

10

Look it up without asking anyone

The lookup table is an object in the network, so an answer comes with a proof instead of a promise.

BUILD STATE

Yours to take

COMPLEXITY

●●●●○

Vision

Taking something off the web usually only means taking it off a list. A page a search engine stops returning is still online, at the same address, on the same server, and its publisher may never learn that anything happened. It has stopped being somewhere anyone arrives.

Europe's right-to-be-forgotten delistings work exactly this way, and the fact that they work at all is the part worth sitting with. Whatever one thinks of the remedy, it establishes where the leverage is: a page one company's index no longer returns has been taken off the web for most practical purposes, and nobody touched the page. The same shape turns up wherever a catalogue stands in front of a collection. An app pulled from a store keeps running on every phone that already has it and becomes unreachable for everyone else. A seller dropped from a marketplace's search still has a storefront, at a URL nobody types.

Systems built to escape this have mostly moved the storage and left the catalogue where it was. The files live in a network nobody operates. The thing that says what is in the network runs on one machine, is maintained by one team, and answers every query anyone asks. Whoever answers the queries decides what exists. Taking the bytes out from under a company while leaving the lookup in place moves the problem without changing its shape, and it tends to leave the part of the design nobody hardened in the position that decides what anyone finds.

On Swarm the lookup table can be content too. The index is a set of pieces with an address of its own, so anyone who would rather answer their own queries fetches it and does, and anyone who would rather ask someone else can check the answer that comes back. The answer carries its own arithmetic: this entry is in the index whose address is this number, and here is the path from that number to this entry. Anyone can redo the sum. So can a smart contract that has never held anything but the number.

What that removes is the index operator as a role anybody has to trust. An index that returns a wrong answer is caught by the first person who checks. An index whose operator loses interest is picked up by anyone holding the address. Two people asking the same question of the same index get the same answer, or one of them can demonstrate that the other is being lied to. None of this removes the work: somebody still writes the entries, somebody still pays to keep them stored, and an index nobody tends goes stale like anything else. What changes is that tending an index and being believed about it stop being the same job.

Architecture

An index here is a set of entries keyed by exactly 32 bytes. One entry sits at each node of a trie, each node is one Swarm chunk, and the whole index comes down to a single 32-byte reference: the address of its root node. Everything below follows from those two facts.

Branching on a measure the network already uses. Swarm's own topology runs on proximity order, the position of the first bit at which two addresses differ, which is how the network decides which nodes are responsible for which chunks. A proximity order trie applies that measure to keys instead of node addresses. Bee already ships a different index, the manifest: a mantaray trie that branches on the next character of a path, which is what makes a website resolve on Swarm and what the access control grantee list is kept in. Paths suit names people type. Keys that are hashes spread evenly across the space, and a trie branching on the first differing bit stays shallow over them without any rebalancing step. In a local run over a thousand random keys, a lookup took about three chunk fetches on average, and opening the index could cost one fetch, for the root.

One number doing two jobs. The root reference is the Swarm address of the root node. It is what a Bee node is given to download the index, and it is what a contract stores as its commitment to the contents, and they are the same 32 bytes. Set that against the usual arrangement for proving membership of a large set on chain: a Merkle root in the contract, and the proofs served as a JSON file from the project's web host, so the contract is permanent while the ability to claim against it lasts exactly as long as somebody keeps paying for the bucket. When the commitment and the retrieval address coincide, anybody can fetch the index, rebuild any proof in it, and act on the contract without asking the party that published it.

How a proof is built. The node layout is deliberate: 32 bytes of key, then a 256-bit map of which forks are present, then the fork references at 32 bytes each, then the entry count under each fork, then the value. Everything lands on a 32-byte boundary because that is the segment size of the Binary Merkle Tree over a chunk. A proof shows that a given fork reference sits at a given segment of a chunk with a given address, and that the matching bit is set in the map. Chain one of those per level from the root down to the entry and the result is a fork path proof. `P0TPProofVerifier.sol` walks the chain and reverts on the first link that does not hold, so a contract holding 32 bytes can act on the contents of an index it has never stored and could not afford to store.

Editing is cheap because nothing is overwritten. Content addressing means an edit rewrites only the nodes whose bytes changed; every untouched node keeps the address it had and is shared with the previous version. In a local run over a thousand entries, adding one wrote two new chunks, and updates and deletes of existing keys averaged under two. Postage covers the new chunks only. Old roots go on resolving, so every version of the index remains an address rather than a backup, and writing the current root into a feed gives the index a stable name whose update history is its version log.

What it answers, and the half that does not work yet. Exact lookup is solid: across local runs from two to a thousand random keys, every key inserted was found. Iteration is not. Iterate walks a subtree in ascending XOR distance from a target key, so a walk from the all-zero key comes out in ascending key order, which is the scan any query layer would be built on. What it actually returns is a partial answer, with no error raised: each node's traversal begins one proximity order past the node's own cursor, so the subtree hanging at that position is never entered, and where keys are spread evenly across the space that is roughly half of them. Local runs returned 485 entries out of 1000, 31 out of 64, and on a hand-built five-entry trie, four. Every skipped key was still retrievable by `Find`, so the entries are there and the walk is at fault. The library's own iteration test packs small integers into the leading bytes, a key shape under which the skipped position is always empty, which is why it passes. Until that traversal is fixed this is a key-value store with proofs rather than something to query. Beyond it there are no joins and no ordering by value, and the library proves that a key is present while proving one absent, which is what would catch an index that had left something out, is not implemented.

The root is a fingerprint of history, not of contents. Insertion policy is a parameter of the structure: the mode decides whether a newly written entry descends into the trie or gets promoted toward the root, and the mode the key-value store ships with promotes it, which is why recent writes are near the top and why edits are as cheap as they are. The cost shows up as history dependence. The same two hundred keys inserted forwards and backwards produced two different root references in a local run, both answering every lookup correctly. Neither index is wrong; they are simply not the same object. Any design that needs two parties to build the same set independently and arrive at the same address needs a mode that fixes the shape from the keys alone, and that mode is open work.

Retrieval says something about the question. Reading an entry means fetching the chunks along its path, and those addresses are visible to the peers that forward the request. Swarm's forwarding conceals who is asking considerably better than it conceals what is being asked, so a reader who wants a private lookup fetches more of the index than they need, or all of it.

Maturity. The library is a standalone Go module in the ethersphere organisation, based on @zelig's implementation on Bee's pottery branch, last pushed in May 2026, with a Solidity verifier that has hardhat tests and a deployment module but no deployed address. It is not part of a Bee release, so nothing in Bee's API answers a lookup against it, and there is no TypeScript port. At the time of writing the repository carries no licence file, which is a smaller problem than it sounds and a real one for anyone planning to depend on it.

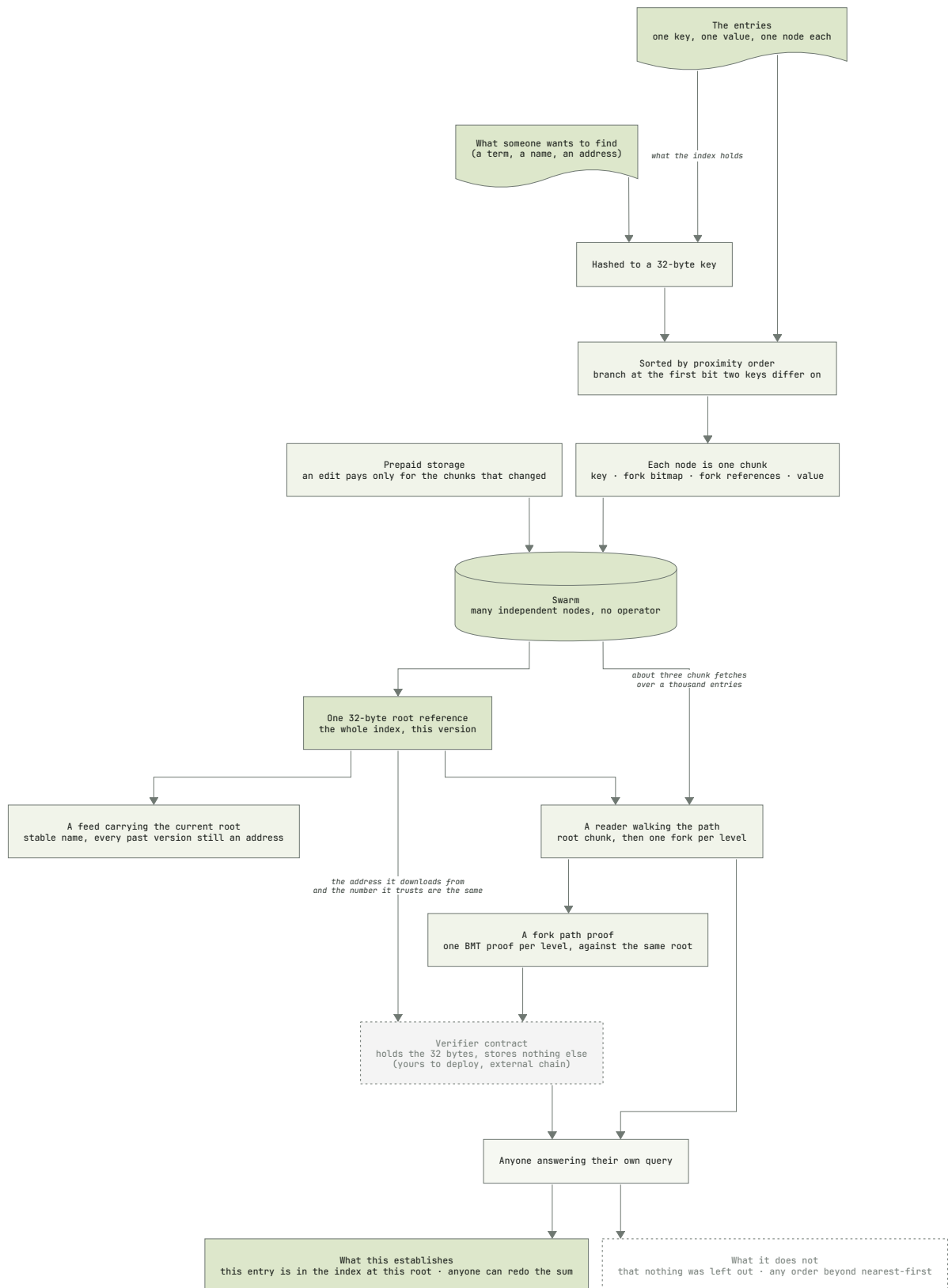
PRIMITIVES

Content addressing Chunks / BMT proofs Feeds / Single Owner Chunks Postage stamps

Manifests

SWARM ECOSYSTEM Proximity Order Trie POTProofVerifier.sol bmt-js

EXTERNAL DEPENDENCIES An EVM chain, if proofs are checked on chain



Data flow. What someone wants to find is hashed to a 32-byte key and the entries are sorted by proximity order, the first bit two keys differ on. Each node of the trie is one chunk holding a key, a map of which forks exist, the fork references and the value, and the whole index reduces to one 32-byte root. A feed carries the current root; an edit pays only for the chunks that changed. A reader walks the path from the root, and the same walk produces a fork path proof a verifier contract checks against the very number it downloads from. The last two boxes separate what this establishes from what it does not.

Development

The store is three calls. Point a load-saver at a Bee node and a funded postage batch, and each node is written with POST /bytes under a Swarm-Postage-Batch-Id header and read back by reference.

```
ls := persister.NewSwarmLoadSaver("http://localhost:1633", batchID)
kvs, _ := pot.NewSwarmKvs(ls)
defer kvs.Close()

kvs.Put(ctx, key32, value)
root, _ := kvs.Save(ctx) // 32 bytes: the whole index
again, _ := pot.NewSwarmKvsReference(ctx, ls, root)
```

Keys are 32 bytes and the value rides in the node's chunk. Marshalling rejects any key that is not exactly 32 bytes, so hash whatever is being keyed on. The value is appended to the node's own serialization, and the prover works over a single 4096-byte chunk, which is the budget a node has to stay inside if it is to remain provable: 32 for the key, 32 for the fork bitmap, 36 for each fork present, and the remainder for the value. Store a Swarm reference and keep the payload outside. An index of references is the useful shape anyway, since it is what turns a search term into a posting list, or an address into a record.

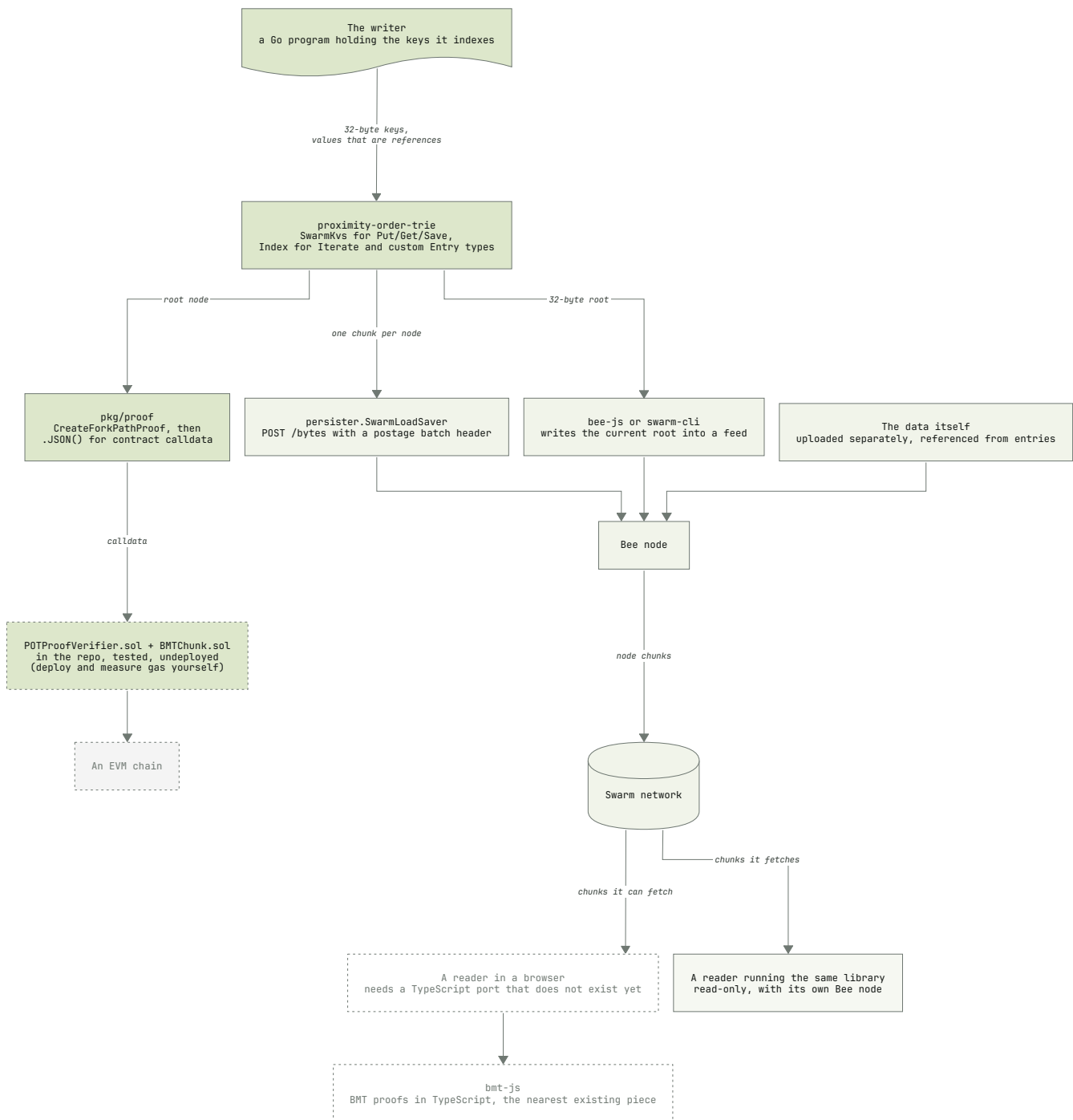
Index is the fuller API, and it is a goroutine rather than a mutex. Constructing one starts a loop that hands the current root to any number of readers and to one writer at a time; Add, Find, Update, Delete, Iterate and Size go through it and Close stops it. Iterate takes two keys, a prefix that selects the subtree to walk and a target that sets the order, walking outwards from the target, and under evenly spread keys it silently returns about half the entries, so treat it as unusable until the traversal is fixed. Entry is an interface, so implementing Key, Value, Equal, MarshalBinary and UnmarshalBinary lets a custom struct be the value instead of a byte slice.

Proofs and the contract. proof.CreateForkPathProof(ctx, rootNode, ls, key) walks from the root to the entry, and ForkPathProof.JSON() emits the hexified structure the verifier takes as calldata. The Solidity is in blockchain/: POTProofVerifier.sol over BMTChunk.sol, a hardhat test that runs a recorded sample proof, and an Ignition module. Nothing is deployed, so deployment is yours, and so is measuring the gas, which should happen early: proof cost per lookup is the number the on-chain half of any design here stands or falls on.

Mutability is a feed. Every Save yields a root for that version; a feed under a key you control gives the sequence of roots a name that does not change. Readers resolve the feed to get the current root, and because every earlier root still resolves, a reader who wants a fixed view pins one and stays on it.

The browser is the missing piece. The library is Go and there is no TypeScript port, so a page cannot walk the trie or check a proof without one, and bee-js has MantarayNode for manifests with no POT equivalent beside it. Nothing in Bee's HTTP API answers a lookup either, which means the reading side today is a Go program with a Bee node next to it, and the promise of the whole design, that a reader verifies for themselves rather than trusting whoever answered, is out of reach for a reader on a phone. Two pieces of work sit in front of everything else here: repairing the traversal so a scan returns every entry, and then that port.

- proximity-order-trie — <https://github.com/ethersphere/proximity-order-trie> (Go; the library, its blockchain/ verifier and the fork path proofs; no licence file at the time of writing)
- bee pottery branch — <https://github.com/ethersphere/bee/tree/pottery> (@zelig's original POT implementation)
- Manifests — <https://docs.ethswarm.org/docs/develop/tools-and-features/manifests> (the mantaray trie Bee ships, and the right structure for anything keyed on paths)
- Chunk types — <https://docs.ethswarm.org/docs/develop/tools-and-features/chunk-types>
- Kademlia and proximity order — <https://docs.ethswarm.org/docs/concepts/DISC/kademlia>
- Bee Feeds documentation — <https://docs.ethswarm.org/docs/develop/tools-and-features/feeds>
- Postage stamps — <https://docs.ethswarm.org/docs/concepts/incentives/postage-stamps>
- Bee API specification — <https://docs.ethswarm.org/api/> (POST /bytes, the call the load-saver writes nodes with)
- bee-js — <https://bee-js.ethswarm.org> (feeds and uploads from TypeScript; MantarayNode is here, a POT equivalent is not)
- swarm-cli — <https://github.com/ethersphere/swarm-cli>
- **Swarm ecosystem, not Bee primitives:** bmt-js <https://github.com/fairDataSociety/bmt-js>, the nearest existing piece to a browser-side verifier
- **External dependencies (not Swarm):** an EVM chain to deploy P0TProofVerifier.sol on, if proofs are checked on chain



Component assembly. A Go writer holds the keys it indexes and drives the library, which serializes one chunk per node through a load-saver that posts to a Bee node under a postage batch. The data itself is uploaded separately and referenced from the entries. The root goes into a feed so the index has a name that does not change. The proof package turns a walk from the root into contract calldata, and the verifier contract in the repository is tested and undeployed. Under the network is the reading side: a Go program today, and a browser once somebody writes the TypeScript port.

Reference implementations

Yours to take. The library exists, works and is tested. Nothing verifiable has been built on it.

Proximity Order Trie (ethersphere) — <https://github.com/ethersphere/proximity-order-trie> — the Go implementation described throughout this document, based on @zelig's version on Bee's pottery branch. Swarm-persisted nodes, a key-value store over them, a mutable index with proximity-ordered iteration, fork path proofs, and a Solidity verifier with hardhat tests and an Ignition deployment module. Last pushed May 2026. `go test ./...` passes, but its iteration test uses a key shape that conceals the traversal defect described above; lookup, update, delete, persistence and the proofs all behave correctly under random keys. No deployed contract, no TypeScript port, and no licence file, which is the first thing to fix for anyone intending to depend on it.

Bee manifests (ethersphere, shipped) — the mantaray trie in `pkg/manifest`, which is the index Swarm ships and uses: path resolution for websites, and the grantee list behind access control, where `pkg/accesscontrol/kvs` is a key-value store built on a manifest. Worth reading before choosing, because for anything keyed on paths it is the right structure and it is already in every Bee node. The case for the proximity order trie is the proof system and the hash-keyed workload, not the lookup on its own.