Lasso

Unlocking the lookup singularity with Lasso

Srinath Setty*　　Justin Thaler†　　Riad Wahby‡

Fengrun Liu(刘冯润) 2023.9

Outline

- Relation in Lasso and Sparse-poly-commit
- KZG + Gemini: PCS for dense multilinear poly
- Spark: Spartan's sparse PCS
  - start from a simple case (c=2) to a general result
  - main tech: offline memory-checking [BEG+91]
  - finally, specialing the Spark to Lasso
- Surge: a generalization of Spark, providing Lasso
  - prover commits to an $m \times N$ matrix with each row is an unit vector (indeed commits to a sparse vector of size N with sparsity m)
  - establish the sparse vector's inner product with any dense, structured vector

LASSO-of-Truth

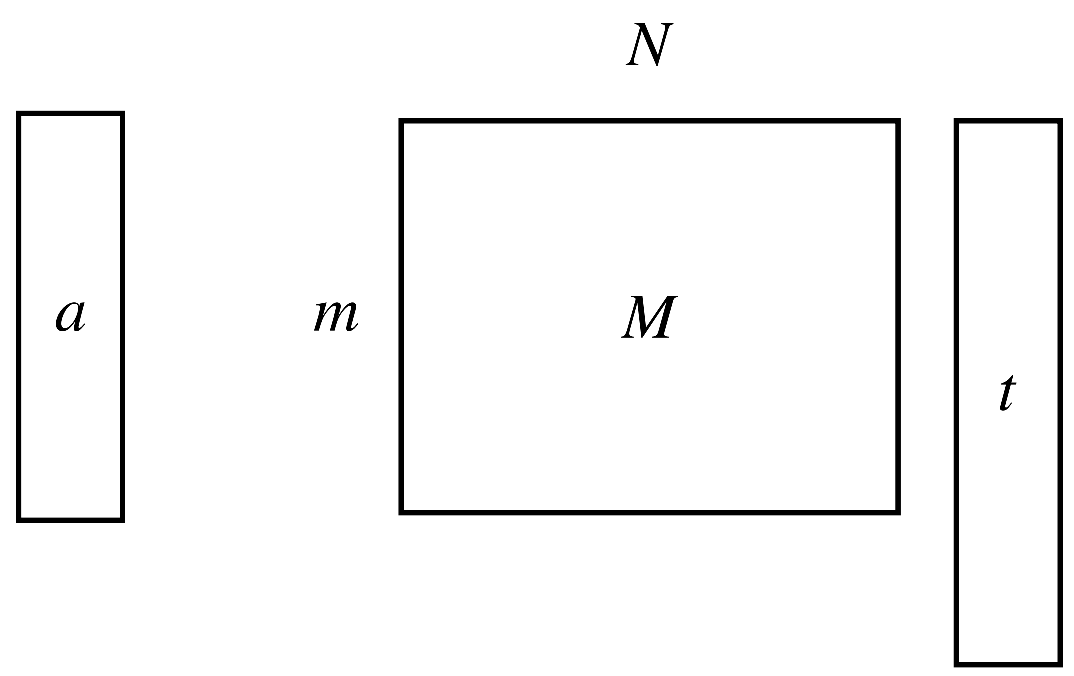
Lookup Arguments via Sparse-poly-commit and the Sum-check protocol, including for Oversized Tables

Reduce lookup to a matrix-vector multiplication with a sparse matrix.

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. A simple observation in prior works [ZBK+22, ZGK+22] is that the prover can prove that it knows a sparse matrix $M \in \mathbb{F}^{m \times n}$ such that for each row of M, only one cell has a value of 1 and the rest are zeros and that $M \cdot t = a$, where $\cdot$ is the matrix-vector multiplication. This turns out to be equivalent, up to negligible soundness error, to confirming that

$$\sum_{y \in \{0,1\}^{\log N}} \widetilde{M}(r, y) \cdot \widetilde{t}(y) = \widetilde{a}(r), \tag{5}$$

for an $r \in \mathbb{F}^{\log m}$ chosen at random by the verifier. Here, $\widetilde{M}$, $\widetilde{a}$ and $\widetilde{t}$ are the so-called *multilinear extension polynomials* (MLEs) of M, t, and a (see Section 2.1 for details).



1. **Commit to the sparse matrix M**
2. Reduced to a sum-check protocol
3. **Evaluation on a random point**

Sparse multilinear polynomial.

Definition 2.1. *A multilinear polynomial g in ℓ variables is a sparse multilinear polynomial if $|\mathsf{DenseRepr}(g)|$ is sub-linear in $O(2^\ell)$. Otherwise, it is a dense multilinear polynomial.*

As an example, suppose $g : \mathbb{F}^{2s} \to \mathbb{F}$. Suppose $|\mathsf{DenseRepr}(g)| = O(2^s)$, then g is a sparse multilinear polynomial because $O(2^s)$ is sublinear in $O(2^{2s})$.

PCS for dense multilinear poly

KZG-based PCS for multilinear poly

Costs for committing to a ℓ-variate multilinear polynomial

Scheme	Commit Size	Proof Size	$\mathcal{V}$ time	Commit time	$\mathcal{P}$ time
KZG + Gemini	$1\ \lvert\mathbb{G}_1\rvert$	$O(\log N)\ \lvert\mathbb{G}_1\rvert$	$O(\log N)\ \mathbb{G}_1$	$O(N)\ \mathbb{G}_1$	$O(N)\ \mathbb{G}_1$
Brakedown-commit	$1\ \lvert\mathbb{H}\rvert$	$O(\sqrt{N \cdot \lambda})\ \lvert\mathbb{F}\rvert$	$O(\sqrt{N \cdot \lambda})\ \mathbb{F}$	$O(N)\ \mathbb{F},\ \mathbb{H}$	$O(N)\ \mathbb{F},\ \mathbb{H}$
Orion-commit	$1\ \lvert\mathbb{H}\rvert$	$O(\lambda \log^2 N)\ \lvert\mathbb{H}\rvert$	$O(\lambda \log^2 N)\ \mathbb{H}$	$O(N)\ \mathbb{F},\ \mathbb{H}$	$O(N)\ \mathbb{F},\ \mathbb{H}$
Hyrax-commit	$O(\sqrt{N})\ \lvert\mathbb{G}\rvert$	$O(\sqrt{N})\ \lvert\mathbb{G}\rvert$	$O(\sqrt{N})\ \mathbb{G}$	$O(N)\ \mathbb{G}$	$O(N)\ \mathbb{F}$
Dory	$1\ \lvert\mathbb{G}_T\rvert$	$O(\log N)\ \lvert\mathbb{G}_T\rvert$	$O(\log N)\ \mathbb{G}_T$	$O(N)\ \mathbb{G}_1$	$O(N)\ \mathbb{F}$
Sona (this work)	$1\ \lvert\mathbb{H}\rvert$	$O(1)\ \lvert\mathbb{G}\rvert$	$O(\sqrt{N})\ \mathbb{G}$	$O(1)\ \mathbb{G}$	$O(N)\ \mathbb{F},\ O(\sqrt{N})\mathbb{G}$

Eval

Figure 1: Costs of polynomial commitment schemes when committing to a multilinear ℓ-variate polynomial over $\mathbb{F}$, with $\boxed{N = 2^\ell.}$ All are transparent. $\mathcal{P}$ time refers to the time to compute evaluation proofs. In addition to the reported $O(N)$ field operations, Hyrax and Dory require roughly $O(N^{1/2})$ cryptographic work to compute evaluation proofs. $\mathbb{F}$ refers to a finite field, $\mathbb{H}$ refers to a collision-resistant hash, $\mathbb{G}$ refers to a cryptographic group where DLOG is hard, and $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ refer to pairing-friendly groups. Columns with a suffix of "size" depict to the number of elements of a particular type, and columns with a suffix of "time" depict the number of operations (e.g., field multiplications or the size of multiexponentiations). Orion also requires $O(\sqrt{N})$ pre-processing time for the verifier.

PCS for dense multilinear poly

KZG-based PCS for multilinear poly

Scheme	Commit Size	Proof Size	$\mathcal{V}$ time	Commit time	$\mathcal{P}$ time
KZG + Gemini	$1\ \lvert\mathbb{G}_1\rvert$	$O(\log N)\ \lvert\mathbb{G}_1\rvert$	$O(\log N)\ \mathbb{G}_1$	$O(N)\ \mathbb{G}_1$	$O(N)\ \mathbb{G}_1$

Eval

The structured reference string (SRS) now consists of encodings in $\mathbb{G}$ of all powers of all Lagrange basis polynomials evaluated at a randomly chosen input $r \in \mathbb{F}^\ell$. That is, if $\chi_1, \dots, \chi_{2^\ell}$ denotes an enumeration of the 2^ℓ Lagrange basis polynomials, the SRS equals $(g^{\chi_1(r)}, \dots, g^{\chi_{2^\ell}(r)})$. Once again, the toxic waste that must be discarded because it can be used to destroy binding is the value r.

1. Transparent Setup with secret r

a similar commitment scheme for *multilinear* polynomials q over $\mathbb{F}_p$, proposed by Papamanthou, Shi, and Tamassia [PST13]. Let ℓ denote the number of variables of q, so $q: \mathbb{F}_p^\ell \to \mathbb{F}_p$. In applications of multilinear

As in the univariate commitment scheme, to commit to a multilinear polynomial q over $\mathbb{F}_p$, the committer sends a value c claimed to equal $g^{q(r)}$. Note that while the committer does not know r, it is still able to compute $g^{q(r)}$ using the SRS: if $q(X) = \sum_{i=0}^{2^\ell} c_i \chi_i(X)$, then $g^{q(r)} = \prod_{i=0}^{2^\ell} \left(g^{\chi_i(r)}\right)^{c_i}$, which can be computed given the values $g^{\chi_i(r)}$ for all $i = 0, \dots, 2^\ell$ even without knowing r.

2. Commit to q

- commit size: $O(1)$
- commit time: $O(N)$

To open the commitment at input $z \in \mathbb{F}_p^\ell$ to some value v, i.e., to prove that $q(z) = v$, the committer computes a series of ℓ "witness polynomials" $w_1, \dots, w_\ell$, defined in the following fact.

Fact 15.1 (Papamanthou, Shi, and Tamassia [PST13]). *For any fixed $z = (z_1, \dots, z_\ell) \in \mathbb{F}_p^\ell$ and any multilinear polynomial q, $q(z) = v$ if and only if there is a unique set of ℓ multilinear polynomials $w_1, \dots, w_\ell$ such that*

$$q(X) - v = \sum_{i=1}^{\ell} (X_i - z_i) w_i(X). \quad (15.4)$$

3. Evaluation on $q(z)$

1. compute ℓ multilinear polys $w_1, \dots, w_\ell$
2. commit to $w_1, \dots, w_\ell$ -> proof size $O(\ell)$
3. check the relation of exponent using pairing

$$e(c \cdot g^{-v}, g) = \prod_{i=1}^{\ell} e(y_i, g^{r_i} \cdot g^{-z_i}).$$

V time: $O(\ell)$

P time(1+2): $O(N)$ Zhang et al. [ZGK+] vRAM

refer to https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf

PCS for dense multilinear poly

Prover time for computing and committing with $O(N)$

Warning for notation change !! Now prover is required to computes $q_1, \ldots, q_\ell$ when evaluating multilinear poly $f(t_1, \ldots, t_\ell)$!

Recall that during Evaluate the prover computes polynomials $q_i(x_i,\ldots,x_\ell)$ for $i = 1,\ldots,\ell$, such that $f(x_1,\ldots,x_\ell) = \sum_{i=1}^{\ell}(x_i - t_i) \cdot q_i(x_i,\ldots,x_\ell) + f(t_1,\ldots,t_\ell)$ and proof $\pi = \{g^{q_i(s_i,\ldots,s_\ell)}, g^{\alpha q_i(s_i,\ldots,s_\ell)}\}_{i=1}^{\ell}$. We start by

Proof: $q_1(x_1, \ldots, x_\ell) = h(x_2, \ldots, x_\ell)$ with no monomial with x_1

$$f(x_1,\ldots,x_\ell) = g(x_2,\ldots,x_\ell) + x_1 \cdot h(x_2,\ldots,x_\ell)$$
$$= (g(x_2,\ldots,x_\ell) + t_1 \cdot h(x_2,\ldots,x_\ell)) + (x_1 - t_1)h(x_2,\ldots,x_\ell)$$
$$= R_1(x_2,\ldots,x_\ell) + (x_1 - t_1)h(x_2,\ldots,x_\ell)\,.$$

We set $q_1(x_1,\ldots,x_\ell) = h(x_2,\ldots,x_\ell)$ (which means q_1 contains no monomial with x_1), and proceed to decompose the multi-linear polynomial $R_1(x_2,\ldots,x_\ell)$ with $\ell - 1$ variables in the same way as f to compute $q_2(x_2,\ldots,x_\ell)$. Regarding the

Proof: It holds for $q_{i-1}(x_i, \ldots, x_\ell)$ for $i - 1 = 1,\ldots,\ell$.

and $g^{\alpha q_1(s_1,\ldots,s_\ell)}$ in the proof, respectively. The exact same reasoning applies for all of $q_3,\ldots,q_\ell$. At the last step after computing $q_\ell(x_\ell)$, the remaining constant term is equal to the answer $f(t_1,\ldots,t_\ell)$. In general, in the ith step, we are

1. compute q_i and R_i in $O(2^{\ell-i})$ —» $O(N)$ in total

Solve the following equations to compute multilinear R_1 and h:

$(x_2, \ldots, x_\ell$ range over $\{0,1\}^{\ell-1})$

$f(0,x_2, \ldots, x_\ell) = R_1(x_2, \ldots, x_\ell) + (0 - t_1)h(x_2, \ldots, x_\ell)$

$f(1,x_2, \ldots, x_\ell) = R_1(x_2, \ldots, x_\ell) + (1 - t_1)h(x_2, \ldots, x_\ell)$

...

2. commit to q_i in $O(2^{\ell-i})$ —» $O(N)$ in total

$q_1(x_1, \ldots, x_\ell) = h(x_2, \ldots, x_\ell)$

$$\sum_{i=1}^{2^\ell} c_i\chi_i(x_1, \ldots, x_\ell) = \sum_{i=1}^{2^{\ell-1}} 2c_i\chi_i(0,x_2, \ldots, x_\ell) = \sum_{i=1}^{2^{\ell-1}} 2c_i\chi_i(1,x_2, \ldots, x_\ell)$$

$$g^{q_1(r)} = \prod_{1}^{2^{\ell-1}} (g^{\chi_i(r)})^{2c_i}$$

refer to https://faculty.cc.gatech.edu/~genkin/papers/vram.pdf

Spark: Spartan's sparse PCS

Notations & Overview

Lasso's starting point is Spark, an optimal sparse polynomial commitment scheme from Spartan [Set20]. 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. Spartan established security of

Dense representation: specifies all multilinear Lagrange basis polys with non-zero coefficients

Dense representation for multilinear polynomials. Since the MLE of a function is unique, it offers the following method to represent any multilinear polynomial. Given a multilinear polynomial $g : \mathbb{F}^\ell \to \mathbb{F}$, it can be represented uniquely by the list of tuples L such that for all $i \in \{0,1\}^\ell$, $(\mathsf{to\text{-}field}(i), g(i)) \in L$ if and only if $g(i) \neq 0$, where to-field is the canonical injection from $\{0,1\}^\ell$ to $\mathbb{F}$. We denote such a representation of g as $\mathsf{DenseRepr}(g)$.

Lagrange basis polynomial

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

unique MLE for $x \in \mathbb{F}^s$

$$\widetilde{\mathsf{eq}}(x, e) = \prod_{i=1}^{s} (x_i e_i + (1 - x_i)(1 - e_i)) .$$

Notations:

N denotes the size of $\log N$-variate multilinear polynomial g.

m denotes the sparsity, then $g(x) = \sum_{i \in \{0,1\}^{\log N} : g(i) \neq 0} g(i)\tilde{eq}(i, x)$

Let c be such that $N = m^c$ (or $\log N = c \log m$)

$\log N$ variables is decomposed to c blocks, each of $\log m$.

Commitment: commit to a "dense" representation of the sparse polynomial.

Evaluation $g(r)$ of the committed polynomial g:

A naive solution. Consider an algorithm that iterates over each Lagrange basis polynomials specified in the committed dense representation, evaluates that basis polynomial at r, multiplies by the corresponding coefficient, and adds the result to the evaluation. Unfortunately, a naive evaluation of a $(\log N)$-variate Lagrange basis polynomial at r would take $O(\log N)$ time, resulting in a total runtime of $O(m \cdot \log N)$.

A naive solution: compute term-by-term

Spark: Spartan's sparse PCS

Notations & Overview

Evaluation $g(r)$ of the committed polynomial g in $O(c \cdot m)$.

Main idea: Represent the $\log N$-variate Lagrange basis polynomial at r as a product of c "smaller" Lagrange basis polynomials, each defined over $\log m$-variate. (Reminiscent of Pippenger's time-optimal algorithm for multiexponentiation)

Decompose the $\log N = c \cdot \log m$ variables of r into c blocks, each of size $\log m$, writing $\boxed{r = (r_1, \ldots, r_c)} \in \left(\mathbb{F}^{\log m}\right)^c$. 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

$$g(r) = \sum_{x \in \{0,1\}^{\log N} : g(x) \neq 0} g(x)\tilde{eq}(x, r) = \sum_{(x_1,\ldots,x_c) \in \{0,1\}^{c \log m} : g(x) \neq 0} g(x) \prod_{i=1}^{c} \tilde{eq}(x_i, r_i)$$

$\log N$ variables is decomposed to c blocks, each of $\log m$.

1. Evaluate c **write-once memory** M, each consisting m evaluations of $\tilde{eq}(x, r_i)$ for $x \in \{0,1\}^{\log m}$. ——> in $O(c \cdot m)$ total time.
2. Given all memory M, any $\log N$-variate Lagrange basis polynomial at r (i.e. $\tilde{eq}(x, r)$) can be evaluated by **performing c lookups into memory**, one for each r_i, and multiplying together the results. ——> in $O(c \cdot m)$ total time.

How the Spark prover proves it correctly ran the above time-optimal algorithm. To enable an untrusted prover to efficiently prove that it correctly ran the above algorithm to compute an evaluation of a sparse polynomial g at r, Spark uses *offline memory checking* [BEG+91] to prove read-write consistency.

General case: decompose $\log N$ variables to c blocks, each of $(\log N)/c$ variables.

1. Evaluate c memory of size $M = N^{1/c}$ in $c \cdot N^{1/c} = O(c \cdot m)$ time. (assuming $m \geq N^{1/c}$)
2. Given all memory, evaluate $g(r)$ by performing $c \cdot m$ lookups in $O(c \cdot m)$ time.

Spark: Spartan's sparse PCS

is adapted from an exposition of Spartan's result by Golovnev et al. [GLS+21]. It is natural for the reader to conceptualize the Spark sparse polynomial commitment scheme as a bespoke SNARK for a prover to prove it correctly ran the sparse ($\log N$)-variate multilinear polynomial evaluation algorithm described in Section 3.1 using c memories of size $N^{1/c}$.

A (slightly) simpler result: $c = 2$

Theorem 1 (Special case of Theorem 2 with $c = 2$). *Let $\mathsf{M} = N^{1/2}$. Given a polynomial commitment scheme for $(\log \mathsf{M})$-variate multilinear polynomials with the following parameters (where M is a positive integer and WLOG a power of 2):*

- *the size of the commitment is $\mathsf{c}(\mathsf{M})$;*
- *the running time of the commit algorithm is $\mathsf{tc}(\mathsf{M})$;*
- *the running time of the prover to prove a polynomial evaluation is $\mathsf{tp}(\mathsf{M})$;*
- *the running time of the verifier to verify a polynomial evaluation is $\mathsf{tv}(\mathsf{M})$;*
- *the proof size is $\mathsf{p}(\mathsf{M})$,*

PCS for dense multilinear polys (KZG extension)

there exists a polynomial commitment scheme for multilinear polynomials over $2\log \mathsf{M} = \log N$ variables that evaluate to a non-zero value at at most m locations over the Boolean hypercube $\{0,1\}^{2\log \mathsf{M}}$, with the following parameters:

- *the size of the commitment is $7\mathsf{c}(m) + 2\mathsf{c}(\mathsf{M})$;*
- *the running time of the commit algorithm is $O(\mathsf{tc}(m) + \mathsf{tc}(\mathsf{M}))$;*
- *the running time of the prover to prove a polynomial evaluation is $O(\mathsf{tp}(m) + \mathsf{tc}(\mathsf{M}))$;*
- *the running time of the verifier to verify a polynomial evaluation is $O(\mathsf{tv}(m) + \mathsf{tv}(\mathsf{M}))$; and*
- *the proof size is $O(\mathsf{p}(m) + \mathsf{p}(\mathsf{M}))$.*

PCS for a $\log N$-variate multilinear polynomial of sparsity m.
(decompose $\log N$ variables to $c = 2$ blocks)

Dominate costs for prover:
- committing to 7 dense multilinear polys over $\log m$-vars
- committing to 2 dense multilinear polys over $\log(N^{1/c})$-vars

As long as $m \geq N^{1/c}$,
prover time is linear in the sparsity of the committed poly.

Spark: Spartan's sparse PCS

The full result

For each memory checked, the prover has to commit to three multilinear polynomials defined over $\log(m)$-many variables, and one defined over $\log(\mathsf{M}) = \log(N)/c$ variables. We obtain the following theorem.

Theorem 2. *Given a polynomial commitment scheme for* $(\log \mathsf{M})$*-variate multilinear polynomials with the following parameters (where* M *is a positive integer and WLOG a power of 2):*

- *the size of the commitment is* $\mathsf{c}(\mathsf{M})$*;*
- *the running time of the commit algorithm is* $\mathsf{tc}(\mathsf{M})$*;*
- *the running time of the prover to prove a polynomial evaluation is* $\mathsf{tp}(\mathsf{M})$*;*
- *the running time of the verifier to verify a polynomial evaluation is* $\mathsf{tv}(\mathsf{M})$*;*
- *the proof size is* $\mathsf{p}(\mathsf{M})$*,*

there exists a polynomial commitment scheme for $(c \log \mathsf{M})$*-variate multilinear polynomials that evaluate to a non-zero value at at most* m *locations over the Boolean hypercube* $\{0,1\}^{c \log \mathsf{M}}$*, with the following parameters:*

- *the size of the commitment is* $\boxed{(3c+1)\mathsf{c}(m) + c \cdot \mathsf{c}(\mathsf{M});}$
- *the running time of the commit algorithm is* $O\left(c \cdot (\mathsf{tc}(m) + \mathsf{tc}(\mathsf{M}))\right)$*;*
- *the running time of the prover to prove a polynomial evaluation is* $O\left(c\left(\mathsf{tp}(m) + \mathsf{tc}(\mathsf{M})\right)\right)$*;*
- *the running time of the verifier to verify a polynomial evaluation is* $O\left(c\left(\mathsf{tv}(m) + \mathsf{tv}(\mathsf{M})\right)\right)$*;*
- *the proof size is* $O\left(c\left(\mathsf{p}(m) + \mathsf{p}(\mathsf{M})\right)\right)$*.*

Many polynomial commitment schemes have efficient batching properties for evaluation proofs. For such schemes, the factor c can be omitted in the final three bullet points of Theorem 2 (i.e., prover and verifier costs for verifying polynomial evaluation do not grow with c).

PCS for a $\log N$-variate polynomial of sparsity m, using c memories of size $M = N^{1/c}$.
(decompose $\log N$ variables to c blocks)

Dominate costs for prover: committing to

- $3c+1$dense multilinear polys over $\log m$-vars
- c dense multilinear polys over $\log(N^{1/c})$-vars

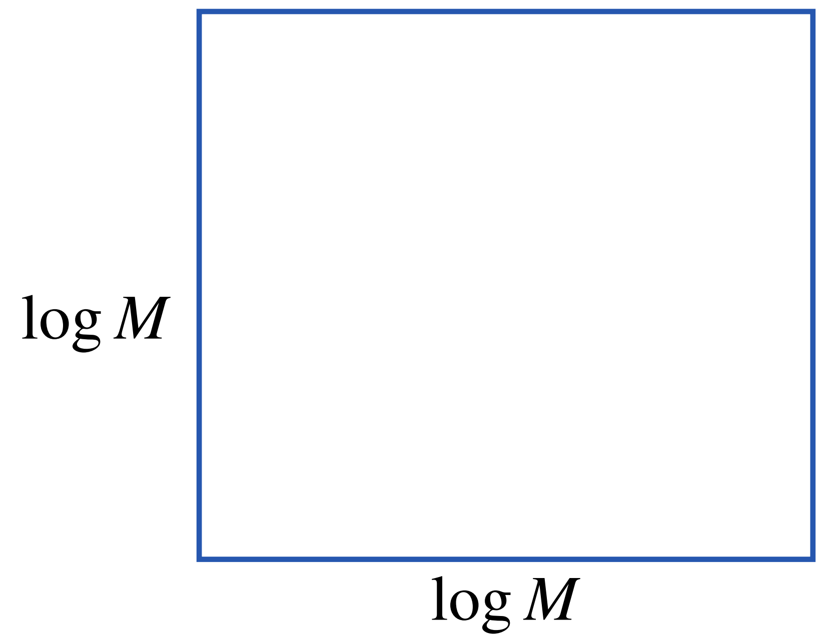
Spark: Spartan's sparse PCS

Special case ($c = 2$): detailed commit phase

Recall the notations:

- A $\log N$-variate multilinear polynomial of sparsity m, sub-linear to N.
- Decompose $\log N$ variables to c blocks.
- Evaluate c memories of size $M = N^{1/c}$. —> relation: $\log N = c \log M$

Original representation:

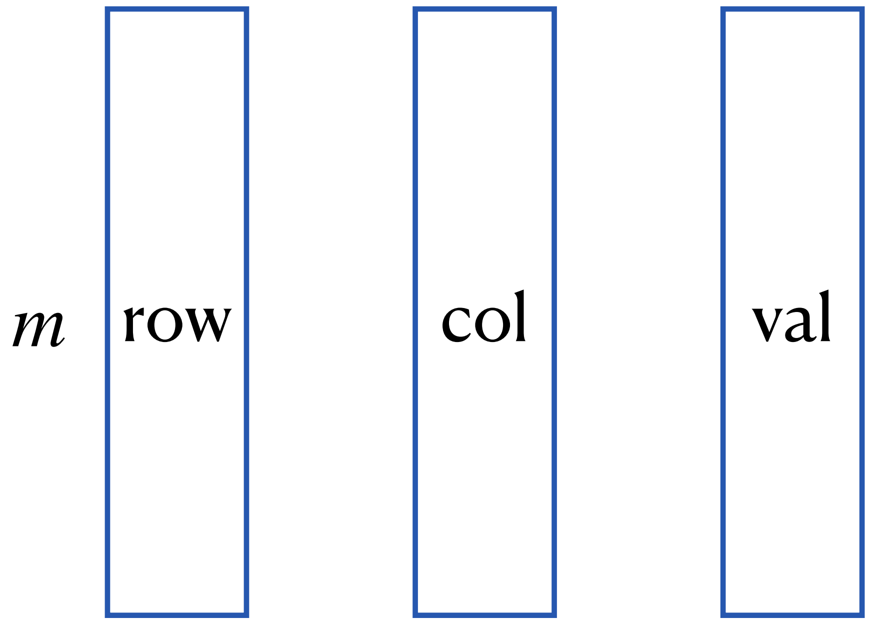


It represents a $\log N$-variate Lagrange basis polynomial at r as a product of $c = 2$ "smaller" Lagrange basis polynomials, each defined over $\log M$-variate.

Representing sparse polynomials with dense polynomials. Let D denote a $(2 \log \mathsf{M})$-variate multilinear polynomial that evaluates to a non-zero value at at most m locations over $\{0,1\}^{2\log \mathsf{M}}$. For any $r \in \mathbb{F}^{2\log \mathsf{M}}$, we can express the evaluation of $D(r)$ as follows. Interpret $r \in \mathbb{F}^{2\log \mathsf{M}}$ as a tuple (r_x, r_y) in a natural manner, where $r_x, r_y \in \mathbb{F}^{\log \mathsf{M}}$. Then by multilinear Lagrange interpolation (Lemma $\boxed{1}$), we can write

$$D(r_x, r_y) = \sum_{(i,j) \in \{0,1\}^{\log \mathsf{M}} \times \{0,1\}^{\log \mathsf{M}} :\ D(i,j) \neq 0} D(i,j) \cdot \boxed{\widetilde{eq}(i, r_x) \cdot \widetilde{eq}(j, r_y)}. \quad (7)$$

Dense representation:



Claim 1. *Let* to-field *be the canonical injection from* $\{0,1\}^{\log \mathsf{M}}$ *to* $\mathbb{F}$ *and* to-bits *be its inverse. Given a* $2\log \mathsf{M}$*-variate multilinear polynomial* D *that evaluates to a non-zero value at at most* m *locations over* $\{0,1\}^{2\log \mathsf{M}}$*, there exist* $\boxed{\text{three } (\log m)\text{-variate multilinear polynomials } \mathsf{row}, \mathsf{col}, \mathsf{val}}$ *such that the following holds for all* $r_x, r_y \in \mathbb{F}^{\log \mathsf{M}}$.

$$D(r_x, r_y) = \sum_{k \in \{0,1\}^{\log m}} \mathsf{val}(k) \cdot \widetilde{eq}(\boxed{\text{to-bits}(\mathsf{row}(k))}, r_x) \cdot \widetilde{eq}(\boxed{\text{to-bits}(\mathsf{col}(k)),} r_y). \quad (8)$$

Commit phase: commit to 3 $\log m$-variate polys

Commit costs: $O(m)$ field operations

Spark: Spartan's sparse PCS

Special case ($c = 2$): detailed evaluation phase

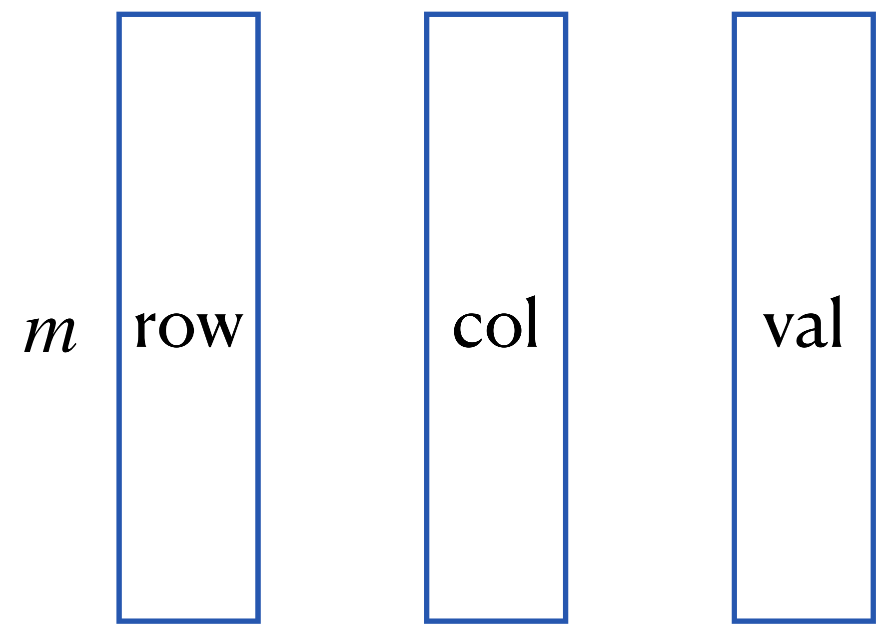
Claim 1. *Let* to-field *be the canonical injection from* $\{0,1\}^{\log \mathsf{M}}$ *to* $\mathbb{F}$ *and* to-bits *be its inverse. Given a* $2\log \mathsf{M}$*-variate multilinear polynomial* D *that evaluates to a non-zero value at at most* m *locations over* $\{0,1\}^{2\log \mathsf{M}}$*, there exist three* $(\log m)$*-variate multilinear polynomials* $\mathsf{row}, \mathsf{col}, \mathsf{val}$ *such that the following holds*

for all $r_x, r_y \in \mathbb{F}^{\log \mathsf{M}}$.

$$D(r_x, r_y) = \sum_{k \in \{0,1\}^{\log m}} \mathsf{val}(k) \cdot \widetilde{eq}(\boxed{\text{to-bits}(\mathsf{row}(k))}, r_x) \cdot \widetilde{eq}(\boxed{\text{to-bits}(\mathsf{col}(k)),} r_y). \qquad (8)$$

A first attempt at the evaluation phase. Given $r_x, r_y \in \mathbb{F}^{\log \mathsf{M}}$, to prove an evaluation of a committed polynomial, i.e., to prove that $D(r_x, r_y) = v$ for a purported evaluation $v \in \mathbb{F}$, consider the polynomial IOP in Figure 2, where the polynomial IOP assumes that the verifier has oracle access to the three $(\log m)$-variate multilinear polynomial oracles that encode D (namely $\mathsf{row}, \mathsf{col}, \mathsf{val}$).

Dense representation:



Commit phase: commit to 3 $\log m$-variate polys

Evaluation procedure to prove $D(r_x, r_y) = v$:

1. (Write) Evaluate $c = 2$ memory of size M.
   - $\tilde{eq}(i, r_x)$ as i ranged over $\{0,1\}^{\log M}$
   - $\tilde{eq}(j, r_y)$ as j ranged over $\{0,1\}^{\log M}$
2. (Read) Evaluate D at point $(r_x, r_y) \in \mathbb{F}^{2\log M}$ term-by-term with $c \cdot m$ lookups into memories.
   - Prover needs to sends the oracles E_{rx} and E_{ry}, thought as the purported multilinear extensions of the values returned by each memory.
   - If **prover is honest**, E_{rx} and E_{ry} are defined as follows.
   - But malicious prover may send arbitrary oracles.
   - As a result, verifier is required to additionally check the two conditions hold.

- $\forall k \in \{0,1\}^{\log m}$, $E_{\mathsf{rx}}(k) = \widetilde{eq}(\text{to-bits}(\mathsf{row}(k)), r_x)$; and
- $\forall k \in \{0,1\}^{\log m}$, $E_{\mathsf{ry}}(k) = \widetilde{eq}(\text{to-bits}(\mathsf{col}(k)), r_y)$.

Spark: Spartan's sparse PCS

Special case ($c = 2$): A first attempt at the evaluation phase

A first attempt at the evaluation phase. Given $r_x, r_y \in \mathbb{F}^{\log \mathsf{M}}$, to prove an evaluation of a committed polynomial, i.e., to prove that $D(r_x, r_y) = v$ for a purported evaluation $v \in \mathbb{F}$, consider the polynomial IOP in Figure 2, where the polynomial IOP assumes that the verifier has oracle access to the three $(\log m)$-variate multilinear polynomial oracles that encode D (namely $\mathsf{row}, \mathsf{col}, \mathsf{val}$).

1. $\mathcal{P} \to \mathcal{V}$: two $(\log m)$-variate multilinear polynomials E_{rx} and E_{ry} as oracles. These polynomials are purported to respectively equal the multilinear extensions of the functions mapping $k \in \{0,1\}^{\log m}$ to $\widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{row}(k)), r_x)$ and $\widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{col}(k)), r_y)$.
2. $\mathcal{V} \leftrightarrow \mathcal{P}$: run the sum-check reduction to reduce the check that

$$v = \sum_{k \in \{0,1\}^{\log m}} \mathsf{val}(k) \cdot E_{\mathsf{rx}}(k) \cdot E_{\mathsf{ry}}(k)$$

   to checking if the following hold, where $r_z \in \mathbb{F}^{\log m}$ is chosen at random by the verifier over the course of the sum-check protocol:
   - $\mathsf{val}(r_z) \stackrel{?}{=} v_{\mathsf{val}}$;
   - $E_{\mathsf{rx}}(r_z) \stackrel{?}{=} v_{E_{\mathsf{rx}}}$ and $E_{\mathsf{ry}}(r_z) \stackrel{?}{=} v_{E_{\mathsf{ry}}}$. Here, v_{val}, $v_{E_{\mathsf{rx}}}$, and $v_{E_{\mathsf{ry}}}$ are values provided by the prover at the end of the sum-check protocol.
3. $\mathcal{V}$: check if the three equalities hold with an oracle query to each of $\mathsf{val}, E_{\mathsf{rx}}, E_{\mathsf{ry}}$.

Figure 2: A first attempt at a polynomial IOP for revealing a requested evaluation of a $(2\log(\mathsf{M}))$-variate multilinear polynomial p over $\mathbb{F}$ such that $p(x) \neq 0$ for at most m values of $x \in \{0,1\}^{2\log(\mathsf{M})}$.

If **prover is honest,**
E_{rx} and E_{ry} are purported as follows:

- $\forall k \in \{0,1\}^{\log m}$, $E_{\mathsf{rx}}(k) = \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{row}(k)), r_x)$; and
- $\forall k \in \{0,1\}^{\log m}$, $E_{\mathsf{ry}}(k) = \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{col}(k)), r_y)$.

But malicious prover may send arbitrary oracles.

As a result, V is required to additionally check the two conditions hold.

Spartan [Set20]: check the two conditions using **memory-checking techniques [BEG+91]**

which confirms that every memory read over the course of an algorithm's execution returns the value last written to that location.

Spark: Spartan's sparse PCS

Offline memory-checking techniques [BEG+91]

Detour: Offline memory checking. Recall that in the offline memory checking algorithm of [BEG+91], a *trusted checker* issues operations to an untrusted memory. For our purposes, it suffices to consider only operation sequences in which each memory address is initialized to a certain value, and all subsequent operations are read operations. To enable efficient checking using multiset-fingerprinting techniques, the

Two operations for our purpose.

- initialized to a certain value
- read operations

enable checking with hash

operations are read operations. To enable efficient checking using multiset-fingerprinting techniques, the memory is modified so that in addition to storing a value at each address, the memory also stores a timestamp with each address. Moreover, each read operation is followed by a write operation that updates the timestamp associated with that address (but not the value stored there).

+ stores a **timestamp** with each address
+ modified read operations
  + followed by a write operation that updates the timestamp associated with that address

In prior descriptions of offline memory checking [BEG+91, CDD+03, SAGL18], the trusted checker maintains a single timestamp counter and uses it to compute write timestamps, whereas in Spark and our description below, the trusted checker does not use any local timestamp counter; rather, each memory cell maintains its own counter, which is incremented by the checker every time the cell is read.[15] For this reason, we depart from the standard terminology in the memory-checking literature and henceforth refer to these quantities as *counters* rather than timestamps.

In Spark and [this work]

+ each memory cell maintains a **counter**
+ modified read operations
  + followed by a write operation where the **counter is incremented**

Spark: Spartan's sparse PCS

Offline memory-checking techniques [BEG+91]

Goal:

A trusted checker issues operations to an untrusted memory (provided by prover).

- **Prover** executes an algorithm with purported functions, which are indeed read operations into memory.
- **Verifier** is convinced that every memory read over the course of an algorithm's execution returns the value last written to that location.

In Spark and [this work]

+ each memory cell maintains a **counter**
+ modified read operations
  + followed by a write operation where the **counter is incremented**

Local state of the checker: Two sets: RS and WS, which are initialized as follows.[16] $\boxed{\mathsf{RS} = \{\},}$ and for an $\boxed{\text{M-sized memory,}}$ $\boxed{\text{WS is initialized}}$ to the following set of tuples: for all $i \in [N^{1/c}]$, the tuple $(i, v_i, 0)$ is included in WS, where v_i is the value stored at address i, and the third entry in the tuple, 0, is an "initial count" associated with the value (intuitively capturing the notion that when v_i was written to address i, it was the first time that address was accessed). Here, [M] denotes the set $\{0, 1, \ldots, \mathsf{M}-1\}$.

- Untrusted M-sized memory: each cell stores a value-count pair (v, t) where t is initialized to 0.

- Modified read operation: (recorded by the local state of the checker)
  1. checker queries a read operation at address a. (RS)
  2. the untrusted memory responds with a value-count pair (v, t) (value is responded via the purported oracle E_{rx} an E_{ry})
  3. the untrusted memory increment the counter at address a (WS)

1. $\mathsf{RS} \leftarrow \mathsf{RS} \cup \{(a, v, t)\}$;
2. store $(v, t+1)$ at address a in the untrusted memory; and
3. $\mathsf{WS} \leftarrow \mathsf{WS} \cup \{(a, v, t+1)\}$.

Spark: Spartan's sparse PCS

Offline memory-checking techniques [BEG+91]

Goal:
A trusted checker issues operations to an untrusted memory (provided by prover).
- **Prover** executes an algorithm with purported functions, which are indeed read operations into memory.
- **Verifier** is convinced that every memory read over the course of an algorithm's execution returns the value last written to that location.

- Untrusted M-sized memory: each cell stores a value-count pair (v, t) where t is initialized to 0.
- Modified read operation: (recorded by the local state of the checker)
  1. checker queries a read operation at address a. (RS)
  2. the untrusted memory responds with a value-count pair (v, t)
     (value is responded via the purported oracle E_{rx} an E_{ry})
  3. the untrusted memory increment the counter at address a (WS)

Initialization: RS={} and WS={$(i, v_i, 0)$|for all $i \in [M]$}

1. $\mathsf{RS} \leftarrow \mathsf{RS} \cup \{(a, v, t)\}$;
2. store $(v, t+1)$ at address a in the untrusted memory; and
3. $\mathsf{WS} \leftarrow \mathsf{WS} \cup \{(a, v, t+1)\}$.

Invariant maintained on the sets of the checker.

Claim 2. *Let $\mathbb{F}$ be a prime order field. Assuming that the domain of counts is $\mathbb{F}$ and that m (the number of reads issued) is smaller than the field characteristic $|\mathbb{F}|$. Let WS and RS denote the multisets maintained by the checker in the above algorithm at the conclusion of m read operations. If for every read operation, the untrusted memory returns the tuple last written to that location, then there exists a set S with cardinality M consisting of tuples of the form (k, v_k, t_k) for all $k \in [\mathsf{M}]$ such that $\mathsf{WS} = \mathsf{RS} \cup S$. Moreover, S is computable in time linear in M.*

Conversely, if the untrusted memory ever returns a value v for a memory call $k \in [\mathsf{M}]$ such v does not equal the value initially written to cell k, then there does not exist any set S such that $\mathsf{WS} = \mathsf{RS} \cup S$.

Prove two directions:

exist a set S with cardinality M such that $WS = RS \cup S$

dose not exist any set with cardinality M such that $WS = RS \cup S$

Spark: Spartan's sparse PCS

Offline memory-checking techniques [BEG+91]

Invariant maintained on the sets of the checker.

Claim 2. *Let $\mathbb{F}$ be a prime order field. Assuming that the domain of counts is $\mathbb{F}$ and that m (the number of reads issued) is smaller than the field characteristic $|\mathbb{F}|$. Let* WS *and* RS *denote the multisets maintained by the checker in the above algorithm at the conclusion of m read operations. If for every read operation, the untrusted memory returns the tuple last written to that location, then there exists a set S with cardinality* M *consisting of tuples of the form (k, v_k, t_k) for all $k \in [\mathsf{M}]$ such that $\mathsf{WS} = \mathsf{RS} \cup S$. Moreover, S is computable in time linear in* M.

Conversely, if the untrusted memory ever returns a value v for a memory call $k \in [\mathsf{M}]$ such v does not equal the value initially written to cell k, then there does not exist any set S such that $\mathsf{WS} = \mathsf{RS} \cup S$.

Prove two directions:

exist a set S with cardinality M such that $WS = RS \cup S$

dose not exist any set with cardinality M such that $WS = RS \cup S$

Proof. If for every read operation, the untrusted memory returns the tuple last written to that location, then it is easy to see the existence of the desired set S. It is simply the current state of the untrusted memory viewed as the set of address-value-count tuples.

Prove the converse direction by contradiction:

We now prove the other direction in the claim. For notational convenience, let WS_i and RS_i $(0 \leq i \leq m)$ denote the multisets maintained by the trusted checker at the conclusion of the ith read operation (i.e., WS_0 and RS_0 denote the multisets before any read operation is issued). Suppose that there is some read operation

Notation

and RS_0 denote the multisets before any read operation is issued). Suppose that there is some read operation i that reads from address k, and the untrusted memory responds with a tuple (v, t) such that v differs from the value initially written to address k. This ensures that $(k, v, t) \in \mathsf{RS}_j$ for all $j \geq i$, and in particular that $(k, v, t) \in \mathsf{RS}$, where recall that RS is the read set at the conclusion of the m read operations. Hence, to

By contradiction:

We have $(k, v, t) \in RS_j$ for all $j \geq i$

and $(k, v, t) \in RS$

Spark: Spartan's sparse PCS

Offline memory-checking techniques [BEG+91]

Prove the converse direction by contradiction:

We now prove the other direction in the claim. For notational convenience, let $\boxed{\mathsf{WS}_i \text{ and } \mathsf{RS}_i\ (0 \le i \le m)}$ denote the multisets maintained by the trusted checker at the conclusion of the ith read operation (i.e., WS_0 and RS_0 denote the multisets before any read operation is issued). Suppose that there is some read operation

Notation

and RS_0 denote the multisets before any read operation is issued). Suppose that there is some read operation i that reads from address k, and the untrusted memory responds with a tuple (v, t) such that v differs from the value initially written to address k. This ensures that $(k, v, t) \in \mathsf{RS}_j$ for all $j \ge i$, and in particular that $(k, v, t) \in \mathsf{RS}$, where recall that RS is the read set at the conclusion of the m read operations. Hence, to

By contradiction:

We have $(k, v, t) \in RS_j$ for all $j \ge i$

and $(k, v, t) \in RS$

Initialization phase for two **multisets**: RS={ } and WS={ $(i, v_i, 0)$ | for all $i \in [M]$ }

1. $\mathsf{RS} \leftarrow \mathsf{RS} \cup \{(a, v, t)\}$;
2. store $(v, t+1)$ at address a in the untrusted memory; and
3. $\mathsf{WS} \leftarrow \mathsf{WS} \cup \{(a, v, t+1)\}$.

To construct a set S such that $RS \cup S = WS$, we need to ensure $RS \subseteq WS$.
——» Then S is the difference set.

Assumption : $(k, v, t) \in RS$ where v differs from the value initially written to address k.

We want to ensure $(k, v, t) \in WS$**:**

- But outside of the initialization phase, WS is only updated with (k, v, t) by a read operation to address k, which returns $(v, t-1)$.
- Accordingly, we want to ensure $(k, v, t-i) \in WS$ for $i = 1, \ldots, \mathrm{char}(\mathbb{F})$.
- But there are only $m < \mathrm{char}(\mathbb{F})$ many read operations.

Contradiction !!!

Claim 2. *Let $\mathbb{F}$ be a prime order field. Assuming that the domain of counts is $\mathbb{F}$ and that m (the number of reads issued) is smaller than the field characteristic $|\mathbb{F}|$. Let* WS *and* RS *denote the multisets maintained by*

Spark: Spartan's sparse PCS

Offline memory-checking techniques [BEG+91]

Invariant maintained on the sets of the checker.

Claim 2. *Let $\mathbb{F}$ be a prime order field. Assuming that the domain of counts is $\mathbb{F}$ and that m (the number of reads issued) $\boxed{\text{is smaller than the field characteristic } |\mathbb{F}|}$. Let WS and RS denote the multisets maintained by the checker in the above algorithm at the conclusion of m read operations. If for every read operation, the untrusted memory returns the tuple last written to that location, then there exists a set S with cardinality M consisting of tuples of the form (k, v_k, t_k) for all $k \in [\mathsf{M}]$ such that $\mathsf{WS} = \mathsf{RS} \cup S$. Moreover, S is computable in time linear in M.*

Conversely, if the untrusted memory ever returns a value v for a memory call $k \in [\mathsf{M}]$ such v does not equal the value initially written to cell k, then there does not exist any set S such that $\mathsf{WS} = \mathsf{RS} \cup S$.

Prove two directions:

exist a set S with cardinality M such that $WS = RS \cup S$

dose not exist any set with cardinality M such that $WS = RS \cup S$

Characteristic: https://en.wikipedia.org/wiki/Characteristic_(algebra)

In mathematics, the **characteristic** of a ring R, often denoted $\mathrm{char}(R)$, is defined to be the smallest number of times one must use the ring's multiplicative identity (1) in a sum to get the additive identity (0). If this sum never reaches the additive identity the ring is said to have characteristic zero.

That is, $\mathrm{char}(R)$ is the smallest positive number n such that:[1](p 198, Thm. 23.14)

$$\underbrace{1 + \cdots + 1}_{n \text{ summands}} = 0$$

if such a number n exists, and 0 otherwise.

FACTS about characteristic of fields:

- The characteristic of any field is either 0 or a prime number.
- The finite field $GF(p^n)$ has characteristic p.

Spark: Spartan's sparse PCS

Offline memory-checking techniques [BEG+91]

Invariant maintained on the sets of the checker.

Claim 2. *Let $\mathbb{F}$ be a prime order field. Assuming that the domain of counts is $\mathbb{F}$ and that m (the number of reads issued) is smaller than the field characteristic $|\mathbb{F}|$. Let* WS *and* RS *denote the multisets maintained by the checker in the above algorithm at the conclusion of m read operations. If for every read operation, the untrusted memory returns the tuple last written to that location, then there exists a set S with cardinality* M *consisting of tuples of the form (k, v_k, t_k) for all $k \in [\mathsf{M}]$ such that $\mathsf{WS} = \mathsf{RS} \cup S$. Moreover, S is computable in time linear in* M.

Conversely, if the untrusted memory ever returns a value v for a memory call $k \in [\mathsf{M}]$ such v does not equal the value initially written to cell k, then there does not exist any set S such that $\mathsf{WS} = \mathsf{RS} \cup S$.

Prove two directions:

exist a set S with cardinality M such that $WS = RS \cup S$

dose not exist any set with cardinality M such that $WS = RS \cup S$

Remark: Claim 2 applies as long as $|\mathbb{F}| > m$ to work over fields of smaller characteristic

for all $i \in 1, \ldots, char(\mathbb{F})$). We can nonetheless work over fields of smaller characteristic by modifying the procedure by which the checker updates the counts returned by each read operation. Specifically, rather than initializing counts to 0 and replacing a count t returned by a read operation with $t+1$, we instead initialize the counts to 1, and replace a returned count t with $t \cdot g$, where g is a fixed generator of the multiplicative group of the field $\mathbb{F}$. With this modification, Claim 2 applies so long as $|\mathbb{F}| > m$.

FACTS about characteristic of fields:

- The characteristic of any field is either 0 or a prime number.
- The finite field $GF(p^n)$ has characteristic p.

Remark: Addition to the value, Claim 2 holds for the indices as well to avoid a read to "invalid" memory.

Remark 3. *The proof of Claim 2 implies that, if the checker ever performs a read to an "invalid" memory cell k, meaning a cell indexed by $k \notin [\mathsf{M}]$, then regardless of the value and timestamp returned by the untrusted prover in response to that read, there does not exist any set S such that $\mathsf{WS} = \mathsf{RS} \cup S$.*

Spark: Spartan's sparse PCS

Special case ($c = 2$): back to the evaluation phase

Evaluation procedure to prove $D(r_x, r_y) = v$:

1. (Write) Evaluate $c = 2$ memory of size M.
   - $\tilde{eq}(i, r_x)$ as i ranged over $\{0,1\}^{\log M}$
   - $\tilde{eq}(j, r_y)$ as j ranged over $\{0,1\}^{\log M}$
2. (Read) Evaluate D at point $(r_x, r_y) \in \mathbb{F}^{2\log M}$ term-by-term with $c \cdot m$ lookups into memories.
   - Prover needs to sends the oracles E_{rx} and E_{ry}, thought as the purported multilinear extensions of the values returned by each memory.
   - If **prover is honest**, E_{rx} and E_{ry} are defined as follows.
   - But malicious prover may send arbitrary oracles.
   - As a result, verifier is required to additionally check the two conditions hold.

$$D(r_x, r_y) = \sum_{k \in \{0,1\}^{\log m}} \mathsf{val}(k) \cdot \tilde{eq}(\text{to-bits}(\mathsf{row}(k)), r_x) \cdot \tilde{eq}(\text{to-bits}(\mathsf{col}(k)), r_y).$$

- $\forall k \in \{0,1\}^{\log m}$, $E_{\mathsf{rx}}(k) = \tilde{eq}(\text{to-bits}(\mathsf{row}(k)), r_x)$; and
- $\forall k \in \{0,1\}^{\log m}$, $E_{\mathsf{ry}}(k) = \tilde{eq}(\text{to-bits}(\mathsf{col}(k)), r_y)$.

Reduced to prove the multi-set equality, i.e. $RS \cup S = WS$, with aid of **counter polynomials.**

Observe that given the size M of memory and a list of m addresses involved in read operations, one can compute two vectors $C_r \in \mathbb{F}^m, C_f \in \mathbb{F}^{\mathsf{M}}$ defined as follows. For $k \in [m]$, $C_r[k]$ stores the count that would have been returned by the untrusted memory if it were honest during the kth read operation. Similarly, for $j \in [\mathsf{M}]$, let $C_f[j]$ store 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. Computing these three vectors requires computation comparable to $O(m)$ operations over $\mathbb{F}$.

Computation costs: $O(m)$

It includes a final "read pass" over the memory.
That's why we refer to it as "offline" memory-checking.

Given the M-sized memory and m read operations, prover computes two vectors $C_r \in \mathbb{F}^m$ and $C_f \in \mathbb{F}^M$ in $O(m)$.

- $C_r[k]$: the count returned by the untrusted memory during kth read operation.
- $C_f[j]$: final count stored at memory location j after m read operations.

Spark: Spartan's sparse PCS

Special case ($c = 2$): Reduce evaluation to proof of multi-set equality

Reduced to prove the multi-set equality, i.e. $RS \cup S = WS$, with aid of **counter polynomials.**

Observe that given the size M of memory and a list of m addresses involved in read operations, one can compute two vectors $\boxed{C_r \in \mathbb{F}^m, C_f \in \mathbb{F}^{\mathsf{M}}}$ defined as follows. For $k \in [m]$, $C_r[k]$ stores the count that would have been returned by the untrusted memory if it were honest during the kth read operation. Similarly, for $j \in [\mathsf{M}]$, let $C_f[j]$ store 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. Computing these three vectors requires computation comparable to $O(m)$ operations over $\mathbb{F}$.

Given the M-sized memory and m read operations, prover computes two vectors $C_r \in \mathbb{F}^m$ and $C_f \in \mathbb{F}^M$ in $O(m)$.

- $C_r[k]$: the count returned by the untrusted memory during kth read operation.
- $C_f[j]$: final count stored at memory location j after m read operations.

Counter Polynomials

Let $\mathsf{read\_ts} = \widetilde{C_r}, \mathsf{write\_cts} = \widetilde{C_r} + 1, \mathsf{final\_cts} = \widetilde{C_f}$. We refer to these polynomials as *counter polynomials*, which are unique for a given memory size M and a list of m addresses involved in read operations.

Commit to counter polynomials

The actual evaluation proof. To prove the evaluation of a given a $(2\log \mathsf{M})$-variate multilinear polynomial D that evaluates to a non-zero value at at most m locations over $\{0,1\}^{2\log \mathsf{M}}$, the prover sends the following polynomials in addition to $\boxed{E_{\mathsf{rx}} \text{ and } E_{\mathsf{ry}}}$: two $(\log m)$-variate multilinear polynomials as oracles $(\mathsf{read\_ts}_{\mathsf{row}}, \mathsf{read\_ts}_{\mathsf{col}})$, and two $(\log \mathsf{M})$-variate multilinear polynomials $(\mathsf{final\_cts}_{\mathsf{row}}, \mathsf{final\_cts}_{\mathsf{col}})$, where $(\mathsf{read\_ts}_{\mathsf{row}}, \mathsf{final\_cts}_{\mathsf{row}})$ and $(\mathsf{read\_ts}_{\mathsf{col}}, \mathsf{final\_cts}_{\mathsf{col}})$ are respectively the counter polynomials for the m addresses specified by row and col over a memory of size M.

$(3c+1)\mathsf{c}(m) + c \cdot \mathsf{c}(\mathsf{M});$

Recall the commitment:

- c(m) for log m-variate
  - 1 for val
  - for each memory checked (decompose log N to c blocks)
    - row
    - E_{rx} for evaluation
    - read_ts
- c(M) for log M-variate
  - for each memory
    - final_cts

Spark: Spartan's sparse PCS

Special case ($c = 2$): Reduce evaluation to proof of multi-set equality

Claim 3. *Given a* $(2\log \mathsf{M})$*-variate multilinear polynomial, suppose that* $(\mathsf{row}, \mathsf{col}, \mathsf{val})$ *denote multilinear polynomials committed by the commit algorithm. Furthermore, suppose that*

$$(E_{\mathsf{rx}}, E_{\mathsf{ry}}, \mathsf{read\_ts}_{\mathsf{row}}, \mathsf{final\_cts}_{\mathsf{row}}, \mathsf{read\_ts}_{\mathsf{col}}, \mathsf{final\_cts}_{\mathsf{col}})$$

denote the additional polynomials sent by the prover at the beginning of the evaluation proof.

Prove the condition holds. ——> Prove the multi-set equality via these committed polynomials.

For any $r_x \in \mathbb{F}^{\log \mathsf{M}}$, *suppose that*

$$\forall k \in \{0,1\}^{\log m},\ \ E_{\mathsf{rx}}(k) = \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{row}(k)), r_x). \tag{9}$$

Then the following holds: $\mathsf{WS} = \mathsf{RS} \cup S$, *where*

- $\mathsf{WS} = \{(\mathsf{to\text{-}field}(i), \widetilde{eq}(i, r_x), 0) \colon i \in \{0,1\}^{\log(\mathsf{M})}\} \cup \{(\mathsf{row}(k), E_{\mathsf{rx}}(k), \mathsf{write\_cts}_{\mathsf{row}}(k) = \mathsf{read\_ts}_{\mathsf{row}}(k) + 1) \colon k \in \{0,1\}^{\log m}\}$;
- $\mathsf{RS} = \{(\mathsf{row}(k), E_{\mathsf{rx}}(k), \mathsf{read\_ts}_{\mathsf{row}}(k)) \colon k \in \{0,1\}^{\log m}\}$; *and*
- $S = \{(\mathsf{to\text{-}field}(i), \widetilde{eq}(i, r_x), \mathsf{final\_cts}_{\mathsf{row}}(i)) \colon i \in \{0,1\}^{\log(\mathsf{M})}\}$.

Meanwhile, if Equation (9) does not hold, then there is no set S *such that* $\mathsf{WS} = \mathsf{RS} \cup S$, *where* WS *and* RS *are defined as above.*

Similarly, it holds for another condition.
And the proof is the application of Claim 2.

4 sum-check-based protocols for grand products:
(can be computed in parallel)

- 2 are over vectors of size M
- 2 are over vectors of size m

Subtlety for Remark 3

Here, we clarify the following subtlety. The expression $\mathsf{to\text{-}bits}(\mathsf{row}(k))$ appearing in Equation (9) is not defined if $\mathsf{row}(k)$ is outside of $[\mathsf{M}]$ for any $k \in \{0,1\}^{\log m}$. But in this event, Remark 3 nonetheless implies the conclusion of the theorem, namely that there is no set S such that $\mathsf{WS} = \mathsf{RS} \cup S$. The analogous conclusion holds by the same reasoning if $\mathsf{col}(k)$ is outside of $[\mathsf{M}]$ for any $k \in \{0,1\}^{\log m}$. □

Spark: Spartan's sparse PCS

Special case ($c = 2$): Reduce evaluation to proof of multi-set equality

Reduced to grand products with hashing.

Claim 4 ([Set20]). *Given two multisets A, B where each element is from $\mathbb{F}^3$, checking that $A = B$ is equivalent to checking the following, except for a soundness error of $O(|A| + |B|)/|\mathbb{F}|)$ over the choice of γ, τ: $\mathcal{H}_{\tau,\gamma}(A) = \mathcal{H}_{\tau,\gamma}(B)$, where $\mathcal{H}_{\tau,\gamma}(A) = \prod_{(a,v,t)\in A}(h_\gamma(a,v,t) - \tau)$, and $h_\gamma(a,v,t) = a\cdot\gamma^2 + v\cdot\gamma + t$. That is, if $A = B$, $\mathcal{H}_{\tau,\gamma}(A) = \mathcal{H}_{\tau,\gamma}(B)$ with probability 1 over randomly chosen values τ and γ in $\mathbb{F}$, while if $A \neq B$, then $\mathcal{H}_{\tau,\gamma}(A) = \mathcal{H}_{\tau,\gamma}(B)$ with probability at most $O(|A| + |B|)/|\mathbb{F}|)$.*

//During the commit phase, $\mathcal{P}$ has committed to three $(\log m)$-variate multilinear polynomials $\mathsf{row}, \mathsf{col}, \mathsf{val}$.

1. $\mathcal{P} \to \mathcal{V}$: four $(\log m)$-variate multilinear polynomials $E_{\mathsf{rx}}, E_{\mathsf{ry}}, \mathsf{read\_ts}_{\mathsf{row}}, \mathsf{read\_ts}_{\mathsf{col}}$ and two $(\log \mathsf{M})$-variate multilinear polynomials $\mathsf{final\_cts}_{\mathsf{row}}, \mathsf{final\_cts}_{\mathsf{col}}$.
2. Recall that Claim 1 (see Equation (8)) shows that $D(r_x, r_y) = \sum_{k\in\{0,1\}^{\log m}} \mathsf{val}(k)\cdot E_{\mathsf{rx}}(k)\cdot E_{\mathsf{ry}}(k)$ assuming that
   - $\forall k \in \{0,1\}^{\log m}$, $E_{\mathsf{rx}}(k) = \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{row}(k)), r_x)$; and
   - $\forall k \in \{0,1\}^{\log m}$, $E_{\mathsf{ry}}(k) = \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{col}(k)), r_y)$.

   Hence, $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to the polynomial $\mathsf{val}(k)\cdot E_{\mathsf{rx}}(k)\cdot E_{\mathsf{ry}}(k)$, which reduces the check that $\boxed{v = \sum_{k\in\{0,1\}^{\log m}} \mathsf{val}(k)\cdot E_{\mathsf{rx}}(k)\cdot E_{\mathsf{ry}}(k)}$ to checking that the following equations hold, where $r_z \in \mathbb{F}^{\log m}$ chosen at random by the verifier over the course of the sum-check protocol:
   - $\mathsf{val}(r_z) \stackrel{?}{=} v_{\mathsf{val}}$; and
   - $E_{\mathsf{rx}}(r_z) \stackrel{?}{=} v_{E_{\mathsf{rx}}}$ and $E_{\mathsf{ry}}(r_z) \stackrel{?}{=} v_{E_{\mathsf{ry}}}$. Here, v_{val}, $v_{E_{\mathsf{rx}}}$ and $v_{E_{\mathsf{ry}}}$ are values provided by the prover at the end of the sum-check protocol.

sum-check protocol for $\log m$-variate poly of degree 3
- round complexity: $O(\log m)$
- communication cost: $O(\log m)$ field elements

$\tilde{O}$ hides the doubly-logarithmic factors

Spark: Spartan's sparse PCS

Special case ($c = 2$): Reduce evaluation to proof of multi-set equality

//During the commit phase, $\mathcal{P}$ has committed to three $(\log m)$-variate multilinear polynomials row, col, val.

1. $\mathcal{P} \to \mathcal{V}$: four $(\log m)$-variate multilinear polynomials $E_{\mathsf{rx}}, E_{\mathsf{ry}}, \mathsf{read\_ts}_{\mathsf{row}}, \mathsf{read\_ts}_{\mathsf{col}}$ and two $(\log \mathsf{M})$-variate multilinear polynomials $\mathsf{final\_cts}_{\mathsf{row}}$, $\mathsf{final\_cts}_{\mathsf{col}}$.
2. Recall that Claim $\boxed{1}$ (see Equation ($\boxed{8}$)) shows that $D(r_x, r_y) = \sum_{k \in \{0,1\}^{\log m}} \mathsf{val}(k) \cdot E_{\mathsf{rx}}(k) \cdot E_{\mathsf{ry}}(k)$ assuming that
   - $\forall k \in \{0,1\}^{\log m}$, $E_{\mathsf{rx}}(k) = \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{row}(k)), r_x)$; and
   - $\forall k \in \{0,1\}^{\log m}$, $E_{\mathsf{ry}}(k) = \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{col}(k)), r_y)$.

   Hence, $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to the polynomial $\mathsf{val}(k) \cdot E_{\mathsf{rx}}(k) \cdot E_{\mathsf{ry}}(k)$, which reduces the check that $v = \sum_{k \in \{0,1\}^{\log m}} \mathsf{val}(k) \cdot E_{\mathsf{rx}}(k) \cdot E_{\mathsf{ry}}(k)$ to checking that the following equations hold, where $r_z \in \mathbb{F}^{\log m}$ chosen at random by the verifier over the course of the sum-check protocol:
   - $\mathsf{val}(r_z) \stackrel{?}{=} v_{\mathsf{val}}$; and
   - $E_{\mathsf{rx}}(r_z) \stackrel{?}{=} v_{E_{\mathsf{rx}}}$ and $E_{\mathsf{ry}}(r_z) \stackrel{?}{=} v_{E_{\mathsf{ry}}}$. Here, v_{val}, $v_{E_{\mathsf{rx}}}$ and $v_{E_{\mathsf{ry}}}$ are values provided by the prover at the end of the sum-check protocol.
3. $\mathcal{V}$: check if the three equalities above hold with one oracle query each to each of $\mathsf{val}, E_{\mathsf{rx}}, E_{\mathsf{ry}}$.
4. // The following checks if E_{rx} is well-formed as per the first bullet in Step 2 above.
5. $\mathcal{V} \to \mathcal{P}$: $\tau, \gamma \in_R \mathbb{F}$.
6. $\mathcal{V} \leftrightarrow \mathcal{P}$: run a sum-check-based protocol for "grand products" ([Tha13], Proposition 2] or [SL20], Section 5 or 6]) to reduce the check that $\mathcal{H}_{\tau,\gamma}(\mathsf{WS}) = \mathcal{H}_{\tau,\gamma}(\mathsf{RS}) \cdot \mathcal{H}_{\tau,\gamma}(S)$, where $\mathsf{RS}, \mathsf{WS}, S$ are as defined in Claim $\boxed{3}$ and $\mathcal{H}$ is defined in Claim $\boxed{4}$ to checking if the following hold, where $r_{\mathsf{M}} \in \mathbb{F}^{\log \mathsf{M}}$, $r_m \in \mathbb{F}^{\log m}$ are chosen at random by the verifier over the course of the sum-check protocol:
   - $\widetilde{eq}(r_{\mathsf{M}}, r_x) \stackrel{?}{=} v_{eq}$
   - $E_{\mathsf{rx}}(r_m) \stackrel{?}{=} v_{E_{\mathsf{rx}}}$
   - $\mathsf{row}(r_m) \stackrel{?}{=} v_{\mathsf{row}}$; $\mathsf{read\_ts}_{\mathsf{row}}(r_m) \stackrel{?}{=} v_{\mathsf{read\_ts}_{\mathsf{row}}}$; and $\mathsf{final\_cts}_{\mathsf{row}}(r_{\mathsf{M}}) \stackrel{?}{=} v_{\mathsf{final\_cts}_{\mathsf{row}}}$
7. $\mathcal{V}$: directly check if the first equality holds, which can be done with $O(\log \mathsf{M})$ field operations; check the remaining equations hold with an oracle query to each of $E_{\mathsf{rx}}, \mathsf{row}, \mathsf{read\_ts}_{\mathsf{row}}, \mathsf{final\_cts}_{\mathsf{row}}$.
8. // The following steps check if E_{ry} is well-formed as per the second bullet in Step 2 above.

Completeness: perfect completeness

Soundness: $O(m)/|\mathbb{F}|$
- introduced by hash in multi-set equality
- introduced by sum-check protocol

Round and Communication Complexity:
(3 invocations of the sum-check protocol)
- round complexity: $\tilde{O}(\log m + \log N)$
- communication cost: $\tilde{O}(\log m + \log N)$
- prover commits to an extra $O(m/\log^3 m)$ field elements.

Verifier Time: $\tilde{O}(\log m)$ field operations
dominated by the grand product sum-check reductions

Prover Time: $O(N)$ field operations for untrusted tables
dominated by linear-time sum-checks

Question about these complexity ?

$\tilde{O}$ hides the doubly-logarithmic factors

Spark: Spartan's sparse PCS

Special case ($c = 2$): More discussion

//During the commit phase, $\mathcal{P}$ has committed to three $(\log m)$-variate multilinear polynomials row, col, val.

1. $\mathcal{P} \to \mathcal{V}$: four $(\log m)$-variate multilinear polynomials $E_{\mathsf{rx}}, E_{\mathsf{ry}}, \mathsf{read\_ts}_{\mathsf{row}}, \mathsf{read\_ts}_{\mathsf{col}}$ and two $(\log \mathsf{M})$-variate multilinear polynomials $\mathsf{final\_cts}_{\mathsf{row}}, \mathsf{final\_cts}_{\mathsf{col}}$.
2. Recall that Claim 1 (see Equation (8)) shows that $D(r_x, r_y) = \sum_{k \in \{0,1\}^{\log m}} \mathsf{val}(k) \cdot E_{\mathsf{rx}}(k) \cdot E_{\mathsf{ry}}(k)$ assuming that
   - $\forall k \in \{0,1\}^{\log m}$, $E_{\mathsf{rx}}(k) = \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{row}(k)), r_x)$; and
   - $\forall k \in \{0,1\}^{\log m}$, $E_{\mathsf{ry}}(k) = \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{col}(k)), r_y)$.

   Hence, $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to the polynomial $\mathsf{val}(k) \cdot E_{\mathsf{rx}}(k) \cdot E_{\mathsf{ry}}(k)$, which reduces the check that $v = \sum_{k \in \{0,1\}^{\log m}} \mathsf{val}(k) \cdot E_{\mathsf{rx}}(k) \cdot E_{\mathsf{ry}}(k)$ to checking that the following equations hold, where $r_z \in \mathbb{F}^{\log m}$ chosen at random by the verifier over the course of the sum-check protocol:
   - $\mathsf{val}(r_z) \stackrel{?}{=} v_{\mathsf{val}}$; and
   - $E_{\mathsf{rx}}(r_z) \stackrel{?}{=} v_{E_{\mathsf{rx}}}$ and $E_{\mathsf{ry}}(r_z) \stackrel{?}{=} v_{E_{\mathsf{ry}}}$. Here, v_{val}, $v_{E_{\mathsf{rx}}}$ and $v_{E_{\mathsf{ry}}}$ are values provided by the prover at the end of the sum-check protocol.
3. $\mathcal{V}$: check if the three equalities above hold with one oracle query each to each of $\mathsf{val}, E_{\mathsf{rx}}, E_{\mathsf{ry}}$.
4. // The following checks if E_{rx} is well-formed as per the first bullet in Step 2 above.
5. $\mathcal{V} \to \mathcal{P}$: $\tau, \gamma \in_R \mathbb{F}$.
6. $\mathcal{V} \leftrightarrow \mathcal{P}$: run a sum-check-based protocol for "grand products" ([Tha13, Proposition 2] or [SL20, Section 5 or 6]) to reduce the check that $\mathcal{H}_{\tau,\gamma}(\mathsf{WS}) = \mathcal{H}_{\tau,\gamma}(\mathsf{RS}) \cdot \mathcal{H}_{\tau,\gamma}(S)$, where $\mathsf{RS}, \mathsf{WS}, S$ are as defined in Claim 3 and $\mathcal{H}$ is defined in Claim 4 to checking if the following hold, where $r_{\mathsf{M}} \in \mathbb{F}^{\log \mathsf{M}}, r_m \in \mathbb{F}^{\log m}$ are chosen at random by the verifier over the course of the sum-check protocol:
   - $\boxed{\widetilde{eq}(r_{\mathsf{M}}, r_x) \stackrel{?}{=} v_{eq}}$
   - $E_{\mathsf{rx}}(r_m) \stackrel{?}{=} v_{E_{\mathsf{rx}}}$
   - $\mathsf{row}(r_m) \stackrel{?}{=} v_{\mathsf{row}}$; $\mathsf{read\_ts}_{\mathsf{row}}(r_m) \stackrel{?}{=} v_{\mathsf{read\_ts}_{\mathsf{row}}}$; and $\mathsf{final\_cts}_{\mathsf{row}}(r_{\mathsf{M}}) \stackrel{?}{=} v_{\mathsf{final\_cts}_{\mathsf{row}}}$
7. $\mathcal{V}$: directly check if the first equality holds, which can be done with $O(\log \mathsf{M})$ field operations; check the remaining equations hold with an oracle query to each of $E_{\mathsf{rx}}, \mathsf{row}, \mathsf{read\_ts}_{\mathsf{row}}, \mathsf{final\_cts}_{\mathsf{row}}$.
8. // The following steps check if E_{ry} is well-formed as per the second bullet in Step 2 above.

Evaluation procedure to prove $D(r_x, r_y) = v$:

1. (Write) Evaluate $c = 2$ memory of size M.
   - $\tilde{eq}(i, r_x)$ as i ranged over $\{0,1\}^{\log M}$
   - $\tilde{eq}(j, r_y)$ as j ranged over $\{0,1\}^{\log M}$

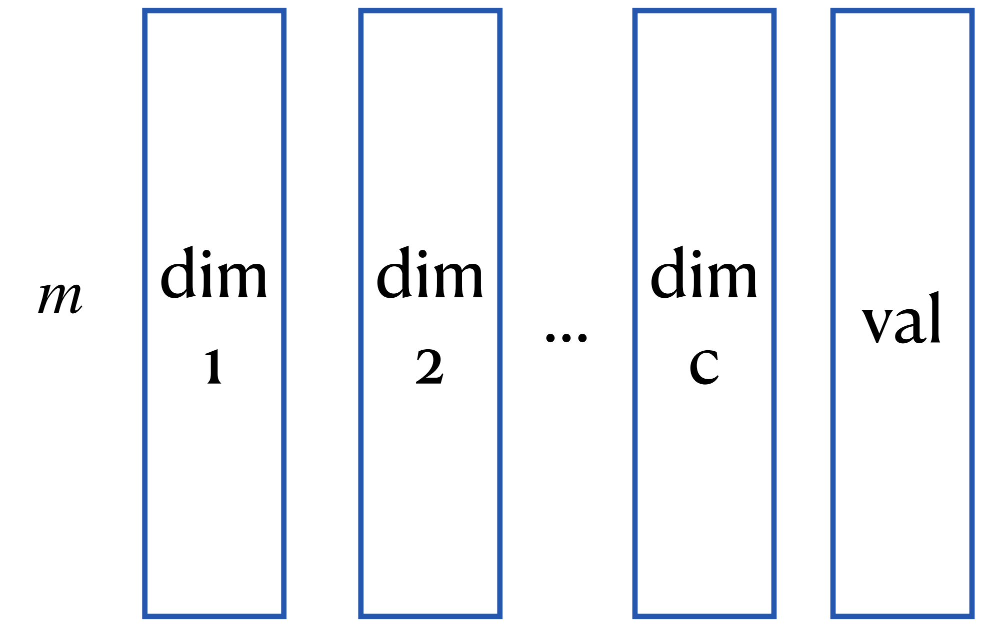
Prover **dose not** have to commit to the values written to memory (or lookup tables), albeit dynamically determined by the evaluation point (r_x, r_y).

Because these lookup tables are MLE-structured, meaning that verifier can quickly evaluate the MLE at a random point on its own.

Intuitively, prover only cryptographically commits to the **values and counters** returned by the aforementioned operations.

Spark: Spartan's sparse PCS

Dense representation:



General case

$c = 2$

$$\widetilde{D}(r_x, r_y) = \sum_{(i,j) \in \{0,1\}^{\log(\mathsf{M})} \times \{0,1\}^{\log(\mathsf{M})}} D(i,j) \cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(i, r_x) \cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(j, r_y).$$

$$\widetilde{\mathsf{eq}}_{2\log(\mathsf{M})}\left((i,j),(r_x,r_y)\right) = \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(i, r_x) \cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(j, r_y).$$

$c = 3$

$$\widetilde{D}(r_x, r_y, r_z) = \sum_{(i,j,k) \in \{0,1\}^{\log(\mathsf{M})} \times \{0,1\}^{\log(\mathsf{M})} \times \{0,1\}^{\log(\mathsf{M})}} D(i,j,k) \cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(i, r_x) \cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(j, r_y) \cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(k, r_z). \tag{11}$$

$$\widetilde{\mathsf{eq}}_{3\log(\mathsf{M})}\left((i,j,k),(r_x,r_y,r_z)\right) = \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(i, r_x) \cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(j, r_y) \cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(k, r_z).$$

Decompose $\log N$ variables into c blocks.

Suppose we want to support sparse polynomials over $c\log(\mathsf{M})$ variables for constant $c > 2$, while ensuring that the prover still only commits to $3c + 1$ many dense multilinear polynomials over $\log m$ many variables, and c many over $\log(N^{1/c})$ many variables. We can proceed as follows.

Commitment phase:

- prover commits to $c + 1$ multilinear polynomials defined over $\log m$-variables.

At the beginning of evaluation phase: $\widetilde{D}(r_1, \ldots, r_c)$

- lookup tables: c memories of size $M = N^{1/c}$
- verifier needs to check c different untrusted memories.
- **for each memory checked**, the prover has to commit to two multilinear polynomials defined over $\log m$-many variables, and one defined over $\log M = \log N/c$ variables. (values and counters)

Spark: Spartan's sparse PCS

Back to our general result

For each memory checked, the prover has to commit to three multilinear polynomials defined over $\log(m)$-many variables, and one defined over $\log(\mathsf{M}) = \log(N)/c$ variables. We obtain the following theorem.

Theorem 2. *Given a polynomial commitment scheme for $(\log \mathsf{M})$-variate multilinear polynomials with the following parameters (where M is a positive integer and WLOG a power of 2):*

- *the size of the commitment is $\mathsf{c}(\mathsf{M})$;*
- *the running time of the commit algorithm is $\mathsf{tc}(\mathsf{M})$;*
- *the running time of the prover to prove a polynomial evaluation is $\mathsf{tp}(\mathsf{M})$;*
- *the running time of the verifier to verify a polynomial evaluation is $\mathsf{tv}(\mathsf{M})$;*
- *the proof size is $\mathsf{p}(\mathsf{M})$,*

there exists a polynomial commitment scheme for $(c \log \mathsf{M})$-variate multilinear polynomials that evaluate to a non-zero value at at most m locations over the Boolean hypercube $\{0,1\}^{c \log \mathsf{M}}$, with the following parameters:

- *the size of the commitment is $\boxed{(3c+1)\mathsf{c}(m) + c \cdot \mathsf{c}(\mathsf{M});}$*
- *the running time of the commit algorithm is $O\left(c \cdot (\mathsf{tc}(m) + \mathsf{tc}(\mathsf{M}))\right)$;*
- *the running time of the prover to prove a polynomial evaluation is $O\left(c\left(\mathsf{tp}(m) + \mathsf{tc}(\mathsf{M})\right)\right)$;*
- *the running time of the verifier to verify a polynomial evaluation is $O\left(c\left(\mathsf{tv}(m) + \mathsf{tv}(\mathsf{M})\right)\right)$;*
- *the proof size is $O\left(c\left(\mathsf{p}(m) + \mathsf{p}(\mathsf{M})\right)\right)$.*

Many polynomial commitment schemes have efficient batching properties for evaluation proofs. For such schemes, the factor c can be omitted in the final three bullet points of Theorem 2 (i.e., prover and verifier costs for verifying polynomial evaluation do not grow with c).

PCS for a $\log N$-variate polynomial of sparsity m, using c memories of size $M = N^{1/c}$.
(decompose $\log N$ variables to c blocks)

Dominate costs for prover: committing to

- $3c + 1$dense multilinear polys over $\log m$-vars
- c dense multilinear polys over $\log(N^{1/c})$-vars

Spark: Spartan's sparse PCS

Specializing the Spark to Lasso

Reduce lookup to a matrix-vector multiplication with a sparse matrix.

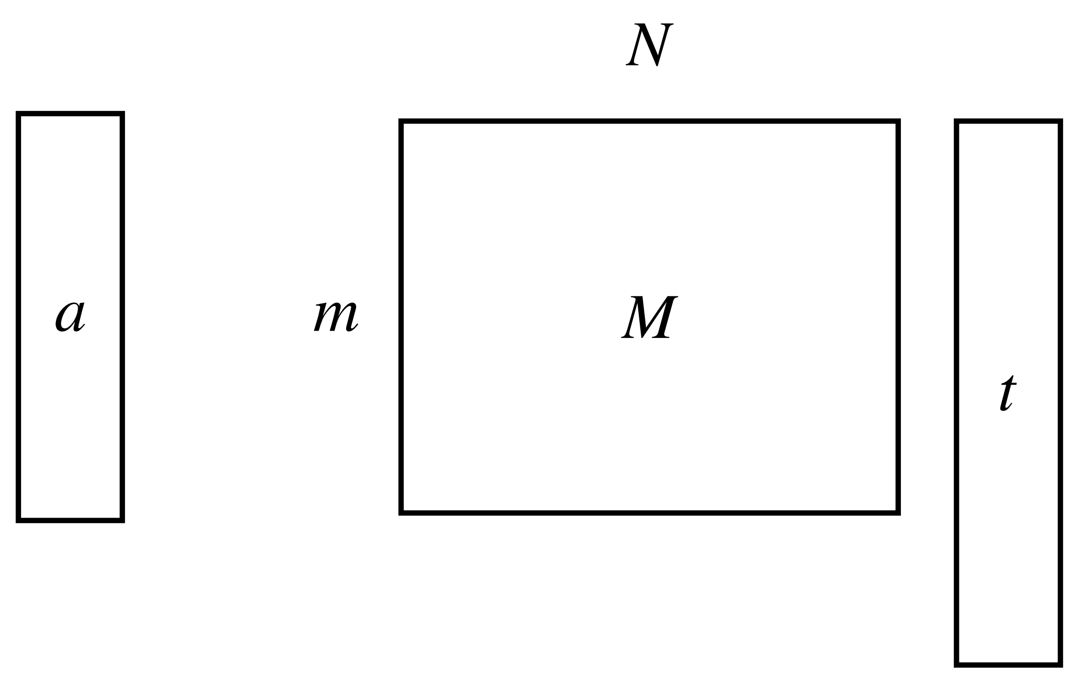
Suppose that the verifier has a commitment to a table $t \in \mathbb{F}^n$ as well as a commitment to another vector $a \in \mathbb{F}^m$. Suppose that a prover wishes to prove that all entries in a are in the table t. A simple observation in prior works [ZBK+22, ZGK+22] is that the prover can prove that it knows a sparse matrix $M \in \mathbb{F}^{m \times n}$ such that for each row of M, only one cell has a value of 1 and the rest are zeros and that $M \cdot t = a$, where $\cdot$ is the matrix-vector multiplication. This turns out to be equivalent, up to negligible soundness error, to confirming that

$$\sum_{y \in \{0,1\}^{\log N}} \widetilde{M}(r, y) \cdot \tilde{t}(y) = \tilde{a}(r), \tag{5}$$

for an $r \in \mathbb{F}^{\log m}$ chosen at random by the verifier. Here, $\widetilde{M}$, $\tilde{a}$ and $\tilde{t}$ are the so-called *multilinear extension polynomials* (MLEs) of M, t, and a (see Section 2.1 for details).

In Lasso, if the prover is honest then the sparse polynomial commitment scheme is applied to the multilinear extension of a matrix M with m rows and N columns, where m is the number of lookups and N is the size of the table. If the prover is honest then each row of M is a unit vector.

In fact, we require the commitment scheme to enforce these properties even when the prover is potentially malicious. Achieving this simplifies the commitment scheme and provides concrete efficiency benefits. It also keeps Lasso's polynomial IOP simple as it does not need additional invocations of the sum-check protocol to prove that M satisfies these properties.



1. **Commit to the sparse matrix M**
2. Reduced to a sum-check protocol
3. **Evaluation on a random point** (r, r') where $r' \in \mathbb{F}^{\log N}$

Instead of committing to a $\log m + \log N$ -variate polynomial with sparsity m,

we can commit to a $\log N$-variate polynomial $M(r, \cdot\,)$ with sparsity m.

$$D(r_x, r_y) = \sum_{k \in \{0,1\}^{\log m}} \mathsf{val}(k) \cdot \widetilde{eq}(\text{to-bits}(\text{row}(k)), r_x) \cdot \widetilde{eq}(\text{to-bits}(\text{col}(k)), r_y).$$

$$; D(r_x, r_y) = \sum_{k \in \{0,1\}^{\log m}} \mathsf{val}(k) \cdot E_{\mathsf{rx}}(k) \cdot E_{\mathsf{ry}}(k)$$

Spark: Spartan's sparse PCS

Specializing the Spark to Lasso

1. val(k)=1 is a constant polynomial —» no need to commit to val(k)

First, the multilinear polynomial $\mathsf{val}(k)$ is fixed to 1, and it is not committed by the prover. Recall from Claim 1 that $\mathsf{val}(k)$ extends the function that maps a bit-vector $k \in \{0,1\}^{\log m}$ to the value of the k'th non-zero evaluation of the sparse function. Since M is a $\{0,1\}$-valued matrix, $\mathsf{val}(k)$ is just the constant polynomial that evaluates to 1 at all inputs.

2. to-bits(row(k))=k —» no need to commit to row(k), $E_{rx}(k)$, nor prove E_{rx} is well-formed.

Second, for any $k = (k_1, \ldots, k_{\log m}) \in \{0,1\}^{\log m}$, the k'th non-zero entry of M is in row $\text{to-field}(k) = \sum_{j=1}^{\log m} 2^{j-1} \cdot k_j$. Hence, in Equation (8) of Claim 1, to-bits(row(k)) is simply k.[18] This means that $E_{\mathsf{rx}}(k) = \widetilde{\mathsf{eq}}(k, r_x)$, which the verifier can evaluate on its own in logarithmic time. With this fact in hand, the prover does not commit to E_{rx} nor prove that it is well-formed.

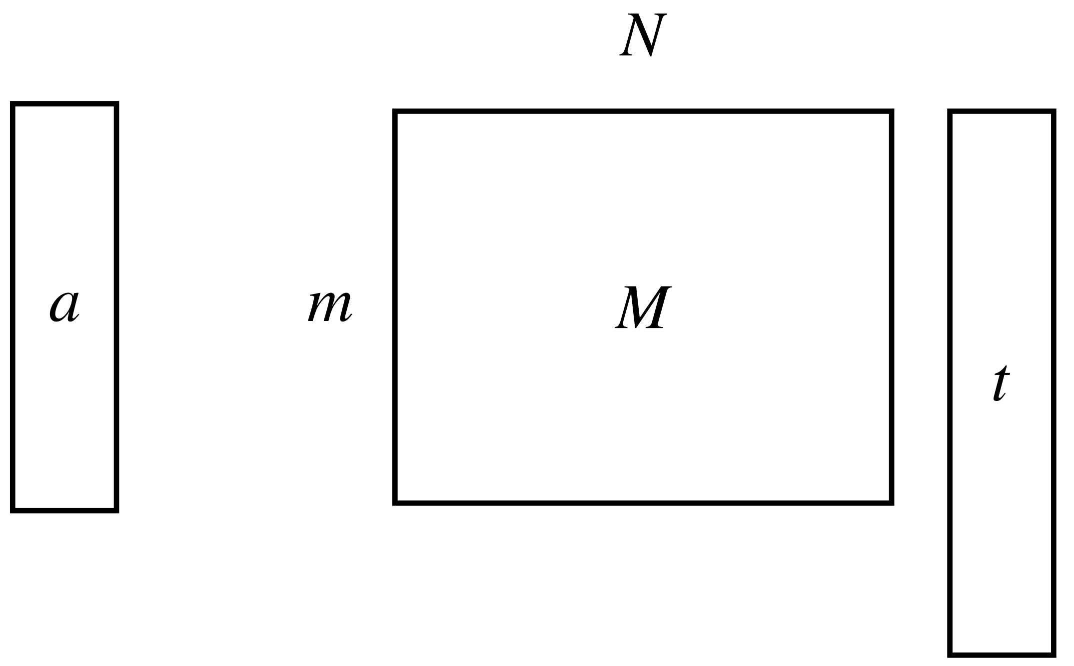
It indeed effectively removes the contribution of the first $\log m$-variables of $\tilde{M}$ to the costs.

As a result, the prover simply commits to a $\log N$-variate polynomial with sparsity m.

Then we can use the aforementioned PCS for sparse polynomials: (Decompose $\log N$ variables to c blocks.)

This means that, setting $c = 2$ for illustration, the prover commits to 6 multilinear polynomials with $\log(m)$ variables each and to two multilinear polynomials with $(1/2)\log N$ variables each.

Figure 4 describes Spark specialized for Lasso to commit to $\widetilde{M}$. The prover commits to $3c$ dense $(\log(m))$-variate multilinear polynomials, called $\dim_1, \ldots, \dim_c$ (the analogs of the row and col polynomials of Section 4.1), $E_1, \ldots, E_c$, and $\mathsf{read\_ts}_1, \ldots, \mathsf{read\_ts}_c$, as well as c dense multilinear polynomials in $\log(N^{1/c}) = \log(N)/c$ variables, called $\mathsf{final\_cts}_1, \ldots, \mathsf{final\_cts}_c$. Each $\dim_i$ is purported to be the memory cell from the i'th memory that the sparse polynomial evaluation algorithm (§3.1) reads at each of its m timesteps, $E_1, \ldots, E_c$ the values returned by those reads, and $\mathsf{read\_ts}_1, \ldots, \mathsf{read\_ts}_c$ the associated counts. $\mathsf{final\_cts}_1, \ldots, \mathsf{final\_cts}_c$ are purported to be to counts returned by the memory checking procedure's final pass over each of the c memories.



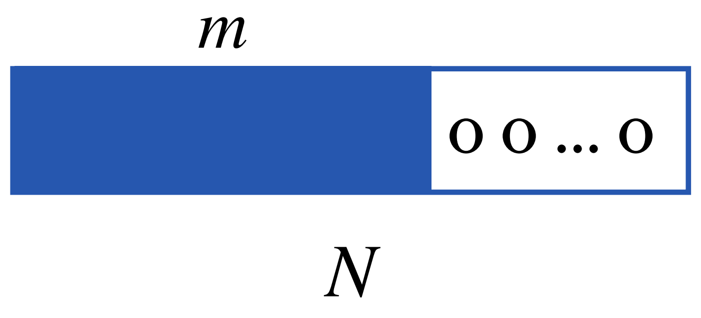
1. **Commit to the sparse matrix M**
2. Reduced to a sum-check protocol
3. **Evaluation on a random point** (r, r') where $r' \in \mathbb{F}^{\log N}$

Here is my understanding. $r_x \in \mathbb{F}^{\log m}, r_y \in \mathbb{F}^{\log N}$

$$\tilde{M}(r_x, r_y) = \sum_{k \in \{0,1\}^{\log m}} \widetilde{eq}(k, r_x) \cdot \boxed{\widetilde{eq}(\text{to-bits}(\text{col}(k)), r_y)}$$

The $\log N$-variate polynomial with sparsity m.

$M(r_x, r_y) =$



for each $r_y \in \{0,1\}^{\log N}$

Spark: Spartan's sparse PCS

Specializing the Spark to Lasso: full evaluation procedure

//During the commit phase applied to the multilinear extension $\widetilde{M}$ of $m \times N$ matrix M with each row a unit vector, $\mathcal{P}$ has committed to c different ℓ-variate multilinear polynomials $\mathsf{dim}_1, \ldots, \mathsf{dim}_c$, where $\ell = \log(N^{1/c})$. These are analogs of the polynomials row and col from Figure 3. dim_i is purported to provide the indices of the cells of the i'th memory that are read by the sparse polynomial evaluation algorithm of Section 3.1. Note that these indices depend only on the the locations of the non-zero entries of $\widetilde{M}$.

//If $\mathcal{P}$ is honest, then each dim_i maps $\{0,1\}^{\log m}$ to $\{0, \ldots, N^{1/c} - 1\}$. For each $j \in \{0,1\}^{\log m}$, $(\mathsf{dim}_1(j), \ldots, \mathsf{dim}_c(j))$ is interpreted as specifying the identity of the unique non-zero entry of row j of M.

//$\mathcal{V}$ requests to evaluate $\widetilde{M}$ at input (r, r') where $r' = (r'_1, \ldots, r'_c) \in (\mathbb{F}^\ell)^c$.

1. $\mathcal{P} \to \mathcal{V}$: $2c$ different $(\log m)$-variate multilinear polynomials $E_1, \ldots, E_c$, $\mathsf{read\_ts}_1, \ldots \mathsf{read\_ts}_c$ and c different ℓ-variate multilinear polynomials $\mathsf{final\_cts}_1, \ldots, \mathsf{final\_cts}_c$.
   //If $\mathcal{P}$ is honest, then $\mathsf{read\_ts}_1, \ldots \mathsf{read\_ts}_c$ and $\mathsf{final\_cts}_1, \ldots, \mathsf{final\_cts}_c$ map $\{0,1\}^{\log m}$ to $\{0, \ldots, m - 1\}$, as these are "counter polynomials" for each of the c memories.
   //If $\mathcal{P}$ is honest, then $E_1, \ldots, E_c$ contain the values returned by each read operation that the sparse polynomial evaluation algorithm of Section 3.1 makes to each of the c memories.
2. Recall (Equation 11) that $\widetilde{M}(r, r') = \sum_{k \in \{0,1\}^{\log m}} \widetilde{\mathsf{eq}}(r, k) \cdot \prod_{i=1}^{c} E_i(k)$, assuming that
   - $\forall k \in \{0,1\}^{\log m}$, $E_i(k) = \widetilde{eq}(\text{to-bits}(\mathsf{dim}_i(k)), r'_i)$.

   Hence, $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to the polynomial $g(k) := \widetilde{\mathsf{eq}}(r, k) \cdot \prod_{i=1}^{c} E_i(k)$, which reduces the check that $v = \sum_{k \in \{0,1\}^{\log m}} \widetilde{\mathsf{eq}}(r, k) \prod_{i=1}^{c} E_i(k)$ to checking that the following equations hold, where $r_z \in \mathbb{F}^{\log m}$ chosen at random by the verifier over the course of the sum-check protocol:
   - $E_i(r_z) \stackrel{?}{=} v_{E_i}$ for $i = 1, \ldots, c$. Here, $v_{E_1}, \ldots, v_{E_c}$ are values provided by the prover at the end of the sum-check protocol.
3. $\mathcal{V}$: check if the above equalities hold with one oracle query to each E_i.
   // The following checks if E_i is well-formed as per the first bullet in Step 2 above.

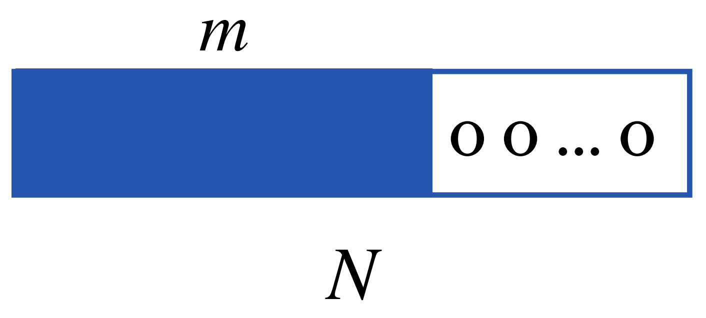
Here is my understanding.

$r_x \in \mathbb{F}^{\log m}, r_y \in \mathbb{F}^{\log N}$

$$\tilde{M}(r_x, r_y) = \sum_{k \in \{0,1\}^{\log m}} \tilde{eq}(k, r_x) \cdot \boxed{\tilde{eq}(\text{to-bits}(\text{col}(k)), r_y)}$$

The $\log N$-variate polynomial with sparsity m.

$M(r_x, r_y) =$



for each $r_y \in \{0,1\}^{\log N}$

1. **Commit to the sparse vector $M(r, \cdot)$ of size N**
2. Reduced to a sum-check protocol
3. **Evaluation on a random point $r' \in \mathbb{F}^{\log N}$**

Spark: Spartan's sparse PCS

Specializing the Spark to Lasso: full evaluation procedure

same as the aforementioned steps

3. $\mathcal{V}$: check if the above equalities hold with one oracle query to each E_i.
   // The following checks if E_i is well-formed as per the first bullet in Step 2 above.
4. $\mathcal{V} \to \mathcal{P}$: $\tau, \gamma \in_R \mathbb{F}$.
   //In practice, one would apply a single sum-check protocol to a random linear combination of the below polynomials. For brevity, we describe the protocol as invoking c independent instances of sum-check.
5. $\mathcal{V} \leftrightarrow \mathcal{P}$: For $i = 1, \ldots, c$, run a sum-check-based protocol for "grand products" ([Tha13, Proposition2] or [SL20, Section 5 or 6]) to reduce the check that $\mathcal{H}_{\tau,\gamma}(\mathsf{WS}) = \mathcal{H}_{\tau,\gamma}(\mathsf{RS}) \cdot \mathcal{H}_{\tau,\gamma}(S)$, where $\mathsf{RS}, \mathsf{WS}, S$ are as defined in Claim 3 and $\mathcal{H}$ is defined in Claim 4 to checking if the following hold, where $r_i'' \in \mathbb{F}^{\ell}, r_i''' \in \mathbb{F}^{\log m}$ are chosen at random by the verifier over the course of the sum-check protocol:
   - $E_i(r_i''') \stackrel{?}{=} v_{E_i}$
   - $\dim_i(r_i''') \stackrel{?}{=} v_i$; $\mathsf{read\_ts}_i(r_i''') \stackrel{?}{=} v_{\mathsf{read\_ts}_i}$; and $\mathsf{final\_cts}_i(r_i'') \stackrel{?}{=} v_{\mathsf{final\_cts}_{\mathsf{row}}}$
6. $\mathcal{V}$: check that the remaining equations hold with an oracle query to each of $E_i, \dim_i, \mathsf{read\_ts}_i, \mathsf{final\_cts}_i$.

Here is my understanding.

$r_x \in \mathbb{F}^{\log m}, r_y \in \mathbb{F}^{\log N}$

$$\tilde{M}(r_x, r_y) = \sum_{k \in \{0,1\}^{\log m}} \tilde{eq}(k, r_x) \cdot \boxed{\tilde{eq}(\text{to-bits}(\text{col}(k)), r_y)}$$

The log N-variate polynomial with sparsity m.

m

$M(r_x, r_y) =$ 

N

for each $r_y \in \{0,1\}^{\log N}$

1. **Commit to the sparse vector $M(r, \cdot\,)$ of size N**
2. Reduced to a sum-check protocol
3. **Evaluation on a random point $r' \in \mathbb{F}^{\log N}$**

Surge

A generalization of Spark, providing Lasso

Lasso with Spark proving evaluations of the sparse poly $\tilde{M}(r, r')$

Overview of Lasso. In Lasso, after committing to $\widetilde{M}$, the Lasso verifier picks a random $r \in \mathbb{F}^{\log m}$ and seeks to confirm that

$$\boxed{\sum_{j \in \{0,1\}^{\log N}} \widetilde{M}(r, j) \cdot t(j)} = \widetilde{a}(r). \quad (12)$$

Indeed, if $M \cdot t$ and a are the same vector, then Equation (12) holds for every choice of r, while if $Mt \neq a$, then by the Schwartz-Zippel lemma, Equation (12) holds with probability at most $\frac{\log m}{|\mathbb{F}|}$. So up to soundness error $\frac{\log m}{|\mathbb{F}|}$, checking that $Mt = a$ is equivalent to checking that Equation (12) holds.

The $\log N$-variate polynomial with sparsity m.

m

$M(r_x, r_y) =$ 

N

for each $r_y \in \{0,1\}^{\log N}$

1. Commit to the sparse vector $\tilde{M}(r, \cdot)$
2. Reduced to a sum-check protocol
3. **Spark: Evaluation on a random point $r' \in \mathbb{F}^{\log N}$**

Surge: directly proves the evaluation of a large class of statements about the committed polynomial $\tilde{M}$

Recall from Section 4 and Figure 4 that Spark allows the untrusted Lasso prover to commit to $\widetilde{M}$, purported to be the multilinear extension of an $m \times N$ matrix M, with each row equal to a unit vector, such that $M \cdot t = a$. The commitment phase of Surge is same as that of Spark. Surge generalizes Spark in that the Surge prover proves a larger class of statements about the committed polynomial $\widetilde{M}$ (Spark focused only on proving *evaluations* of the sparse polynomial $\widetilde{M}$).

Exploiting this perspective, we describe Surge, a generalization of Spark that allows an untrusted prover to commit to any sparse vector and establish the sparse vector's inner product with any dense, structured vector. We refer to the structure required for this to work as *Spark-only structure* (SOS for short). We also

for Spark-only structured(SOS) table

1. Commit to the sparse vector $\tilde{M}(r, \cdot)$
2. Verifier obtain $\tilde{a}(r)$ via the commitment
3. **Surge: directly proves the LHS**

$$\sum_{j \in \{0,1\}^{\log N}} \tilde{M}(r, j) T[j] = v$$

Surge

A roughly $O(\alpha m)$-time algorithm for computing LHS

$$\sum_{j\in\{0,1\}^{\log N}} \tilde{M}(r,j)\cdot t(j) = \sum_{i\in\{0,1\}^{\log m}} \tilde{eq}(i,r)\cdot T[nz(i)]$$

Overview of Lasso. In Lasso, after committing to $\widetilde{M}$, the Lasso verifier picks a random $r \in \mathbb{F}^{\log m}$ and seeks to confirm that

$$\sum_{j\in\{0,1\}^{\log N}} \widetilde{M}(r,j)\cdot t(j) = \widetilde{a}(r). \tag{12}$$

Indeed, if $M \cdot t$ and a are the same vector, then Equation (12) holds for every choice of r, while if $Mt \neq a$, then by the Schwartz-Zippel lemma, Equation (12) holds with probability at most $\frac{\log m}{|\mathbb{F}|}$. So up to soundness error $\frac{\log m}{|\mathbb{F}|}$, checking that $Mt = a$ is equivalent to checking that Equation (12) holds.

Hence, letting $\mathsf{nz}(i)$ denote the unique column in row i of M that contains a non-zero value (namely, the value 1), the left hand side of Equation (12) equals

$$\sum_{i\in\{0,1\}^{\log m}} \widetilde{eq}(i,r)\cdot T[\mathsf{nz}(i)]. \tag{13}$$

$$\widetilde{M}(r,y) = \sum_{(i,j)\in\{0,1\}^{\log m+\log N}} M_{i,j}\cdot \widetilde{eq}(i,r)\cdot \widetilde{eq}(j,y).$$

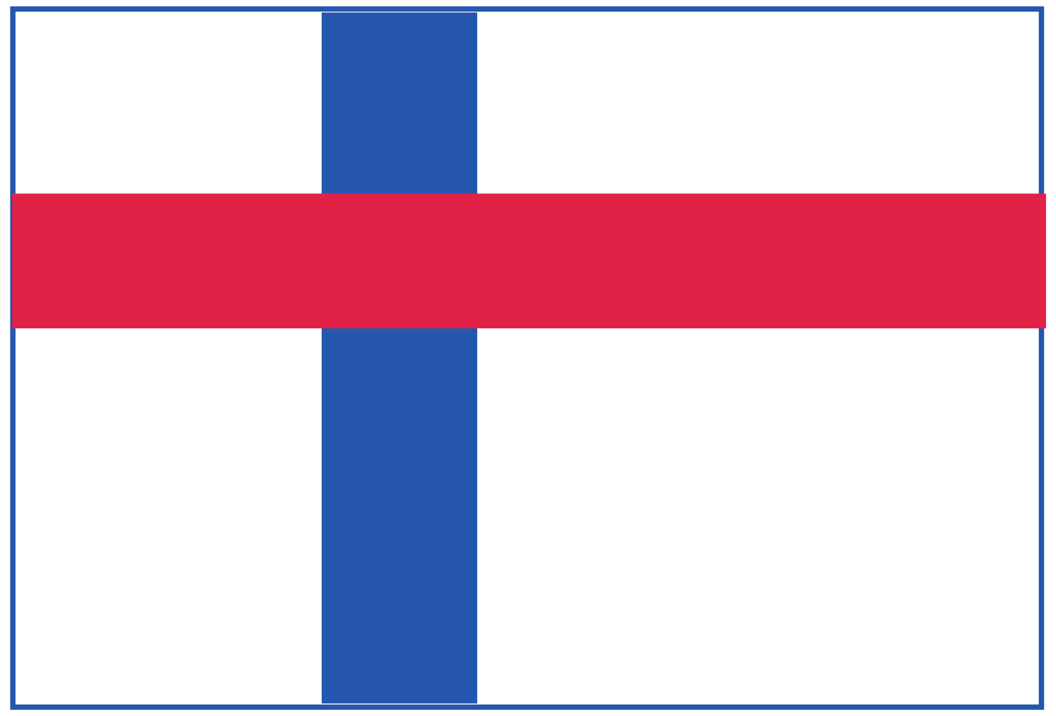
$$\tilde{M}(r,j) = \sum_{i\in\{0,1\}^{\log m}} M_{i,j}\cdot \tilde{eq}(i,r) \quad \text{for } j \in \{0,1\}^{\log N}$$

$$\tilde{M}(r,j)\cdot t(j) = \sum_{i\in\{0,1\}^{\log m}} M_{i,j}\cdot \tilde{eq}(i,r)\cdot t(j) \quad \text{for } j \in \{0,1\}^{\log N}$$

$$\sum_{j\in\{0,1\}^{\log N}} \tilde{M}(r,j)\cdot t(j) = \quad \text{(sum of a matrix as follows)}$$

$j \in \log N$

$i \in \log m$



$\tilde{M}(r,j)\cdot t(j)$

each entry is $M_{i,j}\cdot \tilde{eq}(i,r)\cdot t(j)$

$$= \sum_{i\in\{0,1\}^{\log m}} \tilde{eq}(i,r)\cdot T[nz(i)]$$

(since each row of M is an unit vector)

Surge

A roughly $O(\alpha m)$-time algorithm for computing LHS

Computes $\sum_{y\in\{0,1\}^{\log N}} \tilde{M}(r,y)T[y] = v$ in roughly $O(\alpha m)$-time

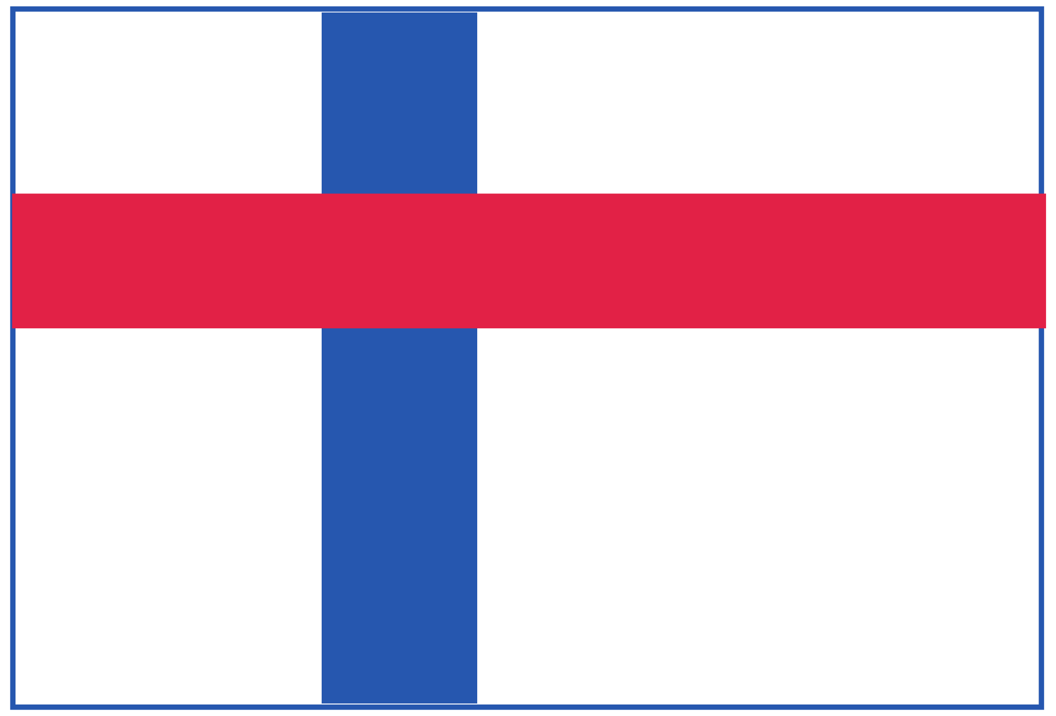
$\sum_{j\in\{0,1\}^{\log N}} \tilde{M}(r,j)\cdot t(j) =$ (sum of a matrix as follows)

Hence, letting $\boxed{\mathsf{nz}(i) \text{ denote the unique column in row } i \text{ of } M \text{ that contains a non-zero value}}$ (namely, the value 1), the left hand side of Equation ($\boxed{12}$) equals

$$\boxed{\sum_{i\in\{0,1\}^{\log m}} \widetilde{eq}(i,r)\cdot T[\mathsf{nz}(i)].} \tag{13}$$

$j \in \log N$

$i \in \log m$



$\tilde{M}(r,j)\cdot t(j)$

each entry is $M_{i,j}\cdot \tilde{eq}(i,r)\cdot t(j)$

SOS table with decomposability

Suppose that T is a SOS table. This means that there is an integer $k \geq 1$ and $\alpha = k\cdot c$ tables $T_1,\ldots,T_\alpha$ of size $N^{1/c}$, as well as an α-variate multilinear polynomial g such that the following holds. Suppose that for every $r = (r_1,\ldots,r_c) \in \left(\{0,1\}^{\log(N)/c}\right)^c$,

$$\boxed{T[r] = g\left(T_1[r_1],\ldots,T_k[r_1],T_{k+1}[r_2],\ldots,T_{2k}[r_2],\ldots,T_{\alpha-k+1}[r_c],\ldots,T_\alpha[r_c]\right).} \tag{14}$$

refer to this property as *decomposability*. In more detail, an SOS table T is one that can be decomposed into $\alpha = O(c)$ "sub-tables" $\{T_1,\ldots,T_\alpha\}$ of size $N^{1/c}$ satisfying the following two properties. First, any entry $T[j]$ of T can be expressed as a simple expression of a corresponding entry into each of $T_1,\ldots,T_\alpha$. Second, the so-called *multilinear extension polynomial* of each T_i can be evaluated quickly (for any such table, we call T_i *MLE-structured*, where MLE stands for multilinear extension). For example, as noted above, the table T arising in Spark itself is simply the tensor product of MLE-structured sub-tables $\{T_1,\ldots,T_\alpha\}$, where $\alpha = c$.

$$= \sum_{i\in\{0,1\}^{\log m}} \tilde{eq}(i,r)\cdot T[nz(i)]$$

(since each row of M is an unit vector)

T in Spark: $\alpha = c$

$$\widetilde{D}(r_x,r_y) = \sum_{(i,j)\in\{0,1\}^{\log(\mathsf{M})}\times\{0,1\}^{\log(\mathsf{M})}} D(i,j)\cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(i,r_x)\cdot \widetilde{\mathsf{eq}}_{\log(\mathsf{M})}(j,r_y).$$

$$D(r_x,r_y) = \sum_{k\in\{0,1\}^{\log m}} \mathsf{val}(k)\cdot \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{row}(k)),r_x)\cdot \widetilde{eq}(\mathsf{to\text{-}bits}(\mathsf{col}(k)),r_y).$$

$$g(T_1[r_1],T_2[r_2],\ldots,T_c[r_c]) = \prod_{i=1}^{c} T_i[r_i]$$

Surge

A roughly $O(\alpha m)$-time algorithm for computing LHS

Computes $\sum_{y\in\{0,1\}^{\log N}} \tilde{M}(r,y)T[y] = v$ in roughly $O(\alpha m)$-time

Hence, letting $\boxed{\mathsf{nz}(i) \text{ denote the unique column in row } i \text{ of } M \text{ that contains a non-zero value}}$ (namely, the value 1), the left hand side of Equation (12) equals

$$\boxed{\sum_{i\in\{0,1\}^{\log m}} \tilde{eq}(i,r)\cdot T[\mathsf{nz}(i)].} \tag{13}$$

SOS table with decomposability

Suppose that T is a SOS table. This means that there is an integer $k\geq 1$ and $\alpha = k\cdot c$ tables $T_1,\ldots,T_\alpha$ of size $N^{1/c}$, as well as an α-variate multilinear polynomial g such that the following holds. Suppose that for every $r=(r_1,\ldots,r_c)\in\left(\{0,1\}^{\log(N)/c}\right)^c$,

$$\boxed{T[r] = g\left(T_1[r_1],\ldots,T_k[r_1],T_{k+1}[r_2],\ldots,T_{2k}[r_2],\ldots,T_{\alpha-k+1}[r_c],\ldots,T_\alpha[r_c]\right).} \tag{14}$$

For each $i\in\{0,1\}^{\log m}$, decompose $\boxed{\mathsf{nz}(i) \text{ and } (\mathsf{nz}_1(i),\ldots,\mathsf{nz}_c(i))\in[N^{1/c}]^c.}$ Then Expression (13) equals

$$\sum_{i\in\{0,1\}^{\log m}} \tilde{\mathsf{eq}}(i,r)\cdot g\left(T_1[\mathsf{nz}_1(i)],\ldots,T_k[\mathsf{nz}_1(i)],T_{k+1}[\mathsf{nz}_2(i)],\ldots,T_{2k}[\mathsf{nz}_2(i)],\ldots,T_{\alpha-k+1}[\mathsf{nz}_c(i)],\ldots,T_\alpha[\mathsf{nz}_c(i)]\right).$$

Compute $\sum_{i\in\{0,1\}^{\log m}} \tilde{eq}(i,r)\cdot T[nz(i)]$ in roughly $O(\alpha m)$-time:

The algorithm to compute Expression (15) simply initializes all tables $T_1,\ldots,T_\alpha$, then iterates over every $i\in\{0,1\}^m$ and computes the i'th term of the sum with a single lookup into each table (of course, the algorithm evaluates g at the results of the lookups into $T_1,\ldots,T_\alpha$, and multiplies the result by $\tilde{\mathsf{eq}}(i,r)$).

1. initialize all tables $T_1, \ldots, T_\alpha$
2. iterates over every $i \in \{0,1\}^m$ to compute the i'th term
    1. evaluates g at α lookups into $T_1, \ldots, T_\alpha$
    2. multiplies the result by $\tilde{eq}(i, r)$

Surge

Description of Surge

Surge: prove $\sum_{y \in \{0,1\}^{\log N}} \tilde{M}(r, y)T[y] = v$ in roughly $O(\alpha m)$-time

1. The Surge prover commit to $\tilde{M}$, purported to be the MLE of an $m \times N$ matrix with each row is an unit vector

Description of Surge. The commitment to $\widetilde{M}$ in Surge consists of commitments to c multilinear polynomials $\dim_1, \ldots, \dim_c$, each over $\log m$ variables. $\dim_i$ is purported to be the multilinear extension of nz_i.

(Consider M as a sparse vector $M(r, \cdot\,)$ of size N with sparsity m.)

2. Verifier chooses $r \in \{0,1\}^{\log m}$, and reduce the proof of $\sum_{y \in \{0,1\}^{\log N}} \tilde{M}(r, y)T[y] = v$ to $\sum_{i \in \{0,1\}^{\log m}} \tilde{eq}(i, r) \cdot T[nz(i)] = v$

The verifier chooses $r \in \{0,1\}^{\log m}$ at random and requests that the Surge prover prove that the committed polynomial $\widetilde{M}$ satisfy Equation (13). The prover does so by proving it ran the aforementioned algorithm

3. Prover dose so by proving it ran the $O(\alpha m)$-time algorithm correctly with some purported oracles (via the sum-check protocol)

polynomial $\widetilde{M}$ satisfy Equation (13). The prover does so by proving it ran the aforementioned algorithm for evaluating Expression (15). Following the memory-checking procedure in Section 4, with each table

(assuming each condition $E_1(i) = T_1[nz(i)]$ ... holds)

$$\sum_{j \in \{0,1\}^{\log m}} \widetilde{\mathsf{eq}}(r, j) \cdot g\left(E_1(j), \ldots, E_\alpha(j)\right). = \sum_{i \in \{0,1\}^{\log m}} \widetilde{\mathsf{eq}}(i, r) \cdot g\left(T_1[\mathsf{nz}_1(i)], \ldots, T_k[\mathsf{nz}_1(i)], T_{k+1}[\mathsf{nz}_2(i)], \ldots, T_{2k}[\mathsf{nz}_2(i)], \ldots, T_{\alpha-k+1}[\mathsf{nz}_c(i)], \ldots, T_\alpha[\mathsf{nz}_c(i)]\right).$$

4. Reduced to evaluate at a random point $r' \in \mathbb{F}^{\log m}$

At the end of the sum-check protocol, the verifier needs to evaluate $\boxed{\widetilde{\mathsf{eq}}(r, r') \cdot g(E_1(r'), \ldots, E_\alpha(r'))}$ at a random point $r' \in \mathbb{F}^{\log m}$, which it can do with $\boxed{\text{one evaluation query to each } E_i}$ (the verifier can compute $\widetilde{\mathsf{eq}}(r, r')$ on its own in $O(\log m)$ time).

Surge

Description of Surge

Surge: prove $\sum_{y \in \{0,1\}^{\log N}} \tilde{M}(r, y)T[y] = v$ in roughly $O(\alpha m)$-time

(assuming each condition $E_1(i) = T_1[nz(i)]$ ... holds)

$$\sum_{j \in \{0,1\}^{\log m}} \widetilde{\mathsf{eq}}(r, j) \cdot g\left(E_1(j), \ldots, E_\alpha(j)\right). = \sum_{i \in \{0,1\}^{\log m}} \widetilde{\mathsf{eq}}(i, r) \cdot g\left(T_1[\mathsf{nz}_1(i)], \ldots, T_k[\mathsf{nz}_1(i)], T_{k+1}[\mathsf{nz}_2(i)], \ldots, T_{2k}[\mathsf{nz}_2(i)], \ldots, T_{\alpha-k+1}[\mathsf{nz}_c(i)], \ldots, T_\alpha[\mathsf{nz}_c(i)]\right).$$

5. Prove each E_i is well-formed by memory-checking procedure

The verifier must still check that each E_i is well-formed, in the sense that $\boxed{E_i(j) \text{ equals } T_i[\mathrm{dim}_i(j)]}$ for all $j \in \{0,1\}^{\log m}$. This is done exactly as in Spark to confirm that for each of the α memories, $\boxed{\mathsf{WS} = \mathsf{RS} \cup S}$

for evaluating Expression ($\boxed{15}$). Following the memory-checking procedure in Section $\boxed{4}$, with each table $\boxed{T_i \colon i = 1, \ldots, \alpha \text{ viewed as a memory of size } N^{1/c}}$, this entails committing for each i to $\log(m)$-variate multilinear polynomials $\boxed{E_i \text{ and } \mathsf{read\_ts}_i}$ (purported to capture the value and count returned by each of the m lookups into T_i) and a $\log(N^{1/c})$-variate multilinear polynomial $\boxed{\mathsf{final\_cts}_i}$ (purported to capture the final count for each memory cell of T_i.)

6. reduced to evaluate at a random point

(see Claims $\boxed{3}$ and $\boxed{4}$ and Figure $\boxed{4}$). At the end of this procedure, for each $i = 1, \ldots, \alpha$, the verifier needs to evaluate each of $\boxed{\mathrm{dim}_i, \mathsf{read\_ts}_i, \mathsf{final\_cts}_i \text{ at a random point, which it can do with one query}}$ to each. The verifier also needs to evaluate the multilinear extension $\boxed{\tilde{t}_i \text{ of each sub-table } T_i \text{ for each } i = 1, \ldots, \alpha}$ at a single point. T being SOS guarantees that the verifier can compute each of these evaluations in $O(\log(N)/c)$ time.

***T* being SOS** enables that verifier can evaluate each $\tilde{t}_i$ at a random point in $O(\log(N)/c)$ time

Surge

Surge's polynomial IOP for proving $\sum_{y\in\{0,1\}^{\log N}} \tilde{M}(r,y)T[y] = v$

Theorem 3. *Figure 5 is a complete and knowledge-sound polynomial IOP for establishing that the prover knows an $m \times N$ matrix $M \in \{0,1\}^{m\times N}$ with exactly one entry equal to 1 in each row, such that*

$$\sum_{y\in\{0,1\}^{\log N}} \widetilde{M}(r,y)T[y] = v. \tag{17}$$

T is an SOS lookup table of size N, meaning there are $\alpha = kc$ tables $T_1,\dots,T_\alpha$, each of size $N^{1/c}$, such that for any $r \in \{0,1\}^{\log N}$, $T[r] = g(T_1[r_1],\dots,T_k[r_1],T_{k+1}[r_2],\dots,T_{2k}[r_2],\dots,T_{\alpha-k+1}[r_c],\dots,T_\alpha[r_c])$. During the commit phase, $\mathcal{P}$ commits to c multilinear polynomials $\mathsf{dim}_1,\dots,\mathsf{dim}_c$, each over $\log m$ variables. dim_i is purported to provide the indices of $T_{(i-1)k+1},\dots,T_{ik}$ the natural algorithm computing $\sum_{i\in\{0,1\}^{\log m}} \widetilde{\mathsf{eq}}(i,r)\cdot T[\mathsf{nz}[i]]$ (see Equation (15)).

//$\mathcal{V}$ requests $\langle u,t\rangle$, where the ith entry of t is $T[i]$ and the yth entry of u is $\widetilde{M}(r,y)$.

1. $\mathcal{P}\to\mathcal{V}$: 2α different $(\log m)$-variate multilinear polynomials $E_1,\dots,E_\alpha$, $\mathsf{read\_ts}_1,\dots\mathsf{read\_ts}_\alpha$ and α different $(\log(N)/c)$-variate multilinear polynomials $\mathsf{final\_cts}_1,\dots,\mathsf{final\_cts}_\alpha$.
   //E_i is purported to specify the values of each of the m reads into T_i.
   //$\mathsf{read\_ts}_1,\dots\mathsf{read\_ts}_\alpha$ and $\mathsf{final\_cts}_1,\dots,\mathsf{final\_cts}_\alpha$, are "counter polynomials" for each of the α sub-tables T_i.
2. $\mathcal{V}$ and $\mathcal{P}$ apply the sum-check protocol to the polynomial $h(k) := \widetilde{\mathsf{eq}}(r,k)\cdot g(E_1(k),\dots,E_\alpha(k))$, which reduces the check that $v = \sum_{k\in\{0,1\}^{\log m}} g(E_1(k),\dots,E_\alpha(k))$ to checking that the following equations hold, where $r_z \in \mathbb{F}^{\log m}$ chosen at random by the verifier over the course of the sum-check protocol:
   - $E_i(r_z) \stackrel{?}{=} v_{E_i}$ for $i = 1,\dots,\alpha$. Here, $v_{E_1},\dots,v_{E_\alpha}$ are values provided by the prover at the end of the sum-check protocol.

In summary, it commit to a sparse vector and, establish the **sparse vector's inner product with any dense, structured (SOS) vector.**

Surge

Surge's polynomial IOP for proving $\sum_{y \in \{0,1\}^{\log N}} \tilde{M}(r, y) T[y] = v$

3. $\mathcal{V}$: check if the above equalities hold with one oracle query to each E_i.
4. // The following checks if E_i is well-formed, i.e., that $E_i(j)$ equals $T_i[\dim_i(j)]$ for all $j \in \{0,1\}^{\log m}$.
5. $\mathcal{V} \to \mathcal{P}$: $\tau, \gamma \in_R \mathbb{F}$.
   //In practice, one would apply a single sum-check protocol to a random linear combination of the below polynomials. For brevity, we describe the protocol as invoking c independent instances of sum-check.
6. $\mathcal{V} \leftrightarrow \mathcal{P}$: For $i = 1, \ldots, \alpha$, run a sum-check-based protocol for "grand products" ([Tha13, Proposition2] or [SL20, Section 5 or 6]) to reduce the check that $\mathcal{H}_{\tau,\gamma}(\mathsf{WS}) = \mathcal{H}_{\tau,\gamma}(\mathsf{RS}) \cdot \mathcal{H}_{\tau,\gamma}(S)$, where $\mathsf{RS}, \mathsf{WS}, S$ are as defined in Claim 3 and $\mathcal{H}$ is defined in Claim 4 to checking if the following hold, where $r_i'' \in \mathbb{F}^{\ell}, r_i''' \in \mathbb{F}^{\log m}$ are chosen at random by the verifier over the course of the sum-check protocol:
   - $E_i(r_i''') \stackrel{?}{=} v_{E_i}$
   - $\dim_i(r_i''') \stackrel{?}{=} v_i$; $\mathsf{read\_ts}_i(r_i''') \stackrel{?}{=} v_{\mathsf{read\_ts}_i}$; and $\mathsf{final\_cts}_i(r_i'') \stackrel{?}{=} v_{\mathsf{final\_cts}_i}$
7. $\mathcal{V}$: Check the equations hold with an oracle query to each of $E_i, \dim_i, \mathsf{read\_ts}_i, \mathsf{final\_cts}_i$.

memory-checking procedure

Question: it omits the evaluation of the MLE of each sub-table ?

verifier also needs to evaluate the multilinear extension $\tilde{t}_i$ of each sub-table T_i for each $i = 1, \ldots, \alpha$ at a single point. T being SOS guarantees that the verifier can compute each of these evaluations in $O(\log(N)/c)$ time.

***T* being SOS** enables that verifier can evaluate each $\tilde{t}_i$ at a random point in $O(\log(N)/c)$ time

Surge

Lasso lookup argument: a straightforward use of Surge

- Input: A polynomial commitment to the multilinear polynomials $\widetilde{a}\colon \mathbb{F}^{\log m} \to \mathbb{F}$, and a description of an SOS table T of size N.
- The prover $\mathcal{P}$ sends a Surge-commitment to the multilinear extension $\widetilde{M}$ of a matrix $M \in \{0,1\}^{m \times N}$. This consists of c different $(\log(m))$-variate multilinear polynomials $\dim_1, \ldots, \dim_c$ (see Figure 5 for details).
- The verifier $\mathcal{V}$ picks a random $r \in \mathbb{F}^{\log m}$ and sends r to $\mathcal{P}$. The verifier makes one evaluation query to $\widetilde{a}$, to learn $\widetilde{a}(r)$.
- $\mathcal{P}$ and $\mathcal{V}$ apply Surge (Figure 5), allowing $\mathcal{P}$ to prove that $\sum_{y \in \{0,1\}^{\log N}} \widetilde{M}(r,y)T[y] = \widetilde{a}(r)$.

Figure 6: Description of the Lasso lookup argument. Here, a denotes the vector of lookups and t the vector capturing the lookup table (Definition 1.1). A polynomial commitments to the multilinear extension polynomial $\widetilde{a}\colon \mathbb{F}^{\log m} \to \mathbb{F}$ is given to the verifier as input. If t is unstructured, then c will be set to 1.

Surge

Costs of Surge

$$\sum_{j\in\{0,1\}^{\log m}} \widetilde{\mathsf{eq}}(r,j)\cdot g\left(E_1(j),\ldots,E_\alpha(j)\right). \quad (16)$$

Prover time. Besides committing to the polynomials $\mathsf{dim}_i, E_i, \mathsf{read\_ts}_i, \mathsf{final\_cts}_i$ for each of the α memories and producing one evaluation proof for each (in practice, these would be batched), the prover must compute its messages in the sum-check protocol used to compute Expression (16) and the grand product arguments (which can be batched). Using the linear-time sum-check protocol [CTY11, Tha13, Set20], the prover can compute its messages in the sum-check protocol used to compute Expression (16) with $O(b\cdot k\cdot\alpha\cdot m)$ field operations, where recall that $\alpha = k\cdot c$ and b is the number of monomials in g. If $k = O(1)$, then this is $O(b\cdot c\cdot m)$ time. For many tables of practical interest, the factor b can be eliminated (e.g., if the *total degree* of g is a constant independent of b, such as 1 or 2). The costs for the prover in the memory checking argument is similar to Spark: $O(\alpha\cdot m + \alpha\cdot N^{1/c})$ field operations, plus committing to a low-order number of field elements.

Prover time:
- commit to polynomials
- produce evaluation proof
- compute messages in sum-check protocol
- memory checking argument

Verification costs. The sum-check protocol used to compute Expression (16) consists of $\log m$ rounds in which the prover sends a univariate polynomial of degree at most $1+\alpha$ in each round. Hence, the prover sends $O(c\cdot k\cdot\log m)$ field elements, and the verifier performs $O(k\cdot\log m)$ field operations. The costs of the memory checking argument (which can be batched) for the verifier are identical to Spark.

Verifier time:
- sum-check protocol
- memory checking argument

Completeness and knowledge soundness of the polynomial IOP. Completeness holds by design and by the completeness of the sum-check protocol, and of the memory checking argument.

By the soundness of the sum-check protocol and the memory checking argument, if the prover passes the verifier's checks in the polynomial IOP with probability more than an appropriately chosen threshold $\gamma = O(m + N^{1/c}/|\mathbb{F}|)$, then $\sum_{y\in\{0,1\}^{\log N}} \widetilde{M}(r,y)T[y] = v$, where $\widetilde{M}$ is the multilinear extension of the following matrix M. For $i\in\{0,1\}^{\log m}$, row i of M consists of all zeros except for entry $M_{i,j} = 1$, where $j = (j_1,\ldots,j_c)\in\{0,1,\ldots,N^{1/c}\}^c$ is the unique column index such that $j_1 = \mathsf{dim}_1(i),\ \ldots,\ j_c = \mathsf{dim}_c(i)$.

Comparison of Lasso's costs

Scheme	Proof size	Prover work group, field	Verifier work
Plookup [GW20b]	$5\mathbb{G}_1, 9\mathbb{F}$	$O(N)$, $O(N\log N)$	$2P$
Halo2 [BGH20]	$6\mathbb{G}_1, 5\mathbb{F}$	$O(N)$, $O(N\log N)$	$2P$
Caulk [ZBK+22]	$14\mathbb{G}_1, 1\mathbb{G}_2, 4\mathbb{F}$	$15m$, $O(m^2 + m\log(N))$	$4P$
Caulk+ [PK22]	$7\mathbb{G}_1, 1\mathbb{G}_2, 2\mathbb{F}$	$8m$, $O(m^2)$	$3P$
Flookup [GK22]	$7\mathbb{G}_1, 1\mathbb{G}_2, 4\mathbb{F}$	$O(m)$, $O(m\log^2 m)$	$3P$
Baloo [ZGK+22]	$12\mathbb{G}_1, 1\mathbb{G}_2, 4\mathbb{F}$	$14m$, $O(m\log^2 m)$	$5P$
cq [EFG22]	$8\mathbb{G}_1, 3\mathbb{F}$	$7m + o(m)$, $O(m\log m)$	$5P$
Lasso w/ Dory (SOS table)	$O(\log(m))\ \mathbb{G}_T$ $\tilde{O}(\log(m))\ \mathbb{F}$	$o(cm + cN^{1/c})$, $O(cm)$ $O(\sqrt{m})\ P$	$O(\log(m))\ \mathbb{G}_T$ $\tilde{O}(\log(m))\ \mathbb{F}$
Lasso w/ Dory (unstructured table)	$O(\log m)\ \mathbb{G}_T$ $\tilde{O}(\log(m))\ \mathbb{F}$	$\min\{2m + O(\sqrt{N}), m + o(N)\}$, $O(m+N)$ $O(\sqrt{N})\ P$	$O(\log m)\ \mathbb{G}_T$ $\tilde{O}(\log(m))\ \mathbb{F}$
Lasso w/ Sona (SOS table)	$\tilde{O}(\log(m))\ \mathbb{F}$ $O(1)\ \mathbb{G}$	$o(cm + cN^{1/c})$, $O(cm)$	$\tilde{O}(\log(m))\ \mathbb{F}$ $O(1)\ \mathbb{G}$
Lasso w/ Sona (unstructured table)	$\tilde{O}(\log(m))\ \mathbb{F}$ $O(1)\ \mathbb{G}$	$\min\{2m+O(\sqrt{N}), N\}$, $O(m+N)$	$\tilde{O}(\log(m))\ \mathbb{F}$ $O(1)\ \mathbb{G}$
Lasso w/ KZG+Gemini (SOS table)	$O(\log m)\ \mathbb{G}_1$ $\tilde{O}(\log(m))\ \mathbb{F}$	$(c+1)m + cN^{1/c}$, $O(m)$ $O(\log m)\ \mathbb{G}_1$	$\tilde{O}(\log(m))\ \mathbb{F}$ $2P$
Lasso w/ KZG+Gemini (unstructured table)	$O(\log m)\ \mathbb{G}_1$ $\tilde{O}(\log(m))\ \mathbb{F}$	$(c+1)m + cN^{1/c}$, $O(m+N)$ $O(\log m)\ \mathbb{G}_1$	$\tilde{O}(\log(m))\ \mathbb{F}$ $2P$

Figure 7: Dominant costs of prior lookup arguments vs. our work. Sona is the polynomial commitment scheme proposed in this work (Section 1.5). Other cost profiles for our schemes are possible by using other polynomial commitments. **Notation**: m is the number of lookups, N is the size of the lookup table. We assume $N \geq m$ for simplicity. For verification costs only, we assume that $m \leq \text{poly}(N)$, so that $\log m = \Theta(\log N)$. The notation $\tilde{O}(\log m)$

Notation:

- m: number of lookups
- N: size of the lookup table
- Assume $N \geq m$ for simplicity.
- For verification costs only
  - assume $m \leq \text{poly}(N)$
  - so that $\log m = \Theta(\log N)$
- For prover work
  - m "group work" for prover refers to a multiexponentiation of size m
  - m "exps" refers to m group exponentiations
  - c dentoes an arbitrary positive integer
- P refers to pairing operations

Efficient properties in Lasso

Described in Abstract

- For m lookups into a table of size n, Lasso's prover commits to just $m + n$ field elements. Moreover, the committed field elements are *small*, meaning that, no matter how big the field $\mathbb{F}$ is, they are all in the set $\{0, \ldots, m\}$. When using a multiexponentiation-based commitment scheme, this results in the prover's costs dominated by only $O(m + n)$ group *operations* (e.g., elliptic curve point additions), plus the cost to prove an evaluation of a multilinear polynomial whose evaluations over the Boolean hypercube are the table entries. This represents a significant improvement in prover costs over prior lookup arguments (e.g., plookup, Halo2's lookups, lookup arguments based on logarithmic derivatives).

- Unlike all prior lookup arguments, if the table t is structured (in a precise sense that we define), then no party needs to commit to t, enabling the use of much larger tables than prior works (e.g., of size 2^{128} or larger). Moreover, Lasso's prover only "pays" in runtime for table entries that are accessed by the lookup operations. This applies to tables commonly used to implement range checks, bitwise operations, big-number arithmetic, and even transitions of a full-fledged CPU such as RISC-V. Specifically, for any integer parameter $c > 1$, Lasso's prover's dominant cost is committing to $3 \cdot c \cdot m + c \cdot n^{1/c}$ field elements. Furthermore, all these field elements are "small", meaning they are in the set $\{0, \ldots, \max\{m, n^{1/c}, q\} - 1\}$, where q is the maximum value in a.