical to the data parallel setting we consider here. However, a downside to the solution of Vu et al. is that the verifier's work, as well as the total communication cost of the protocol, grows linearly with the "batch size" – the number of applications of the sub-computation that are being outsourced. We wish to develop a protocol whose costs to both the prover and verifier grow much more slowly with the batch size.

8.6.1 Motivation

As discussed above, existing interactive proof protocols for circuit evaluation either apply only to circuits with highly regular wiring patterns or incur large overheads for the prover and

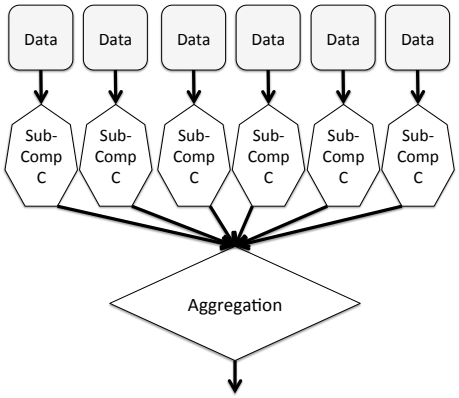


Figure 8.2: Schematic of a data parallel computation.

verifier. Here, we show how to mitigate the bottlenecks of existing protocols by leveraging some general structure underlying many real-world computations. Specifically, the structure we focus on exploiting is data-parallelism.

By data parallel computation, we mean any setting in which the same sub-computation is applied independently to many pieces of data, before possibly aggregating the results. Crucially, we do not want to make significant assumptions on the sub-computation that is being applied (in particular, we want to handle sub-computations computed by circuits with highly irregular wiring patterns), but we are willing to assume that the sub-computation is applied independently to many pieces of data. See Figure 8.2 for a schematic of a data parallel computation.

We have already seen a very simple example of a data parallel computation: the DISTINCT problem. The circuit C from Section 8.4 used to solve this problem takes as input a vector a and computes $a_i^{q-1} \mod q$ for all i (this is the data parallel phase of the computation), before summing the results (this is the aggregation phase). Notice that if the data stream consists of a sequence of words, then the DISTINCT problem becomes the word-count problem, a classic data parallel application.

By design, the protocol of this section also applies to more complicated data parallel

computations. For example, it applies to arbitrary *counting queries* on a database. In a counting query, one applies some function independently to each row of the database and sums the results. For example, one may ask "How many people in the database satisfy Property P?" Our protocol allows one to verifiably outsource such a counting query with overhead that depends minimally on the size of the database, but that necessarily depends on the complexity of the property P.

8.6.2 Overview of the Protocol

Let C be a circuit of size $S(n)$ with an arbitrary wiring pattern, and let C^* be a "super-circuit" that applies C independently to B different inputs before aggregating the results in some fashion. For example, in the case of a counting query, the aggregation phase simply sums the results of the data parallel phase. We assume that the aggregation step is sufficiently simple that the aggregation itself can be verified using existing techniques, and we focus on verifying the data parallel part of the computation.

If we naively apply the GKR protocol to the super-circuit C^*, $\mathcal{V}$ might have to perform an expensive pre-processing phase to evaluate the wiring predicate of C^* at the necessary locations – this would require time $\Omega(B \cdot S)$. Moreover, when applying the basic GKR protocol to C^*, $\mathcal{P}$ would require time $\Theta\left(B \cdot S \cdot \log(B \cdot S)\right)$. A different approach was taken by Vu et al [104], who applied the GKR protocol B independent times, once for each copy of C. This causes both the communication cost and $\mathcal{V}$'s online check time to grow linearly with B, the number of sub-computations.

In contrast, our protocol achieves the best of both prior approaches. We observe that although each sub-computation C can have a very complicated wiring pattern, the circuit is maximally regular between sub-computations, as the sub-computations do not interact at all. Therefore, each time the basic GKR protocol would apply the sum-check protocol to a

polynomial derived from the wiring predicate of C^*, we can instead use a simpler polynomial derived only from the wiring predicate of C. By itself, this is enough to ensure that $\mathcal{V}$'s pre-processing phase requires time only $O(S)$, rather than $O(B \cdot S)$ as in a naive application of the basic GKR protocol. That is, the cost of $\mathcal{V}$'s pre-processing phase is essentially proportional to the cost of applying the GKR protocol only to C, not to the super-circuit C^*.

Furthermore, by combining this observation with the methods of Section 8.4, we can bring the runtime of $\mathcal{P}$ down to $O(B \cdot S \cdot \log S)$. That is, the blowup in runtime suffered by the prover, relative to performing the computation without a guarantee of correctness, is just a factor of $\log S$ – the same as it would be if the prover had run the basic GKR protocol on a single instance of the sub-computation.

8.6.3 Technical Details

Notation

Let C be an arithmetic circuit over $\mathbb{F}$ of depth d and size S with an arbitrary wiring pattern, and let C^* be the circuit of depth d and size $B \cdot S$ obtained by laying B copies of C side-by-side, where $B = 2^b$ is a power of 2. We assume that the in-neighbors of all of the S_i gates at layer i can be enumerated in $O(S_i)$ time. We will use the same notation as in Section 8.4, using *'s to denote quantities referring to C^*. For example, layer i of C has size $S_i = 2^{s_i}$ and gate values specified by the function V_i, while layer i of C^* has size $S_i^* = 2^{s_i^*}$ and gate values specified by the function V_i^*. We denote the length of the input to C^* by $n^* = Bn$.

Main Theorem

Our main theorem gives a protocol for compute $\tilde{V}_1^*(z)$, for any point $z \in \mathbb{F}^{s_1^*}$. The idea is that the verifier would first apply simpler techniques (such as the protocol of Theorem

8.4.10) to the aggregation phase of the computation to obtain a claim about $\tilde{V}_1^*(z)$, and then use our main theorem to verify this claim. Hence, in principle $\mathcal{V}$ need not look at the entire output of the data parallel phase, only the output of the aggregation phase, which we anticipate to be much smaller.

Theorem 8.6.1. *For any point $z \in \mathbb{F}^{s_1^*}$, there is a valid interactive proof protocol for computing $\tilde{V}_1^*(z)$ with the following costs. $\mathcal{V}$ spends $O(S)$ time in a pre-processing phase, and $O(n^* \log n^* + d \cdot \log(B \cdot S))$ time in an online verification phase, where the $n^* \log n^*$ term is due to the time required to evaluate the multilinear extension of the input to C^* at a point. $\mathcal{P}$ runs in total time $O(S \cdot B \cdot \log S)$. The total communication is $O(d \cdot \log(B \cdot S))$ field elements.*

Proof. Consider layer i of C^*. Let $p = (p_1, p_2) \in \{0,1\}^{s_i} \times \{0,1\}^b$ be the label of a gate at layer i of C^*, where p_2 specifies which "copy" of C the gate is in, while p_1 designates the label of the gate within the copy. Similarly, let $\omega = (\omega_1, \omega_2) \in \{0,1\}^{s_{i+1}} \times \{0,1\}^b$ and $\gamma = (\gamma_1, \gamma_2) \in \{0,1\}^{s_{i+1}} \times \{0,1\}^b$ be the labels of two gates at layer $i+1$.

It is straightforward to check that for all $(p_1, p_2) \in \{0,1\}^{s_i} \times \{0,1\}^b$,

$$V_i^*(p_1, p_2) = \sum_{\omega_1 \in \{0,1\}^{s_{i+1}}} \sum_{\gamma_1 \in \{0,1\}^{s_{i+1}}} g_z^{(i)}(p_1, p_2, \omega_1, \gamma_1),$$

where

$$g_z^{(i)}(p_1, p_2, \omega_1, \gamma_1) = \beta_{s_i^*}(z, (p_1, p_2)) \cdot$$
$$\left(\tilde{\text{add}}_i(p_1, \omega_1, \gamma_1) \left(\tilde{V}_{i+1}^*(\omega_1, p_2) + \tilde{V}_{i+1}^*(\gamma_1, p_2) \right) + \tilde{\text{mult}}_i(p_1, \omega_1, \gamma_1) \tilde{V}_{i+1}^*(\omega_1, p_2) \cdot \tilde{V}_{i+1}^*(\gamma_1, p_2) \right)$$

Essentially, this equation says that an addition (respectively, multiplication) gate $p = (p_1, p_2) \in \{0,1\}^{s_i + b}$ is connected to gates $\omega = (\omega_1, \omega_2) \in \{0,1\}^{s_{i+1} + b}$ and $\gamma = (\gamma_1, \gamma_2) \in \{0,1\}^{s_{i+1} + b}$ if and only if p, ω, and γ are all in the same copy of C, and p is connected to ω and γ within the copy.

Lemma 8.4.1 then implies that for any $z \in \mathbb{F}^{s_i^*}$,

$$\tilde{V}_i^*(z) = \sum_{(p_1,p_2,\omega_1,\gamma_1) \in \{0,1\}^{s_i} \times \{0,1\}^b \times \{0,1\}^{s_{i+1}} \times \{0,1\}^{s_{i+1}}} g_z^{(i)}(p_1, p_2, \omega_1, \gamma_1).$$

Thus, in iteration i of our protocol, we apply the sum-check protocol to the polynomial $g_z^{(i)}$. The communication costs of this protocol are immediate.

Costs for $\mathcal{V}$. In order to run her part of the sum-check protocol of iteration i, $\mathcal{V}$ only needs to perform the required checks on each of $\mathcal{P}$'s messages. $\mathcal{V}$'s check requires $O(1)$ time in each round of the sum-check protocol except the last. In the last round of the sum-check protocol, $\mathcal{V}$ must evaluate the polynomial $g_z^{(i)}$ at a single point. This requires evaluating $\beta_{s_i^*}$, $\tilde{\text{add}}_i$, $\tilde{\text{mult}}_i$, and $\tilde{V}_{i+1}^*$ at a constant number of points. The $\tilde{V}_{i+1}^*$ evaluations are provided by $\mathcal{P}$ in all iterations i of the protocol except the last, while the $\beta_{s_i^*}$ evaluation can be done in $O(\log(B \cdot S))$ time.

The $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ computations can be done in pre-processing in time $O(S_i)$ by enumerating the in-neighbors of each of the S_i gates at layer i. Adding up the pre-processing time across all iterations i of our protocol, $\mathcal{V}$'s pre-processing time is $O(\sum_i S_i) = O(S)$ as claimed.

In the final iteration of the protocol, $\mathcal{P}$ no longer provides the $\tilde{V}_{i+1}^*$ evaluation for $\mathcal{V}$; instead, $\mathcal{V}$ must evaluate the multilinear extension of the input at a point on her own. This can be done in a streaming manner using $O(\log n^* \log |\mathbb{F}|)$ bits of space in time $O(n^* \log n^*)$. The time cost for $\mathcal{V}$ in the online phase follows.

Costs for $\mathcal{P}$. It remains to show that $\mathcal{P}$ can perform the required computations in iteration i of the protocol in time $O((S_i + S_{i+1}) \cdot B \cdot \log(S))$. To this end, notice $g_z^{(i)}$ is a polynomial in $v := s_i + 2s_{i+1} + b$ variables. We order the sum in this sum-check protocol so that the $s_i + 2s_{i+1}$ variables in p_1, ω_1, and γ_1 are bound first in arbitrary order, followed by the variables of p_2. $\mathcal{P}$ can compute the prescribed messages in the first $s_i + 2s_{i+1} = O(\log S)$

rounds exactly as in the implementation of Chapter 7. They show that each gate at layers i and $i+1$ of C^* contributes to exactly one term in the sum defining $\mathcal{P}$'s message in any given round of the sum-check protocol, and moreover the contribution of a given gate can be determined in $O(1)$ time. Hence the total time devoted required by $\mathcal{P}$ to handle these rounds is $O(B \cdot (S_i + S_{i+1}) \cdot \log S)$. It remains to show how $\mathcal{P}$ can compute the prescribed messages in the final b rounds of the sum-check protocol while investing $O((S_i + S_{i+1}) \cdot B)$ across all rounds of the protocol.

Recall that in order to compute $\mathcal{P}$'s message in round j of the sum-check protocol applied to $g_z^{(i)}$, it suffices for $\mathcal{P}$ to evaluate $g_z^{(i)}$ at 2^{v-j} points of the form $(r_1, \ldots, r_{j-1}, t, b_{j+1}, \ldots, b_v)$, with $t \in \{0, \ldots, \deg_j(g_z^{(i)})\}$ and $(b_{j+1}, \ldots, b_v) \in \{0,1\}^{v-j}$. Each of these evaluations of $g_z^{(i)}$ can be computed in $O(1)$ time given the evaluations of $\beta_{s_i^*}$, $\tilde{\text{add}}_i$, $\tilde{\text{mult}}_i$, and $\tilde{V}_{i+1}^*$ at the relevant points.

Notice that once the variables in p_1, ω_1, and γ_1 are bound to specific values, say $r_1^{(p)}$, $r_1^{(\omega)}$, and $r_1^{(\gamma)}$, $\tilde{\text{add}}_i(p_1, \omega_1, \gamma_1)$ and $\tilde{\text{mult}}_i(p_1, \omega_1, \gamma_1)$ are themselves bound to specific values, namely $\tilde{\text{add}}_i(r_1^{(p)}, r_1^{(\omega)}, r_1^{(\gamma)})$ and $\tilde{\text{mult}}_i(r_1^{(p)}, r_1^{(\omega)}, r_1^{(\gamma)})$. So $\mathcal{P}$ only needs to evaluate these polynomials once, and both of these evaluations can be computed by $\mathcal{P}$ in $O(S_i)$ time. Thus, the $\tilde{\text{add}}_i$, $\tilde{\text{mult}}_i$ evaluations in the last b rounds require just $O(S_i)$ time in total.

$\mathcal{P}$ can evaluate the function $\beta_{s_i^*}$ at the relevant points exactly as in the proof of Theorem 8.4.10 using the $C^{(j)}$ arrays to ensure that this computation is done quickly. The array $C^{(0)}$ has size $2^{s_i^*} = O(S_i \cdot B)$, and $C^{(j-1)}$ gets updated to $C^{(j)}$ whenever a variable in p_1 or p_2 becomes bound. This ensures that across all rounds of the sum-check protocol, the $\beta_{s_i^*}$ evaluations require $O(S_i \cdot B)$ time in total.

Likewise, the $\tilde{V}_{i+1}^*$ evaluations can be handled exactly as in Theorem 8.4.10, using the the $V^{(j)}$ arrays to ensure that this computation is done quickly. The array $V^{(0)}$ has size $2^{s_{i+1}^*} = O(S_{i+1} \cdot B)$, and $V^{(j-1)}$ gets updated to $V^{(j)}$ whenever a variable in ω_1 becomes

bound (and similarly for the variables in γ_1). This ensures that across all rounds of the sum-check protocol, the $\tilde{V}^*_{i+1}$ evaluations take $O((S_i + S_{i+1}) \cdot B)$ in total.

Reducing to Verification of a Single Point. After executing the sum-check protocol at layer i as described above, $\mathcal{V}$ is left with a claim about $\tilde{V}_{i+1}(\omega_1, p_2)$ and $\tilde{V}_{i+1}(\gamma_1, p_2)$, for $\omega_1, \gamma_1 \in \mathbb{F}^{s_i}$, and $p_2 \in \mathbb{F}^b$. This requires $\mathcal{P}$ to send $\tilde{V}_{i+1}(\ell(t))$ for a canonical line $\ell(t)$ that passes through (ω_1, p_2) and (γ_1, p_2). It is easily seen that $\tilde{V}_{i+1}(\ell(t))$ is a univariate polynomial of degree at most s_i. Here, we are exploiting the fact that the final b coordinates of (ω_1, p_2) and (γ_1, p_2) are equal.

Hence $\mathcal{P}$ can specify $\tilde{V}_{i+1}(\ell(t))$ by sending $\tilde{V}_{i+1}(\ell(t_j))$ for $O(s_i)$ many points $t_j \in \mathbb{F}$. Using the method of Lemma 8.3.1, $\mathcal{P}$ can evaluate $\tilde{V}_{i+1}$ at each point $\ell(t_j)$ in $O(S_{i+1})$ time, and hence can perform all $\tilde{V}_{i+1}(\ell(t_j))$ evaluations in $O(S_{i+1} \cdot s_i) = O(S_{i+1} \cdot \log S)$ time in total. This ensures that across all iterations of our protocol, $\mathcal{P}$ devotes at most $O(S \cdot B \cdot \log S)$ time to the "reducing to verification of a single point" phase of the protocol. This completes the proof.

□

In practice we would expect the results of the data parallel phase of computation represented by the super-circuit C^* to be aggregated in some fashion. We assume this aggregation step is amenable to verification via other techniques. In the case of counting queries, the aggregation step simply sums the outputs of the data parallel step, which can be handled via Theorem 8.4.10, or slightly more efficiently via Proposition 8.7.1 described below in Section 8.7. More generally, if this aggregation step is computed by a circuit C' of size $O(S \cdot B \cdot \log S / \log B)$ such that $\mathcal{V}$ can efficiently evaluate the multilinear extension of the wiring predicate of C', then we can simply apply the basic GKR protocol to C' with asymptotic costs smaller than those of the protocol described in Theorem 8.6.1. This application of the GKR protocol to C' ends with a claim about the value of $\tilde{V}^*_1(z)$ for some $z \in \mathbb{F}^{s^*_1}$. The

verifier can then invoke the protocol of Theorem 8.6.1 to verify this claim.

We stress that the protocol of Theorem 8.6.1 can be applied if there are multiple data parallel stages interleaved with aggregation stages.

8.7 Extensions

In this section we describe two final optimizations that are much more specialized than Theorems 8.4.10 and 8.6.1, but have a significant effect in practice when they apply. In particular, Section 8.7.2 culminates in a protocol for matrix multiplication that is of interest in its own right. It is hundreds of times faster than the protocol implied by Theorem 8.4.10 and studied experimentally in Section 8.5.

8.7.1 Binary Tree of Addition Gates

Cormode et al. [57] describe an optimization that applies to any circuit C with a single output that culminates in a binary tree of addition gates; at a high level, they directly apply a single sum-check protocol to the entire binary tree, thereby treating the entire tree as a single addition gate with very large fan-in. In contrast, the optimization described here applies to circuits with multiple outputs and allows the binary tree of addition gates to occur anywhere in the circuit, not just at the layers immediately preceding the output.

At first blush, our optimization might seem quite specialized since it only applies to circuits with a specific wiring pattern. However, this is one of the most commonly occurring wiring patterns, as evidenced by its appearance within the circuits computing MatMult, Distinct, Pattern Matching, and counting queries. Notice that our optimization also applies to verifying multiple independent instances of any problem with a single output whose circuit ends with a binary tree of sum-gates, such as verifying the number of distinct items in

multiple distinct data streams, or posing multiple separate counting queries to a database. This is because, similar to Theorem 8.6.1, one can lay the circuits for each of the individual problem instances side-by-side and treat the result as a single "super-circuit" culminating in a binary tree of addition gates with multiple outputs.

The starting point for our optimization is the observation that in order to verify that $\mathcal{P}$ has correctly evaluated a circuit with many output gates, $\mathcal{P}$ may simply send $\mathcal{V}$ the (claimed) values of all output gates, thereby specifying a function $V_1' : \{0,1\}^{s_1} \to \mathbb{F}$ claimed to equal V_1. $\mathcal{V}$ can pick a random point $z \in \mathbb{F}^{s_1}$ and evaluate $\tilde{V}_1'(z)$ on her own in $O(S_1)$ time. The Schwartz-Zippel Lemma (Lemma 2.2.1) implies that it is safe for $\mathcal{V}$ to believe that V_1 is as claimed as long as $\tilde{V}_1(z) = \tilde{V}_1'(z)$. Our protocol as described in Section 8.4 would then proceed in iterations, with one iteration per layer of the circuit and one application of the sum-check protocol per iteration. This would ultimately reduce $\mathcal{P}$'s claim about the value of $\tilde{V}_1(z)$ to a claim about $\tilde{V}_d(z')$ for some $z' \in \mathbb{F}^{s_d}$, where d is the input layer of the circuit.

Instead, our final refinement uses a single sum-check protocol to directly reduce $\mathcal{P}$'s claim about $\tilde{V}_1(z)$ to a claim about $\tilde{V}_d(z')$ for some random points $z' \in \mathbb{F}^{s_d}$.

Proposition 8.7.1. *Let C be a depth-d circuit consisting of a binary tree of addition gates, 2^k inputs, and 2^{k-d} outputs. For any points $z \in \mathbb{F}^{k-d}$, $\tilde{V}_1(z) = \sum_{p \in \{0,1\}^k} g_z(p)$, where*

$$g_z(p) = \tilde{V}_d(z, p_{k-d+1}, \ldots, p_k).$$

Proof. At layer i of C, the gate with label $p \in \{0,1\}^{s_i}$ is the sum of the gates with labels $(p,0)$ and $(p,1)$ at layer $i+1$. It is then straightforward to observe that the for any $p \in \{0,1\}^{k-d}$, the pth output gate has value

$$V_1(p_1, \ldots, p_{k-d}) = \sum_{(p_{k-d+1}, \ldots, p_d) \in \{0,1\}^d} \tilde{V}_d(p_1, \ldots, p_{k-d}, p_{k-d+1}, \ldots, p_k). \tag{8.4}$$

Notice that the right hand side of Equation (8.4) is a multilinear polynomial in the variables $(p_1, \ldots, p_{k-d})$ that agrees with $V_1(p_1, \ldots, p_{k-d})$ at all Boolean inputs. Hence, the right hand

Table 8.4: Experimental results for $n \times n$ MatMult, with and without the refinement of Section 8.7.1.

Implementation	Problem Size	$\mathcal{P}$ Time	$\mathcal{V}$ Time	Rounds	Total Communication	Circuit Eval Time
Thm. 8.4.10	256 x 256	4.37 s	0.02 s	190	4.4 KBs	0.73 s
Prop. 8.7.1	256 x 256	2.52 s	0.02 s	35	0.76 KBs	0.73 s
Thm. 8.4.10	512 x 512	37.85 s	0.10 s	236	5.48 KBs	6.07 s
Prop. 8.7.1	512 x 512	22.98 s	0.10 s	39	0.86 KBs	6.07 s

As in Table 8.1, the Total Communication column does not count the n^2 field elements required to specify the answer.

side is the (unique) multilinear extension $\tilde{V}_1$ of the function $V_1 : \{0,1\}^{k-d} \to \{0,1\}$. The theorem follows. □

In applying the sum-check protocol to the polynomial g_z in Proposition 8.7.1, it is straightforward to use the methods of Section 8.4.4 to implement the honest prover in time $O(2^k)$. We omit the details for brevity.

Experimental Results. Let C be the circuit for naive matrix multiplication described in Section 8.4.5. To demonstrate the efficiency gains implied by Proposition 8.7.1, we modified our MatMult implementation of Section 8.5.2 to use the protocol of Proposition 8.7.1 to verify the sub-circuit of C consisting of a binary tree of addition gates. The results are shown in Table 8.4. Our optimizations in this section shave $\mathcal{P}$'s runtime by a factor of 1.5x-2x, the total number of rounds by a factor of more than 5, and the total communication (not counting the cost of specifying the output of the circuit) by a factor of more than 5.

8.7.2 Optimal Space and Time Costs for MatMult

We describe a final optimization here on top of Proposition 8.7.1. While this optimization is specific to the MatMult problem, its effects are substantial and the underlying observation may be more broadly applicable.

Suppose we are given an unverifiable algorithm for $n \times n$ matrix multiplication that requires time $T(n)$ and space $s(n)$. Our refinements reduce the prover's runtime from $O(n^3)$ in the case of Sections 8.4 and 8.7.1 to $T(n) + O(n^2)$, and lowers $\mathcal{P}$'s space requirement to $s(n) + o(n^2)$. That is, in the protocol the prover sends the correct output and performs just $O(n^2)$ more work to provide a guarantee of correctness on top. It is irrelevant what algorithm the prover uses to arrive at the correct output – in particular, algorithms much more sophisticated than naive matrix multiplication are permitted. This runtime and space usage for $\mathcal{P}$ are optimal even up to the leading constant assuming matrix multiplication cannot be computed in $O(n^2)$ time.

The final protocol is extremely natural, as it consists of a single invocation of the sum-check protocol. We believe this protocol is of interest in its own right. The proof and technical details are in Section 8.7.2.

Theorem 8.7.2. *There is a valid interactive proof protocol for $n \times n$ matrix multiplication over the field $\mathbb{F}_q$ with the following costs. The communication cost is $n^2 + O(\log n)$ field elements. The runtime of the prover is $T(n) + O(n^2)$ and the space usage is $s(n) + o(n^2)$, where $T(n)$ and $s(n)$ are the time and space requirements of any (unverifiable) algorithm for $n \times n$ matrix multiplication. The verifier can make a single streaming pass over the input as well as over the claimed output in time $O(n^2 \log n)$, storing $O(\log n)$ field elements.*

Using the observation of Vu et al. described in Lemma 8.3.1, the runtime of the verifier can be brought down to $O(n^2)$ at the cost of increasing $\mathcal{V}$'s space usage to $O(n^2)$. Furthermore, by Remark 4, the runtime of the verifier can be brought down to $O(n^2)$ while maintaining the streaming property if the input matrices are presented in row-major order.

The prover's runtime in Theorem 8.7.2 is within an additive low-order term of any unverifiable algorithm for matrix multiplication; this is essential in many practical scenarios where even a 2x slowdown is too steep a price to pay for verifiability. Notice also that the space

usage bounds in Theorem 8.7.2 are in stark contrast to protocols based on circuit-checking: the prover in a general circuit-checking protocol may have to store the entire circuit, and this can result in space requirements that are much larger than those of an unverifiable algorithm for the problem. For example, naive matrix multiplication requires time $O(n^3)$, but only $O(n^2)$ space, while the provers in our MatMult protocols of Sections 8.4 and 8.7.1 require both space and time $O(n^3)$. As implementations of interactive proofs become faster, the prover is likely to run out of space long before she runs out of time.

Comparison to Prior Work

It is worth comparing Theorem 8.7.2 to a well-known protocol due to Freivalds [53]. Let D^* denote the claimed output matrix. In Freivalds' algorithm, the verifier stores a random vector $x \in \mathbb{F}^n$, and computes D^*x and ABx, accepting if and only if $ABx = D^*x$. Freivalds showed that this is a valid protocol. In both Freivalds' protocol and that of Theorem 8.7.2, the prover runs in time $T(n) + O(n^2)$ (in the case of Freivalds' algorithm, the $O(n^2)$ term is 0), and the verifier runs in linear or quasilinear time.

We now highlight several properties of our protocol that are not achieved by prior work.

Utility as a Primitive. A major advantage of Theorem 8.7.2 relative to prior work is its utility as a primitive that can be used to verify more complicated computations. This is important as many algorithms repeatedly invoke matrix multiplication as a subroutine. For concreteness, consider the problem of computing A^{2^k} via repeated squaring. By iterating the protocol of Theorem 8.7.2 k times, we obtain a valid interactive proof protocol for computing A^{2^k} with communication cost $n^2 + O(k \log(n))$. The n^2 term is due simply to specifying the output A^{2^k}, and can often be avoided in applications – see for example the diameter protocol described two paragraphs hence. The ith iteration of the protocol for computing A^{2^k} reduces a claim about an evaluation of the multilinear extension of $A^{2^{k-i+1}}$ to an analogous claim about

$A^{2^{k-i}}$. Crucially, the prover in this protocol never needs to send the verifier the intermediate matrices $A^{2^{k'}}$ for $k' < k$. In contrast, applying Freivalds' algorithm to this problem would require $O(kn^2)$ communication, as $\mathcal{P}$ must specify each of the intermediate matrices A^{2^i}.

The ability to avoid having $\mathcal{P}$ explicitly send intermediate matrices is especially important in settings where an algorithm repeatedly invokes matrix multiplication, but the desired output of the algorithm is smaller than the size of the matrix. In these cases, it is not necessary for $\mathcal{P}$ to send *any* matrices; $\mathcal{P}$ can instead send just the desired output, and V can use Theorem 8.7.2 to check the validity of the output with only a polylogarithmic amount of additional communication. This is analogous to how the verifier in the GKR protocol can check the values of the output gates of a circuit without ever seeing the values of the "interior" gates of the circuit.

As a concrete example illustrating the power of our matrix multiplication protocol, consider the fundamental problem of computing the diameter of an unweighted (possibly directed) graph G on n vertices. Let A denote the adjacency matrix of G, and let I denote the $n \times n$ identity matrix. Then it is easily verified that the diameter of G is the least positive number d such that $(A + I)^d_{ij} \neq 0$ for all (i, j). We therefore obtain the following natural protocol for diameter. $\mathcal{P}$ sends the claimed output d to V, as well as an (i, j) such that $(A + I)^{d-1}_{ij} = 0$. To confirm that d is the diameter of G, it suffices for $\mathcal{V}$ to check two things: first, that all entries of $(A + I)^d$ are non-zero, and second that $(A + I)^{d-1}_{ij}$ is indeed non-zero.

The first task is accomplished by combining our matrix multiplication protocol of Theorem 8.7.2 with our DISTINCT protocol from Theorem 8.4.10. Indeed, let d_j denote the jth bit in the binary representation of d. Then $(A + I)^d = \prod_j^{\lceil \log d \rceil} (A + I)^{d_j 2^j}$, so computing the number of non-zero entries of $(A + I)^d$ can be computed via a sequence of $O(\log d)$ matrix multiplications, followed by a DISTINCT computation. The second task, of verifying that $(A + I)^{d-1}_{ij} = 0$, is similarly accomplished using $O(\log d)$ invocations of the matrix multi-

plication protocol of Theorem 8.7.2 – since $\mathcal{V}$ is only interested in one entry of $(A+I)^{d-1}$, $\mathcal{P}$ need not send the matrix $(A+I)^{d-1}$ in full, and the total communication here is just polylog (n).

$\mathcal{V}$'s runtime in this diameter protocol is $O(m \log n)$, where m is the number of edges in G. $\mathcal{P}$'s runtime in the above diameter protocol matches the best known unverifiable diameter algorithm up to a low-order additive term [89,107], and the communication is just polylog (n). We know of no other protocol achieving this.

As discussed above, the fact that $\mathcal{P}$'s slowdown is a low-order additive term is critical in the many settings in which even a 2x slowdown to achieve verifiability is unacceptable. Moreover, for a graph with $n = 1$ million nodes, the total communication cost of the above protocol is on the order of KBs – in contrast, if $\mathcal{P}$ had to send the matrices $(I+A)^d$ or $(I+A)^{d-1}$ explicitly (as required in prior work e.g. Cormode et al. [40]), the communication cost would be at least $n^2 = 10^{12}$ words, which translates to terabytes of data.

Small-Space Streaming Verifiers. In Freivalds' algorithm, $\mathcal{V}$ has the store the random vector x, which requires $\Omega(n)$ space. There are methods to reduce $\mathcal{V}$'s space usage by generating x with limited randomness: Kimbrel and Sinha [68] show how to reduce $\mathcal{V}$'s space to $O(\log n)$, but their solution does not work if $\mathcal{V}$ must make a streaming pass over arbitrarily ordered input. In Theorem 3.10.2, we extended the method of Kimbrel and Sinha to work with a streaming verifier, but this requires $\mathcal{P}$ to play back the input matrices A, B in a special order, increasing proof length to $3n^2$. Our protocol works with a streaming verifier using $O(\log n \log |\mathbb{F}|)$ bits of space, and our proof length is $n^2 + O(\log n)$, where the n^2 term is due to specifying AB and can be avoided in applications such as the diameter example considered above.

Protocol Details

The idea behind the optimization is as follows. All of our earlier circuit-checking protocols only make use of the multilinear extension $\tilde{V}_i$ of the function V_i mapping gate labels at layer i of the circuit to their values. In some cases, there is something to be gained by using a higher-degree extension of V_i, and this is precisely what we exploit here. By using a higher-degree extension of the gate values in the circuit, we are able to apply the sum-check protocol to a polynomial that differs from the one used in Section 8.4. In particular, the polynomial we use here avoids referencing the β_{s_i} polynomial used in Section 8.4. Details follow.

When multiplying matrices A and B such that $AB = D$, let $A(i,j)$, $B(i,j)$ and $D(i,j)$ denote functions from $\{0,1\}^{\log n} \times \{0,1\}^{\log n} \to \mathbb{F}_q$ that map input (i,j) to A_{ij}, B_{ij}, and D_{ij} respectively. Let $\tilde{A}$, $\tilde{B}$, and $\tilde{D}$ denote their multilinear extensions.

Lemma 8.7.3. *For all* $(p_1, p_2) \in \mathbb{F}^{\log n} \times \mathbb{F}^{\log n}$,

$$\tilde{D}(p_1, p_2) = \sum_{p_3 \in \{0,1\}^{\log n}} \tilde{A}(p_1, p_3) \cdot \tilde{B}(p_3, p_2)$$

Proof. For all $(p_1, p_2) \in \{0,1\}^{\log n} \times \{0,1\}^{\log n}$, the right hand side is easily seen to equal $D(p_1, p_2)$, using the fact that $D_{ij} = \sum_k A_{ik} B_{kj}$ and the fact that $\tilde{A}$ and $\tilde{B}$ agree with the functions $A(i,j)$ and $B(i,j)$ at all Boolean inputs. Moreover, the right hand side is a multilinear polynomial in the variables of (p_1, p_2). Putting these facts together implies that the right hand side is the unique multilinear extension of the function $D(i,j)$. □

Lemma 8.7.3 implies the following valid interactive proof protocol for matrix multiplication: $\mathcal{P}$ sends a matrix D^* claimed to equal the product $D = AB$. $\mathcal{V}$ evaluates $\tilde{D}^*(r_1, r_2)$ at a random point $(r_1, r_2) \in \mathbb{F}^{\log n} \times \mathbb{F}^{\log n}$. By the Schwartz-Zippel lemma, it is safe for $\mathcal{V}$ to believe D^* is as claimed, as long as $\tilde{D}^*(r_1, r_2) = \tilde{D}(r_1, r_2)$ (formally, if $D^* \neq D$, then $\tilde{D}^*(r_1, r_2) \neq \tilde{D}(r_1, r_2)$ with probability $1 - 2\log n/q$). In order to check that $\tilde{D}^*(r_1, r_2) = \tilde{D}(r_1, r_2)$, we invoke a sum-check protocol on the polynomial $g_{r_1,r_2}(p_3) = \tilde{A}(r_1, p_3) \cdot \tilde{B}(p_3, r_2)$.

$\mathcal{V}$'s final check in this protocol requires her to compute $g_{r_1,r_2}(r_3)$ for a random point $r_3 \in \mathbb{F}^{\log n}$. $\mathcal{V}$ can do this by evaluating both of $\tilde{A}(r_1, r_3)$ and $\tilde{B}(r_3, r_2)$ with a single streaming pass over the input, and then multiplying the results.

The prover can be made to run in time $T(n) + O(n^2)$ across all rounds of the sum-check protocol using the $V^{(j)}$ arrays described in Section 8.4 to quickly evaluate $\tilde{A}$ and $\tilde{B}$ at all of the necessary points. The $V^{(j)}$ arrays are initialized in round 0 to equal the input matrices themselves, and there is no need for $\mathcal{P}$ to maintain an "uncorrupted" copy of the original input (though in practice this may be desirable). Thus, the $V^{(j)}$ arrays can be computed using the storage $\mathcal{P}$ initially devoted to the inputs, and $\mathcal{P}$ needs to store just $O(1)$ additional field elements over the course of the protocol ($\mathcal{P}$ does not even need to store the messages sent by $\mathcal{V}$, as $\mathcal{P}$ need not refer to the jth message once the array $V^{(j)}$ is computed). The claimed $s(n) + o(n^2)$ space usage bound for $\mathcal{P}$ follows.

Remark 11. Let C be the circuit for naive matrix multiplication described in Section 8.4. Notice that the $3 \log n$-variate polynomial $h(p_1, p_2, p_3) = \tilde{A}(p_1, p_3) \cdot \tilde{B}(p_3, p_2)$ extends the function V_i mapping gate labels at layer $i = \log n$ of C to their values. However, h is not the multilinear extension of $\tilde{V}_i$, as h has degree two in the variables of p_3.

Informally, Theorem 8.7.2 cannot be said to perform "circuit checking" on C, since it is not necessary for $\mathcal{P}$ to evaluate all of the gates in C; indeed, the prover in Theorem 8.7.2 can run in sub-cubic time using fast matrix multiplication algorithms. However, the use of a low-degree extension of the gate values at layer $\log n$ of C allows one to view the protocol of Theorem 8.7.2 as a direct extension of the circuit-checking methodology.

Remark 12. Consider the problem of computing a matrix power M^{2^k} via repeated squaring. We may apply the protocol of Theorem 8.7.2 in k iterations, with the ith iteration applied to inputs $A = B = M^{2^{k-i}}$. The ith iteration of this protocol reduces a claim about an evaluation of the multilinear extension of $M^{2^{k-i+1}}$ to an analogous claim about the multilinear extension

of $M^{2^{k-i}}$ at two points of the form (r_1, r_3), $(r_3, r_2) \in \mathbb{F}^{\log n \times \log n}$. We can further reduce the claims about (r_1, r_3), (r_3, r_2) to a claim about a single point exactly as in the "Reducing to Verification of a Single Point" step of the GKR protocol. We then move onto iteration $i+1$. Notice in particular that the verifier only needs to observe the output matrix M^{2^k} and the input matrix M to run this protocol; in particular, $\mathcal{P}$ does not need to explicitly send the intermediate matrices $M^{2^{k-i}}$ to $\mathcal{V}$.

We implemented the protocol just described (our implementation is sequential). The results are shown in Table 8.5, where the column labelled "Additional Time for $\mathcal{P}$" denotes the time required to compute $\mathcal{P}$'s prescribed messages after $\mathcal{P}$ has already computed the correct answer. We report the naive matrix multiplication time both when the computation is done using standard multiplication of 64-bit integers, as well as when the computation is done using finite field arithmetic over the field with $q = 2^{61} - 1$ elements. The reported verifier runtime is for the $O(n^2 \log n)$ time reported in Theorem 8.7.2. The verifier's runtime could be improved using Lemma 8.3.1 at the cost of increasing $\mathcal{V}$'s space usage to $O(n)$, but we did not implement this optimization. Moreover, if the input matrices are presented in row-major order, then the observation of Vu et al. described in Remark 4 improves $\mathcal{V}$'s runtime with no increase in space usage.

The main takeaways from Table 8.5 are that the verifier does indeed save substantial time relative to performing matrix multiplication locally, and that the runtime of the prover is hugely dominated by the time required simply to compute the answer.

8.8 Discussion

We believe our results substantially advance the goal of achieving a truly practical general purpose implementation of interactive proofs. The $O(\log S(n))$ factor overhead in the runtime

Table 8.5: Experimental results for the $n \times n$ MatMult protocol of Theorem 8.7.2.

Implementation	Problem Size	Naive Matrix Multiplication Time	Additional Time for $\mathcal{P}$	$\mathcal{V}$ Time	Rounds
Thm. 8.7.2	$2^{10} \times 2^{10}$	2.17 s over $\mathbb{Z}$ 9.11 s over $\mathbb{F}_q$	0.03 s	0.67 s	11
Thm. 8.7.2	$2^{11} \times 2^{11}$	18.23 s over $\mathbb{Z}$ 73.65 s over $\mathbb{F}_q$	0.13 s	2.89 s	12

of the prover within prior implementations of the GKR protocol is too steep a price to pay in practice, and our refinements (formalized in Theorem 8.4.10) remove this logarithmic factor overhead for circuits with regular wiring patterns. Our experiments demonstrate that this protocols yields a prover that is less than 10x slower than a C++ program that simply evaluates the circuit, and that our protocols are highly amenable to parallelization. Exploiting similar ideas, we have also extended the reach of prior interactive proof protocols by describing an efficient protocol (formalized in Theorem 8.6.1) for general data parallel computation, and given a protocol for matrix multiplication in which the prover's overhead (relative to *any* unverifiable algorithm) is just a low-order additive term. The latter is a powerful primitive for verifying the many algorithms that repeatedly invoke matrix multiplication. A major message of our results is that the more structure that exists in a computation, the more efficiently it can be verified, and that this structure exists in many real-world computations.

We believe two directions in particular are worthy of future work. The first direction is to build a full-fledged system implementing our protocol for data parallel computation. Our vision is to combine our protocol with a high-level programming language allowing the programmer to easily specify data parallel computations, analogous to frameworks such as MapReduce. Any such program could be automatically compiled in the manner of Vu et al. [104] into a circuit, and our protocol could be run automatically on that circuit. The

second direction is to further enable such a compiler to automatically take advantage of our other refinements, which are targeted at computations that are not necessarily data parallel. These refinements apply to a circuit on a layer-by-layer basis, so they may yield substantial speedups in practice even if they apply only to a subset of the layers of a circuit.

Chapter 9

Conclusion

In this thesis, we introduced two new models, dubbed annotated data streams and streaming interactive proofs, for verifiably outsourcing computations on data streams. Both models require the verifier to process the input within the confines of the data streaming paradigm, a restriction that fits the cloud computing setting well, as the verifier's streaming pass over the input can occur while uploading data to the cloud. By utilizing and extending algebraic techniques developed in the interactive proofs literature, we developed highly efficient protocols in both models and demonstrated their practicality in many cases via implementations. We believe the general-purpose protocols developed in Chapters 7 and 8 have the largest potential for practical impact.

9.1 Related Work

In contrast to the work on interactive proofs described in this thesis, all parallel lines of work have focused on the development of argument systems, which are interactive proofs that are only secure against polynomial time adversaries. That is, in an argument system the soundness property is only required to hold against polynomial time provers; it may

be possible for inefficient provers to convince the verifier to accept an incorrect answer. Typically, argument systems utilize cryptographic machinery, and an inefficient prover may be able to "break" the machinery and thereby trick the verifier.

Theoretical Work. A substantial body of theoretical work in this area has focused on the development of protocols targeted at specific problems (e.g. [15, 19, 51]). Other theoretical works have focused on the development of general-purpose argument systems. Several papers in this direction (e.g. [16, 33, 34, 55]) have used fully homomorphic encryption, which unfortunately remains impractical despite substantial recent progress. Work in this category by Chung et al. [33] focuses on streaming settings, and is therefore particularly relevant to our work in this thesis.

Works Refining Theory Toward Implementation. In parallel with the work reported on in this thesis, several other research groups from multiple communities have also been working toward the development of practical general-purpose protocols for verifying outsourced computations. Interestingly, each of these approaches achieves a different tradeoff between efficiency and applicability/functionality. The approaches are therefore likely to prove highly complementary: users of protocols for verifiable computation will choose the most efficient approach that has the "features" they require, and that applies to the class of computations they are outsourcing. We give a brief summary and comparison of the existing approaches, before briefly describing where we think this area of research should head next.

All existing work in this category is based on probabilistically-checkable proofs (PCPs). In a PCP, the prover generates a static proof (which may be long), but the verifier only needs to inspect a small number of randomly chosen bits of the proof in order to determine with high probability whether or not the proof is valid. PCPs can be combined with cryptographic machinery to yield efficient argument systems.

The first line of work that sought to develop a practical implementations of general-

purpose argument systems is due to Setty et al. [90–92]. They begin with a base argument system due to Ishai et al. [66] and substantially refine the theory to achieve an implementation that approaches practicality. The most recent system developed in this line of work is called Zaatar [92, 104]. A related line of work is due to Parno et al. [82], who describe a general-purpose implementation, called Pinocchio, of an argument system proposed by Genarro et al. [56]. The works of Setty et al. and Genarro et al. avoid the use of short PCPs, as these are often a bottleneck in the development of efficient argument systems. In contrast, a line of work by Ben-Sasson et al. [13, 14] focuses on the development of short PCPs that might be suitable for use in practice – such PCPs can be combined with standard cryptographic tools to yield interactive arguments.

9.2 Comparison With Other Approaches

We now provide a detailed comparison between the approach taken in this thesis and the parallel projects on argument systems that have attempted to refine theory toward implementation. The interactive proof protocols developed in this thesis are the most efficient when they are applicable: in [104], Vu et al. empirically compare the protocol developed in Chapter 7 (with some additional refinements) to Zaatar and find our approach to be significantly more efficient for quasi-straight-line computations (e.g., programs with relatively simple control flow), while Zaatar is appropriate for programs with more complicated control flow. The main reasons for the superior efficiency of our approach in those cases is that it avoids the use of cryptography (which can be expensive), and it avoids pre-processing for regular circuits. Furthermore, for certain extremely important computations such as matrix multiplication, our approach is unique in its ability to avoid representing the computation as a circuit, thereby reducing the overhead for the prover to additive low-order terms. The main

disadvantages of our approach are that it applies only to small-depth computations, since the cost to the verifier grows linearly with circuit depth, and that we cannot support "non-deterministic reductions", which can be useful for turning high-level programs into small circuits [13, 91]. Essentially, support for non-deterministic reductions means that, unlike interactive proofs, the argument systems developed by Parno et al., Setty et al., and Ben-Sasson et al. can be run on non-deterministic circuits – i.e., circuits that "take advice" – without the prover having to explicitly send the advice inputs to the verifier. Very recent implementation work by Braun et al. also exploits the ability of these argument systems to support circuits that take advice [20].

The approach taken in Zaatar [92, 104] is the second most efficient, followed closely by Pinocchio [82] (the underlying machinery of these two systems is closely related). Vu et al. [104] estimate that Pinocchio is more expensive than Zaatar by small constant factors in terms of both the prover's runtime and the verifier's per-instance costs, but Pinocchio achieves several features not shared by Zaatar or the general-purpose protocols in this thesis, including non-interactivity, public verifiability, unbounded query reuse, and support for zero-knowledge. While more general than interactive proofs in terms of their support for non-deterministic reductions and applicability to deep circuits, these systems inherently require an expensive pre-processing phase for the verifier.

In contrast, the approach of Ben-Sasson et al. [13, 14] never requires expensive pre-processing for the verifier, at least asymptotically. However, the overhead for the prover may be substantially higher than with other approaches – at the time of writing, definite conclusions cannot be drawn, as experimental results from this approach are not yet available.

We conclude that each of the approaches to verifiable computation being pursued thus far achieves a different tradeoff between efficiency, expressiveness, and support for features such

as public verifiability and zero knowledge properties. This diversity can only be a good thing, as users will be able to choose the approach that best suits their needs. Moreover, different approaches have come out of different communities (including the systems, security, and theory communities) – there has been substantial interplay of ideas across multiple groups, and it is essential that this continue in the future if the area is to continue to progress.

9.3 Future Directions

A major message of this thesis has been that the more structure that exists in a computation, the more efficiently it can be verified, and that this structure exists in real-world computations. Fully general-purpose protocol implementations for verifiable computation already exist and will continue to grow more efficient, but we strongly believe that such structure *must* be leveraged to achieve fully practical protocols. For this reason, further exploration of the kind of structure that can be exploited for efficiency gains in protocols for verifiable computation is an important direction for future research.

Along these lines, we view the matrix multiplication protocol of Theorem 8.7.2 as particularly encouraging, as for many problems, the main bottleneck of the protocols developed in this thesis is in representing the outsourced computation as an arithmetic circuit (i.e., already the overhead due to evaluating the circuit verifiably is much smaller than the overhead in generating the circuit in the first place). The matrix multiplication protocol of of Theorem 8.7.2 is the most striking example we have of a protocol for an extremely important algorithmic primitive that completely avoids representing the computation as a circuit, and consequently reduces the overhead for the prover to low-order additive terms. It would be very interesting to develop a larger set of primitives that can be verified with overhead that are just additive low-order terms – this would allow any computation that can be expressed

as a combination of these primitives to be verified with similarly tiny overhead.

The ultimate goal should be to develop protocols and build systems that can verifiably execute general computations, but that automatically leverage structure within computations for efficiency gains. The end result may be a programming framework analogous to MapReduce: a restricted framework that still allows for the expression of a powerful class of computations, and automatically "extracts" the structure necessary to verify the computation efficiently. Determining the right balance between the level of generality to support and the amount of structure to force upon computations for efficiency gains is perhaps the most critical long-term direction for the area.

Bibliography

[1] List of open problems in sublinear algorithms: Problem 47. `http://sublinear.info/47`.

[2] Scott Aaronson. QMA/qpoly $\subseteq$ PSPACE/poly: De-merlinizing quantum protocols. In *IEEE Conference on Computational Complexity*, pages 261–273. IEEE Computer Society, 2006.

[3] Scott Aaronson and Avi Wigderson. Algebrization: A new barrier in complexity theory. *ACM Trans. Comput. Theory*, 1(1):2:1–2:54, February 2009.

[4] Farid Ablayev. Lower bounds for one-way probabilistic communication complexity and their application to space complexity. *Theoretical Computer Science*, 175(2):139–159, 1996.

[5] Dimitris Achlioptas. Database-friendly random projections. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '01, pages 274–281, New York, NY, USA, 2001. ACM.

[6] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[7] Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Streaming algorithms via precision sampling. In Rafail Ostrovsky, editor, *FOCS*, pages 363–372. IEEE, 2011.

[8] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[9] László Babai. Trading group theory for randomness. In Robert Sedgewick, editor, *STOC*, pages 421–429. ACM, 1985.

[10] Laszlo Babai, Peter Frankl, and Janos Simon. Complexity classes in communication complexity theory. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, pages 337–347, Washington, DC, USA, 1986. IEEE Computer Society.

[11] Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *SODA*, pages 623–632, 2002.

[12] Paul Beame and Trinh Huynh. The value of multiple read/write streams for approximating frequency moments. *ACM Trans. Comput. Theory*, 3(2):6:1–6:22, January 2012.

[13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from rams to delegatable succinct constraint satisfaction problems: extended abstract. In Robert D. Kleinberg, editor, *ITCS*, pages 401–414. ACM, 2013.

[14] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. On the concrete efficiency of probabilistically-checkable proofs. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *STOC*, pages 585–594. ACM, 2013.

[15] Siavosh Benabbas, Rosario Gennaro, and Yevgeniy Vahlis. Verifiable delegation of computation over large datasets. In *Proceedings of the 31st Annual Conference on Advances in Cryptology*, CRYPTO'11, pages 111–131, Berlin, Heidelberg, 2011. Springer-Verlag.

[16] Nir Bitansky and Alessandro Chiesa. Succinct arguments from multi-prover interactive proofs and their efficiency benefits. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 255–272. Springer, 2012.

[17] Eric Blais, Joshua Brody, and Kevin Matulef. Property testing lower bounds via communication complexity. *Computational Complexity*, 21(2):311–358, 2012.

[18] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. *Algorithmica*, pages 90–99, 1995.

[19] Dan Boneh and David Mandell Freeman. Homomorphic signatures for polynomial functions. In Kenneth G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 149–168. Springer, 2011.

[20] Benjamin Braun, Ariel J. Feldman, Srinath Setty Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *24th ACM Symposium on Operating Systems Principles*, November 2013, Preprint: Cryptology ePrint 2013/356, June 2013.

[21] Vladimir Braverman and Rafail Ostrovsky. Recursive sketching for frequency moments. *CoRR*, abs/1011.2571, 2010.

[22] Vladimir Braverman and Rafail Ostrovsky. Zero-one frequency laws. In *STOC '10: Proceedings of the 42nd ACM Symposium on Theory of Computing*, pages 281–290, New York, NY, USA, 2010. ACM.

[23] Joshua Brody, Amit Chakrabarti, and Ranganath Kondapally. Certifying equality with limited interaction. *Electronic Colloquium on Computational Complexity (ECCC)*, 19:153, 2012.

[24] Harry Buhrman, David García-Soriano, Arie Matsliah, and Ronald de Wolf. The non-adaptive query complexity of testing k-parities. *CoRR*, abs/1209.3849, 2012.

[25] Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting triangles in data streams. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, pages 253–262, New York, NY, USA, 2006. ACM.

[26] C. Sidney Burrus and Peter W. Eschenbacher. An in-place, in-order prime factor fft algorithm. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 29:806–817, 1981.

[27] Ran Canetti, Ben Riva, and Guy N. Rothblum. Practical delegation of computation using multiple servers. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 445–454. ACM, 2011.

[28] Ran Canetti, Ben Riva, and Guy N. Rothblum. Refereed delegation of computation. *Inf. Comput.*, 226:16–36, 2013.

[29] Amit Chakrabarti, Graham Cormode, and Andrew McGregor. Robust lower bounds for communication and stream computation. In *STOC*, pages 641–650, 2008.

[30] Amit Chakrabarti, Graham Cormode, Andrew Mcgregor, and Justin Thaler. Annotations in data streams. In *Submitted, 2012. A preliminary version of this paper appeared at ICALP*, pages 222–234, 2009.

[31] Amit Chakrabarti, Subhash Khot, and Xiaodong Sun. Near-optimal lower bounds on the multi-party communication complexity of set disjointness. In *IEEE Conference on Computational Complexity*, pages 107–117, 2003.

[32] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312(1):3–15, 2004.

[33] Kai-Min Chung, Yael Tauman Kalai, Feng-Hao Liu, and Ran Raz. Memory delegation. In Phillip Rogaway, editor, *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 151–168. Springer, 2011.

[34] Kai-Min Chung, Yael Tauman Kalai, and Salil P. Vadhan. Improved delegation of computation using fully homomorphic encryption. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 483–501. Springer, 2010.

[35] Anne Condon. The complexity of space bounded interactive proof systems. *Complexity Theory: Current Research*, pages 47–190, 1993.

[36] James Cooley and John Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, April 1965.

[37] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Finding hierarchical heavy hitters in data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 464–475. VLDB Endowment, 2003.

[38] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Source code. `http://people.seas.harvard.edu/~jthaler/code.htm`, 2011.

[39] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In Shafi Goldwasser, editor, *ITCS*, pages 90–112. ACM, 2012.

[40] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Streaming graph computations with a helpful advisor. *Algorithmica*, 65(2):409–442, 2013.

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

[42] Graham Cormode, Justin Thaler, and Ke Yi. Verifying computations with streaming interactive proofs. *PVLDB*, 5(1):25–36, 2011.

[43] Atish Das Sarma, Richard J. Lipton, and Danupon Nanongkai. Best-order streaming model. In *Theory and Applications of Models of Computation*, pages 178–191, 2009.

[44] Anirban Dasgupta, Ravi Kumar, and D. Sivakumar. Sparse and lopsided set disjointness via information theory. In Anupam Gupta, Klaus Jansen, José D. P. Rolim, and Rocco A. Servedio, editors, *APPROX-RANDOM*, volume 7408 of *Lecture Notes in Computer Science*, pages 517–528. Springer, 2012.

[45] M. Datar and S. Muthukrishnan. Estimating rarity and similarity over data stream windows. In *ESA*, volume 2461 of *Lecture Notes in Computer Science*, pages 323–334, 2002.

[46] Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. Trading off space for passes in graph streaming problems. *ACM Trans. Algorithms*, 6(1):6:1–6:17, December 2009.

[47] David Eppstein and Michael T. Goodrich. Straggler identification in round-trip data streams via newton's identities and invertible bloom filters. *IEEE Trans. Knowl. Data Eng.*, 23(2):297–306, 2011.

[48] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2-3):207–216, 2005.

[49] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. Graph distances in the data-stream model. *SIAM J. Comput.*, 38(5):1709–1727, 2008.

[50] Joan Feigenbaum, Sampath Kannan, and Jian Zhang. Annotation and computational geometry in the streaming model. Technical Report YALEU/DCS/TR-1249, Yale University, 2003.

[51] Dario Fiore and Rosario Gennaro. Publicly verifiable delegation of large polynomials and matrix computations, with applications. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 501–512, New York, NY, USA, 2012. ACM.

[52] M.L. Fredman and J. Komlós. On the size of separating systems and perfect hash functions. SIAM J. Algebra. Discr., 5(1):61–68, 1984.

[53] Rusins Freivalds. Fast probabilistic algorithms. In *MFCS*, pages 57–69, 1979.

[54] Dmitry Gavinsky and Alexander A. Sherstov. A separation of NP and coNP in multiparty communication complexity. *Theory of Computing*, 6(1):227–245, 2010.

[55] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: outsourcing computation to untrusted workers. In *Proceedings of the 30th Annual Conference on Advances in Cryptology*, CRYPTO'10, pages 465–482, Berlin, Heidelberg, 2010. Springer-Verlag.

[56] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 626–645. Springer, 2013.

[57] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, STOC '08, pages 113–122, New York, NY, USA, 2008. ACM.

[58] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989.

[59] Shafi Goldwasser and Michael Sipser. Private coins versus public coins in interactive proof systems. In Juris Hartmanis, editor, *STOC*, pages 59–68. ACM, 1986.

[60] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In Masayuki Abe, editor, *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2010.

[61] Tom Gur and Ran Raz. Arthur-Merlin streaming complexity. In *Proceedings of the 40th International Colloquium on Automata, Languages and Programming: Part I*, ICALP '13, Berlin, Heidelberg, 2013. Springer-Verlag.

[62] Sariel Har-Peled, Piotr Indyk, and Rajeev Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory of Computing*, 8(14):321–350, 2012.

[63] Johan Håstad and Avi Wigderson. The randomized communication complexity of set disjointness. *Theory of Computing*, 3(1):211–219, 2007.

[64] Monika R. Henzinger, Prabhakar Raghavan, and Sridhar Rajagopalan. Computing on data streams. *External memory algorithms*, pages 107–118, 1999.

[65] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2011. Version 1.3.0.

[66] Yuval Ishai, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient arguments without short PCPs. In *Proceedings of the Twenty-Second Annual IEEE Conference on Computational Complexity*, pages 278–291, Washington, DC, USA, 2007. IEEE Computer Society.

[67] Hossein Jowhari and Mohammad Ghodsi. New streaming algorithms for counting triangles in graphs. In Lusheng Wang, editor, *COCOON*, volume 3595 of *Lecture Notes in Computer Science*, pages 710–716. Springer, 2005.

[68] Tracy Kimbrel and Rakesh K. Sinha. A probabilistic algorithm for verifying matrix products using $o(n^2)$ time and $\log_2 n + o(1)$ random bits. *Inf. Process. Lett.*, 45(2):107–110, 1993.

[69] Valerie King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263–270, 1997.

[70] Hartmut Klauck. Rectangle size bounds and threshold covers in communication complexity. In *IEEE Conference on Computational Complexity*, pages 118–134, 2003.

[71] Hartmut Klauck. On Arthur Merlin games in communication complexity. In *IEEE Conference on Computational Complexity*, pages 189–199. IEEE Computer Society, 2011.

[72] Hartmut Klauck and Ved Prakash. Streaming computations with a loquacious prover. In Robert D. Kleinberg, editor, *ITCS*, pages 305–320. ACM, 2013.

[73] Jon Kleinberg and Eva Tardos. *Algorithm Design*. 2005.

[74] Feifei Li, Ke Yi, Marios Hadjieleftheriou, and George Kollios. Proof-infused streams: enabling authentication of sliding window queries on streams. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 147–158. VLDB Endowment, 2007.

[75] Helger Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In *Proceedings of the 9th International Conference on Theory of Cryptography*, TCC'12, pages 169–189, Berlin, Heidelberg, 2012. Springer-Verlag.

[76] Richard J. Lipton. Efficient checking of computations. In *STACS*, pages 207–215, 1990.

[77] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *J. ACM*, 39:859–868, October 1992.

[78] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay – a secure two-party computation system. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[79] Yaron Minsky, Ari Trachtenberg, and Richard Zippel. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory*, 49(9):2213–2218, 2003.

[80] S. Muthukrishnan. *Data Streams: Algorithms And Applications*. Foundations and Trends in Theoretical Computer Science. Now Publishers Incorporated, 2005.

[81] Michael Nüsken and Martin Ziegler. Fast multipoint evaluation of bivariate polynomials. In *ESA*, pages 544–555, 2004.

[82] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, 2013.

[83] Ran Raz and Amir Shpilka. On the power of quantum proofs. In *IEEE Conference on Computational Complexity*, pages 260–274. IEEE Computer Society, 2004.

[84] Alexander Razborov. On the distributional complexity of disjontness. In *ICALP*, pages 249–253, 1990.

[85] Guy Rothblum. *Delegating computation reliably : paradigms and constructions*. PhD thesis, Massachusetts Institute of Technology, 2009.

[86] Mert Sağlam and Gabor Tardos. On the communication complexity of sparse set disjointness and exists-equal problems. *CoRR*, abs/1209.3849, 2013.

[87] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley Series in Discrete Mathematics & Optimization. John Wiley & Sons, 1998.

[88] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, October 1980.

[89] Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Syst. Sci.*, 51(3):400–403, December 1995.

[90] Srinath Setty, Richard McPherson, Andrew J. Blumberg, and Michael Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, 2012.

[91] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 12–12, Berkeley, CA, USA, 2012. USENIX Association.

[92] Srinath T. V. Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. Resolving the conflict between generality and plausibility in verified computation. In Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek, editors, *EuroSys*, pages 71–84. ACM, 2013.

[93] Adi Shamir. IP = PSPACE. *J. ACM*, 39:869–877, October 1992.

[94] Alexander A. Sherstov. The multiparty communication complexity of set disjointness. In Howard J. Karloff and Toniann Pitassi, editors, *STOC*, pages 525–548. ACM, 2012.

[95] Alexander A. Sherstov. Communication lower bounds using directional derivatives. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *STOC*, pages 921–930. ACM, 2013.

[96] Justin Thaler. Source code. `http://people.seas.harvard.edu/~jthaler/Tcode.htm`, 2011.

[97] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *Proceedings of the 33rd Annual Conference on Advances in Cryptology*, CRYPTO'13, Berlin, Heidelberg, 2013. Springer-Verlag.

[98] Justin Thaler, Mike Roberts, Michael Mitzenmacher, and Hanspeter Pfister. Source code. `http://people.seas.harvard.edu/~jthaler/TRMPcode.htm`, 2012.

[99] Justin Thaler, Mike Roberts, Michael Mitzenmacher, and Hanspeter Pfister. Verifying computations with massively parallel interactive proofs. In *HotCloud*, 2012.

[100] Mikkel Thorup. Even strongly universal hashing is pretty fast. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete algorithms*, SODA, pages 496–497, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[101] Mikkel Thorup and Yin Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, pages 615–624, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[102] Peter A. Tucker, David Maier, Lois M. L. Delcambre, Tim Sheard, Jennifer Widom, and Mark P. Jones. Punctuated data streams, 2005.

[103] Victor Vu, Srinath Setty, Andrew J. Blumberg, and M. Walfish. Personal communication. January 2013.

[104] Victor Vu, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE Symposium on Security and Privacy*, 2013.

[105] Mihalis Yannakakis. Expressing combinatorial optimization problems by linear programs. *J. Comput. Syst. Sci.*, 43(3):441–466, 1991.

[106] Ke Yi, Feifei Li, Marios Hadjieleftheriou, George Kollios, and Divesh Srivastava. Randomized synopses for query assurance on data streams. In Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen, editors, *ICDE*, pages 416–425. IEEE, 2008.

[107] Raphael Yuster. Computing the diameter polynomially faster than APSP. *arXiv preprint arXiv:1011.6181*, 2010.

[108] Mariano Zelke. Weighted matching in the semi-streaming model. In *STACS*, pages 669–680, 2008.

[109] Richard Zippel. Probabilistic algorithms for sparse polynomials. In Edward W. Ng, editor, *EUROSAM*, volume 72 of *Lecture Notes in Computer Science*, pages 216–226. Springer, 1979.

Appendix A

Deferred Proofs from Chapter 8

A.1 Proof of Theorem 8.4.10

Proof. Consider layer i of the circuit C. Since $\text{in}_1^{(i)}$ and $\text{in}_2^{(i)}$ are regular, there is a subset of input bits $\mathcal{S}_i \subseteq [v]$ with $|\mathcal{S}_i| = c_i$ for some constant c_i such that each input bit in $[v] \setminus \mathcal{S}$ affects $O(1)$ of the output bits of $\text{in}_1^{(i)}$ and $\text{in}_2^{(i)}$. Number the input variables so that the numbers $\{1, \ldots, c_i\}$ correspond to variables in $\mathcal{S}_i$.

Let $\rho \in \{0,1\}^{c_i}$ be an assignment to the variables in $\mathcal{S}$, and let $I_\rho : \{0,1\}^{s_i} \to \{0,1\}$ denote the indicator function for ρ. For example, if $c_i = 3$ and $\rho = (1,0,1)$, then $I_\rho(x) = 1$ if $x_1 = 1, x_2 = 0$, and $x_3 = 1$, and $I_\rho(x) = 0$ otherwise. Let $\tilde{I}_\rho$ denote the multilinear extension of I_ρ. In the previous example, $\tilde{I}_\rho = x_1(1 - x_2)x_3$. Finally, let $\text{in}_{1,\rho}^{(i)}$ and $\text{in}_{2,\rho}^{(i)}$ denote the functions $\text{in}_1^{(i)}$ and $\text{in}_2^{(i)}$ with the variables in $\mathcal{S}_i$ fixed to the assignment ρ, and for $k \in \{1,2\}$, let $b_{\rho,k,j}$ denote the jth output bit of $\text{in}_{k,\rho}^{(i)}$.

By regularity, for each assignment $\rho \in \{0,1\}^{c_i}$ to the variables in $\mathcal{S}_i$, the jth output bit $b_{\rho,k,j}$ of in_ρ^k depends on only one variable $x_{q(\rho,k,j)} \in [s_i] \setminus \mathcal{S}_i$ for some function $q(\rho,k,j)$. Let $\tilde{b}_{\rho,k,j}(x_{q(\rho,k,j)}) : \mathbb{F} \to \mathbb{F}$ denote the multilinear extension of the function $b_{\rho,k,j}(x_{q(\rho,k,j)}) : \{0,1\} \to \{0,1\}$. If $b_{\rho,k,j}$ is not identically 0 or identically 1, then either $\tilde{b}_{\rho,k,j}(x_{q(\rho,k,j)}) = x_{q(\rho,k,j)}$

or $\tilde{b}_{\rho,k,j} = 1 - x_{q(\rho,k,j)}$.

For any $\rho \in \{0,1\}^{s_i}$, define $\tilde{\text{in}}_{1,\rho}^{(i)}$ to be the concatenation of the $\tilde{b}_{\rho,1,j}$ functions for all $j \in [s_{i+1}]$. Under this definition, $\tilde{\text{in}}_{1,\rho}^{(i)}$ is a collection of s_{i+1} linear polynomials, where each of the polynomials depends on a single variable, and we may view $\tilde{\text{in}}_{1,\rho}^{(i)}$ as a single function mapping $\mathbb{F}^{s_i}$ to $\mathbb{F}^{s_{i+1}}$. We define $\tilde{\text{in}}_{2,\rho}^{(i)}$ and $\tilde{\text{type}}_{\rho}^{(i)}$ analogously to $\tilde{\text{in}}_1$.

Now let

$$W^{(i)}(p) = \sum_{\rho \in L^{(i)}} \tilde{I}_\rho(p) \cdot \Bigg(\tilde{\text{type}}_{\rho}^{(i)}(p) \cdot \tilde{V}_{i+1}\left(\tilde{\text{in}}_{1,\rho}^{(i)}(p)\right) \cdot \tilde{V}_{i+1}\left(\tilde{\text{in}}_{2,\rho}^{(i)}(p)\right) + \left(1 - \tilde{\text{type}}_{\rho}^{(i)}(p)\right)\left(\tilde{V}_{i+1}\left(\tilde{\text{in}}_{1,\rho}^{(i)}(p)\right) + \tilde{V}_{i+1}\left(\tilde{\text{in}}_{2,\rho}^{(i)}(p)\right)\right)\Bigg).$$

It is easily checked that for all $p \in \{0,1\}^{s_i}$, $V_i(p) = W^{(i)}(p)$. Lemma 8.4.1 then implies that $\tilde{V}_i(z) = \sum_{p \in \{0,1\}^{s_i}} g_z^{(i)}(p)$, where $g_z^{(i)}(p) = \beta_{s_i}(z,p) \cdot W^{(i)}(p)$. Our protocol follows precisely the description of Section 8.4.1, with $\mathcal{P}$ and $\mathcal{V}$ applying the sum-check protocol to the polynomial $g_z^{(i)}$ at iteration i.

Communication Costs and Costs to $\mathcal{V}$. Notice that our polynomial $g_z^{(i)}(p) = \beta(z,p) \cdot W^{(i)}(p)$ has degree $O(1)$ in each variable. Indeed, $\beta(z,p)$ has degree 1 in each variable. Moreover, $W^{(i)}(p)$ is a sum of polynomials that each have degree $O(1)$ in each variable, and hence $W^{(i)}(p)$ itself has degree $O(1)$ in each variable.

This latter fact can be seen by observing that for each assignment $\rho \in \{0,1\}^{c_i}$ to the variables in $\mathcal{S}_i$, it holds that $\tilde{I}_\rho(p)$, $\tilde{\text{type}}_{\rho}^{(i)}(p)$, $\tilde{V}_{i+1}\left(\tilde{\text{in}}_{1,\rho}^{(i)}(p)\right)$ and $\tilde{V}_{i+1}\left(\tilde{\text{in}}_{2,\rho}^{(i)}(p)\right)$ all have constant degree in each variable. That $\tilde{V}_{i+1}\left(\tilde{\text{in}}_{1,\rho}^{(i)}(p)\right)$ and $\tilde{V}_{i+1}\left(\tilde{\text{in}}_{2,\rho}^{(i)}(p)\right)$ have constant degree in each variable follows from the facts that $\tilde{V}_{i+1}$ is a multilinear polynomial, and that each input variable $j \in [s_i] \setminus \mathcal{S}_i$ affects at most a constant number of outputs for $\tilde{\text{in}}_{1,\rho}$ and $\tilde{\text{in}}_{2,\rho}$ by Property 1 of Definition 8.4.8.

Since $g_z^{(i)}(p)$ has degree $O(1)$ in each variable, the claimed communication cost and the costs to the verifier follow immediately by summing the corresponding costs of the sum-check

protocols over all iterations $i \in \{1, \ldots, d(n)\}$ (see Section 2.3).

Time Cost for $\mathcal{P}$. It remains to demonstrate how $\mathcal{P}$ can compute her prescribed messages when applying the sum-check protocol to the polynomial $g_z^{(i)}$ in time $O(S_i + S_{i+1})$. It will follow that $\mathcal{P}$'s runtime over all $d(n)$ invocations of the sum-check protocol is $O(\sum_{i=1}^{d(n)} S_i) = O(S(n))$.

As in our analysis of Section 8.4.4, it suffices to show how $\mathcal{P}$ can quickly evaluate $g_z^{(i)}$ at all points in $S^{(j)}$, where $S^{(j)}$ consists of all points of the form $p = (r_1, \ldots, r_{j-1}, t, p_{j+1}, \ldots, p_{s_i})$ with $t \in \{0, 1, \ldots, \deg_j(g_z^{(i)})\}$ and $(p_{j+1}, \ldots, p_{s_i}) \in \{0,1\}^{s_i - j}$. As $g_z^{(i)}(p) = \beta_{s_i}(z, p) \cdot W^{(i)}(p)$, it suffices for $\mathcal{P}$ to evaluate $\beta_{s_i}(z, \cdot)$ and $W(\cdot)$ at all such points p. The $\beta_{s_i}(z, \cdot)$ computations can be done in $O(S_i)$ total time across all iterations of the sum-check protocol, exactly as in Section 8.4.4.

To see how $\mathcal{P}$ can efficiently evaluate all of the $W^{(i)}(p)$ values efficiently, notice that for any fixed point $p \in \mathbb{F}^{s_i}$, $W^{(i)}(p)$ can be computed efficiently given $\tilde{\text{type}}_\rho^{(i)}(p)$, $\tilde{V}_{i+1}(\tilde{\text{in}}_{1,\rho}(p))$, and $\tilde{V}_{i+1}(\tilde{\text{in}}_{2,\rho}(p))$ for all $\rho \in \{0,1\}^{c_i}$. As $|\mathcal{S}_i| = c_i = O(1)$, modulo a constant-factor blowup in runtime it suffices to explain how to perform these evaluations for a fixed restriction $\rho \in \{0,1\}^{c_i}$ to the variables in $\mathcal{S}_i$.

It is easy to see that $\tilde{\text{type}}_\rho^{(i)}(p)$ can be evaluated in constant time, since this function depends on only 1 input variable $x_{q(\rho,3,1)}$. All that remains is to show how $\mathcal{P}$ can evaluate $\tilde{V}_{i+1}(\tilde{\text{in}}_{1,\rho}(p))$ quickly; the case for $\tilde{V}_{i+1}(\tilde{\text{in}}_{2,\rho}(p))$ is similar.

To this end, we follow the approach of Section 8.4.4.

Pre-processing. $\mathcal{P}$ will begin by computing an array $V^{(0)}$, which is simply defined to be the vector of gate values at layer $i+1$ i.e., identifying a number $0 < j < S_{i+1}$ with its binary representation in $\{0,1\}^{s_{i+1}}$, $\mathcal{P}$ sets $V^{(0)}[(j_1, \ldots, j_{s_{i+1}})] = V_{i+1}(j_1, \ldots, j_{s_{i+1}})$ for each $(j_1, \ldots, j_{s_{i+1}}) \in \{0,1\}^{s_{i+1}}$. The right hand side of this equation is simply the value of the jth gate at layer $i+1$ of C. So $\mathcal{P}$ can fill in the array $V^{(0)}$ when she evaluates the circuit C,

before receiving any messages from $\mathcal{V}$.

Overview of Online Processing. Assume without loss of generality that the output bits of $\tilde{\text{in}}_{1,\rho}(p)$ are labelled in increasing order of the input bits they are affected by. So for example if p_1 affects 2 output bits of $\tilde{\text{in}}_{1,\rho}$ and p_2 affects 3 output bits, then the bits affected by p_1 are labelled 1 and 2 respectively, while the bits affected by p_2 are labelled 3, 4, and 5.

In round j of of the sum-check protocol, $\mathcal{P}$ needs to evaluate the polynomial $\tilde{V}_{i+1}$ at the $O(2^{s_{i+1}-j})$ points in the sets $\tilde{\text{in}}_{1,\rho}(S^{(j)})$ and $\tilde{\text{in}}_{2,\rho}(S^{(j)})$. $\mathcal{P}$ will do this using the help of intermediate arrays as follows.

Efficiently Constructing $V^{(j)}$ Arrays. Let a_{j-1} denote the total number of output bits affected by the first $j-1$ input variables. Inductively, assume $\mathcal{P}$ has computed in the previous round an array $V^{(j-1)}$ of length $2^{s_{i+1}-a_{j-1}}$, such that for each $p = (p_{a_{j-1}+1}, \ldots, p_{s_{i+1}}) \in \{0,1\}^{s_{i+1}-a_{j-1}}$, the pth entry of $V^{(j-1)}$ equals

$$V^{(j-1)}[(p_{a_{j-1}+1}, \ldots, p_{s_{i+1}})] =$$

$$\sum_{(c_1,\ldots,c_{a_{j-1}})\in\{0,1\}^{a_{j-1}}} V_{i+1}(c_1, \ldots, c_{a_{j-1}}, p_{a_{j-1}+1}, \ldots, p_{s_{i+1}}) \cdot \prod_{k=1}^{j-1} \chi_{c_k}(\tilde{b}_{\rho,1,k}(r_{q(\rho,1,k)})),$$

where recall that $q(\rho, 1, k)$ is the input bit that output bit k of $\text{in}_{1,\rho}$ depends on. As the base case, we explained how $\mathcal{P}$ can fill in $V^{(0)}$ in the process of evaluating the circuit C.

Let $x_1, \ldots, x_{s_i}$ denote the input variables to in_1, and let $b_1, \ldots, b_{s_{i+1}}$ denote the outputs of in_1. Intuitively, at the end of round j of the sum-check protocol, $\mathcal{P}$ must "bind" input variable x_j to value $r_j \in \mathbb{F}$. This has the effect of binding the output variables affected by x_j, since each such output variable depends only on x_j. For illustration, suppose the variable x_1 affects output variable b_1; specifically, suppose that $b_1 = 1 - x_1$. Then binding x_1 to value r_1 has the effect of binding b_1 to value $1 - r_1$. $V^{(j)}$ is obtained from $V^{(j-1)}$ by taking this into account. We formalize this as follows.

Assume that variable x_j affects only one output variable $b_{\rho,1,a_{j-1}+1}$, and thus $a_j = a_{j-1}+1$;

if this is not the case, we can compute $V^{(j)}$ by applying the following update once for each output variable affected by x_j. Observe that $\mathcal{P}$ can compute $V^{(j)}$ given $V^{(j-1)}$ in $O(2^{s_{i+1}-a_{j-1}})$ time using the following recurrence:

$$V^{(j)}[(p_{a_j+1},\ldots,p_{s_{i+1}})] =$$
$$V^{(j-1)}[(0,p_{a_j+1},\ldots,p_{s_{i+1}})]\cdot\chi_0(\tilde{b}_{\rho,1,a_j}(r_j)) + V^{(j-1)}[(1,p_{a_j+1},\ldots,p_{s_{i+1}})]\cdot\chi_1(\tilde{b}_{\rho,1,a_j}(r_j)).$$

Thus, at the end of round j of the sum-check protocol, when $\mathcal{V}$ sends $\mathcal{P}$ the value r_j, $\mathcal{P}$ can compute $V^{(j)}$ from $V^{(j-1)}$ in $O(2^{s_{i+1}-a_{j-1}})$ time.

Using the $V^{(j)}$ Arrays. We now show how to use the array $V^{(j-1)}$ to evaluate $\tilde{V}_{i+1}(\tilde{\text{in}}_{1,\rho}(p))$ in $O(1)$ time for any point p of the form $p=(r_1,\ldots,r_{j-1},t,p_{j+1},\ldots,p_{s_i})$ with $(p_{j+1},\ldots,p_{s_i})\in\{0,1\}^{s_i-j}$. In order to ease notation in the following derivation, we make the simplifying assumption that $\tilde{b}_{\rho,1,k}(x_{q(\rho,1,k)}) = x_{q(\rho,1,k)}$ for all output bits $k\in[s_{i+1}]$. The derivation when this assumption does not hold is similar.

We exploit the following sequence of equalities:

$$\tilde{V}_{i+1}(\tilde{\text{in}}_{1,\rho}(p)) = \sum_{c\in\{0,1\}^{s_{i+1}}} V_{i+1}(c)\chi_c(\tilde{\text{in}}_{1,\rho}(p))$$
$$= \sum_{(c_1,\ldots,c_{a_{j-1}})\in\{0,1\}^{a_{j-1}}} \sum_{(c_{a_{j-1}+1},\ldots,c_{s_{i+1}})\in\{0,1\}^{s_{i+1}-a_{j-1}}} V_{i+1}(c)\chi_c(\tilde{\text{in}}_{1,\rho}(p))$$
$$= \sum_{(c_1,\ldots,c_{a_{j-1}})\in\{0,1\}^{a_{j-1}}} \sum_{(c_{a_{j-1}+1},\ldots,c_{s_{i+1}})\in\{0,1\}^{s_{i+1}-a_{j-1}}} V_{i+1}(c)\left(\prod_{k=1}^{a_{j-1}}\chi_{c_k}(\tilde{b}_{\rho,1,k}(r_{q(\rho,1,k)}))\right)\cdot$$
$$\left(\prod_{k=a_{j-1}+1}^{a_j}\chi_{c_k}(\tilde{b}_{\rho,1,k}(t))\right)\left(\prod_{k=a_j+1}^{s_{i+1}}\chi_{c_k}(p_{q(\rho,1,k)})\right)$$
$$= \sum_{(c_1,\ldots,c_{a_j})\in\{0,1\}^{a_j}} V_{i+1}(c_{j+1},\ldots,c_{a_j},p_{q(\rho,1,a_j+1)},\ldots,p_{q(\rho,1,s_{j+1})})\left(\prod_{k=1}^{a_{j-1}}\chi_{c_k}(r_k)\right)\cdot\left(\prod_{k=a_{j-1}+1}^{a_j}\chi_{c_k}(t)\right)$$
$$= \sum_{(p_{a_{j-1}+1},\ldots,p_{a_j})\in\{0,1\}^{a_j-a_{j-1}}} V^{(j-1)}[(p_{q(\rho,1,a_{j-1}+1)},\ldots,p_{q(\rho,1,s_{j+1})})]\cdot\prod_{k=a_{j-1}+1}^{a_j}\chi_{p_k}(t).$$

Here, the first equality holds by Equation (8.3). The third holds by definition of the functions χ_c and $\tilde{\text{in}}_1$, as well as the assumption that $\tilde{b}_{\rho,1,k}(x_{q(\rho,1,k)}) = x_{q(\rho,1,k)}$ for all $k\in[s_{i+1}]$. The

fourth holds because for Boolean values $c_k, p_{q(\rho,1,k)} \in \{0,1\}$, $\chi_{c_k}(p_{q(\rho,1,k)}) = 1$ if $c_k = p_{q(\rho,1,k)}$, and $\chi_{c_k}(p_{q(\rho,1,k)}) = 0$ otherwise. The final equality holds by definition of the array $V^{(j-1)}$.

The final expression above can be computed with $O(2^{a_j - a_{j-1}})$ time given the array $V^{(j-1)}$. Since $a_j - a_{j-1}$ is constant by Property 1 of Definition 8.4.8, $O(2^{a_j - a_{j-1}}) = O(1)$.

Putting Things Together. In round j of the sum-check protocol, $\mathcal{P}$ uses the array $V^{(j-1)}$ to evaluate $\tilde{V}_{i+1}(\tilde{\text{in}}_1(p))$ for all $O(2^{s_i - j})$ points $p \in S^{(j)}$, which requires constant time per point and hence $O(2^{s_i - j})$ time over all points in $S^{(j)}$. At the end of round j, $\mathcal{V}$ sends $\mathcal{P}$ the value r_j, and $\mathcal{P}$ computes $V^{(j)}$ from $V^{(j-1)}$ in $O(2^{s_{i+1} - a_{j-1}})$ time. By ordering input variables in such a way that $a_j > a_{j-1}$ for all j, we ensure that in total across all rounds of the sum-check protocol, $\mathcal{P}$ spends $O(\sum_{j=1}^{s_i} 2^{s_i - j} + 2^{s_{i+1} - j}) = O(2^{s_i} + 2^{s_{i+1}})$ time to evaluate $\tilde{V}_{i+1}$ at the relevant points. When combined with our $O(2^{s_i})$-time algorithm for computing all the relevant $\beta(z,p)$ values, we see that $\mathcal{P}$ takes $O(2^{s_i} + 2^{s_{i+1}}) = O(S_i + S_{i+1})$ time to run the entire sum-check protocol for iteration i of our circuit-checking protocol.

Reducing to Verification of a Single Point. After executing the sum-check protocol at layer i as described above, $\mathcal{V}$ is left with a claim about $\tilde{V}_{i+1}(\omega_1)$ and $\tilde{V}_{i+1}(\omega_2)$ from two points $\omega_1, \omega_2 \in \mathbb{F}^{s_{i+1}}$. If i is a layer for which $\text{in}_1^{(i)}$ and $\text{in}_2^{(i)}$ are similar (see Definition 8.4.9), we run the reducing to verification of a single point phase exactly as in the basic GKR protocol. This requires $\mathcal{P}$ to send $\tilde{V}_{i+1}(\ell(t))$ for a canonical line $\ell(t)$ that passes through the points ω_1 and ω_2. Because $\text{in}_1^{(i)}$ and $\text{in}_2^{(i)}$ are similar, it is easily seen that $\tilde{V}_{i+1}(\ell(t))$ is a univariate polynomial of constant degree. Hence $\mathcal{P}$ can specify $\tilde{V}_{i+1}(\ell(t))$ by sending $\tilde{V}_{i+1}(\ell(t_j))$ for $O(1)$ many points $t_j \in \mathbb{F}$. Using the method of Lemma 8.3.1, $\mathcal{P}$ can evaluate $\tilde{V}_{i+1}$ at each point $\ell(t_j)$ in $O(S_{i+1})$ time, and hence can perform all $\tilde{V}_{i+1}(\ell(t_j))$ evaluations in $O(S_{i+1})$ time in total.

Let $c = O(1)$ be the number of layers i for which $\text{in}_1^{(i)}$ and $\text{in}_2^{(i)}$ are not similar. At each such layer i, we skip the "reducing to verification at a single point" phase of the protocol.

Each time we do this, it doubles the number of points $\omega \in \mathbb{F}^{s_{i+1}}$ that must be considered at the next iteration. However, we only skip the "reducing to verification at a single point" phase c times, and thus at all layers i of the circuit, $\mathcal{V}$ needs to check $\tilde{V}_i(\omega_j)$ for at most $2^c = O(1)$ points. This affects $\mathcal{P}$'s and $\mathcal{V}$'s runtime by at most a $2^c = O(1)$ factor, and the $O(S)$ time bound for $\mathcal{P}$, and the $O(n \log n + d(n) \log S(n))$ time bound for $\mathcal{V}$ follow. □

A.2 Analysis for Pattern Matching

Let C be the circuit for pattern matching described in Section 8.4.5. Our goal in this appendix is to handle the layer of the circuit adjacent to the input layer. Call this layer ℓ. Layer ℓ computes $t_{i+k} - p_k$ for each pair $(i,k) \in \{0,\ldots,n-1\} \times \{0,\ldots,q-1\}$. We want to show how to use a sum-check protocol to reduce a claim about the value of $\tilde{V}_\ell(z)$ for some $z \in \mathbb{F}^{s_\ell}$ to a claim about $\tilde{V}_{\ell+1}(r)$ for some $r \in \mathbb{F}^{s_{\ell+1}}$, while ensuring that $\mathcal{P}$ runs in time $O(S_\ell) = O(nm)$.

The idea underlying our analysis here is the following. The reason Theorem 8.4.10 does not apply to layer ℓ is that the first in-neighbor of a gate with label

$$p = (i_1,\ldots,i_{\log n}, k_1,\ldots,k_{\log m}) \in \{0,1\}^{\log n + \log m}$$

has label equal to the binary representation of the integer $i+k$, and a single bit i_k can affect many bits in the binary representation of $i+k$ (likewise, each bit in the binary representation of $i+k$ may be affected by many bits in the binary representation of i and k). In order to ensure that each bit of p affects only a single bit of $y = \mathrm{in}_1^{(\ell)}(p)$, we introduce $\log n$ dummy variables $(c_1,\ldots,c_{\log n})$ and force the jth dummy variable c_j to have value equal to the jth *carry bit* when adding numbers i and k in binary. Now each bit of p affects only one output bit, and each output bit y_j is only affected by at most three "input bits": i_j, k_j, and c_j if $j \leq \log m$, and just i_j and c_j if $j > \log m$.

To this end, let $\phi : \{0,1\}^4 \to \{0,1\}$ be the function that evaluates to 1 on input (i_1, k_1, c_0, c_1) if and only if $c_1 = 0$ and $i_1 + k_1 + c_0 < 2$ or $c_1 = 1$ and $i + k + c_0 \geq 2$. That is, ϕ outputs 1 if and only if c_1 is equal to the carry bit when adding i_1, k_1, and c_0. Let $\tilde{\phi}$ be the multilinear extension of ϕ. Notice $\tilde{\phi}$ can be evaluated at any point $r \in \mathbb{F}^4$ in $O(1)$ time.

Now let (i, k, c) denote a vector in $\mathbb{F}^{\log n} \times \mathbb{F}^{\log m} \times \mathbb{F}^{\log n}$, and define

$$\Phi(i, k, c) := \prod_{j=1}^{\log n} \tilde{\phi}(i_j, k_j, c_{j-1}, c_j),$$

where it is understood that $c_{-1} = 0$ and $k_j = 0$ for $j > \log m$.

For any Boolean vector $(i, k, c) \in \{0,1\}^{\log n} \times \{0,1\}^{\log m} \times \{0,1\}^{\log n}$, it is easily verified that $\Phi(i, k, c) = 1$ if and only if for all j, c_j equals the jth *carry bit* when adding numbers i and k in binary.

Finally, let $\gamma : \{0,1\}^3 \to \{0,1\}$ be the function that evaluates to 1 on input (i_1, k_1, c_1) if and only if $i_1 + k_1 + c_1 = 1 \mod 2$. Let $\tilde{\gamma}$ be the multilinear extension of γ. Notice $\tilde{\gamma}$ can be evaluated at any point $r \in \mathbb{F}^3$ in $O(1)$ time.

Now consider the following $\log n + \log m$-variate polynomial over the field $\mathbb{F}$:

$$W^{(\ell)}(i, k) = \sum_{(c_1, \ldots, c_{\log n}) \in \{0,1\}^{\log n}} \Phi(i, k, c) \cdot$$
$$\left(\tilde{T}(\tilde{\gamma}(i_1 + k_1 + c_0), \ldots, \tilde{\gamma}(i_{\log n} + k_{\log n} + c_{\log n - 1})) - \tilde{P}(k_1, \ldots, k_{\log m}) \right),$$

where again it is understood that $c_{-1} = 0$ and $k_j = 0$ for $j > \log m$. Here, $\tilde{T}$ is the multilinear extension of the input T, viewed as a function from $\{0,1\}^{\log n}$ to $[n]$, and $\tilde{P}$ is the multilinear extension of the input pattern P, viewed as a function from $\{0,1\}^{\log m}$ to $[n]$.

It can be seen that for all Boolean vectors $(i, k) = \{0,1\}^{\log n} \times \{0,1\}^{\log m}$, $W^{(\ell)}(i, k) = V_\ell(i, k)$. This is because for any $(i, k) \in \{0,1\}^{\log n} \times \{0,1\}^{\log m}$, $\Phi(i, k, c)$ will be zero for all c except the c consisting of the correct carry bits for i and k, and for this input

c, $\tilde{T}(\tilde{\gamma}(i_1 + k_1 + c_0), \ldots, \tilde{\gamma}(i_{\log n} + k_{\log n} + c_{\log n - 1}))$ will equal $T(i + k)$ when interpreting i, k as integers in the natural way.

Lemma 8.4.1 then implies that for all $z \in \mathbb{F}^{\log n + \log m}$,

$$\tilde{V}_\ell(z) = \sum_{(i,k) \in \{0,1\}^{\log n} \times \{0,1\}^{\log m}} \beta_{\log n + \log m}(z, (i,k)) \cdot W^{(\ell)}(i,k)$$

$$= \sum_{(i,k,c) \in \{0,1\}^{\log n} \times \{0,1\}^{\log m} \times \{0,1\}^{\log n}} \beta_{\log n + \log m}(z, (i,k)) \cdot$$

$$\Phi(i,k,c) \cdot \left(\tilde{T}(\tilde{\gamma}(i_1 + k_1 + c_0), \ldots, \tilde{\gamma}(i_{\log n} + k_{\log n} + c_{\log n - 1})) - \tilde{P}(j_1, \ldots, j_{\log m}) \right).$$

Therefore, in order to reduce a claim about $\tilde{V}_\ell(z)$ to a claim about $\tilde{T}(r_1)$ and $\tilde{P}(r_2)$ for random vectors $r_1 \in \mathbb{F}^{\log n}$ and $r_2 \in \mathbb{F}^{\log m}$, it suffices to apply the sum-check protocol to the $2 \log n + \log m$-variate polynomial

$$g_z(i,k,c) = \beta_{\log n + \log m}(z, (i,k)) \cdot \Phi(i,k,c) \cdot$$
$$\left(\tilde{T}(\tilde{\gamma}(i_1 + k_1 + c_0), \ldots, \tilde{\gamma}(i_{\log n} + k_{\log n} + c_{\log n - 1})) - \tilde{P}(j_1, \ldots, j_{\log m}) \right).$$

It remains to show how to extend the techniques underlying Theorem 8.4.10 to allow $\mathcal{P}$ to compute all of the required messages in this sum-check protocol in $O(nm)$ time. For brevity, we restrict ourselves to a sketch of the techniques.

The first obvious complication is that the sum defining $\mathcal{P}$'s message in a given round of the sum-check protocol has as many as $2^{2 \log n + \log m} = \Omega(mn^2) > nm$ terms. Fortunately, the Φ polynomial ensures that almost all of these terms are zero: when considering any Boolean setting of the variables i_j, k_j, and c_{j-1}, the only setting of c_j that $\mathcal{P}$ must consider is the one corresponding to the carry bit of $i_j + k_j + c_{j-1}$ i.e., the unique setting of c_j such that $\phi(i_j, k_j, c_{j-1}, c_j) = 1$. This ensures that at round $3j$, $3j + 1$, and $3j + 2$ of the sum-check protocol applied to g_z, $\mathcal{P}$ must only evaluate g_z at $O(2^{\log n + \log m - j})$ terms, which is falling geometrically quickly with j.

We now turn to explaining how $\mathcal{P}$ can evaluate g_z at all necessary points in round $3j$, $3j+1$ and $3j+2$ in total time $O(2^{\log n+\log m-j})$. To accomplish this, it is sufficient for $\mathcal{P}$ to evaluate $\beta_{\log n+\log m}$ at the necessary points, as well as Φ, $\tilde{T}$, and $\tilde{P}$ at the necessary points. The $\beta_{\log n+\log m}$ evaluations are handled exactly as in Theorem 8.4.10 i.e., by using $C^{(j)}$ arrays (but these arrays only get updated every time a variable i_j or k_j gets bound within the sum-check protocol; no update is necessary when a variable c_j gets bound). The $\tilde{P}$ evaluations are also handled exactly as in Theorem 8.4.10, using $V^{(j)}$ arrays that only need to be updated when a variable k_j gets bound.

The $\tilde{T}$ evaluations require some additional explanation on top of the analysis of Theorem 8.4.10. We want $\mathcal{P}$ to be able to use $V^{(j)}$ arrays as in Theorem 8.4.10 to evaluate $\tilde{T}$ at the necessary points in constant time per point, but we need to make sure that $\mathcal{P}$ can compute array $V^{(j)}$ from $V^{(j-1)}$ in time that falls geometrically quickly with j. In order to do this, it is essential to choose a specific ordering for the sum in the sum-check protocol.

Specifically, we write the sum as:

$$\sum_{i_1}\sum_{k_1}\sum_{c_1}\sum_{i_2}\sum_{k_2}\sum_{c_2}\cdots\sum_{i_{\log n}}\sum_{c_{\log n}} g_z(i,k,c).$$

This ensures that e.g. (i_1,k_1,c_1) are the first three variables in the sum-check protocol to become bound to random values in $\mathbb{F}$. The reason we must do this is so that every 3 rounds, another value $\tilde{\gamma}(i_j+k_j+c_{j-1})$ feeding into $\tilde{T}$ becomes bound to a specific value (and moreover the outputs of $\tilde{\gamma}(i_{j'}+k_{j'}+c_{j'-1})$ are unaffected by the bound variables for all $j'>j$). This is precisely the property we exploited in the protocol of Theorem 8.4.10 to ensure that the $V^{(j)}$ arrays there halved in size every round, and that $V^{(j)}$ could be computed from $V^{(j-1)}$ in time proportional to its size. So we can use $V^{(j)}$ arrays to efficiently perform the $\tilde{T}$ evaluations, updating the arrays every time another value $\tilde{\gamma}(i_j+k_j+c_{j-1})$ feeding into $\tilde{T}$ becomes bound to a specific value.

Finally, the Φ evaluations can be handled as follows. Consider for simplicity round $3j$ of

the protocol. Recall that $\mathcal{P}$ only needs to evaluate Φ at points for which $\phi_{j'}(i_{j'}, k_{j'}, c_{j'-1}, c_{j'}) = 1$ for all $j' > j$. Thus, for all $j' > j$, $\phi_{j'}$ does not affect the product defining Φ. So in order to evaluate Φ at the relevant points, it suffices for $\mathcal{P}$ to evaluate the $\phi_{j'}$s for $j' \leq j$. Now at round $3j$ of the protocol, all triples $(i_{j'}, k_{j'}, c_{j'})$ for $j' < j$ are already bound, say to the values $(r_{j'}^{(i)}, r_{j'}^{(k)}, r_{j'}^{(c)})$, and hence all the $\phi_{j'}$ functions for $j' < j$ are themselves already bound to specific values. So in order to quickly determine the contribution of the $\phi_{j'}$s for $j' < j$ to the product defining Φ, it suffices for $\mathcal{P}$ to maintain the quantity $\prod_{j'<j} \phi_{j'}(r_{j'}^{(i)}, r_{j'}^{(k)}, r_{j'}^{(c)})$ over the course of the protocol, which takes just $O(\log n)$ time in total. Finally, the contribution of ϕ_j to the product defining Φ can be computed in constant time per point. This completes the proof that Φ can be evaluated by $\mathcal{P}$ at all of the necessary points in $O(1)$ time per point over all rounds of the sum-check protocol, and completes the proof of the theorem.