Understanding Lasso – A Novel Lookup Argument Protocol

Oleg Fomenko
Distributed Lab
oleg.fomenko@distributedlab.com

Anton Levochko
Distributed Lab
anton.levochko@distributedlab.com

Abstract

In 2023, Srinath Setty, Justin Thaler, and Riad Wahby published a paper that describes a novel lookup argument with efficient verification called Lasso. We present a focused and accessible overview of the Lasso lookup argument that stands for the foundational component of the Jolt ZK-VM. This article distills the core principles behind Lasso: the sum-check protocol, multilinear polynomials and their extensions, Spark commitment, offline memory-checking, and the evolution of Spark called Surge. By clarifying the underlying protocols and their relationship to innovations like Spark and Surge, we aim to provide researchers and engineers with practical insights into the cryptographic foundations powering both Lasso and the Jolt virtual machine.

1 Introduction

This document presents a clear, streamlined overview of the Lasso lookup argument, first introduced in [STW23]. At its core, Lasso is a family of lookup arguments that enables efficient verification of "out-of-circuit" table lookups, crucial for stateful and memory-heavy computations. It serves as the centerpiece of Jolt, a ZK-VM designed by the a16zCrypto team. Our goal is to make the underlying protocol more accessible by concentrating on its core component: the cryptographic commitment to a large, sparse matrix. We hope this concise treatment will serve as a practical guide for researchers and engineers seeking to understand the cryptographic foundations of both Lasso and Jolt.

Firstly, we begin by examining the lookup protocol. Suppose that the verifier has a commitment to a table $t \in \mathbb{F}^n$ as well as a commitment to another vector $a \in \mathbb{F}^m$. Suppose that a prover wishes to prove that all entries in a are in the table t. There is a simple, commonly known way to prove this statement: prove the existence of a matrix $M \in \mathbb{F}^{m \times n}$ where each row has only one non-zero element (namely the 1) such that the following statement works:

$$M \cdot t = a$$

We call such matrix M a **unit-row matrix**. This statement can be easily verified by proving that for a random row r selected by the verifier, the following holds:

$$\langle M_r, t \rangle = a_r$$

where M_r denotes the r-th row and a_r denotes r-th element of a. More precisely, in Lasso, the prover provides a commitment to the multilinear extensions of matrix M and vectors a, t that enables this check for a random coin $r \in \mathbb{F}^m$ up to a negligible soundness error (following Schwartz–Zippel lemma):

$$\sum_{y \in \{0,1\}^{\log n}} \widetilde{M}(r, y) \cdot \widetilde{t}(y) = \widetilde{a}(r)$$

where r and y denote row and column indexes of the matrix.

The main magic happens in the way Lasso commits to the multilinear extension of matrix M and provides the opening at random row r. While the matrix M is naturally sparse, Lasso [STW23] presents **Spark** and its extension called **Surge** that allows us to commit to the sparse matrix M efficiently.

In the paragraphs below, we focus on the Spark construction and foundational protocols such as the sum-check protocol and the offline memory check. We also describe the Sona multilinear polynomial commitment, which is referenced as the primary commitment method in the original paper. Finally, after presenting Spark and its underlying components, we describe Surge – a generalization of Spark – and introduce some notes about the Spark protocol complexity.

2 Basic Knowledge

This section introduces the key protocols and concepts necessary to understand how Lasso works. Most of the definitions are adapted from [Tha22] with minor modifications; refer to it for more detailed explanations. We assume that the reader is already acquainted with fundamental cryptographic primitives (including commitment schemes, oracle access, and interactive proof systems) as well as core concepts from linear algebra (such as groups, fields, and polynomials).

2.1 Multilinear Polynomial

We call an ℓ-variate polynomial $f\colon \mathbb{F}^\ell \to \mathbb{F}$ **multilinear** if it has degree at most one in each variable. Let $f\colon \{0,1\}^\ell \to \mathbb{F}$ be any function mapping the ℓ-dimensional boolean hypercube to a field $\mathbb{F}$. By definition, it equals

$$f(x) = \sum_{k \subseteq [\ell]} a_k \prod_{i \in k} x_i$$

For f we can observe an equivalence between the notions of a "multilinear polynomial" and "ordered list". Indeed, any ℓ-variate multilinear polynomial f can be represented as a vector of 2^ℓ elements, where the element at position $i = \sum_{j=0}^{\ell-1} 2^j \cdot r_j$ equals to the $f(r_0, \dots, r_{\ell-1})$. This list can be naturally expressed as

$$t = \{f(0,\dots,0), f(1,0,\dots,0), \dots, f(0,\dots,0,1), \dots, f(0,1,\dots,1), f(1,\dots,1)\}$$

where the arguments are lexicographically ordered. To perform the inverted transformation, we can define the equality function $\mathrm{eq}\colon \{0,1\}^\ell \times \{0,1\}^\ell \to \mathbb{F}$ as follows:

$$\mathrm{eq}(x, e) = \begin{cases} 1 & \text{if } x = e, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

If we can represent this function as a polynomial, then we can build a multilinear polynomial from our array t as follows (here, the notation $t[i]$ denotes the i-th element of the array t):

$$f(r_0, \dots, r_{\ell-1}) = \sum_{x \in \{0,1\}^\ell} t\Big[\sum_{j=0}^{\ell-1} 2^j \cdot x_j\Big] \cdot \mathrm{eq}(x_0, \dots, x_{\ell-1}, r_0, \dots, r_{\ell-1})$$

Remark 1. Here, we used the notation $\sum 2^j x_j$ to denote the conversion from a binary string to its corresponding value in $\mathbb{F}$. For simplicity, in the remainder of this document, we will generally omit this explicit notation and assume that the conversion from a binary array x to its value in $\mathbb{F}$ is performed.

Remark 2. Since we've shown the equivalence between ordered lists and multilinear polynomials, we will henceforth treat these terms as interchangeable on both the prover's and verifier's sides. In particular, we assume that any multilinear polynomial can be accessed in either of these two forms (and their derivative forms) at any time.

2.2 Multilinear Extension (MLE)

We say that $\widetilde{f}\colon \mathbb{F}^\ell \to \mathbb{F}$ **extends** $f\colon \{0,1\}^\ell \to \mathbb{F}$ if $\widetilde{f}(x) = f(x)$ for all $x \in \{0,1\}^\ell$. The total degree of an ℓ-variate polynomial f refers to the maximum sum of the exponents in any monomial of f. Observe that if the ℓ-variate polynomial f is multilinear, then its total degree is at most ℓ. It is well-known that for any f, there is a unique multilinear polynomial $\widetilde{f}$ that extends f. The polynomial $\widetilde{f}$ is referred to the **multilinear extension (MLE)** of f.

A particular multilinear extension that arises frequently in the design of proof systems is $\widetilde{\mathrm{eq}}$, which is the MLE of the function $\mathrm{eq}\colon \{0,1\}^\ell \times \{0,1\}^\ell \to \mathbb{F}$ defined in Equation (1). An explicit expression for $\widetilde{\mathrm{eq}}$ is:

$$\widetilde{\mathrm{eq}}(x, e) = \prod_{i=1}^{\ell} \left(x_i e_i + (1 - x_i)(1 - e_i)\right). \tag{2}$$

Indeed, one can easily check that the right hand side of Equation (2) is a multilinear polynomial, and that if evaluated at any input $(x, e) \in \{0,1\}^\ell \times \{0,1\}^\ell$, it outputs 1 if $x = e$ and 0 otherwise. Hence, the right-hand side of Equation (2) is the unique multilinear polynomial extending eq. Equation (2) implies that $\widetilde{\text{eq}}(r_1, r_2)$ can be evaluated at any point $(r_1, r_2) \in \mathbb{F}^\ell \times \mathbb{F}^\ell$ in $\mathcal{O}(\ell)$ time. This polynomial is also called a **Lagrange basis polynomial** for ℓ-variate multilinear polynomial.

2.3 Sona: Multilinear Polynomial Commitment Scheme

Throughout this article walkthrough, the discussed protocols relied on oracle access to the polynomials generated by the prover. So, we began by specifying the required type of oracle access: a cryptographic commitment to multilinear polynomials. This allowed us to finally define the concrete scheme presented in [STW23], known as **Sona**. It builds on top of the Hyrax protocol from [Wah+18] and the Nova framework [KST22], introducing a simplified, non-zero-knowledge variant of Hyrax called BabyHyrax, which is then integrated into Nova's zero-knowledge setting.

Note that, in general, the Lasso protocol does not restrict the usage of a certain multilinear polynomial commitment scheme. For example, the implementation of Lasso in Jolt ZK-VM currently leverages a KZG-based polynomial commitment for multilinear polynomials. One may note that the selected commitment protocol affects the prover and verifier complexity directly.

2.3.1 BabyHyrax

We start by defining a non-zero-knowledge version of Hyrax multilinear polynomial commitment called BabyHyrax. In this section, we define the commitment itself and the protocol to evaluate a committed ℓ-variate multilinear polynomial at a random point $r \in \{0,1\}^\ell$ sampled by the verifier in a non-zero-knowledge environment.

Let's put $f\colon \{0,1\}^\ell \to \mathbb{F}$ an ℓ-variate multilinear polynomial. Instead of observing its coefficients, the commitment scheme works with the witness vector $t \in \mathbb{F}^{2^\ell}$, where $t_r = f(r)$ for each $r \in \{0,1\}^\ell$ (in other words – the ordered list representation of our multilinear polynomial). For simplicity, we put $2^\ell = m$. We start by restructuring the witness vector t of size m into a matrix T of size $\sqrt{m} \times \sqrt{m}$. Then, using the vector Pedersen commitment in group $\mathbb{G}$ with generators $g_0, \ldots, g_{\sqrt{m}-1}$, we commit to the rows of this matrix:

$$c_i = \prod_{j=0}^{\sqrt{m}-1} g_j^{T_{i,j}}$$

The resulting commitment will be a hash of vector c:

$$C_f = \mathsf{Hash}(c_0, \ldots, c_{\sqrt{m}-1})$$

Now, we can consider the opening protocol. First of all we redefine the polynomial f using the witness vector t for the query point $r \in \{0,1\}^\ell$, using the Lagrange basis polynomial:

$$f(r) = \sum_{k \in \{0,1\}^\ell} t_k \cdot \widetilde{\text{eq}}(k, r) = \sum_{k \in \{0,1\}^\ell} t_k \prod_{i=0}^{\ell-1} (k_i r_i + (1-k_i)(1-r_i))$$

The last product can be rewritten as follows:

$$\widetilde{\text{eq}}(k, r) = \prod_{i=0}^{\ell-1} (k_i r_i + (1-k_i)(1-r_i)) = \prod_{i=0}^{\ell/2-1} (k_i r_i + (1-k_i)(1-r_i)) \cdot \prod_{\ell/2}^{\ell-1} (k_i r_i + (1-k_i)(1-r_i))$$

Then, we define:

$$u_k = \prod_{i=0}^{\ell/2-1} (k_i r_i + (1-k_i)(1-r_i))$$
$$v_k = \prod_{\ell/2}^{\ell-1} (k_i r_i + (1-k_i)(1-r_i))$$

So, we can replace the $\widetilde{\text{eq}}$ by multiplication of u and v (where k_l and k_r define the left and right halves of k):

$$\widetilde{\text{eq}}(k, r) = u_{k_l} \cdot v_{k_r}$$

Thus, our original equation changes as follows:

$$f(r) = \sum_{k \in \{0,1\}^\ell} t_k \cdot \widetilde{\mathrm{eq}}(k, r) = \sum_{i \in [\sqrt{m}]} \sum_{j \in [\sqrt{m}]} T_{i,j} \cdot u_i \cdot v_j = \sum_{j \in [\sqrt{m}]} \left(\sum_{i \in [\sqrt{m}]} T_{i,j} \cdot u_i \right) \cdot v_j = \langle w, v \rangle$$

where $w = (w_0, \dots, w_{\sqrt{m}-1})$ and $w_j = \sum_{i \in [\sqrt{m}]} T_{i,j} \cdot u_i$.

Observe that for each $i \in [\sqrt{m}]$, a Pedersen commitment to w_i can be obtained by raising c_i to the exponent u_i. In particular,

$$c_{w_i} = c_i^{u_i} = \left(\prod_{j=0}^{\sqrt{m}-1} g_j^{T_{i,j}} \right)^{u_i} = \prod_{j=0}^{\sqrt{m}-1} g_j^{T_{i,j} \cdot u_i}$$

In the original Hyrax, the prover runs a Bulletproof protocol instance to prove the inner product $\langle w, v \rangle = f(r)$ using a commitment to w_i. In the BabyHyrax, instead, we send the w to the verifier as it is (assuming that the protocol does not need to be zero-knowledge or that it will be used in a zero-knowledge environment).

2.3.2 Sona

So, for an ℓ-variate multilinear polynomial $f\colon \{0,1\}^\ell \to \mathbb{F}$ the commitment and the evaluation proof consists of:

$$C_f = \mathsf{Hash}(c_0, \dots, c_{\sqrt{m}-1}) \in \mathbb{F}$$
$$w = (w_0, \dots, w_{\sqrt{m}-1}) \in \mathbb{F}^{\sqrt{m}}$$

The original Sona leverages the Nova backend [KST22] to prove that the following relation holds for the arbitrary point r and prover answer $v^\star = f(r)$:

$$\mathcal{R} = \left\{ \begin{array}{l} C_f = \mathsf{Hash}(c_0, \dots, c_{\sqrt{m}-1}) \\ \prod_{i \in [\sqrt{m}]} c_i^{u_i} = \prod_{j \in [\sqrt{m}]} g_j^{w_j} \\ v^\star = \langle w, v \rangle \end{array} \right\}$$

Following the definition of Sona commitment protocol, we can describe a bunch of functions that leverage an arbitrary collision-resistant hash function $\mathsf{Hash}\colon \mathbb{G}^{\sqrt{m}} \to \mathbb{F}$, a Nova prover ProveNova and a verifier VerifyNova functions, and generator points $g \in \mathbb{G}^{\sqrt{m}}$:

```
CommitSona($f\colon \{0,1\}^\ell \to \mathbb{F}$)
1: Put $m = 2^\ell$
2: Evaluate witness vector $t \in \mathbb{F}^m$ from $f$ and restructure it into the matrix T.
3: for $i \leftarrow 0$ to $\sqrt{m} - 1$ do
4:     $c_i \leftarrow \prod_{j=0}^{\sqrt{m}-1} g_j^{T_{i,j}} \in \mathbb{G}$
5: end for
6: $C_f \leftarrow \mathsf{Hash}(c_0, \dots, c_{\sqrt{m}-1})$
7: return $C_f$
```

```
OpenSona($C_f \in \mathbb{F}$, $r \in \{0,1\}^\ell$)
1: Put $m = 2^\ell$
2: For each $k \in \{0,1\}^{\ell/2}$ derive $u$ and $v$ according to BabyHyrax:

       $u_k = \prod_{i=0}^{\ell/2-1} (k_i r_i + (1 - k_i)(1 - r_i))$
```

```
$$v_k = \prod_{\ell/2}^{\ell-1} (k_i r_i + (1 - k_i)(1 - r_i))$$
3: // While $u$ and $v$ vectors seem to be calculated in $\mathcal{O}(2^{\ell/2})$ time, they consume only $\mathcal{O}(1)$ because of an obvious structure.
4: for $j \leftarrow 0$ to $\sqrt{m} - 1$ do
5:     // It is assumed that we can obtain a witness matrix $T$ from memory for the queried commitment $C_f$.
6:     $w_j \leftarrow \sum_{i=0}^{\sqrt{m}-1} T_{i,j}\, u_i$
7: end for
8: $v^\star \leftarrow \langle w, v \rangle$ // Remember that $\langle w, v \rangle = f(r)$
9: $\pi \leftarrow$ ProveNova$(c_0, \ldots, c_{\sqrt{m}-1}, T \mid C_f, r, u, v, v^\star)$
10: return $(v^\star, \pi)$
```

```
VerifySona($C_f \in \mathbb{F}$, $r \in \{0,1\}^\ell$, $v^\star$, $\pi$)
1: For each $k \in \{0,1\}^{\ell/2}$ derive $u$ and $v$ according to BabyHyrax:
$$u_k = \prod_{i=0}^{\ell/2-1} (k_i r_i + (1 - k_i)(1 - r_i))$$
$$v_k = \prod_{\ell/2}^{\ell-1} (k_i r_i + (1 - k_i)(1 - r_i))$$
2: // While $u$ and $v$ vectors seem to be calculated in $\mathcal{O}(2^{\ell/2})$ time, they consume only $\mathcal{O}(1)$ because of an obvious structure.
3: if VerifyNova$(C_f, r, u, v, v^\star, \pi) \neq$ True then
4:     $\mathcal{V}$ rejects
5: end if
```

Thus, the evaluation of an oracle access "open" function at some point consists of a combined running of OpenSona and VerifySona, which is preceded by a single run of CommitSona. Also, one can note that the protocol environment strictly fixes the degree of the polynomial (i.e., multilinear polynomial).

2.4 Sum-Check Protocol

A sum-check protocol [Lun+92] is an interactive proof protocol used in theoretical computer science to verify the correctness of a sum of values computed by a multivariate polynomial over a finite field, while having oracle access to the aforementioned polynomial. It allows a verifier to check a complex computation (like a huge sum over an exponential number of values) by interacting only a few times with a prover and doing logarithmic work. More precisely, let g be some ℓ-variate polynomial defined over a finite field $\mathbb{F}$. The purpose of the sum-check protocol is for $\mathcal{V}$ to make sure that the prover $\mathcal{P}$ has calculated the following sum correctly (the definition below is taken from [Tha17]):

$$H := \sum_{b \in \{0,1\}^\ell} g(b)$$

During the protocol walkthrough, we assume that $\mathcal{V}$ has oracle access to the polynomial $g \in \mathbb{F}^\ell[x]$. The protocol looks as follows:

1. At the beginning of the protocol, the prover sends a value C_1 claimed to equal the value H.

2. In the first round, $\mathcal{P}$ sends to $\mathcal{V}$ a univariate polynomial $g_1(X_1)$ with the following construction:

$$g_1(X_1) = \sum_{(x_2,\ldots,x_\ell) \in \{0,1\}^{\ell-1}} g(X_1, x_2, \ldots, x_\ell).$$

The sum in this polynomial is already computed by $\mathcal{P}$, so $\mathcal{V}$ can evaluate it with $\mathcal{O}(1)$ complexity. Then, $\mathcal{V}$ checks that $C_1 = g_1(0) + g_1(1)$ and that g_1 is a univariate polynomial of degree at most $\deg_1(g)$, rejecting if not.

3. $\mathcal{V}$ chooses a random element $r_1 \in \mathbb{F}$, and sends r_1 to $\mathcal{P}$.
4. In the jth round, for $1 < j < \ell$, $\mathcal{P}$ sends $\mathcal{V}$ a univariate polynomial $g_j(X_j)$ with the following construction:
$$g_j(X_j) = \sum_{(x_{j+1},\dots,x_\ell)\in\{0,1\}^{\ell-j}} g(r_1,\dots,r_{j-1},X_j,x_{j+1},\dots,x_\ell).$$
$\mathcal{V}$ 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.
5. $\mathcal{V}$ chooses a random element $r_j \in \mathbb{F}$, and sends r_j to $\mathcal{P}$.
6. In Round ℓ, $\mathcal{P}$ sends $\mathcal{V}$ a univariate polynomial $g_\ell(X_\ell) = g(r_1,\dots,r_{\ell-1},X_\ell)$. $\mathcal{V}$ checks that g_ℓ is a univariate polynomial of degree at most $\deg_l(g)$, rejecting if not, and also checks that $g_{\ell-1}(r_{\ell-1}) = g_\ell(0) + g_\ell(1)$.
7. $\mathcal{V}$ chooses a random element $r_\ell \in \mathbb{F}$ and evaluates $g(r_1,\dots,r_\ell)$ with a single oracle query to g. $\mathcal{V}$ checks that $g_l(r_\ell) = g(r_1,\dots,r_\ell)$, rejecting if not.
8. If $\mathcal{V}$ has not yet rejected, $\mathcal{V}$ halts and accepts.

While the definition of the protocol is universal, for future purposes, we need only a version that works with ℓ-variate multilinear polynomials and leverages the Sona commitment protocol as an oracle access provider:

```
Query($C_f \in \mathbb{F}, x\colon \{0,1\}^\ell$)
 1: // This procedure is executed by both $\mathcal{P}$ and $\mathcal{V}$
 2: $\mathcal{P}$ puts $(f(x),\pi) \leftarrow$ OpenSona($C_f$, $x$)
 3: $\mathcal{V}$ executes VerifySona($C_f$, $x$, $f(x)$, $\pi$)
 4: return $f(x)$
```

Finally, we describe the sum-check algorithm, executed by both prover $\mathcal{P}$ and verifier $\mathcal{V}$ for the polynomial g, its commitment C_g, and the sum H:

```
SumCheckProtocol($H \in \mathbb{F}, g\colon \{0,1\}^\ell \to \mathbb{F}$, $C_g \in \mathbb{F}$)
 1: Put $g_1(X_1) = \sum_{(x_2,\dots,x_\ell)\in\{0,1\}^{\ell-1}} g(X_1,x_2,\dots,x_\ell)$.
 2: $\mathcal{P}$ shares $g_1$ with $\mathcal{V}$.
 3: $\mathcal{V}$ evaluates $g_1(0)$ and $g_1(1)$
 4: if $H \neq g_1(0) + g_1(1)$ then
 5:     $\mathcal{V}$ rejects
 6: end if
 7: $\mathcal{V}$ chooses a random element $r_1 \in \mathbb{F}$, and evaluates $g_1(r_1)$
 8: for $j \leftarrow 2$ to $\ell$ do
 9:     Put $g_j(X_j) = \sum_{(x_{j+1},\dots,x_\ell)\in\{0,1\}^{\ell-j}} g(r_1,\dots,r_{j-1},X_j,x_{j+1},\dots,x_\ell)$
10:     $\mathcal{P}$ shares $g_j$ with $\mathcal{V}$.
11:     $\mathcal{V}$ evaluates $g_j(0)$ and $g_j(1)$
12:     if $g_{j-1}(r_{j-1}) \neq g_j(0) + g_j(1)$ then
13:         $\mathcal{V}$ rejects
14:     end if
15:     $\mathcal{V}$ chooses a random element $r_j \in \mathbb{F}$, and evaluates $g_j(r_j)$
16: end for
17: $\mathcal{V}$ puts $g(r_1,\dots,r_\ell) \leftarrow$ Query($C_g$, $(r_1,\dots,r_\ell)$)
18: if $g_\ell(r_\ell) \neq g(r_1,\dots,r_\ell)$ then
19:     $\mathcal{V}$ rejects
20: end if
```

Remark. Sometimes, g is not committed directly; instead, it is specified as a predefined combination of other committed polynomials. In this case, **we override the query/commit methods by committing/querying these polynomials and combining them according to the pre-defined structure.**

2.5 Sum-Check-based Protocol for Grand Products

Following [SL20], this subsection describes a transparent SNARK, which may be of independent interest, for proving grand product relations of the following form:

$$\mathcal{R}_{GP} = \{(P \in \mathbb{F}, V \in \mathbb{F}^m) \colon P = \prod_{i=0}^{m} V_i\}$$

Let $m = |V|$. WLOG, assume that m is a power of 2. Let V denote a table of evaluations of a $\log m$-variate multilinear polynomial $v(x)$ over $\{0,1\}^{\log m}$ in a natural fashion. We assume the prover first opens P and then commits to $v(x)$; the ensuing protocol then verifies both the polynomial's validity and the corresponding grand-product equality derived from that commitment.

Lemma 1. *Following [SL20]:*

$$P = \prod_{x \in \{0,1\}^{\log m}} v(x)$$

satisfied if and only if there exists a multilinear polynomial f in $\log m+1$ variables such that $f(1,\dots,1,0) = P$ and $\forall x \in \{0,1\}^{\log m}$ the following hold:

$$f(0,x) = v(x)$$
$$f(1,x) = f(x,0) \cdot f(x,1)$$

Such polynomial f has the following construction:

- $f(1,\dots,1) = 0$
- *For all $\ell \in [\log m]$ and $x \in \{0,1\}^{\log m - \ell}$: $f(1^\ell, 0, x) = \prod_{y \in \{0,1\}^\ell} v(x,y)$*

Then, to check that $\forall x \in \{0,1\}^{\log m}$ the equation $f(1,x) = f(x,0) \cdot f(x,1)$ holds, we can use a sum-check protocol to prove the evaluation of g that is referred to a MLE of $f(1,x) - f(x,0) \cdot f(x,1)$:

$$g(t) = \sum_{x \in \{0,1\}^{\log m}} \widetilde{\mathrm{eq}}(t,x) \cdot (f(1,x) - f(x,0) \cdot f(x,1))$$

By the Schwartz–Zippel lemma, except for a soundness error of $\frac{\log m}{|\mathbb{F}|}$ (which should be negligible), $g(\tau) = 0$ for τ uniformly random in $\mathbb{F}^{\log m}$ if and only if $g = 0$, which implies that $f(1,x) - f(x,0) \cdot f(x,1) = 0$ for all $x \in \{0,1\}^{\log m}$.

Similarly, to prove that $v(x) = f(0,x)$ for all $x \in \{0,1\}^{\log m}$ it suffices to prove that $v(\gamma) = f(0,\gamma)$ for a public coin $\gamma \in \mathbb{F}^{\log m}$.

Thus, to prove the existence of f and hence the grand product relationship, it suffices to prove, for some verifier selected random $\tau, \gamma \in \mathbb{F}^\ell$, that:

$$0 = \sum_{x \in \{0,1\}^{\log m}} \widetilde{\mathrm{eq}}(x,\tau) \cdot (f(1,x) - f(x,0) \cdot f(x,1)) \tag{3}$$

$$f(0,\gamma) = v(\gamma) \tag{4}$$

$$f(1,\dots,1,0) = P \tag{5}$$

This can be achieved by running the sum-check protocol between $\mathcal{P}$ and $\mathcal{V}$, where $\mathcal{V}$ has oracle access to v and f. Additionally, $\mathcal{V}$ evaluates the f, v at the random point γ to verify that $f(0,x) = v(x)$. So, the sum-check-based protocol for the grand products looks as follows:

```
GrandProductsProtocol($V \in \mathbb{F}^m, P \in \mathbb{F}$)
1: $\mathcal{P}$ compute polynomials $v \in \mathbb{F}^{\log m}[x], f \in \mathbb{F}^{\log m+1}[x]$ such that $P = \prod_{x\in\{0,1\}^{\log m}} v(x)$ and $v, f$ satisfies Equations (3), (4), (5).
2: $\mathcal{P}$ puts $C_f \leftarrow$ CommitSona($f$) and $C_v \leftarrow$ CommitSona($v$) and shares $C_f, C_v$ with $\mathcal{V}$.
3: $\mathcal{V}$ chooses a random elements $\tau, \gamma \in \mathbb{F}^{\log m}$ and sends $\tau, \gamma$ to $\mathcal{P}$.
4: $\mathcal{P}$ puts $g^*(x) = \widetilde{\text{eq}}(x, \tau) \cdot (f(1, x) - f(x, 0) \cdot f(x, 1))$
5: $\mathcal{P}$ and $\mathcal{V}$ run SumCheckProtocol($0, g^*, C_f$) // We assume that the instance of the sum-check protocol here operates over commitment to $f$ instead of the separate commitment to the input polynomial $g^*$
6: $\mathcal{V}$ puts $f(1, \dots 1, 0) \leftarrow$ Query($C_f$, $(1, \dots, 1, 0)$)
7: if $f(1, \dots 1, 0) \neq P$ then
8:     $\mathcal{V}$ rejects
9: end if
10: $\mathcal{V}$ puts $f(0, \gamma) \leftarrow$ Query($C_f$, $(0, \gamma)$) and $v(\gamma) \leftarrow$ Query($C_f$, $\gamma$)
11: if $f(0, \gamma) \neq v(\gamma)$ then
12:     $\mathcal{V}$ rejects
13: end if
```

We noted that during the call to the sum-check protocol, we pass the polynomial g^* in a couple with the commitment to f. Here we assume that each invocation of the Query method to g^* inside the sum-check instance will be more complex: **instead of evaluating a single function and returning result, we evaluate f at three different points, check the proofs and return results according to the construction of g^*.**

Also, note that the definition of f strictly depends on the nature of v (i.e, V). In the protocol that follows, we construct V from commitments to several polynomials; evaluations of f and v are therefore carried out through oracle access to those committed polynomials, removing the need for separate commitments to f and v. Since we no longer have to commit to the f, we do not need to build it for an arbitrary v – instead, we can utilize its known structure to evaluate at a random point during the sum-check protocol:

$$\forall \ell \in [\log m], \forall x \in \{0,1\}^{\log m - \ell} \colon f(1^\ell, 0, x) = \prod_{y \in \{0,1\}^\ell} v(x, y)$$

So, the protocol can be modified as follows for future usage:

1. We do not commit to v directly – instead, we override its query function by querying the evaluation of certain, committed polynomials and combining them according to the pre-defined structure.

2. We do not commit to f directly and prover do not build f directly – instead, we leverage the known structure of f, evaluating it at certain points by querying v at these points.

2.6 Offline Memory Checking

The goal of the memory-checking protocol is to enable a verifier to audit a huge, untrusted memory transcript using only a tiny proof: the prover records every read, write, and the verifier checks with a single grand-product that the multiset of reads equals the multiset of writes plus the initial image. If the equality holds, then each read has returned the most recently written value. Using this approach, the verifier can ensure that the committed sparse-matrix vectors representing the unit-row matrix are properly structured, without ever scanning the entire memory.

In the present version of this protocol, each operation is followed by an increment of the corresponding memory cell counter (unlike the original version, where the counter was global). The present protocol also utilizes only reads, pairing them with "synthetic" writes. The implementation of this protocol for Spark requires only three sets of tuples to work with:

- WS – contains tuples that represent write operations: $(\mathsf{addr}, \mathsf{val}, \mathsf{counter}_{\mathsf{addr}})$, where val is the value stored at the memory address addr after the specified counter;

- RS – contains tuples that represent read operations: $(\mathsf{addr}, \mathsf{val}, \mathsf{counter}_{\mathsf{addr}})$, where val is the value read from the memory address addr at the specified counter;

- S – contains tuples such that $\mathsf{WS} = \mathsf{RS} \cup \mathsf{S}$ if and only if all operations were executed correctly; otherwise, no such S exists that satisfies this equality.

You can think of a memory as a vector of values, and a counter as a memory cell timestamp, separately for read and write operations. At the beginning, WS is populated with the initial memory state, where all counters are set to 0, indicating that the values were first accessed during the initialization phase, while the RS, S are empty. Then, for each read operation, the untrusted memory is queried at addr, returning a pair $(\mathsf{val}, \mathsf{counter})$. The tuple $(\mathsf{addr}, \mathsf{val}, \mathsf{counter})$ is written in RS, and a corresponding write operation $(\mathsf{addr}, \mathsf{val}, \mathsf{counter} + 1)$ is added to WS. After all the read operations are done, one can notice that S equals the final memory state. The equality of multisets WS and $\mathsf{RS} \cup \mathsf{S}$ can be verified by a single grand-product check.

Claim 1 ([STW23] Randomized Permutation Check)**.** *Let A and B be two multisets of tuples in $\mathbb{F}^3$. Define*

$$h_\gamma(a, v, t) \;=\; a \cdot \gamma^2 \;+\; v \cdot \gamma \;+\; t, \qquad H_{\tau,\gamma}(A) \;=\; \prod_{(a,v,t)\in A} \bigl(h_\gamma(a, v, t) - \tau\bigr).$$

Then comparing $\mathcal{H}_{\tau,\gamma}(A)$ and $\mathcal{H}_{\tau,\gamma}(B)$ yields a randomized test for whether A and B are permutations of one another. Concretely: 1. If $A = B$ (as multisets), then

$$\mathcal{H}_{\tau,\gamma}(A) = \mathcal{H}_{\tau,\gamma}(B)$$

with probability 1 over uniform $\tau, \gamma \in \mathbb{F}$. 2. If $A \neq B$, then

$$\Pr\bigl[\mathcal{H}_{\tau,\gamma}(A) = \mathcal{H}_{\tau,\gamma}(B)\bigr] \;\le\; O\bigl((|A| + |B|)/|\mathbb{F}|\bigr).$$

In other words, by first "hashing" each tuple via the map function h_γ and then taking the τ-shifted product, one obtains a fingerprint that is invariant under permutation but unlikely to collide on distinct multisets.

Thus, $\mathsf{WS} = \mathsf{RS} \cup \mathsf{S}$ can be written as:

$$\prod_{(a,v,t)\in WS} \bigl((a\gamma^2 + v\gamma + t) - \tau\bigr) = \prod_{(a,v,t)\in RS} \bigl((a\gamma^2 + v\gamma + t) - \tau\bigr) \prod_{(a,v,t)\in S} \bigl((a\gamma^2 + v\gamma + t) - \tau\bigr),$$

$$\prod_{(a,v,t)\in WS} (h_\gamma(a, v, t) - \tau) = \prod_{(a,v,t)\in RS} (h_\gamma(a, v, t) - \tau) \prod_{(a,v,t)\in S} (h_\gamma(a, v, t) - \tau),$$

$$\mathcal{H}_{\tau,\gamma}(\mathsf{WS}) = \mathcal{H}_{\tau,\gamma}(\mathsf{RS})\mathcal{H}_{\tau,\gamma}(\mathsf{S})$$

Example ($N^{1/c} = 4,\ m = 3$)

We track a single *row-memory.* The algorithm performs three reads.

Address	0	1	2	3
Value	2	5	7	9

Initial memory (counter 0): $\mathsf{WS}_{\text{init}} = \{(0, 2, 0),\ (1, 5, 0),\ (2, 7, 0),\ (3, 9, 0)\}$, while $\mathsf{RS} = \varnothing$ and $\mathsf{S} = \varnothing$ are both empty.

Trace 1 – consistent execution

step	**operation**	$\Delta\mathsf{RS}_{\text{step}}$	$\Delta\mathsf{WS}_{\text{step}}$
1	read(1) $\to$ $(5, 0)$	$(1, 5, 0)$	$(1, 5, 1)$
2	read(0) $\to$ $(2, 0)$	$(0, 2, 0)$	$(0, 2, 1)$
3	read(2) $\to$ $(7, 0)$	$(2, 7, 0)$	$(2, 7, 1)$

After the three steps

$$\mathsf{RS} = \{(1, 5, 0),\ (0, 2, 0),\ (2, 7, 0)\},$$
$$\mathsf{WS} = \mathsf{WS}_{\text{init}} \cup \{(1, 5, 1),\ (0, 2, 1),\ (2, 7, 1)\}.$$

And we can fill S with

$$\mathsf{S} = \{(0, 2, 1),\ (1, 5, 1),\ (2, 7, 1),\ (3, 9, 0)\}$$

One can clearly see that $\mathsf{WS} = \mathsf{RS} \cup \mathsf{S}$.

$$\begin{aligned}\mathcal{H}_{\tau,\gamma}(\mathsf{WS}) &= \underbrace{(1\gamma^2 + 5\gamma + 0 - \tau)\,(0\gamma^2 + 2\gamma + 0 - \tau)\,(2\gamma^2 + 7\gamma + 0 - \tau)}_{\mathcal{H}_{\tau,\gamma}(\mathsf{RS})} \\ &\quad\times \underbrace{(0\gamma^2 + 2\gamma + 1 - \tau)\,(1\gamma^2 + 5\gamma + 1 - \tau)\,(2\gamma^2 + 7\gamma + 1 - \tau)\,(3\gamma^2 + 9\gamma + 0 - \tau)}_{\mathcal{H}_{\tau,\gamma}(\mathsf{S})} \\ &= \mathcal{H}_{\tau,\gamma}(\mathsf{RS}) \;\times\; \mathcal{H}_{\tau,\gamma}(\mathsf{S}).\end{aligned}$$

Trace 2 – inconsistent execution
Let the second reading return a wrong value – a, instead of 2. So, the new trace would look like:

step	**operation**	$\Delta\mathsf{RS}_{\text{step}}$	$\Delta\mathsf{WS}_{\text{step}}$
1	read(1) $\to$ $(5,0)$	$(1,5,0)$	$(1,5,1)$
2	read(0) $\to$ $(a,0)$	$(0,a,0)$	$(0,a,1)$
3	read(2) $\to$ $(7,0)$	$(2,7,0)$	$(2,7,1)$

After the three steps

$$\begin{aligned}\mathsf{RS} &= \{(1,5,0),\ (0,a,0),\ (2,7,0)\},\\ \mathsf{WS} &= \mathsf{WS}_{\text{init}} \cup \{(1,5,1),\ (0,a,1),\ (2,7,1)\},\\ \mathsf{S} &\to \text{ does not exist}\end{aligned}$$

Now, RS contains $(0,\ a,\ 0)$ while WS still contains the untouched initial tuple $(0,\ 2,\ 0)$. No multiset S that depends on the valid values and addresses can reconcile them, and the verifier *rejects.*

3 Spark Commitment Protocol

Spark is a sparse polynomial commitment scheme. It allows an untrusted prover to prove evaluations of a sparse multilinear polynomial with costs proportional to the size of the dense representation of the sparse multilinear polynomial.

It was originally introduced in the Spartan [Set19] as a tool for committing to R1CS matrices A, B, C, which are inherently **sparse** (it contains much more zero values then non-zero), effectively reducing the prover's commitment work by committing to m non-zero elements instead of the full matrix.

3.1 Multilinear Extension of Matrix

Firstly, we start from the MLE of a square matrix without any restrictions on the values inside. These assumptions will be changed in the next sections to fit the Lasso matrix format. Let $D \in \mathbb{F}^{M\times M}$ be a sparse matrix over the field $\mathbb{F}$. For convenience in some of the upcoming transitions, we can view it as a function $D(i,j)\colon \mathbb{F}^2 \to \mathbb{F}$. Let's define a polynomial that returns elements of our matrix by its indexes:

$$D(r_i, r_j) = \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} D(i,j) \cdot \mathrm{eq}(i, r_i) \cdot \mathrm{eq}(j, r_j)$$

Here, $\mathrm{eq}(a,b)\colon \mathbb{F}^2 \to \{0,1\}$ stands for the equality-check function that returns 1 if $a = b$, and 0 otherwise. These functions serve as Lagrange basis polynomials, ensuring that the expression evaluates to $D(i,j)$ at the point $(r_i, r_j) = (i,j)$, and zero elsewhere. Consequently, evaluating the sum, we receive the desired matrix value at the specified indices.

Utilizing the $\widetilde{\mathrm{eq}}$ introduced in the MLE subsection we can build MLE of D in the following way, assuming that $r_i, r_j \in \{0,1\}^{\log M}$:

$$\widetilde{D}(r_i, r_j) = \sum_{(i,j)\in\{0,1\}^{\log M}\times\{0,1\}^{\log M}} D(i,j) \cdot \widetilde{\mathrm{eq}}(i, r_i) \cdot \widetilde{\mathrm{eq}}(j, r_j)$$

As a drawback, we would need to go through all the M^2 cells with this option, which is pretty inefficient.

Since D is sparse, let us denote m as the number of non-zero entries in the matrix. These non-zero entries can be represented as an array of tuples: $T = \{(i_k, j_k, D(i_k, j_k)) : k = 0, 1, \ldots, m-1\}$.

It can also be represented as three polynomials $row, col, val : \{0,1\}^{\log(m)} \to \{0,1\}^{\log(M)}$ such that for each $k \in [m]$ a tuple $(row(k), col(k), val(k)) \in T$ identifies a valid non-zero element of our initial matrix D. With this structure, we can refactor our previous definition:

$$\widetilde{D}(r_i, r_j) = \sum_{k \in \{0,1\}^{\log m}} val(k) \cdot \widetilde{\mathrm{eq}}(row(k), r_i) \cdot \widetilde{\mathrm{eq}}(col(k), r_j)$$

The usage of such polynomials to represent our matrix describes the basic idea behind the Spark commitment protocol: *if the matrix contains a lot of zeros, we can replace commitment to the matrix with commitment to several polynomials that describe the positions of non-zero elements.*

3.2 Eliminating the Logarithmic Factor

The section above describes how one can build an m-sparse $\log M$-variate polynomial representation of a sparse matrix at a cost of its dense representation. This section describes a technique to eliminate the $\log m$ factor during the polynomial evaluation.

During the current iteration of the protocol description, we assume that the matrix we want to commit to has only one non-zero element per row equal to 1 (or, in other words, all rows are unit vectors). This simplification will allow us to avoid an unnecessary commitment to the *val* polynomial (we know its values at all interesting points). Then, each non-zero element of the $M \times N$ matrix can be represented as a pair of coordinates $(i, j) \in [M] \times [N]$.

3.2.1 The Idea

Naive evaluation. Let $L_i,\ i \in [M]$ be $\log N$-variate Lagrange-basis polynomials with, consequently, $\log N$ evaluation cost, and $r \in \{0,1\}^{\log N}$, then $L = L_1(r) + L_2(r) + \cdots + L_M(r)$ is a $\log N$-variate, M-sparse polynomial that can be evaluated with a total cost of $\mathcal{O}(M \log N)$.

Eliminating the logarithmic factor. The goal is to achieve $\mathcal{O}(c \cdot M)$ evaluation time by ensuring that each Lagrange basis polynomial can be evaluated in $\mathcal{O}(c)$. Let's decompose the $\log N = c \cdot \log m$ variables of r into c blocks, each of size $\log m$, writing $r = (r_1, \ldots, r_c) \in (\mathbb{F}^{\log m})^c$. Then any $\log N$-variate Lagrange basis polynomial evaluated at r can be expressed as a product of c "smaller" Lagrange basis polynomials, each defined over only $\log m$ variables, with the i'th such polynomial evaluated at r_i. There are only $2^{\log m} = m$ multilinear Lagrange basis polynomials over $\log m$ variables, so, having c inputs, one can evaluate all of them at each r_i in time $\mathcal{O}(cm)$, storing the result in a write-once memory. Now, given the memory, one can evaluate any given $\log N$-variate Lagrange basis polynomial at r by performing c lookups into the memory, one for each block r_i, and multiplying together the results, achieving $\mathcal{O}(M \cdot c)$ total time for all the M polynomials.

It's obvious, yet important to emphasize, that the choice of c uniquely determines m. While the parameter c needs to be chosen in a way to ensure optimal calculations.

3.2.2 Extending Matrix MLE

To apply the knowledge from the previous section, we need to encode a second coordinate of our $M \times N$ matrix as a vector in a c-dimensional hypercube, in effect decomposing a single $\log N$-variate polynomial into smaller ones. This hypercube will have c coordinates each of size $m = N^{\frac{1}{c}}$ giving us $(N^{\frac{1}{c}})^c = N$ points in total. So, each $j \in [N]$ will be represented as a vector $(j_0, \ldots j_{c-1})$, where $j_k \in [m]$. We also represent each hypercube coordinate as a binary vector of $\log m$ elements, so each $j \in [N]$ will be encoded as $(j_0, \ldots j_{c-1})$, where $j_k \in \{0,1\}^{\log m}$.

Now, we can define the $\log M$-variate multilinear polynomials to represent the non-zero matrix elements, where the first coordinate stands for the evaluation point and the second coordinate stands for the evaluation point image:

$$dim_1, \ldots, dim_c \colon \{0,1\}^{\log M} \to \{0,1\}^{\log m}, m = N^{\frac{1}{c}}$$

So, for each $k \in \{0,1\}^{\log M}$ (goes through the all row indexes) vector $(dim_1(k), \ldots, dim_c(k))$ represents an encoded column index (in the hypercube coordinates) of a non-zero element in the row k.

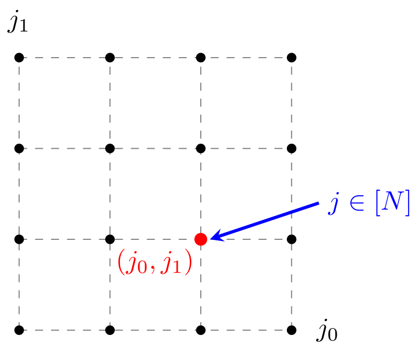


Figure 1: Decomposing a column index $j \in [N]$ into a 2-dimensional ($c = 2, m = N^{\frac{1}{2}}$) grid coordinate ($j_0 \in [m], j_1 \in [m]$)

Thus, let's put $r \in \{0,1\}^{\log M}$ as a row number, and $r' = (r'_1, \ldots, r'_c)$, where $r'_i \in \{0,1\}^{\log m}$, as an encoded column number, then:

$$\widetilde{D}(r, r') = \sum_{k \in \{0,1\}^{\log M}} \widetilde{\text{eq}}(r, k) \cdot \prod_{i=1}^{c} E_i(k) \tag{6}$$

where

$$\forall k \in \{0,1\}^{\log M} : E_i(k) = \widetilde{\text{eq}}(\text{dim}_i(k), r'_i)$$

Here, $\widetilde{\text{eq}}(r, k)$ stands for the MLE of the row-equality check function, as well as $\prod_{i=1}^{c} E_i(k)$ stands for the column-equality check function. We also remove the value multiplier because of our initial assumption – non-zero elements in our matrix can only equal 1.

Conclusion. This section describes an approach to encode the second matrix coordinate in the hypercube coordinates, which allows us to pre-calculate evaluations of small $\log m$-variate Lagrange basis polynomials $\widetilde{\text{eq}}$ in $E_i(k)$. The usage of the pre-calculated evaluations reduces the entire matrix MLE evaluation time by the $\log m$ factor. More precisely, using the naive approach, the evaluation of $\widetilde{D}(r, r')$ consists of the M evaluation of $\log N$-variate polynomials, which consumes $\mathcal{O}(M \log N)$ time. The usage of the described approach requires M evaluations of $c \log m$-variate polynomials, but if all $\log m$-variate polynomial evaluations can be pre-computed and queried from memory with $\mathcal{O}(1)$ complexity, the entire evaluation time will be $\mathcal{O}(M \cdot c)$.

3.3 Ensuring Consistency of Indicator Polynomials

Committing only to a dense matrix representation is a good performance boost, but it turns out that $dim_1, \ldots, dim_c$ and consequently, the indicator polynomials $E_i, i \in [c]$, can be easily made up to satisfy the equation. So, our goal now is to prove that the evaluation of E_i follows from the correct pre-computed evaluations of $\widetilde{\text{eq}}$ stored in both the prover's and verifier's memory and picked with correspondence to the dim_i. To address this issue, memory-checking techniques can be used to enforce consistency and correctness.

Remember $\forall k \in \{0,1\}^{\log M} : E_i(k) = \widetilde{\text{eq}}(dim_i(k), r'_i),\ \ i \in [c]$. Note that the proving of each E_i consistency will be performed separately. So, while utilizing the memory checking approach, we need to define and commit to the vector of memory `counter`-s where the verifier will have an oracle access to them. For each $i \in [c]$ we put M a number of read operations (obviously equals to the number of rows), and define the memory of size m that contains the evaluations of all $\widetilde{\text{eq}}(j, r'_i)$ for address j. So, the results of each $\widetilde{\text{eq}}(j, r'_i)$ as well as $E_i(j)$ are strictly fixed by the $dim_i(k) = j$. Additionally, we put

- $C_r \in \mathbb{F}^M$ – a vector of read-operation counters: $C_r[k]$ contains the count that would have been returned by the untrusted memory if it were honest during the k-th read operation
- $C_f \in \mathbb{F}^m$ – a vector of final memory counters: $C_f[j]$ stores the final count stored at memory location j of the untrusted memory (if the untrusted memory were honest) at the termination of the M read operations.

Let $read\_ts = \widetilde{C_r}$, $write\_ts = \widetilde{C_r} + 1$, $final\_cts = \widetilde{C_f}$ – we refer to these polynomials as counter polynomials, which are unique for a given memory and read operations. Note that $write\_ts$ can be eliminated, as it can be derived from $read\_ts$ with an increment. Therefore, we can build our multisets for the memory checking algorithm over dim_i, $i \in [c]$ as follows:

- $\mathsf{WS}_i = \{(\mathsf{to\text{-}field}(j), \widetilde{\mathrm{eq}}(j, r'_i), 0) \colon j \in \{0,1\}^{\log m}\} \cup \{(dim_i(k), E_i(k), read\_ts_i(k){+}1) \colon k \in \{0,1\}^{\log M}\}$. As mentioned before, we initialize WS_i with the initial memory state and zero counters (the first subset). Then, for each read operation, we put its result in a couple with its address and the incremented counter (the second subset).
- $\mathsf{RS}_i = \{(dim_i(k), E_i(k), read\_ts_i(k)) \colon k \in \{0,1\}^{\log M}\}$. During the evaluation of each $E_i(k)$, we read from the memory a value at $dim_i(k)$ address that should be equal to the $E_i(k)$ if the prover is honest. The reading counter strongly depends on the $dim_i(k)$ result.
- $\mathsf{S}_i = \{(\mathsf{to\text{-}field}(j), \widetilde{\mathrm{eq}}(j, r'_i), final\_cts_i(j)) \colon j \in \{0,1\}^{\log m}\}$. To equalize our subsets, we have to add to the reading set the final counters of each memory cell $j \in [m]$. Note that the memory cell value is fixed by the $\widetilde{\mathrm{eq}}(j, r'_i)$ result.

Finally, we can check the equality $\mathsf{WS}_i = \mathsf{RS}_i \cup \mathsf{S}_i$ by running the sum-check-based protocol for grand products over $\mathcal{H}_{\tau,\gamma}(\mathsf{WS}_i) = \mathcal{H}_{\tau,\gamma}(\mathsf{RS}_i) \cdot \mathcal{H}_{\tau,\gamma}(\mathsf{S}_i)$.

Grand Products Invocation. For example, the verifier receives $P = \mathcal{H}_{\tau,\gamma}(\mathsf{WS}_i)$ from prover and then both executes $\textsc{GrandProductsProtocol}(\{h_\gamma(a,v,t) - \tau\}|_{(a,v,t)\in\mathsf{WS}_i}, P)$. During the grand-products protocol, the verifier makes sure about the structure validity of the WS_i multiset by evaluating the f and v functions (see 2.5) at random points. As mentioned earlier (see 2.5), the f and v functions do not exist – instead, both the verifier and the prover know the structure of multiset WS_i, the evaluations of h_γ over its elements, and the resulting MLE of the input vector $\{h_\gamma(a,v,t) - \tau\}$. So, instead of querying f and v directly, the verifier executes query requests to the committed polynomials E_i, dim_i, $read\_ts_i$, and $final\_cts_i$, combining the results according to the multisets structure.

3.4 Spark Commitment Algorithm

Finally, we can define the Spark commitment protocol procedure. The prover runs it over the unit-row matrix of size $M \times N$, encoding each non-zero element by c polynomials $dim_1, \dots, dim_c$. Then, after committing to these polynomials, the prover provides the proof of the valid matrix decomposition, which includes both an evaluation of the matrix MLE at random coordinates and an instance of the memory check protocol to enable verification of the E_i polynomials' consistency. The evaluations of all $\widetilde{\mathrm{eq}}$ in E_i are precomputed, so they consume $\mathcal{O}(1)$ time. During the protocol, $\mathcal{V}$ has oracle access to all polynomials via a Sona commitment protocol (more precisely, they are: $dim_1, \dots, dim_c$ and $E_1, \dots, E_c$ and $read\_ts_1, \dots, read\_ts_c$ and $final\_cts_1, \dots, final\_cts_c$).

```
$\textsc{CommitSpark}(\mathsf{M}\colon \{0,1\}^{M\times N})$
 1: $\mathcal{P}$ executes $\textsc{CommitSona}$ for $dim_1, \dots, dim_c$ such that satisfy Equation (6) and shares the commitments with $\mathcal{V}$
 2: $\mathcal{V}$ randomly chooses $r \in [M]$ and $r' = (r'_1, \dots, r'_c)$, where $r'_k \in \{0,1\}^{\log m}$ and sends $r, r'$ to $\mathcal{P}$
 3: $\mathcal{P}$ executes $\textsc{CommitSona}$ for $E_1, \dots, E_c$ and $read\_ts_1, \dots, read\_ts_c$ and $final\_cts_1, \dots, final\_cts_c$ and shares the commitments with $\mathcal{V}$
 4: $\mathcal{P}$ opens to $\mathcal{V}$ the value $y = D(r, r')$
 5: $\mathcal{P}$ and $\mathcal{V}$ run $\textsc{SumCheckProtocol}\Big(y,\ \widetilde{\mathrm{eq}}(r,x) \cdot \prod_{i=1}^{c} E_i(x)\Big)$ // We assume that the instance of sum-check protocol here operates over commitments to $E_i$ instead of the separate commitment to the input polynomial
 6: $\mathcal{V}$ chooses a random $\tau, \gamma$ and sends them to $\mathcal{P}$
 7: for $i \in 1, \dots, c$ do
 8:     $\mathcal{P}$ builds $\mathsf{WS}_i, \mathsf{RS}_i$ and $\mathsf{S}_i$ multisets.
 9:     $\mathcal{P}$ sends $\mathcal{V}$ the evaluations of $\mathcal{H}_{\tau,\gamma}(\mathsf{WS}_i), \mathcal{H}_{\tau,\gamma}(\mathsf{RS}_i)$ and $\mathcal{H}_{\tau,\gamma}(\mathsf{S}_i)$
10:     $\mathcal{P}$ and $\mathcal{V}$ run $\textsc{GrandProductsProtocol}\Big(\{h_\gamma(a,v,t) - \tau\}\big|_{(a,v,t)\in\mathsf{WS}_i}, \mathcal{H}_{\tau,\gamma}(\mathsf{WS}_i)\Big)$
11:     $\mathcal{P}$ and $\mathcal{V}$ run $\textsc{GrandProductsProtocol}\Big(\{h_\gamma(a,v,t) - \tau\}\big|_{(a,v,t)\in\mathsf{RS}_i}, \mathcal{H}_{\tau,\gamma}(\mathsf{RS}_i)\Big)$
```

```
12:     $\mathcal{P}$ and $\mathcal{V}$ run GrandProductsProtocol$\Big(\{h_\gamma(a,v,t)-\tau\}\big|_{(a,v,t)\in \mathsf{S}_i}, \mathcal{H}_{\tau,\gamma}(\mathsf{S}_i)\Big)$
13:     if $\mathcal{H}_{\tau,\gamma}(\mathsf{WS}_i) \neq \mathcal{H}_{\tau,\gamma}(\mathsf{RS}_i)\cdot\mathcal{H}_{\tau,\gamma}(\mathsf{S}_i)$ then
14:         $\mathcal{V}$ rejects
15:     end if
16: end for
```

As mentioned before, the presented algorithm runs separate instances of the grand products protocol to check each E_i consistency. But, by grouping c multisets into a one multiset via a random linear combination, we can replace $c \times 3$ grand product executions by only 3 executions.

3.5 Example

Let's show how the Lasso with Spark works on the matrix of size 4×8. The vector t will contain the following values:

$$t = (91, 24, 13, 45, 41, 38, 27, 23)$$

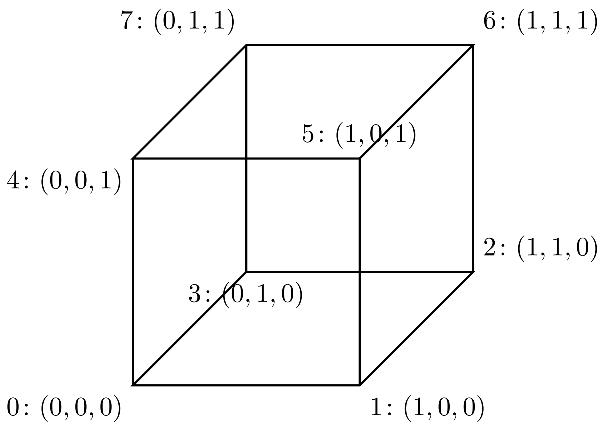
Let's put the $a = (91, 41, 91, 45)$. You can observe that the first value in it as well as the third appear at the first position in t, the second appears at the fifth position in t, and the last value appears at the fourth position. So, we can observe the following Lasso equation to be proven:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 91 \\ 24 \\ 13 \\ 45 \\ 41 \\ 38 \\ 27 \\ 23 \end{pmatrix} = \begin{pmatrix} 91 \\ 41 \\ 91 \\ 45 \end{pmatrix}$$

You can observe that the lookup matrix is sparse and unit-row, so we can encode using a polynomial that returns a column number by the row number:

$$\begin{aligned} col(0) &= 0 \\ col(1) &= 4 \\ col(2) &= 0 \\ col(3) &= 3 \end{aligned}$$

According to the Spark notation, we have: $N = 8$, $\log N = 3$ and $M = 4, \log M = 2$. Let's also put $c = 3$, then $m = N^{\frac{1}{c}} = 8^{\frac{1}{3}} = 2$. Thus, we are going to encode the column coordinate as a point at a 3-dimensional boolean hypercube of size 1. Let's order its points (the ordering can be defined by the proving system in any way, but it should be the same on both verifier's and prover's side):



Then, we can rewrite the results of our *col* polynomial as follows:

$$\begin{aligned} col(0) &= 0 = (0,0,0) \\ col(1) &= 4 = (0,0,1) \\ col(2) &= 0 = (0,0,0) \\ col(3) &= 3 = (0,1,0) \end{aligned}$$

Next, we initialize the $dim_i, i \in [c]$ polynomials to represent each coordinate of the hypercube point:

$$\begin{array}{lll} dim_1(0) = 0 & dim_2(0) = 0 & dim_3(0) = 0 \\ dim_1(1) = 0 & dim_2(1) = 0 & dim_3(1) = 1 \\ dim_1(2) = 0 & dim_2(2) = 0 & dim_3(2) = 0 \\ dim_1(3) = 0 & dim_2(3) = 1 & dim_3(3) = 0 \end{array}$$

After the prover commits to the dim_i polynomials, the verifier samples the challenge points r and r' for row and column. For example, let's take $r = 1, r' = 4$ (it can be any pair of coordinates, including the zero cells). More precisely, $r' = (0,0,1)$. Then, the prover and verifier initialize the memory with the evaluations of $E_i(k)$ at each $k \in \{0,1\}^{\log M} = \{0,1\}^2 = [4]$. For example, let's observe the memory check protocol for the $E_2 = \widetilde{\text{eq}}(dim_2(k), r'_2)$. Firstly, let's check all evaluations of E_2 (where $r'_2 = 0$):

$$\begin{aligned} E_2(0) &= \widetilde{\text{eq}}(dim_2(0), 0) = \widetilde{\text{eq}}(0,0) = 1 \\ E_2(1) &= \widetilde{\text{eq}}(dim_2(1), 0) = \widetilde{\text{eq}}(0,0) = 1 \\ E_2(2) &= \widetilde{\text{eq}}(dim_2(2), 0) = \widetilde{\text{eq}}(0,0) = 1 \\ E_2(3) &= \widetilde{\text{eq}}(dim_2(3), 0) = \widetilde{\text{eq}}(1,0) = 0 \end{aligned}$$

We have a memory of size $m = 2$ with addresses 0 and 1 that stores $\widetilde{\text{eq}}(j, r'_2)$ for $j \in [m] = [2]$. More precisely, the initial memory looks as follows:

Address	0	1
Value	1	0
Counter	0	0

Thus, we can fill the $\mathsf{WS}_2, \mathsf{RS}_2, \mathsf{S}_2$ vectors:

$$\mathsf{RS}_2 = \{(dim_2(k), E_2(k), \mathsf{read\_ts}_2(k)) \colon k \in \{0,1\}^{\log M}\} = \{(0,1,0), (0,1,1), (0,1,2), (1,0,0)\}$$

At the first read operation, we read $E_2(0) = 1$ at address $dim_2(0) = 0$: current memory counter is 0. At the second read operation, we read $E_2(1) = 1$ at address $dim_2(1) = 0$: current memory counter is 1. And so on...

$$\begin{aligned} \mathsf{WS}_2 = &\{(\mathsf{to\text{-}field}(j), \widetilde{\text{eq}}(j, r'_2), 0) \colon j \in \{0,1\}^{\log m}\} \cup \{(dim_2(k), E_2(k), \mathsf{read\_ts}_2(k) + 1) \colon k \in \{0,1\}^{\log M}\} = \\ &\{(0,1,0), (1,0,0)\} \cup \{(0,1,1), (0,1,2), (0,1,3), (1,0,1)\} \end{aligned}$$

Following the definition, we fill the write memory with the initial memory state (with zero counters) and the reading results (with incremented counters).

$$\mathsf{S}_2 = \{(\mathsf{to\text{-}field}(j), \widetilde{\text{eq}}(j, r'_2), \mathsf{final\_cts}_2(j)) \colon j \in \{0,1\}^{\log m}\} = \{(0,1,3), (1,0,1)\}$$

The final memory counter at address 0 equals 3, while at address 1 it equals 1 (check the construction of RS_2). One can easy observe that $\mathsf{WS}_2 = \mathsf{RS}_2 \cup \mathsf{S}_2$:

$$\begin{gathered} \{(0,1,0), (1,0,0)\} \cup \{(0,1,1), (0,1,2), (0,1,3), (1,0,1)\} = \\ \{(0,1,0), (0,1,1), (0,1,2), (1,0,0)\} \cup \{(0,1,3), (1,0,1)\} \end{gathered}$$

The same logic is applied to the E_1 and E_3 during the CommitSpark walkthrough.

4 Protocol Complexity

Sum-Check Complexity. Let's first consider the commitment protocol-agnostic version of the sum-check instance. During the protocol for an ℓ-variate multilinear polynomial, the prover pre-evaluates each of ℓ polynomials g_j with $\mathcal{O}(2^{\ell-j})$ complexity, and the verifier evaluates ℓ univariate polynomials g_j with $\mathcal{O}(1)$ complexity. One can observe that $\mathcal{O}(\sum_1^\ell 2^{\ell-j}) = \mathcal{O}(2^\ell)$. In the last stage of the protocol, the verifier evaluates an ℓ-variate multilinear polynomial at a random point corresponding to the one invocation of the evaluate and the verify functions of the commitment protocol. Thus, the total complexity of the algorithm is:

- Verifier: $\mathcal{O}(\ell + \mathsf{Com}_{\mathsf{Verify}})$
- Prover: $\mathcal{O}(2^\ell + \mathsf{Com}_{\mathsf{Eval}})$

Spark complexity. For the square matrix of size $M \times N$, the Spark protocol complexity then consists of the following parts (here we put $m = N^{\frac{1}{c}}$):

- Commitment to the polynomials (multiplied by factor c). Total: $\mathcal{O}(c \cdot \mathsf{Commit})$;
- The instance of the sum-check protocol (multiplied by factor c because of a complex input polynomial that depends on c different $\log M$-variate multilinear polynomials E_i). Total: $\mathcal{O}(cM + c \cdot \mathsf{Com}_{\mathsf{Eval}})$ for the prover and $\mathcal{O}(c \cdot \log M + c \cdot \mathsf{Com}_{\mathsf{Verify}})$ for the verifier;
- The evaluation of WS_i, RS_i and S_i multisets (multiplied by factor c for linear combination to reduce the number of grand-products invocations). Total: $\mathcal{O}(c(m + M))$ for prover;
- The instance of the grand-products protocol (with a complex function – multiplied by factor c). Complexity is mostly the same as for the sum-check instance.

Commit	$\mathcal{O}(c \cdot \mathsf{Commit})$
Prove	$\mathcal{O}(c \cdot M + c \cdot m + c \cdot \mathsf{Com}_{\mathsf{Eval}})$
Verify	$\mathcal{O}(c \cdot \log M + c \cdot \mathsf{Com}_{\mathsf{Verify}})$

5 Surge: a Generalization of Spark

In Spark we suppose to have an input vector $r \in [N]$ that we represent in a binary form $\{0,1\}^{\log N}$ and then split into the c smaller chunks receiving $(r_1, \ldots, r_c) \in (\{0,1\}^{\frac{\log N}{c}})^c$. We also decompose some $\log N$-variate polynomial t into c smaller $\frac{\log N}{c}$-variate polynomials and pre-compute all possible evaluations of t_i at r_i. Then we can evaluate a global result as follows

$$t(r) = \prod_{i=1}^{c} t_i(r_i)$$

As a final improvement to the Lasso lookup protocol, [STW23] presents **Surge**, which generalizes this approach by leveraging a "decomposable" polynomial T that can be split on α sub-polynomials, where $\alpha = k \cdot c$ for some natural k. In other words, we name T a "table" of size N and decompose this table on α "sub-tables" of size $N^{\frac{1}{c}}$, that we can store in both verifier's and prover's memory. We also assume the existence of some α-variate multilinear polynomial g such that following holds: for each $r \in \{0,1\}^{\log N}$ we write $r = (r_1, \ldots, r_c) \in (\{0,1\}^{\frac{\log N}{c}})^c$ and

$$T[r] = g(T_1[r_1], \ldots, T_k[r_1], T_{k+1}[r_2], \ldots, T_{2k}[r_2], \ldots, T_{\alpha-k+1}[r_c], \ldots, T_\alpha[r_c])$$

Note that this definition can be turned into the Spark case if we put $k = 1$ and $g = \prod_{i=1}^{c}$. Then, by following the general Lasso equation

$$\sum_{j \in \{0,1\}^{\log N}} \widetilde{D}(r,j) \cdot \widetilde{t}(j) = \widetilde{a}(r)$$

where j goes through all columns of $M \times N$ matrix, we can represent the left side of this equation as

$$\sum_{i \in \{0,1\}^{\log M}} \widetilde{\mathrm{eq}}(i,r) \cdot T[\mathsf{nz}(i)] = \widetilde{a}(r) \tag{7}$$

where $\mathsf{nz}(i)$ denotes a unique column in row i that stores a non-zero value (namely, 1). We suppose that T as well as nz is decomposable as described above, then we turn the previous equation into the following form:

$$\sum_{i \in \{0,1\}^{\log M}} \widetilde{\mathrm{eq}}(i, r) \cdot g^*(i) = \widetilde{a}(r)$$

where

$$g^*(i) = g(T_1[\mathsf{nz}_1(i)], \ldots, T_k[\mathsf{nz}_1(i)], T_{k+1}[\mathsf{nz}_2(i)], \ldots, T_{2k}[\mathsf{nz}_2(i)], \ldots, T_{\alpha-k+1}[\mathsf{nz}_c(i)], \ldots, T_\alpha[\mathsf{nz}_c(i)])$$

```
CommitSurge($\mathsf{M}\colon \{0,1\}^{M \times N}$)
 1: Both $\mathcal{P}$ and $\mathcal{V}$ stores $\alpha$ sub-tables $T_i$ each of size $N^{\frac{1}{c}}$ such that for any $r \in \{0,1\}^{\log N}$ the
    following holds:
        $T[r] = g(T_1[r_1], \ldots, T_k[r_1], T_{k+1}[r_2], \ldots, T_{2k}[r_2], \ldots, T_{\alpha-k+1}[r_c], \ldots, T_\alpha[r_c])$
 2: $\mathcal{P}$ executes CommitSona and shares the commitments to $c$ $\log m$-variate polynomials
    $dim_1, \ldots, dim_c$, where $dim_i$ is assumed to provide indexes to the sub-tables $T_{(i-1)k+1} \cdots T_{ik}$
    to satisfy the Equation (7) and it's decomposed form.
 3: $\mathcal{V}$ randomly chooses $r \in \{0,1\}^{\log M}$ and sends $r$ to $\mathcal{P}$.
 4: $\mathcal{P}$ executes CommitSona for $E_1, \ldots, E_\alpha$ and $read\_ts_1, \ldots, read\_ts_\alpha$ and
    $final\_cts_1, \ldots, final\_cts_\alpha$ and shares the commitments with $\mathcal{V}$.
 5: $\mathcal{P}$ opens to $\mathcal{V}$ the value $v = \sum_{k \in \{0,1\}^{\log M}} \widetilde{\mathrm{eq}}(r, k) \cdot g(E_1(k), \ldots, E_\alpha(k))$
 6: $\mathcal{P}$ and $\mathcal{V}$ run SumCheckProtocol$\Big(v,\ \widetilde{\mathrm{eq}}(r, x) \cdot g(E_1(x), \ldots, E_\alpha(x))\Big)$ // We assume that the
    instance of sum-check protocol here operates over commitments to $E_i$ instead of the separate
    commitment to the input polynomial
 7: $\mathcal{V}$ chooses a random $\tau, \gamma$ and sends them to $\mathcal{P}$
 8: for $i \in 1, \ldots, \alpha$ do
 9:     $\mathcal{P}$ builds $\mathsf{WS}_i, \mathsf{RS}_i$ and $\mathsf{S}_i$ multisets.
10:     $\mathcal{P}$ sends $\mathcal{V}$ the evaluations of $\mathcal{H}_{\tau,\gamma}(\mathsf{WS}_i), \mathcal{H}_{\tau,\gamma}(\mathsf{RS}_i)$ and $\mathcal{H}_{\tau,\gamma}(\mathsf{S}_i)$
11:     $\mathcal{P}$ and $\mathcal{V}$ run GrandProductsProtocol$\Big(\{h_\gamma(a,v,t) - \tau\}|_{(a,v,t) \in \mathsf{WS}_i},\ \mathcal{H}_{\tau,\gamma}(\mathsf{WS}_i)\Big)$
12:     $\mathcal{P}$ and $\mathcal{V}$ run GrandProductsProtocol$\Big(\{h_\gamma(a,v,t) - \tau\}|_{(a,v,t) \in \mathsf{RS}_i},\ \mathcal{H}_{\tau,\gamma}(\mathsf{RS}_i)\Big)$
13:     $\mathcal{P}$ and $\mathcal{V}$ run GrandProductsProtocol$\Big(\{h_\gamma(a,v,t) - \tau\}|_{(a,v,t) \in \mathsf{S}_i},\ \mathcal{H}_{\tau,\gamma}(\mathsf{S}_i)\Big)$
14:     if $\mathcal{H}_{\tau,\gamma}(\mathsf{WS}_i) \neq \mathcal{H}_{\tau,\gamma}(\mathsf{RS}_i) \cdot \mathcal{H}_{\tau,\gamma}(\mathsf{S}_i)$ then
15:         $\mathcal{V}$ rejects
16:     end if
17: end for
```

References

[Lun+92] Carsten Lund et al. "Algebraic Methods for Interactive Proof Systems". In: *Journal of the ACM* 39.4 (Oct. 1992), pp. 859–868. DOI: 10.1145/146585.146605.

[Tha17] Justin Thaler. *Sum-Check Protocol.* https://people.cs.georgetown.edu/jthaler/sumcheck.pdf. Lecture notes, COSC 544: Probabilistic Proof Systems. 2017.

[Wah+18] Riad S. Wahby et al. "Doubly-Efficient zkSNARKs Without Trusted Setup". In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P).* IEEE, 2018.

[Set19] Srinath Setty. *Spartan: Efficient and general-purpose zkSNARKs without trusted setup.* Cryptology ePrint Archive, Paper 2019/550. 2019. URL: https://eprint.iacr.org/2019/550.

[SL20] Srinath Setty and Jonathan Lee. *Quarks: Quadruple-efficient transparent zkSNARKs.* Cryptology ePrint Archive, Paper 2020/1275. 2020. URL: https://eprint.iacr.org/2020/1275.pdf.

[KST22] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. "Nova: Recursive Zero-Knowledge Arguments from Folding Schemes". In: *Proceedings of the International Cryptology Conference (CRYPTO).* Lecture Notes in Computer Science. Springer, 2022.

[Tha22] Justin Thaler. "Proofs, Arguments, and Zero-Knowledge". In: *Foundations and Trends in Privacy and Security* 4.2–4 (2022), pp. 117–660.

[STW23] Srinath Setty, Justin Thaler, and Riad Wahby. *Unlocking the lookup singularity with Lasso*. Cryptology ePrint Archive, Paper 2023/1216. 2023. URL: `https://eprint.iacr.org/2023/1216`.