

11

Ask for everything and know you got it

A query comes back as a set, a count and a proof for each, so a missing row is something you catch rather than suspect.

BUILD STATE

Yours to take

COMPLEXITY



Vision

Anything presented as a complete list is really two claims at once: that what is on it belongs there, and that nothing is missing. The first is checkable item by item. The second is not checkable at all.

Most of the time the gap goes unnoticed, because nothing turns on it. It appears the moment somebody has a reason to shorten the list. A search returning nine results looks exactly like a search that returned nine of eleven. A marketplace showing forty sellers looks exactly like one showing forty of the fifty who matched. Nothing in the answer separates the two cases, and the only party in a position to know is the party that produced it. Records offices have understood this for a long time, which is why “no responsive documents were found” is the reply that ends an inquiry rather than advancing it.

This is the last trusted position in systems that have worked hard to remove every other one. An application reading a blockchain through an indexer can check that each event it was handed really happened, because each one is on the chain and each one is signed. It cannot check that it was handed all of them. Verification covers the items and stops at the edge of the set. So a stack that is scrupulous about every individual fact still rests on one act of faith, and it happens to be the one that decides what anybody sees.

An index that lives on Swarm has a shape that can be counted, and the counts are checkable in the same way the entries are. An answer can then arrive as three things rather than one: the entries that matched, the number that should have matched, and a proof for each entry. Count what arrived, check the number, check the proofs. A short answer stops being something to suspect and becomes something to detect, which is a different kind of thing to live with.

None of this decides what a good answer is. Ranking, relevance, which of forty sellers belongs at the top: those are judgements, and no proof reaches them. What changes is narrower and awkward to give up once you have had it. Whoever answers the question loses the ability to shorten the list without leaving evidence. Whether they put the right thing first stays an argument, and arguments are survivable. The silent omission was never an argument, because nobody could see it happen.

Architecture

This space stands on the proximity order trie of space 10, and it is written forward rather than from working code. Two things are being assumed, and they are different sizes.

What is assumed, and what is merely unbuilt. The library’s subtree walk begins one proximity order past each node’s own cursor, so it never enters the fork sitting at that position; with keys spread evenly across the space it returns roughly half of any scan and reports no error. Everything below assumes that walk visits every entry. It is an off-by-one in one function, nothing in the stored format depends on it, and it is unfixed at the time of writing. Separately, and more substantially, none of the proof work described here exists: absence proofs, count proofs, secondary indexes and the planner are all design. The distinction matters, so it is marked at every step. What is genuinely shipped is the part that carries the weight: the node format already stores what these proofs would need, in the places a proof can reach.

The counts are already in the format (shipped). Every node serializes, beside its fork references, a four-byte entry count for each fork: how many entries hang in the subtree at that proximity order. This is what lets the index report its size from one lookup instead of a traversal, and in local runs the figure stayed exact through a thousand inserts, a hundred deletes and repeated updates of existing

keys. Those counts sit inside the 32-byte-aligned region of the node, which is to say inside the part of the chunk a Binary Merkle Tree proof can address. Four bytes is smaller than a segment, so a proof would cover a segment carrying up to eight counts and the verifier would read the four it wants. Nobody has written that proof. The bytes it needs are already there, in the right place, under the right hash.

Absence is a rearrangement of presence (partly shipped). To show that a key is not in the index, walk the path a lookup walks and stop where the lookup stops: at a node whose fork bitmap has the bit cleared at the proximity order that key requires. The bitmap is bytes 32 to 64 of the node, a whole segment of its own, so proving its contents against the chunk address is a single BMT proof. `P0TProofVerifier.sol` already does this, in the negative: one of its revert conditions is that a fork is not set in the parent's bitvector. The contract knows how to read the bit. What is missing is a caller that treats a cleared bit as the thing being proved rather than as an error. This is not a new cryptosystem; it is the existing one pointed the other way.

A complete answer is a set and a number. Put those two together and a query over a subtree returns three things: the entries, a proof of the count at that subtree's root, and a path proof for each entry. Whoever asked checks every proof, counts what arrived, and compares. Entries left out make the two numbers disagree, and there is nowhere to hide the discrepancy, because the count came out of the same chunk as the fork reference the walk followed to get there. What stays possible is refusing to answer, which is a visible act in a way that an answer with nothing to say about its own shortness is not.

Predicates become tries of their own. A query engine over this keeps a trie per queryable field, keyed on the hash of the field and value together, whose entries are posting lists: the primary keys carrying that value. A conjunction of predicates intersects posting lists. Because each posting list is addressed like everything else, an intersection is a computation whose every input is named by hash. This is ordinary secondary-index design and none of it is written.

The planner's statistics are exact, and they can be proved. Here it stops resembling a conventional database. A planner chooses an execution order from estimates of how many rows each predicate will match, and conventionally those estimates come from sampled histograms which are approximate by construction and stale between refreshes; a planner that guesses badly is the ordinary cause of a query that takes a thousand times longer than it should. In this structure the number of entries under any branch is stored, exact, readable in a single chunk fetch, and provable by the same machinery as the entries. A planner reads a count per predicate, orders the intersection cheapest first, and can hand over the figures it planned on. Somebody checking the work can confirm the plan was consistent with the statistics it claims to have used. Whether it was the best plan available is a separate question, and not one any proof answers.

A plan is content. Every input is named by hash, so every intermediate is too, and a plan reduces to a small graph of references: these index roots, these posting lists, this intersection order. That graph has an address. It can be published, cached and cited, and re-running it against the same roots gives anybody the same answer. So a cached result stops requiring trust in whoever cached it, since the cached answer carries the proofs the original carried, and a query result becomes something that can be pointed at rather than something that has to be asked for again.

Snapshots come free; concurrent writers do not. Reads happen against a root, and a root is a version, so a query spanning several indexes pins one root each and has a consistent snapshot without any further machinery, which is a thing conventional databases spend real effort on. Writing

runs the other way. The mutable index hands its current root to any number of readers and to one writer at a time, which makes a single party's indexes safe and says nothing about two parties writing the same index at once. Anyone who needs that brings their own agreement, whether a coordination protocol or entry types that merge, and it is open work rather than an oversight.

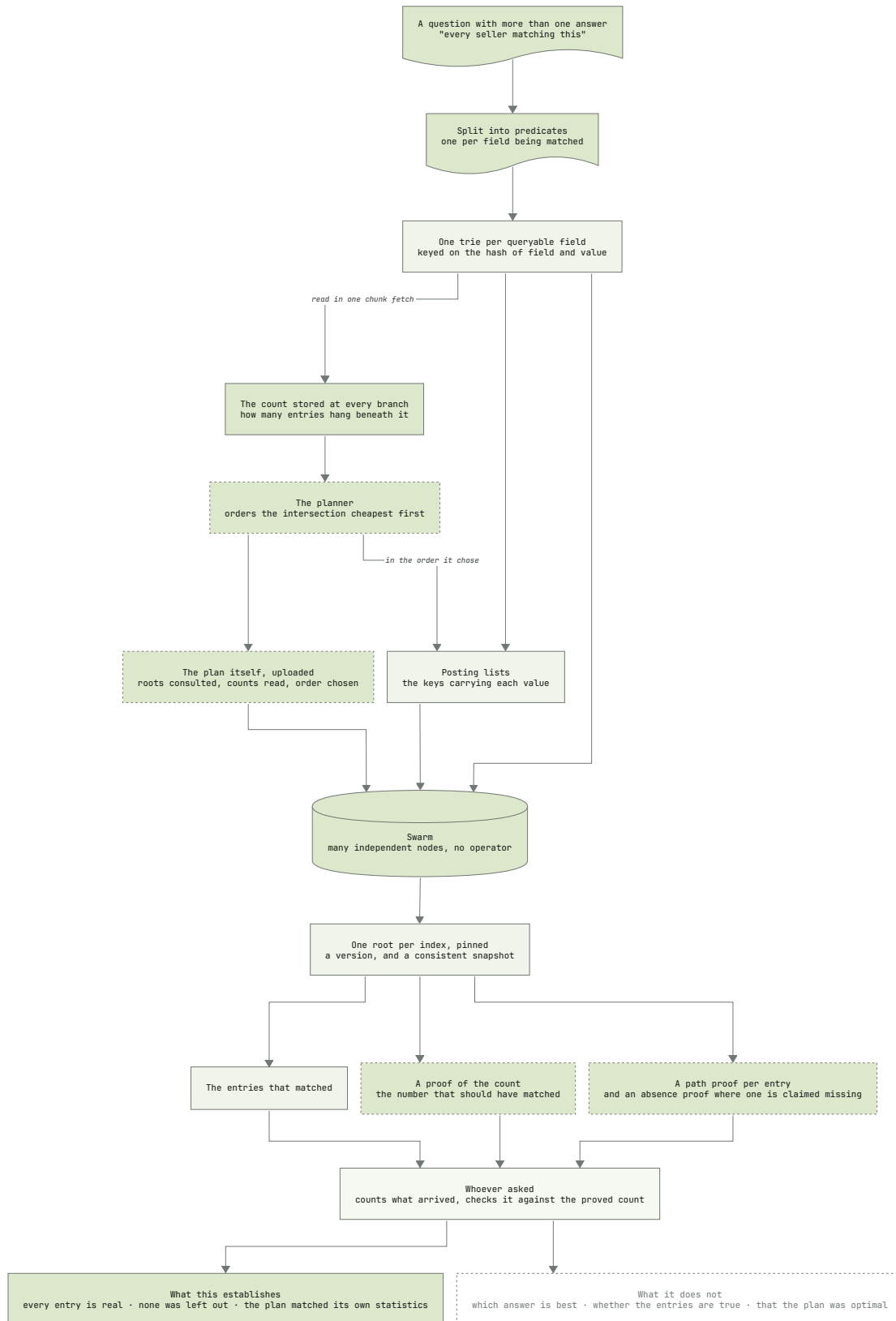
What it does not do. No query language, no execution engine, no optimizer, and no absence or count proof: this section describes what the stored format supports, and every proof in it is unwritten. Ranking and relevance are judgements and sit above anything a proof can settle. Full text needs tokenisation before any of this applies. And the cost is real, because each step is a chunk fetch across a network rather than a page read from a local disk, so a plan that would be free in Postgres is a sequence of round trips here. Measure that before designing around it.

PRIMITIVES

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

SWARM ECOSYSTEM Proximity Order Trie POTProofVerifier.sol

YOURS TO BUILD Absence proofs Count proofs Secondary indexes Query planner



Data flow. A query splits into predicates, each resolved against a secondary trie keyed on the hash of field and value, whose entries are posting lists. The planner reads the exact subtree counts stored in the nodes themselves and orders the intersection cheapest first. What comes back is three things rather than one: the entries, a proof of the count that should have matched, and a path proof per entry, all against roots the asker pinned. Counting what arrived and checking it against the proved count is what turns an omission into something detectable. The last two boxes separate what this establishes from what it does not.

Development

Four pieces of work stand between the library as it is and a query engine, and they are separable enough to be picked up one at a time by different people.

First, the walk. In `pkg/elements/pot.go` the recursive `iterate` computes `po := Compare(n.Node, k, n.At+1)` and slices forks from `n.At+1`, so the fork at `n.At` itself is never entered. A five-entry trie built from keys differing only in the leading byte returns four. Under random 32-byte keys a thousand-entry index returned 485. Whatever the fix, test it against random keys: the existing `TestIterate` builds keys as `binary.BigEndian.PutUint32` of values 0 to 63, 256 to 319 and 512 to 575, so every key has zero high bits, no two keys ever diverge at a low proximity order, the skipped position is always empty, and the suite passes. A property test that compares a walk against a plain Go map over uniformly random keys would have caught this on the first run, and is the right thing to add alongside any repair.

Second, the two proofs. Both are extensions of `pkg/proof`, which already has the BMT prover and the fork path walk. An absence proof is the existing path proof stopped one level early, carrying the bitmap segment for the node where the required bit is clear. A count proof is a segment proof over the fork-size region, plus the offset telling the verifier which four bytes in the segment to read. On the Solidity side, `POTProofVerifier.sol` already contains the bit test; it needs an assertion that a bit is clear, and one that reads a count out of a proved segment. Measure the gas on both before anything is designed around them.

Third, the indexes. One trie per queryable field, keyed on `keccak256(field, value)`, with a posting list as the entry value. Entry is an interface, so the value can be a struct rather than a byte slice, but the node budget rules the design: an entry's value rides inside its own node's 4096-byte chunk alongside the key, the 32-byte bitmap and 36 bytes per fork. A posting list of any size therefore has to be a Swarm reference to separately uploaded bytes, with only short lists inlined, and where that threshold sits is worth deciding early because it decides how many round trips a predicate costs.

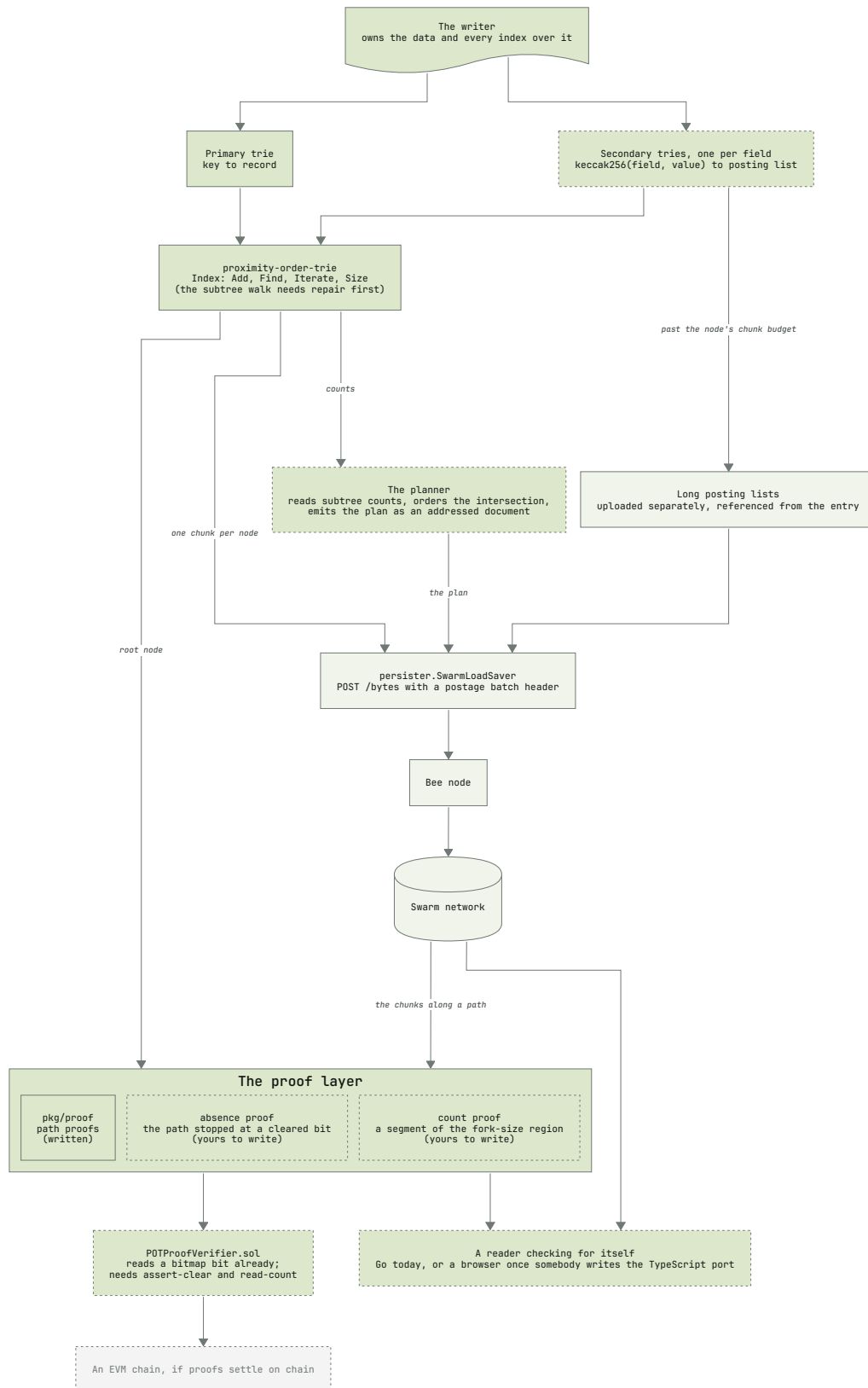
Fourth, the planner. Read the count for each predicate's subtree, order the intersection cheapest first, execute, and emit the plan as a document: the roots consulted, the counts read, the order chosen. Upload it, and the plan has an address. The useful discipline is to make the plan an output rather than an internal detail, because a plan somebody can fetch is what makes the difference between checking a result and re-running a query to see if you get the same thing.

What none of this reaches yet. There is no TypeScript anywhere in this, so every verifier here is server-side, and a reader on a phone is trusting somebody again, which is most of the point going missing at the last step. The same port that space 10 needs is the port this needs, and it is larger here, since a browser would have to walk tries, intersect posting lists and check three kinds of proof.

TOOLS & REFERENCES

- proximity-order-trie — <https://github.com/ethersphere/proximity-order-trie> (the base structure, its `pkg/proof` and its `blockchain/verifier`)
- Possibility space 10, “Look it up without asking anyone” — the trie, the root reference and the path proofs this builds on
- bee pottery branch — <https://github.com/ethersphere/bee/tree/pottery> (@zelig's original POT implementation)
- Bee Feeds documentation — <https://docs.ethswarm.org/docs/develop/tools-and-features/feeds> (a root per version, which is what makes a snapshot)

- Chunk types — <https://docs.ethswarm.org/docs/develop/tools-and-features/chunk-types>
- Postage stamps — <https://docs.ethswarm.org/docs/concepts/incentives/postage-stamps>
- Bee API specification — <https://docs.ethswarm.org/api/>
- bee-js — <https://bee-js.ethswarm.org>
- **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, if proofs are settled on chain rather than checked by the reader



Component assembly. A writer maintains a primary trie and one secondary trie per queryable field, all through the same library and load-saver onto a Bee node. Posting lists longer than a node's chunk budget are uploaded separately and referenced. The planner reads subtree counts to order an intersection and emits the plan as an addressed document. The three proof builders are the work: path proofs exist, absence and count proofs are extensions of them, and the verifier contract needs one assertion added at each end. Dashed boxes are unwritten.

Reference implementations

Yours to take, and further from finished than most spaces in this set. Nothing has been built on this. The honest summary is that the storage format supports it, one repair is needed before any of it can be tried, and the proof layer that gives the space its name is a design with no code behind it.

Proximity Order Trie (ethersphere) — <https://github.com/ethersphere/proximity-order-trie> — the base. Its nodes already carry the per-fork entry counts a count proof would read and the fork bitmap an absence proof would read, both inside the 32-byte-aligned region a BMT proof can address, and its Solidity verifier already tests a bitmap bit. Lookup, update, delete, persistence, size accounting and the existing path proofs behave correctly under random keys; the subtree walk does not, and that is where anyone starting here starts. Last pushed May 2026, no licence file.

Possibility space 10, “Look it up without asking anyone” — the same structure told as single-key lookup, where the root reference is simultaneously the address the index is fetched from and the commitment a contract holds. Everything in that space is a precondition for this one, and a team that gets a query engine working will have built that space’s browser-side verifier along the way.

The comparison worth making is an indexer of the kind blockchain applications read through today. Those are not dishonest and mostly not even suspected; they are simply unfalsifiable on completeness, and the operators of the good ones would probably rather be checkable than trusted. That is the gap this space is about, and it is worth saying that the gap has a plausible commercial shape as well as a political one: an indexer that can prove it returned everything has something to sell that its competitors cannot copy without doing the same work.