A Sharded PIR Design for the Ethereum State

A practical path to private reads of Ethereum data

Ali Atiia
Lead — Privacy of Reads
Ethereum Foundation
privreads.ethereum.foundation

Workshop on Cryptographic Tools for Blockchains
Rome, May 2026

Ethereum users routinely query state data from remote servers

The content and the patterns of these queries leak privacy

a curious/malicious server can profile a user just by tracking what they *read* — undermining other privacy measures the user has worked hard to follow (eg shielding assets).

The edge reads, providers see

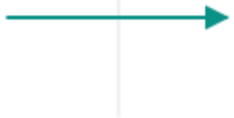
wallets

frontends

dApps / SDKs

light clients

RPC calls



balance nonce allowances

contract code merkle roots

they don't or can't hold the full state, so they query remote providers

RPC provider

Sees: *which* address, *which* slot, *which* block, *when*, *from where*.

— the privacy boundary leaks at the read.

A read is a fingerprint

- Holdings inferred from *which* balances and slots you poll
- On-chain ↔ off-chain identity correlated by IP, timing, query mix
- Frontrunning & MEV: a wallet polling a liquidation price is a tell
- Behavioral profile assembled over time, even without any tx broadcast

Reads defeat the rest of the stack.

Shielding your transactions doesn't help if the read pattern alone tells the same story.

What does “the edge” actually read?

Hot state

“What is my balance now?”

`eth_getBalance`

`eth_getStorageAt`

`eth_call`

`eth_getCode`

Txs & blocks

“Did my tx land?”

`eth_getBlockByNumber`

`eth_getTxByHash`

`eth_getTxReceipt`

Historical state

“What was my balance on Dec 31?”

`eth_getBalance(@blockN)`

`eth_getStorageAt(@blockN)`

archival state

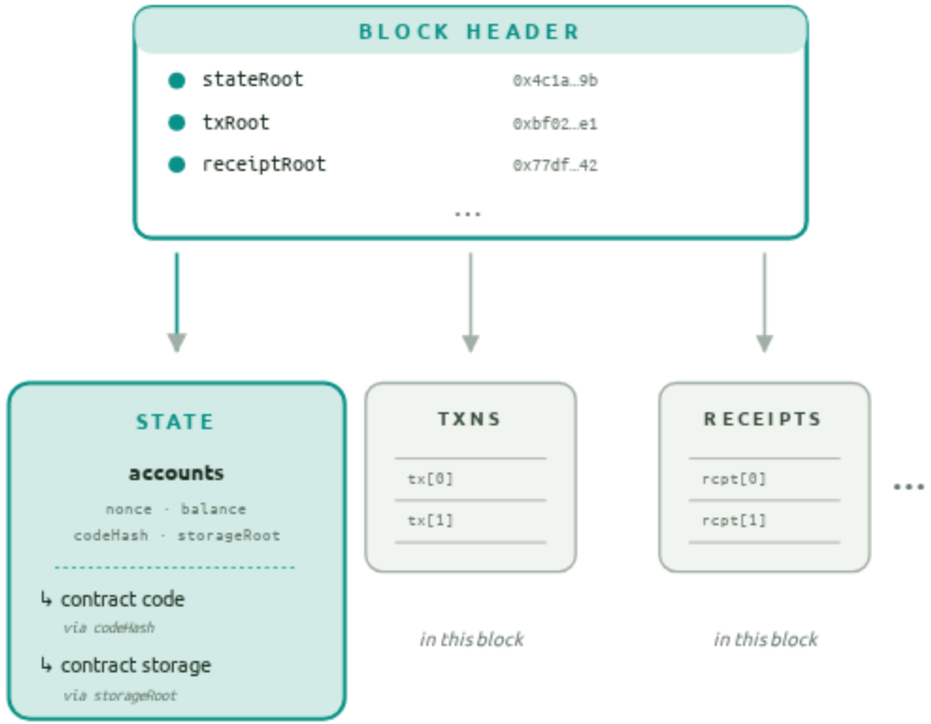
(“data warehouse” in traditional db lingo)

Answer: **a bit of everything** — any complete privacy story has to cover all three.

The Ethereum state

The Ethereum state, briefly

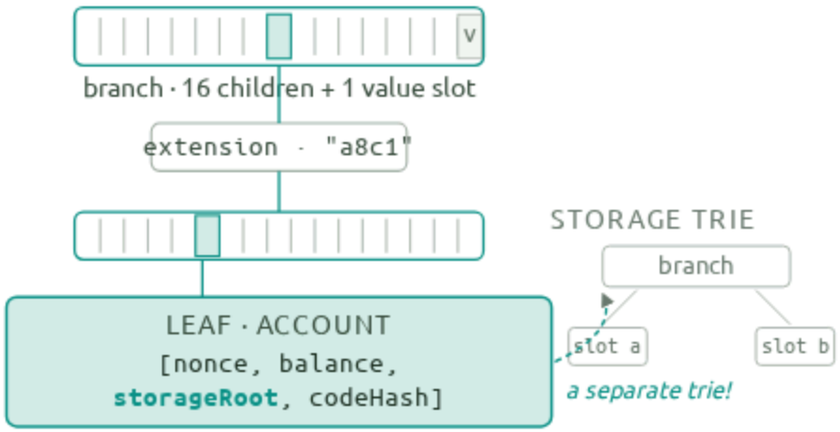
- A **key→value** store of accounts; code and storage live one indirection deeper
- Committed under the **state root** (a Merkle-Patricia trie) in every block header
- Every read is **verifiable** with Merkle roots to the state root in the block header



Reading a value can entail fetching a *merkle proof* anchored to the state root in the block header — if the user wants to independently verify the value (and soon, the zkVM proof of that header).

MPT will be upgraded from, eventually

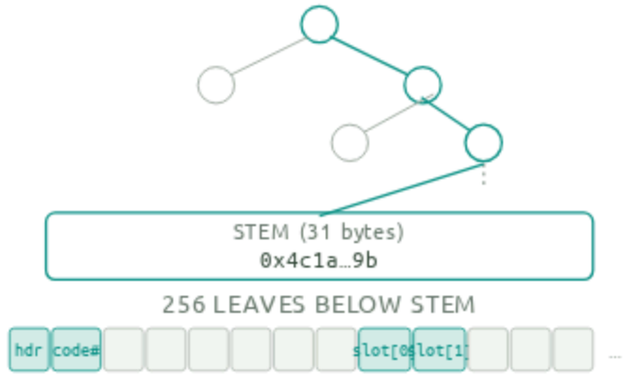
MPT — today arity 16



Disadvantages of MPT

- Heavy proofs — **15 siblings × 32 B** per level
  *typical **~2.4–2.9 KB** (depth 5–6) · worst-case **~3.8 KB** (depth ~8)*
- Worst-case **stateless proof per block ~300 MB**
- Storage slots require a *second trie traversal*
- Keccak — slow in zkVMs

UBT — EIP-7864 (draft) arity 2



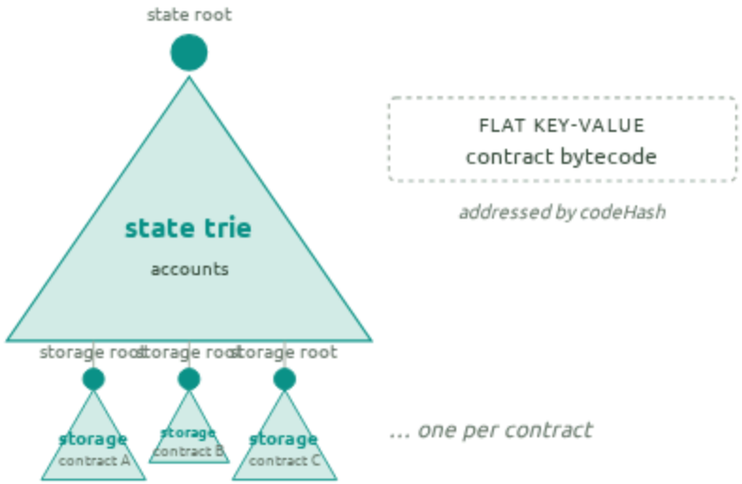
Advantages of UBT

- Lean proofs — **1 sibling × 32 B** per level → **~1 KB** (~4× smaller)
- Single trie — storage & code under one root
- Snark-friendly hash (BLAKE3 / Poseidon — *exact choice TBD*) → **3–100×** proving speedup
- Page-based locality — meaningful **gas savings** under Verkle/UBT witness pricing for storage-heavy dApps

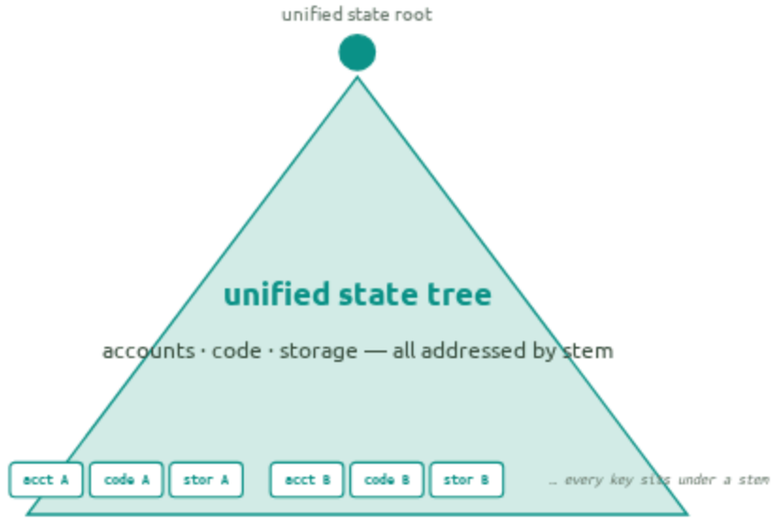
We are using **UBT** as the source DB for PIR, with a **zk proof** of its equivalence to mainnet MPT.

Many roots today, one tomorrow

MPT — many trees, many roots



UBT — one tree, one root



What "tree of trees" costs

- One **state root** + many **storage roots** + a flat code store. Three commitment regimes side by side.
- A balance proof = path through state trie. A storage-slot proof = **two** paths.

What unification buys

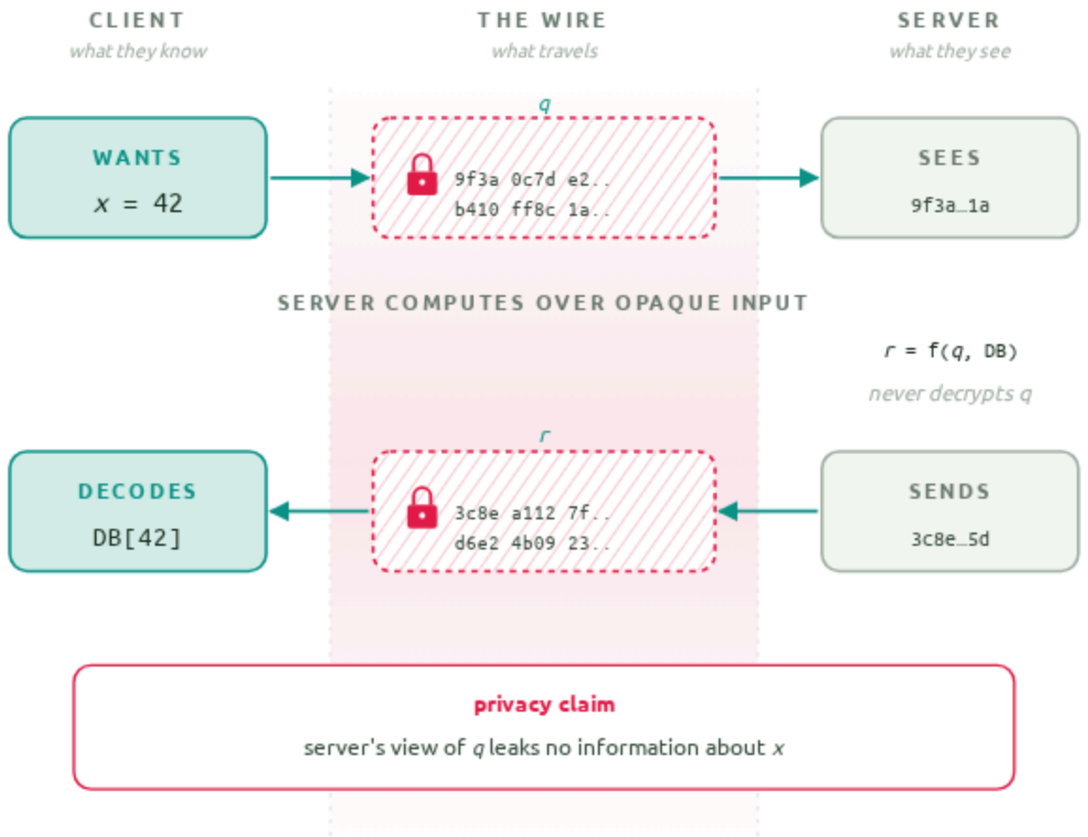
- One root, one keyspace. Account, code chunks, storage slots all addressed by stem.
- For PIR: **one universal interface, one slicing schema**.

Private Information Retrieval (PIR)

PIR at a glance

Private Information Retrieval: read $\mathrm{DB}[x]$ without revealing x.

- Server holds DB; client wants $\mathrm{DB}[x]$
- Client sends a query q that *encodes the index of x under a* ***cryptographic veil***
- Server computes a response r — learning nothing about x
- Client decodes r to get $\mathrm{DB}[x]$



A one-hot vector picks one row

- A vector q of length N
- Zero everywhere except a single 1 at index x
- Inner-product against any vector DB —
- … selects exactly $\mathrm{DB}[x]$, zeroing the rest.

q		DB		$q[i] \cdot \mathrm{DB}[i]$
0	$\times$	$\mathrm{DB}[0]$	$=$	0
0	$\times$	$\mathrm{DB}[1]$	$=$	0
1	$\times$	$\mathrm{DB}[x]$	$=$	$\mathrm{DB}[x]$
0	$\times$	$\mathrm{DB}[3]$	$=$	0
$\vdots$		$\vdots$		$\vdots$
0	$\times$	$\mathrm{DB}[N-1]$	$=$	0
		sum $=$		$\mathrm{DB}[x]$

We can send q to the server while hiding the "1"

Single-server PIR: encrypt the one-hot

- Client encrypts each entry of q under SHE/FHE.
- Server computes $\sum_i q[i] \cdot \mathrm{DB}[i]$ **homomorphically** — an inner product on ciphertexts.
- Client decrypts the response and recovers $\mathrm{DB}[x]$.

$\mathrm{enc}(q)$		$\mathrm{enc}(\mathrm{DB})$		$q[i] \cdot \mathrm{DB}[i]$
$\mathrm{enc}(0)$	$\times$	$\mathrm{enc}(\mathrm{DB}[0])$	$=$	$\mathrm{enc}(0)$
$\mathrm{enc}(0)$	$\times$	$\mathrm{enc}(\mathrm{DB}[1])$	$=$	$\mathrm{enc}(0)$
$\mathrm{enc}(1)$	$\times$	$\mathrm{enc}(\mathrm{DB}[x])$	$=$	$\mathrm{enc}(\mathrm{DB}[x])$
$\mathrm{enc}(0)$	$\times$	$\mathrm{enc}(\mathrm{DB}[3])$	$=$	$\mathrm{enc}(0)$
$\vdots$		$\vdots$		$\vdots$
$\mathrm{enc}(0)$	$\times$	$\mathrm{enc}(\mathrm{DB}[N-1])$	$=$	$\mathrm{enc}(0)$
sum $=$ $\mathrm{enc}(\mathrm{DB}[x])$				

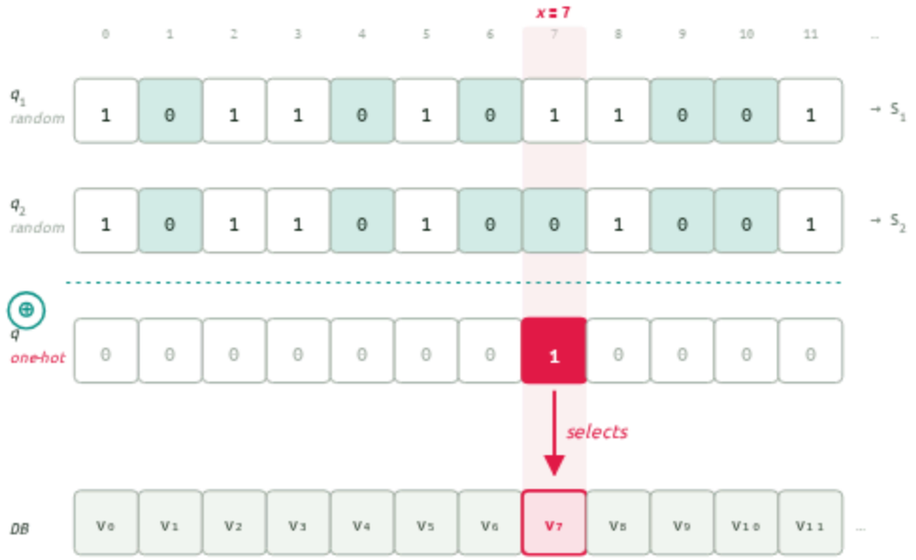
Privacy here is *computational* — it relies on the **semantic security** of the encryption.

The catch: the server must touch every record to compute that sum — cost is $O(N)$ per query. Skipping any record would leak that the client didn't want it.

Two non-colluding servers

PIR with XOR'ing

1. Client wants index x; encodes it as a one-hot vector q over $[N]$.
2. Splits q into two random XOR shares: $q = q_1 \oplus q_2$. Each share alone is uniform random.
3. Sends q_1 to S_1, q_2 to S_2. Each server XORs the DB entries selected by its share.
4. Client XORs the responses: $r_1 \oplus r_2 = \mathrm{DB}[x]$.



EACH SHARE ALONE IS UNIFORM RANDOM

S_1 sees random bits; S_2 sees random bits.

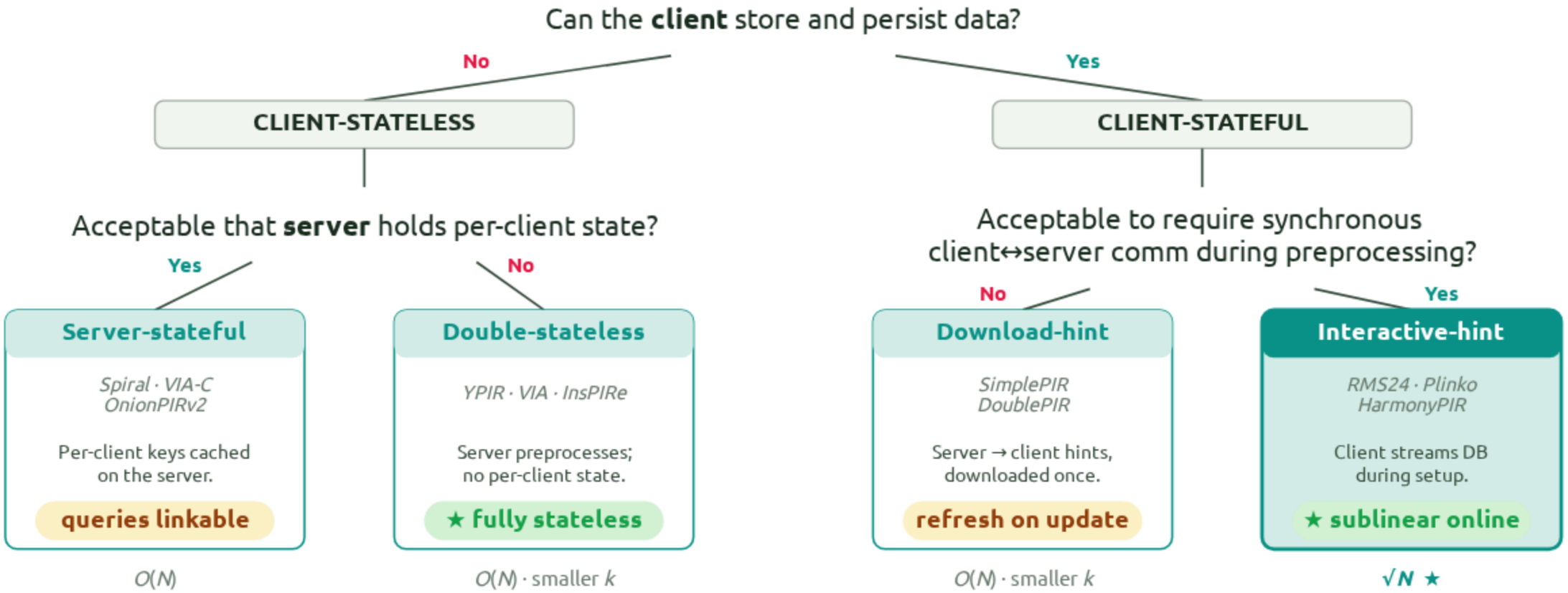
Only the XOR of both reveals *x*.

Privacy here is *information-theoretic* — it follows from how the query is split, not from any encryption.

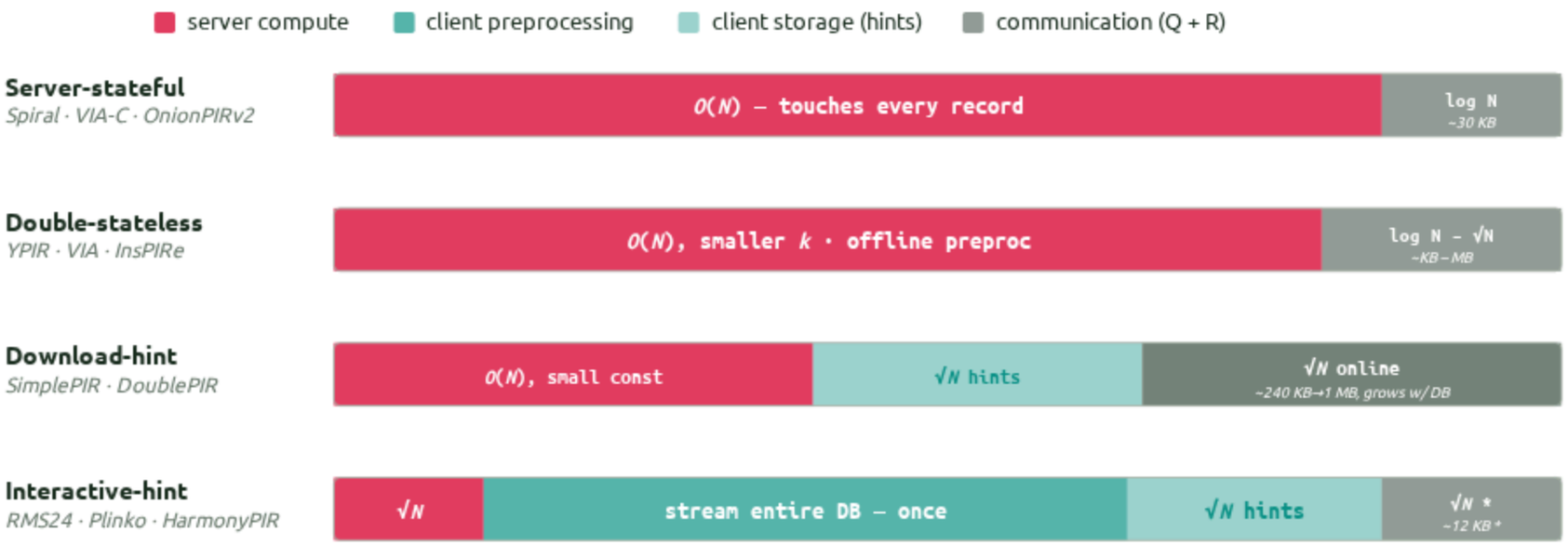
Currently we are focused exclusively on single-server schemes

The Performance Tradeoffs of PIR Schemes

The design space



The $O(N)$ wall, and how preprocessing breaks it



Different cost shapes — and online comm itself differs by an order of magnitude.

Download-hint pays $\sqrt{N}$ per query online; * interactive-hint is small *after* a one-time DB-stream setup.

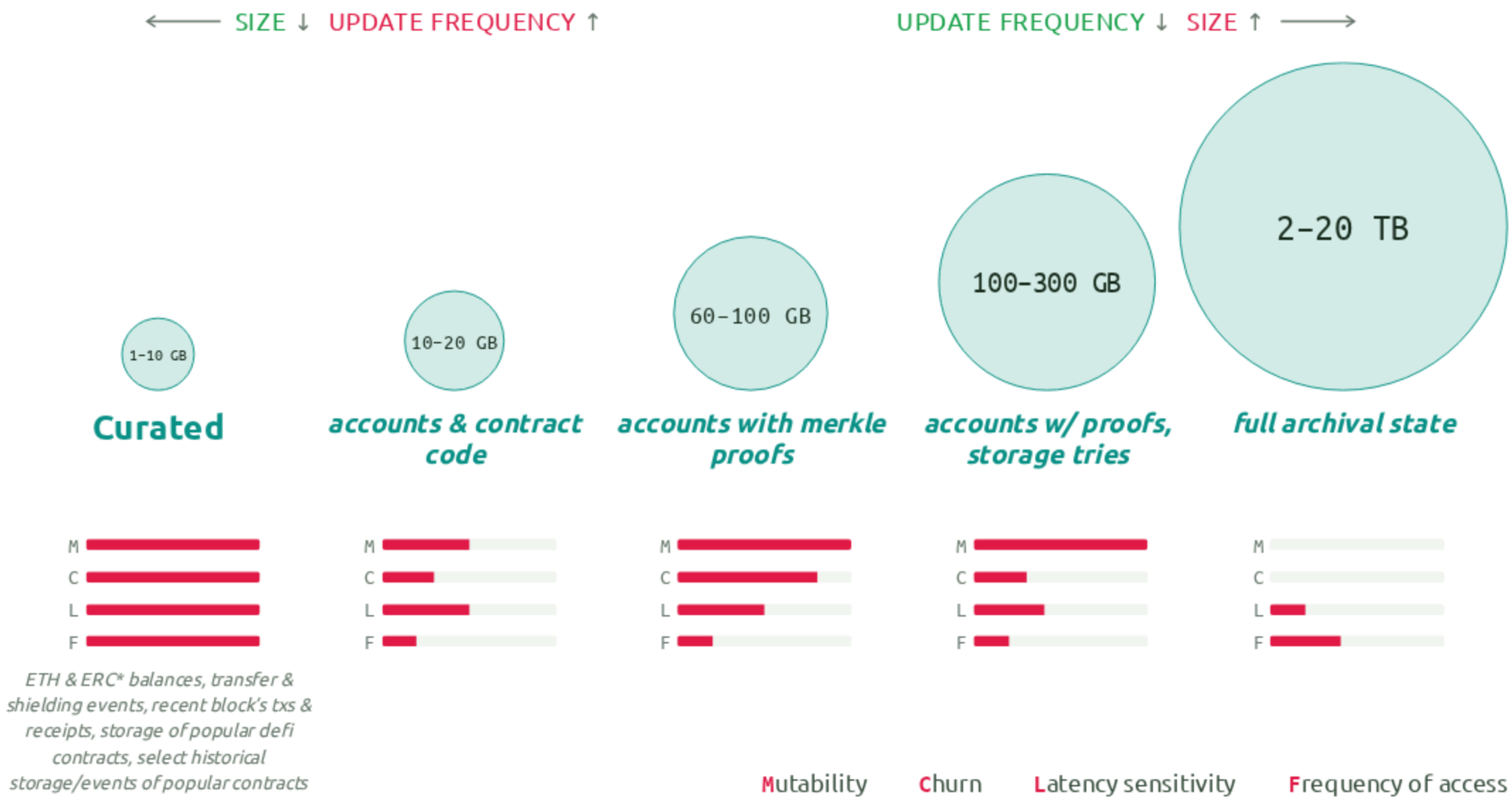
Concrete: 1 GB DB · OnionPIRv2 ~30 KB · InsPIRe ~400 KB · SimplePIR 240 KB (→ ~960 KB at 16 GB) · Plinko ~12 KB. Source: PIR Tutorial & Survey, Table II.

No single PIR scheme dominates

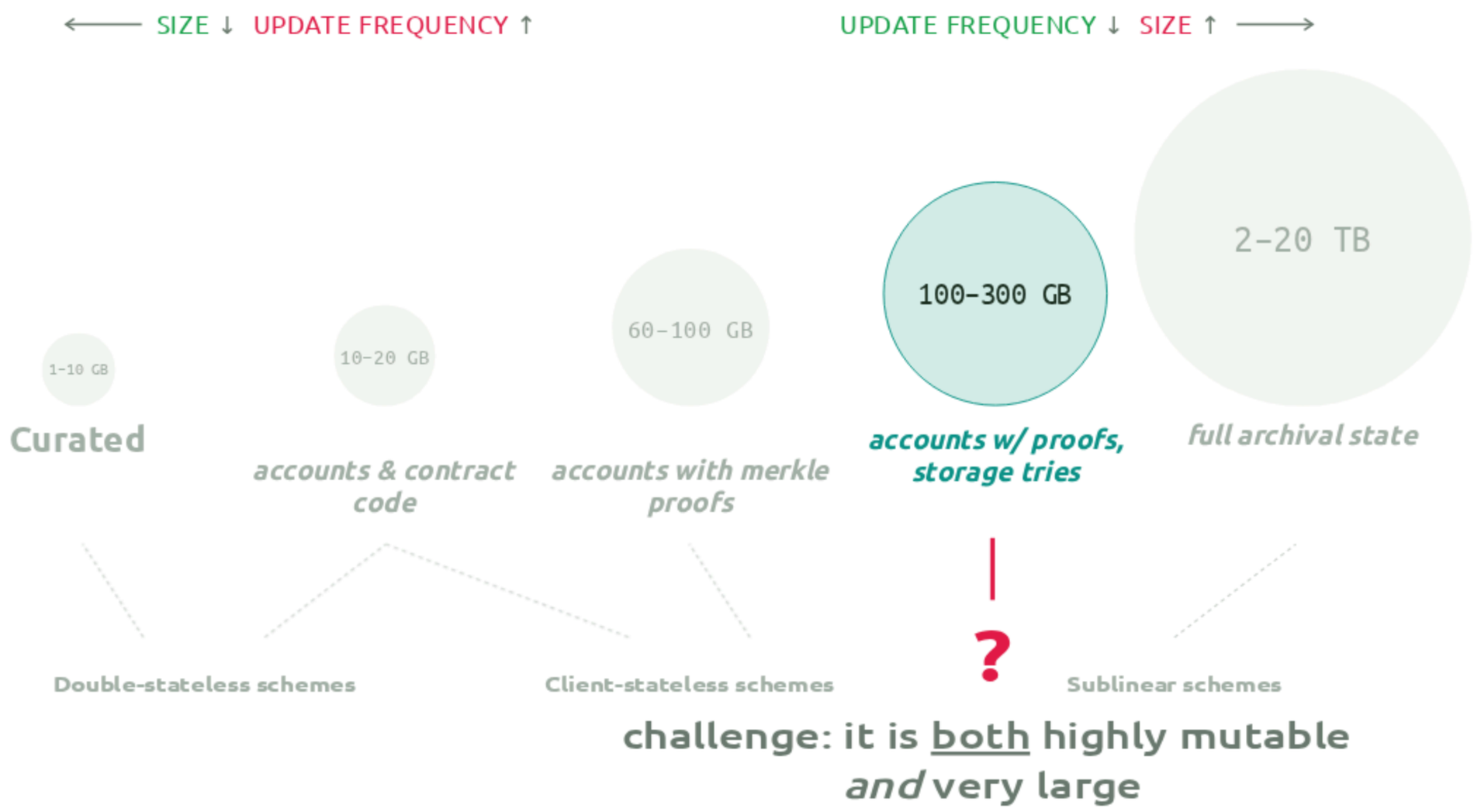
FAMILY / EXAMPLES	ONLINE SERVER COST	CLIENT STORAGE	PER-CLIENT SERVER STATE	UPDATE COST / FRESHNESS	ONLINE COMM
Server-stateful *Spiral · VIA-C · OnionPIRv2*	!	●	●	●	●
Double-stateless *YPIR · VIA · InsPIRe*	!	●	●	●	●
Download-hint *SimplePIR · DoublePIR*	!	!	●	●	●
Interactive-hint *RMS24 · Plinko · HarmonyPIR*	●	!	!	!	●

Sharding

Sharding the Ethereum State



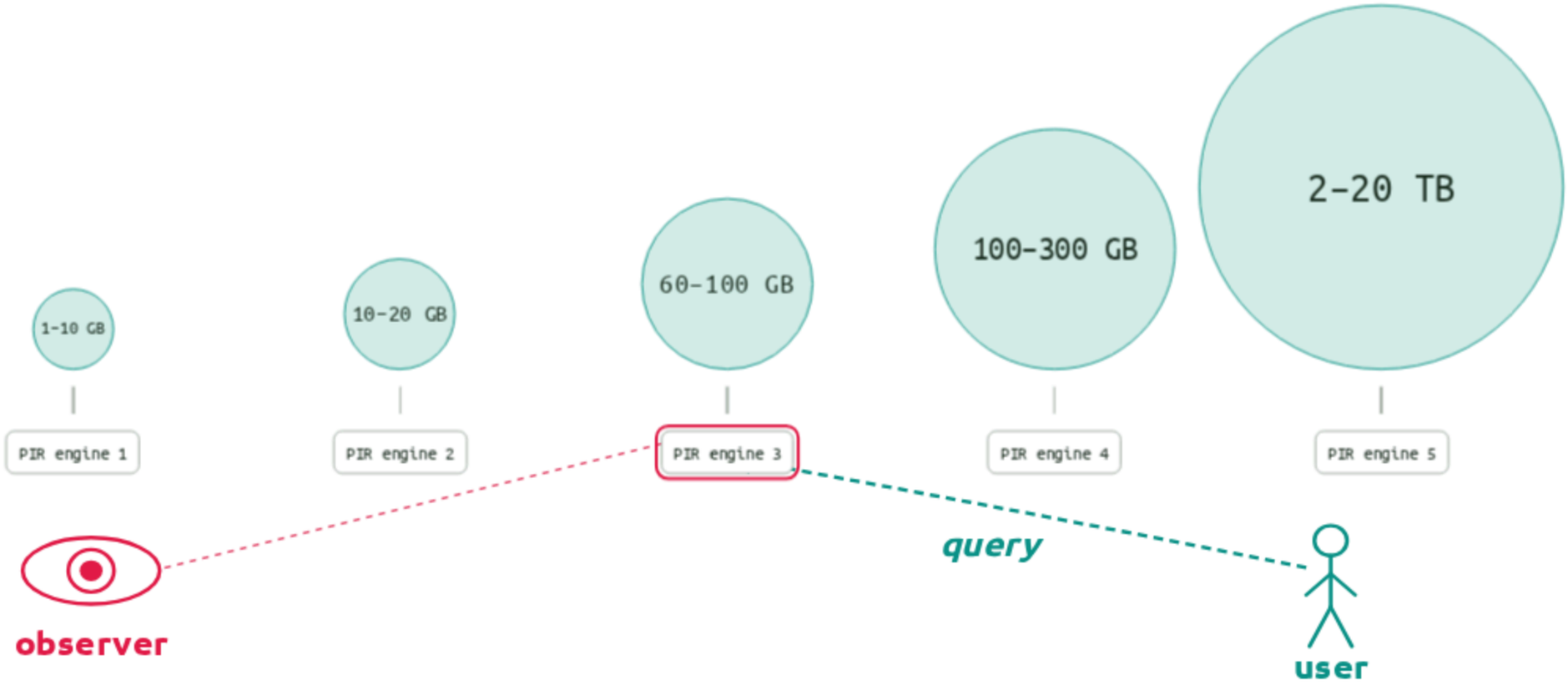
Sharding the Ethereum State



challenge: it is <u>both</u> highly mutable *and* very large

Knowing *which slice* → leakage

which slice is being queried **leaks privacy** — the server knows the content of each slice, and can make correlations over time

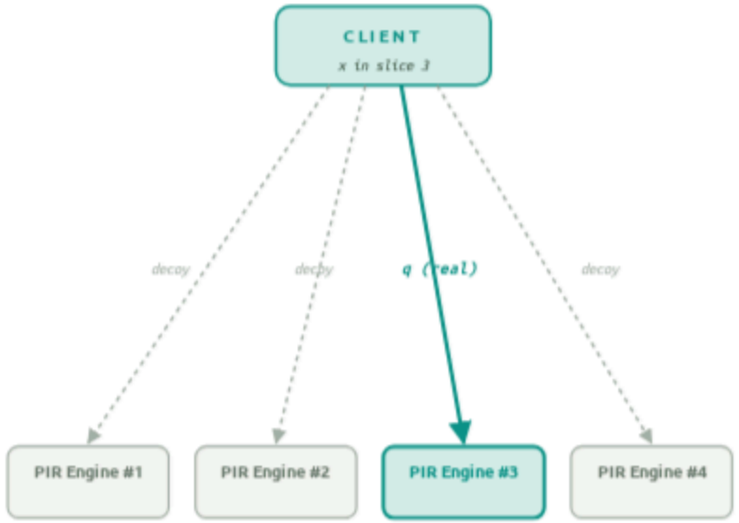


Genuine + decoy queries = full privacy

What the client knows vs. what the network observer sees.

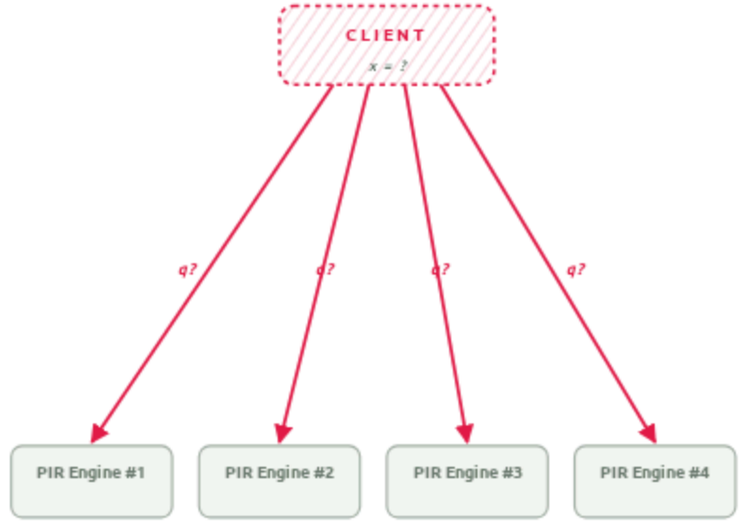
Client's view — ground truth

labelled · asymmetric · 1 real + $k-1$ decoys



Observer's view — on the wire

all rays identical · no labels · uniformly distributed



Observer's view = monolithic PIR over the whole state. Performance = per-slice optimized.

PRIVACY Decoys are *real* PIR queries — not blanks. They cost $k\times$ wire, but each shard sees a uniform access just as before.

The wire is safe, but what about the clock

Decoys hide *which slice* — per-slice *response times* can give it away.

The failure mode

- Response time is different across PIR engines.
- If the client **acts** as soon as it receives the response of the *real* query, the observer correlates time → recovers *which slice*.

Mitigations

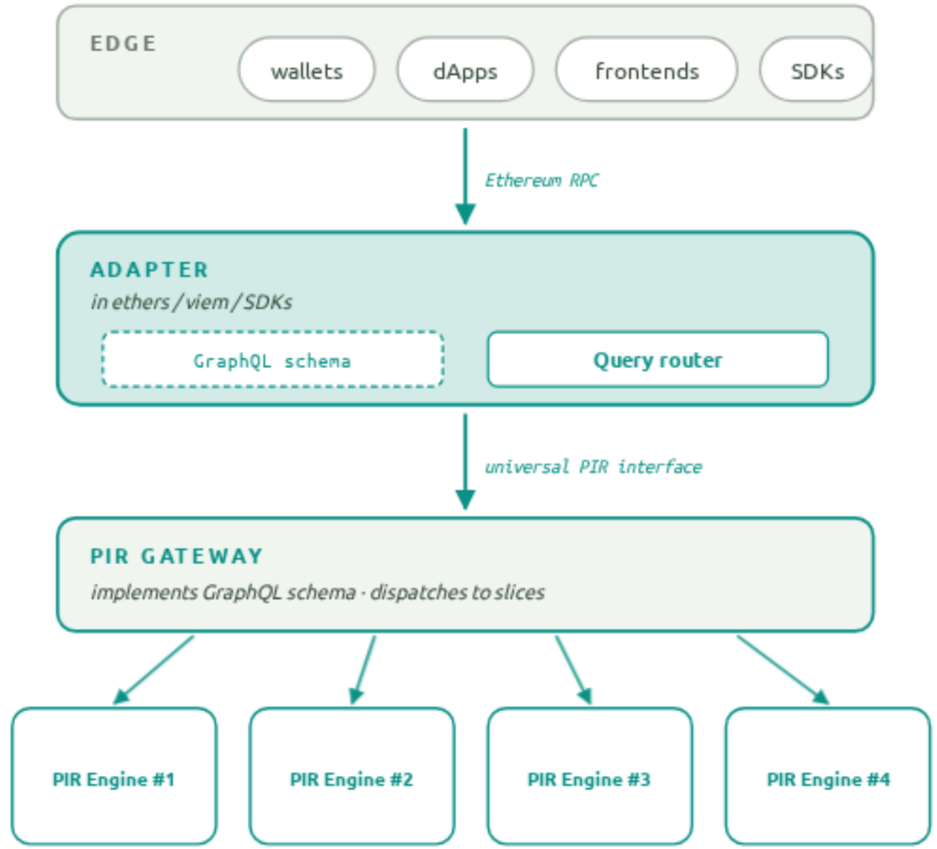
- **Wait-for-all**: client doesn't act until *every* shard has responded.
- m**-of-**k: tolerate slowest few; discard their slices this round.
- **Constant background traffic**: client always queries every slice.

Middleware

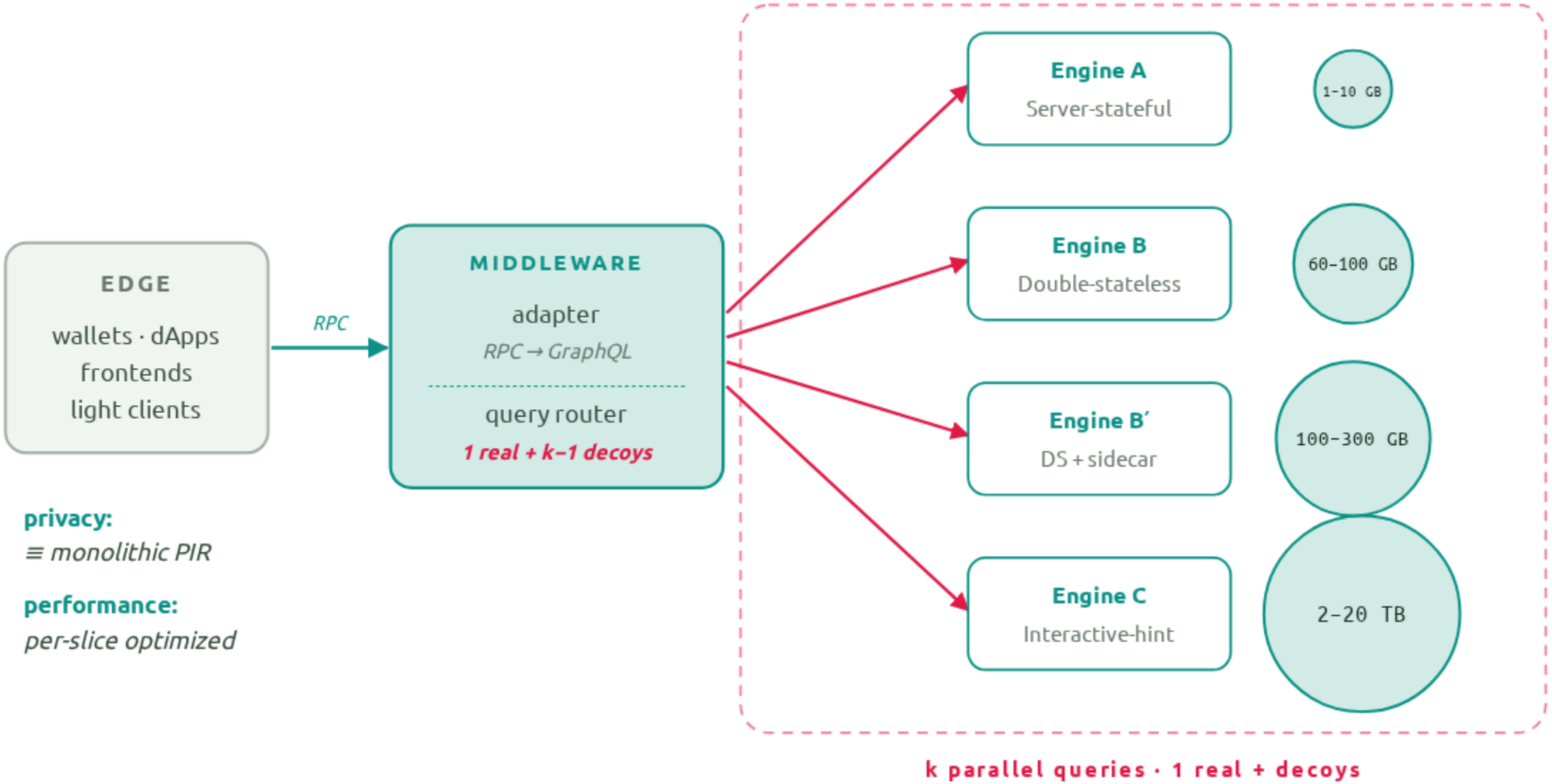
A universal PIR interface

The edge keeps speaking standard Ethereum RPC. The PIR backend evolves independently.

- **Edge unchanged.** Wallets, dApps, light clients keep speaking `eth_getBalance`, `eth_getProof`, …
- **Adapter** in the SDK (ethers/viem) translates RPC into a *GraphQL-shaped* PIR query plan.
- **Query router** constructs *k* queries (1 real + decoys), dispatches per-slice.
- **Decoupling**: PIR families and slice boundaries evolve under the interface.



Summary of Sharded Design



privacy:
≡ monolithic PIR

performance:
per-slice optimized

Optimizations

Three levers to scale sharded PIR

LEVER 1

Shrink the DB

Reduce what each PIR engine has to scan per query.

→ **PIR for Merkle proofs** — serve proofs, not nodes

→ **SNARKify archival roots** — serve proofs of stem nodes instead of Merkle proofs

LEVER 2

Slice smarter

Carve slices along their natural seams — especially mutability.

→ **Sidecar pattern** — isolate the mutable tail

→ **Verifiable UBT equivalence** — capitalize on the uniformity of binary tries

LEVER 3

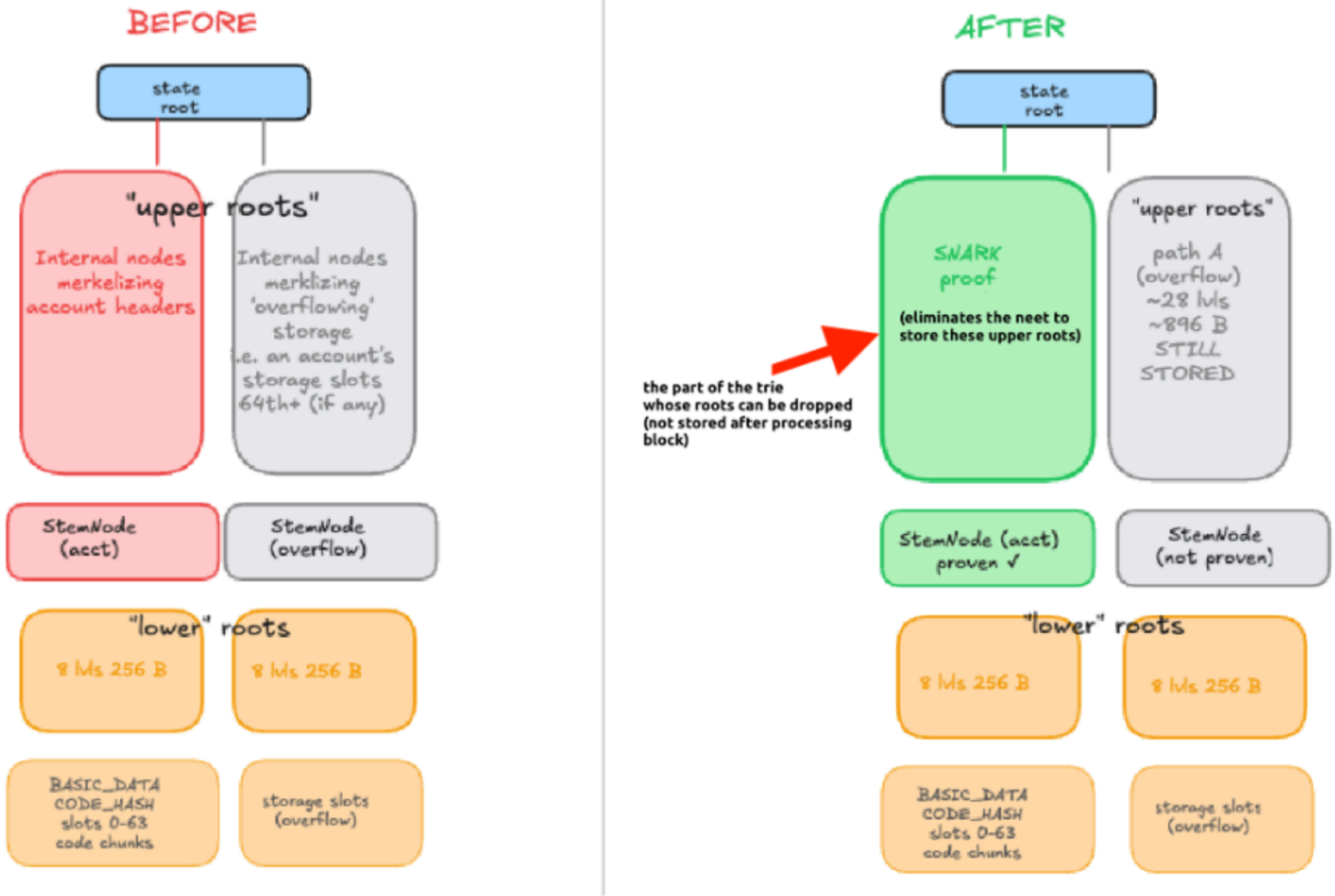
Build better schemes

Push the per-slice scheme harder — especially toward GPU-native designs.

→ **Double-stateless GPU-tailored R&D**

→ **Delegated hints · DEPIR to the rescue?**

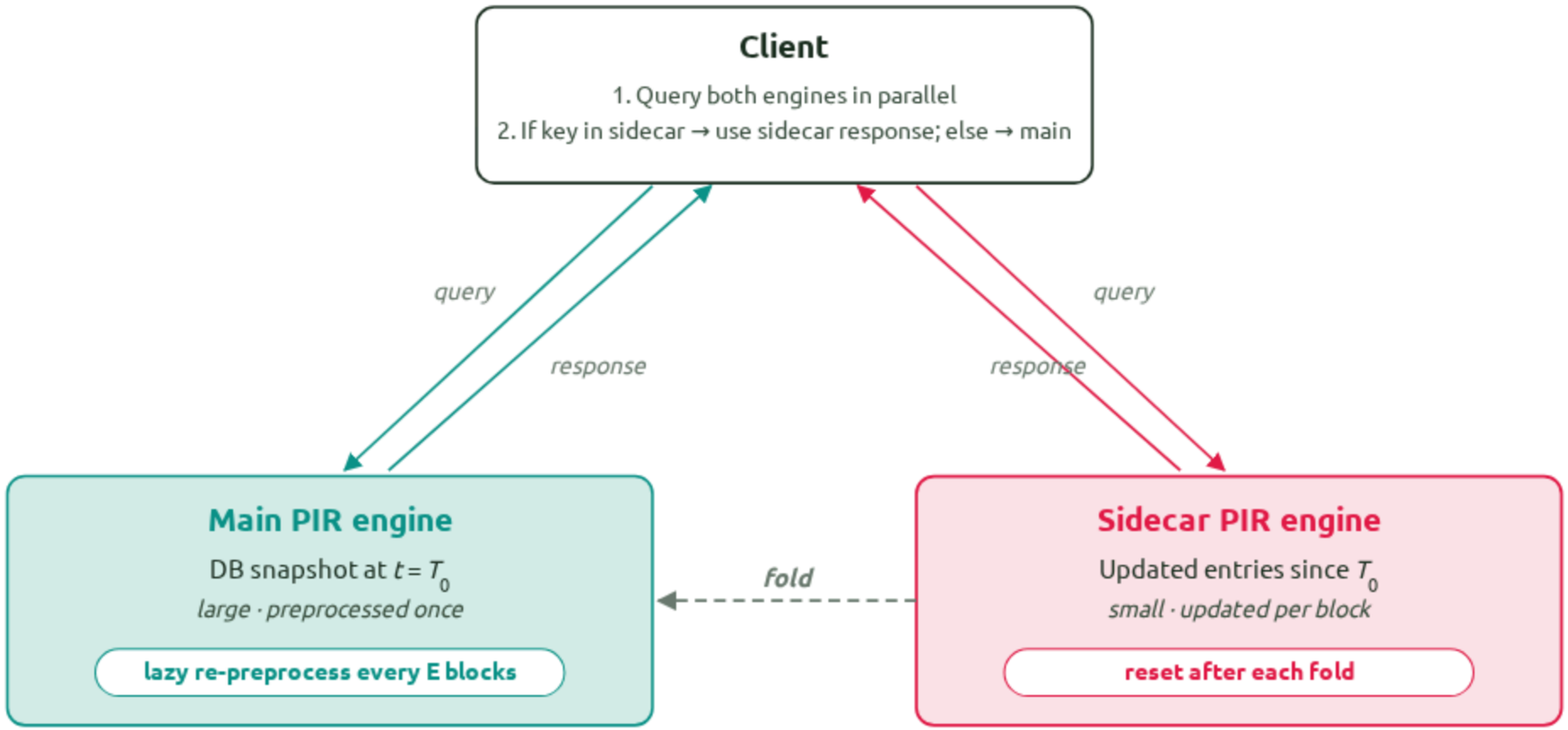
Snarkifying away merkle roots → reduce db size massively



What gets dropped: upper trie levels — most-mutated, biggest by volume. Lower nodes & leaves stay; PIR continues unchanged.

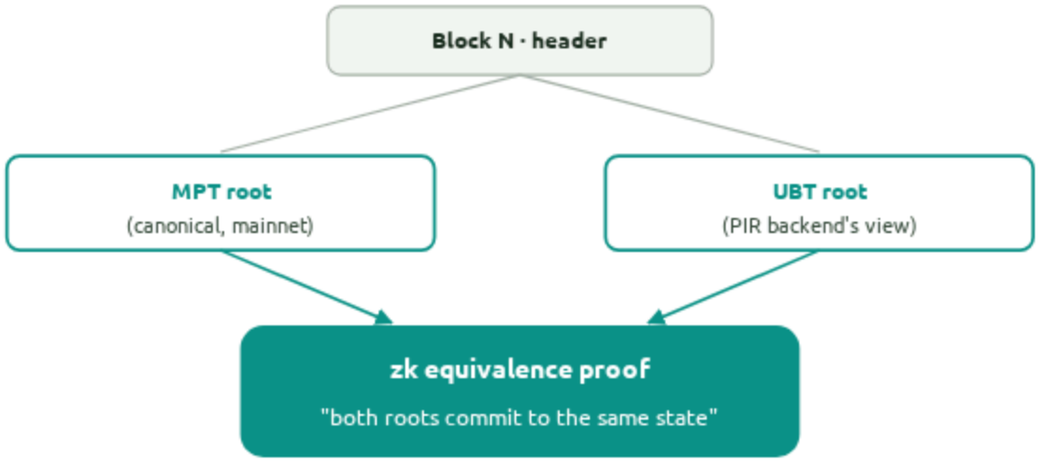
Why it pays: **~50–80% archival DB-size reduction** (§8 of the ethresear.ch post). Smaller DB → cheaper PIR per query.

Isolate mutability — the sidecar pattern



Big snapshot stays cold; fresh writes live in a small, fast engine; client always sees the freshest answer.

Verifiable UBT ↔ MPT equivalence




```
∀ (key, value): (key, value) ∈ MPT  ⇔  (key, value) ∈ UBT
```

Why this anchors everything

- PIR backend can use UBT *before* it lands on mainnet.
- SNARKification (slide 30) operates on the UBT side.
- One verification surface for all PIR engines — the UBT root.

Cost shape

- One proof per block, generated server-side; clients verify it once.
- SNARK-friendly UBT hash (BLAKE3 / Poseidon—*TBD*) keeps prover work tractable.

> **IN PROGRESS** initial UBT-enabled **Geth** / **Ethrex** nodes in testing right now.

Researching double-stateless GPU schemes

Active in-house research direction: PIR schemes where *both* client and server are stateless — and the server kernel is shaped to be GPU-native from the start.

Why double-stateless?

- ✓ **No per-client server state** — server scales horizontally without client tracking.
- ✓ **No client hint to keep fresh** — client carries only its keys.
- ✓ **Clean trust model** — every client looks the same to the server.
- ✓ **Updates are clean** — no broadcast hint to invalidate, no per-client preproc to redo.

Family includes e.g. HintlessPIR, YPIR, VIA, InsPIRe. Online server work is still $O(N)$ — but with much smaller constants than FHE-PIR.

Why GPU-tailored?

- ▸ **Server work is dense linear algebra** — matrix–vector multiply over the DB.
- ▸ **Batch friendly** — many concurrent queries amortize one DB pass.
- ▸ **Small modular params** fit in GPU L2 cache — bandwidth-bound, not compute-bound.
- ▸ **Stateless server** means many GPUs can serve the same DB without coordination.

Designing the scheme around the GPU memory hierarchy — not just porting an existing scheme to CUDA — is what unlocks order-of-magnitude gains. (See Part 4 for current GPU PIR numbers.)

RESEARCH Open target: match or beat GPIR throughput on Ethereum-shaped DBs while staying double-stateless. Construction details forthcoming.

Progress, ongoing work, references

Progress

- Correctness-focused specs of **Plinko**, **RMS24**, **VIA** schemes.
- Speccing Plinko surfaced the **invertible PRF** as an intractable bottleneck before months of impl work.
- **GPU acceleration** of insPIRe scheme.
- Reproducing benchmarks of existing schemes.

Ongoing work

- New **double-stateless** GPU-tailored schemes — can they hit GPIR throughput while keeping the cleaner trust model?
- Universal PIR middleware spec & wallet integration.
- SNARKification cost analysis (binary trie).

References

- Sharded PIR design — the ethresear.ch post.
- GPIR (latest) — SOTA GPU acceleration of PIR.
- PIR-Eng-Notes — per-scheme papers, specs, engineering notes.
- PIR benchmarks & PIR specs on Private Reads.

Thank you