Sly Technologies
Whitepaper

Why Querying 100 EB Takes Seconds

The architecture behind Vantage Query — and why it has nothing to do with a database

0.000019% Data Read Per Query • **8× Window Scaling**
0.021% Index Overhead • **PCAPNG Compatible**

Mark Bednarczyk
CEO, Sly Technologies Inc.
April 2026

Executive Summary

The first question I get from architects evaluating Vantage is always some version of the same thing: how does querying 2 TB of raw packet capture return results in 38 milliseconds? At 100 EB, how is it any different?

The short answer is: because we read 412 KB, not 2 TB. None of it involves a database. That distinction matters more than it might seem, and it is the subject of this whitepaper.

Vantage's capture engine builds a structure called the **Query Analytics Tree** — the QAT — as packets arrive. It is a hierarchical beacon tree written to a companion `.index` file alongside the PCAP capture. At the leaf level, each node covers a 1 MB window of the capture. Each level above covers exactly 8× the window of the level below, yielding a 9-level tree for a 2.1 TB capture (rooted at LOD-8, whose 16 TB window covers 2.1 TB comfortably) and a 16-level tree for 32 EB (rooted at LOD-15, the maximum single-tree capacity). The tree shape, the navigation algorithm, and the storage overhead ratio are identical at both scales.

Every node carries a compact bitmap of keys present in its window. Keys are not hashes or derived summaries — they are **literal tokens from the jNetWorks Token Stream**, the same tokens analyzers emit during decoding, copied byte-for-byte into the QAT with an 8-byte suffix appended. The suffix carries an 8-bit sub-bitmap that tells the query engine which of a node's 8 children contain the key's value. A single bitwise AND across the query's required keys yields a candidate mask. Children whose bit is clear are pruned without ever being read.

Across 9 levels of tree, compound pruning eliminates virtually all irrelevant data before a single raw packet byte is touched. A targeted query against 2.1 TB of capture visits 847 QAT nodes, reads 412 KB of packet data, and completes in 38 milliseconds. The same query against 32 EB would visit a similar number of nodes, because node count scales with the specificity of the question, not the size of the capture — only the tree depth grows, and a few additional levels of single-bit-sub-bitmap navigation add negligible cost.

The QAT file is a PCAPNG copyable custom block. Tools that don't understand it skip it. Tools that do understand it get a 0.021% overhead index that navigates exabytes in milliseconds. The `.events` token stream it indexes is governed by the same format. Every analyzer that emits a token automatically gets QAT indexability with no schema change, no reindex, no format migration.

This whitepaper covers the structure, the pruning algorithm, the write mechanics, the PCAPNG wrapper, the SILO correlation model that joins packets to IDS alerts and threat intelligence at query time, and the storage math that makes retention of the analytical record economically viable at any scale. It closes with a direct comparison to database approaches and a pointer to the companion forensic readiness, Token Stream, and intelligent retention whitepapers.

We did not build a faster database. We built something structurally different.

1. Why “Database for Packets” Is the Wrong Mental Model

When people first hear about querying exabytes of packet data in seconds, the instinct is to reach for a familiar frame: it must be some kind of database. A time-series store. A columnar engine. Elasticsearch tuned up. There is a class of architecture discussion that assumes packet-at-scale is just another ETL pipeline problem — ingest the packets, extract the fields, stuff them into some queryable store, run SQL on top.

This instinct leads to the wrong expectations and the wrong architecture questions.

Consider what actually happens if you try to build it this way. A 100 Gbps link running at 50% utilization produces roughly 54 TB per day of raw packet data, and a full-packet archive of a reasonable retention window runs into the multi-petabyte range quickly. An 800 Gbps capture node, which Vantage supports on a single server, produces ~100 GB per second sustained. Any database approach has to ingest that without falling behind, while simultaneously serving queries.

1.1 Columnar stores

The closest database match to packet query workloads is a columnar store — ClickHouse, DuckDB, Apache Parquet with a query engine on top. They are good at scanning large volumes of structured data with predicate pushdown and compression. And they are genuinely fast for analytical queries on tabular data.

They fall short on three fronts for packet capture.

First, the ingest model is wrong. Columnar stores batch writes and commit in the background, accepting a latency gap between arrival and queryability. Packet capture cannot accept that gap — the whole point is to have the data indexed the moment it arrives, so that when the alert fires 10 days later, the index is already complete. A columnar store’s “eventually indexed” model is a forensic liability, not a feature.

Second, the schema model is wrong. A columnar store needs to know the columns before data arrives. Packet data has dozens of protocols, hundreds of fields, and new analyzers being added constantly. Either you define every possible column up front (paying storage cost for every field in every row, most of which are null), or you add columns reactively when a new analyzer needs to emit a new field type (forcing a schema migration across historical data). Neither scales.

Third, and most importantly, columnar stores read the columns they scan. They are faster than row stores because they read less, but they still **read proportionally to the data being queried**. A query across a year of captures reads a year’s worth of the columns it touches. The QAT reads 412 KB out of 2.1 TB because it proves, through structural pruning, that the rest *cannot* contain the answer. No amount of column-store optimization gets you to four orders of magnitude less data read.

1.2 Time-series databases

Time-series databases — InfluxDB, TimescaleDB, VictoriaMetrics — handle high-rate ingest and time-range queries well. They are good when the data is tagged metrics with known cardinality.

They fall apart on packet data for two reasons. Packet data has extreme cardinality — every TCP flow is unique, every HTTP URI is unique, every TLS session is unique. Time-series databases with millions of distinct series per minute thrash their indexes, bloat their tag tables, and slow down as they accumulate cardinality. Packet capture generates series cardinality several orders of magnitude beyond what time-series databases are designed for.

And time-series databases still fundamentally **store rows**. They compress them, they partition them, they index them by time and by tag, but the retrieval model is “find the rows that match and return them.” The QAT doesn’t retrieve rows; it navigates away from windows that can’t match, and the raw packets stay in native PCAP format throughout.

1.3 Search engines

Elasticsearch, OpenSearch, Splunk. These are architecturally closer to what packet query needs because they build rich inverted indexes up front and query through index traversal rather than row scanning. They can answer “find all events matching these attributes” quickly even at scale.

The problem is storage overhead. An Elasticsearch index over full packet data typically runs 2–5× the size of the data itself, depending on which fields are indexed. At 54 TB/day of capture, that’s 100–270 TB/day of index — and you still have the original packet data to store alongside it. The combined storage cost is prohibitive for anything beyond a few days of retention.

Beyond cost, Elasticsearch ingest has well-known scaling limits. A single Elasticsearch cluster tuned for high ingest rates saturates somewhere in the low-hundreds-of-GB/day per node, and horizontal scaling requires careful sharding that introduces coordination overhead. 800 Gbps on a single server is not a workload Elasticsearch was designed for.

1.4 The pattern

Every database approach to packet query shares a fundamental property: **the index tells you where the data is, but the query reads the data**. Faster indexes reduce the scan, compressed columns reduce the I/O, sharded clusters parallelize the work — but at the bottom, a database query reads some amount of the underlying data proportional to the rows that match.

A targeted packet query against 2.1 TB should not read 21 GB (1% of the data) or even 210 MB (0.01%). It should read the handful of kilobytes that actually contain the answer, and prove the rest is irrelevant without looking at it.

> *A database finds data by looking for it. The Query Analytics Tree finds data by proving where it cannot be — and skipping everything else.*

We did not build a faster database. We built a hierarchical index that is written as a byproduct of the capture process itself, costs less than a quarter of a percent of the underlying data in storage, and navigates by elimination rather than by search.

2. Structure: A Hierarchical Beacon Tree

The QAT lives in a file called `.index`, written alongside the PCAP capture. Internally, it's a single PCAPNG custom block (type `0x00000BAD` — covered in Section 7) wrapping an append-only blob of nodes and keys.

2.1 The 8× window scaling

At the leaf level — LOD-0 — each node describes a 1 MB window of the capture. Each level above covers exactly 8× the window of the level below:

LOD	Window Size	Notes
0	1 MB	Leaf — pruning keys only
1	8 MB	u32 child offsets
2	64 MB	u32 child offsets
3	512 MB	u32 child offsets
4	4 GB	u32 child offsets (ceiling of u32)
5	32 GB	u64 child offsets (switchover)
6	256 GB	
7	2 TB	Typical mid-deployment root
8	16 TB	
9	128 TB	
10	1 PB	
11	8 PB	
12	64 PB	
13	512 PB	
14	4 EB	
15	32 EB	Theoretical maximum

`window_size(L) = 1 << (L × 3 + 20)`.

A 2.1 TB capture produces a tree roughly 9 levels deep — a LOD-7 root covers exactly 2 TB, which is too small for a 2.1 TB capture, so the root promotes to LOD-8 whose 16 TB window has room to spare. A 32 EB deployment — the maximum single-tree capacity — produces a 16-level tree rooted at LOD-15. Captures larger than 32 EB use multi-file or distributed mode (covered in the QAT format specification), where the same tree format is sharded across files or

nodes with u64 cross-file offsets. **Same tree shape, same navigation algorithm, same query performance characteristics** across every scale. The difference between querying 2.1 TB and 32 EB is a few additional levels of pruning — negligible compared to the I/O of reading the data itself.

2.2 Why 8

We evaluated branching factors from 4 to 64 during design. Eight emerged as the sweet spot for three reasons.

First, it fits exactly in a single byte of sub-bitmap. Eight children means eight bits, which means a single `u8` value encodes the full "which children contain this key" information, and a single bitwise AND across a query's required keys produces the child-descent mask. Sixteen children would need a `u16`; sixty-four would need a `u64`. The byte-sized sub-bitmap maps directly onto common CPU instructions and compresses tightly in the chain.

Second, it keeps the tree shallow enough to navigate in a few nanoseconds per level while keeping each node's key chain bounded. A branching factor of 4 would double the tree depth; a branching factor of 64 would produce much sparser nodes with weaker pruning per level. Eight balances depth against density.

Third, and prosaically, the arithmetic is clean. Each level is `8×` the one below, which means level-N window sizes are `1 << (N*3 + 20)` — trivially computable, easy to visualize, and the bit-shift arithmetic compiles to one instruction.

None of these reasons is deep. The design principle is KISS, and 8 is what KISS looked like after evaluating the alternatives.

2.3 Node types

There are two.

Branch nodes (LOD 1+) are 116 bytes at LOD 1–4 and 148 bytes at LOD 5+. They contain a `seq_table` holding temporal and packet-index anchors for their 8 children (described in Section 2.4), an array of 8 child offsets (u32 below LOD 5, u64 above), and a pair of fields pointing at the head and tail of a key chain. The key chain is where the pruning happens.

Leaf nodes (LOD 0) are 12 bytes. No seq_table, no child offsets — just a pair of pointers to the leaf's own key chain. Leaves exist purely for pruning confirmation. LOD-0 is never read during navigation; the leaves are only consulted once the branch above has confirmed their window as a candidate, to finalize the match.

This asymmetry matters for performance. The bulk of query work happens at branch nodes at LOD 4 and above — larger windows, fewer nodes, more pruning per node read. Leaves are cheap enough to skip past unless they're needed.

2.4 The seq_table

Every branch node carries a 72-byte seq_table holding:

- A base timestamp anchor (ts_anchor) for the node's first child, plus an 8-entry array of u32 deltas (ts_delta[8]). Child i's absolute timestamp is ts_anchor + ts_delta[i].
- A base packet-index anchor (pkt_index_anchor) plus an 8-entry array of u16 packet counts per child window. Child i's starting packet index is the anchor plus the sum of counts for children 0 through i-1.

Child byte offsets are fully implicit — they're computed from the section base plus the child index times the child's window size.

The seq_table lets the engine resolve "which of these 8 children might contain a time range" and "which contain a packet-index range" **within a single node read, with zero child I/O**. Time-range queries and index-range queries prune at every level using only the parent node's data. This is why temporal filtering in a Vantage Query costs almost nothing — the seq_table turns "does this child's window overlap with my time filter" into eight arithmetic comparisons on data already in cache.

The u32 and u16 deltas have enough range for realistic line rates. A 1 MB LOD-0 window at 100 Gbps takes 83,886 ns to fill — well within u32 nanosecond range. At 500-byte average packet size, a 1 MB window holds 2,097 packets — well within u16 count range. Both fields are comfortable at 800 Gbps and beyond.

2.5 Root promotion

The root of the QAT is not fixed. When the capture is small, the root is a low-LOD node. As the capture grows past the root's window, the tree grows upward: a new root is appended whose first child is the old root, with the remaining seven children marked UNANALYZED (pending data) or EMPTY (no data expected). Two pwrite() calls update the block header to point at the new root.

Root promotion happens at most once per LOD level crossed — for a capture that starts from a LOD-1 root and grows to 2 TB (LOD-8 root), that's at most seven promotions over the entire lifetime of the capture. For a capture that fills a single QAT to its 32 EB maximum, fourteen. The cost of each promotion is trivial, and the mechanism is identical at every scale. A fresh 32 EB volume and a 2 MB one use the same code path.

3. Keys Are Tokens

This is where the architecture diverges most sharply from database thinking, and where the design pays off most.

In a database, the index is a separate structure: an inverted index, a B-tree, a hash map. The index is built *about* the data — derived, separate, with its own format, its own schema, its own maintenance concerns. Queries traverse the index to find which rows to read.

The QAT doesn't work that way. **QAT keys are literal tokens from the jNetWorks Token Stream** — the same tokens analyzers emit during packet decoding, described in detail in the *AI Defense Needs a Feature Stream, Not Another Model* whitepaper. A token that lands in the .events stream is copied byte-for-byte into the QAT with an 8-byte suffix appended. No new format, no translation layer, no coordination.

```
QAT key = [token header  4 bytes]   standard 4-byte Token Stream header
          [ext length    2 bytes]   only if extended token
          [payload       var]       copied unchanged from the token
          [suffix        8 bytes]   QAT-specific routing data
```

3.1 The suffix

The 8-byte suffix is the only QAT-specific addition. It holds four fields:

- **sub_bitmap (u8)** — one bit per child. Bit i set means "this key's value is present somewhere in child window i." The core of the pruning algorithm.
- **coarseness (u8)** — how many steps of adaptive aggregation this key represents. Zero is natural precision for the LOD.
- **inline_value (u16)** — small payloads (single-byte or double-byte values) live here instead of in a payload field. A proto_bitmap key uses this directly and stores no payload; total key size is 12 bytes.
- **next_key (u32)** — offset of the next key in the chain, or 0 for the tail.

Keys are linked into chains per node. Queries walk the chain, matching keys against predicates, accumulating the candidate_mask via AND.

3.2 Why tokens make good keys

Tokens were designed for compactness, self-description, and emission at wire speed. Those are exactly the properties a scalable index key needs.

A typical Token Stream control token is 4 bytes. An analytical token with a flow key or sequence number is 8–16 bytes. An extended token carrying a URL or identifier can be larger but is the minority. Token sizes are already optimized for CPU register width and cache line alignment. Using them as-is means the QAT inherits those properties for free.

The domain-and-type identification system that lets the Token Stream tolerate new analyzers additively — without reindexing — carries over to the QAT. Any analyzer that emits a token automatically gets QAT indexability. New protocols, new IDS engines, new custom analyzers in

domain 0x07: no format change, no schema migration, no coordination. The tree picks up the output automatically.

3.3 Key types

The spec defines several natural categories. The most common:

- **Address keys** — src_cidr_v4, dst_cidr_v4, src_cidr_v6, dst_cidr_v6. CIDR prefix plus sub-bitmap.
- **Port and protocol keys** — src_port_range, dst_port_range, proto_bitmap, app_proto_pack. Common queries bottom out here.
- **Flow metric keys** — duration_range, packet_count, byte_count, flow_count, retransmit_ratio. Used for flow-characteristic predicates.
- **Flow-token reference keys** — flow_token_refs. Point into the .events token stream for detailed content scan on confirmed windows. This is how LOD-0 leaves connect back to the full analytical record.
- **Protocol anomaly keys** — TLS_ANOMALY, DNS_ANOMALY, HTTP_REQUEST_NO_USER_AGENT, and dozens of others. Emitted as tokens by the Protocol Stack's analyzers.
- **SILO correlation keys** — SURICATA-domain alerts, ZEEK-domain events, VIRUSTOTAL and MISP enrichments. Covered in Section 8.

Every one of these is a Token Stream token with an 8-byte suffix. No exceptions.

3.4 Adaptive coarsening

Key chains per node are capped per type — by default, 64 entries. This matters because some windows produce a lot of values: an IP scan might touch 5,000 distinct /24 networks in a single 1 MB window, and keeping 5,000 individual CIDR keys in that node's chain would bloat it catastrophically.

Instead of bloating, the QAT writer **coarsens**.

When a key chain exceeds the cap, the writer emits a new coarsened key with the coarseness byte incremented, and the older individual keys at the finer precision are silently superseded. A window with 5,000 /24 keys becomes a smaller number of /16 keys covering the same space. A window with thousands of distinct subdomain.slytechs.com DNS queries becomes a single *.slytechs.com wildcard key.

The precision is not lost — it's preserved at the child nodes below. Since each child window is 8× smaller, each child typically holds 8× fewer distinct values, and coarsening is less likely to trigger. Precision is recovered on descent.

This has two practical consequences.

First, node size stays bounded regardless of input. A 1 MB window with 10 flows produces a small key chain. A 1 MB window with 50,000 flows produces a bounded, coarsened key chain. The storage overhead ratio holds at 0.021% across all traffic patterns because coarsening absorbs cardinality explosions.

Second, queries pay a small cost during scan-like predicates but avoid a catastrophic one. A query for `src_cidr == 10.1.2.0/24` against a window that coarsened to `/16` descends into more children than strictly necessary — but each descent reads one small child node, and the child's finer keys either confirm or prune. Compared to the alternative of an unbounded key chain at every node, this is a dramatic win.

The writer decides when to coarsen. Per-key-type coarseness steps are defined in the spec's appendix (e.g., CIDRv4 coarsens by 2 prefix bits per step, port ranges double their bucket size, timing quantization doubles). Analyzers continue to emit natural-precision tokens; the QAT writer handles precision management at write time. Analyzers stay simple. Writer absorbs the complexity.

> *The QAT doesn't get bigger with cardinality. It coarsens gracefully and recovers precision by descent.*

4. Pruning: Two-Phase Navigation

A Vantage Query is translated into a set of required keys. The engine begins at the root of the QAT and descends. At every branch node, it performs two phases of filtering before deciding which children to descend into.

4.1 Phase 1: sequential filter

Using only the node's own `seq_table`, the engine computes an initial `candidate_mask` for the 8 children:

```
candidate_mask = 0xFF  (all children)

for i in 0..7:
   if (ts_anchor + ts_delta[i]) outside query time range:
      candidate_mask &= ~(1 << i)
   if child_offsets[i] == EMPTY:
      candidate_mask &= ~(1 << i)
```

Zero I/O beyond the node read itself. Time-range queries prune here — a query for "last 24 hours" against a 14-day capture eliminates most children at the root and near-root, without touching child nodes at all. Windows marked EMPTY (confirmed empty after analysis) are pruned likewise.

4.2 Phase 2: key chain pruning

If the candidate_mask is still non-zero, the engine walks the key chain:

```
for key in node.key_chain:
   if key matches a query predicate:
      candidate_mask &= key.suffix.sub_bitmap

if candidate_mask == 0:
   prune entire subtree, return
```

Every matching key contributes its sub_bitmap to the candidate_mask via AND. Keys that don't match predicates are ignored. After walking the chain, the candidate_mask is the exact set of children that could satisfy all query predicates.

4.3 Phase 3: descend

For each bit still set in the candidate_mask, descend into the corresponding child. Recurse.

The elegance of this is that **every level performs the same two-phase filter against its own data**. A query that's highly specific — a particular source IP, a particular destination port, a particular time window — prunes aggressively at every level. A query for "any TCP traffic in the last year" prunes less but still prunes reasonably.

4.4 A concrete example

```
Query: FIND packets WHERE ip.src == 185.234.72.19

Translation: one required key, type = src_cidr_v4, value = 185.234.72.19/32

Root node (LOD-8, 16 TB window — covers 2.1 TB capture):
  Phase 1: no time filter, all 8 children candidate
  Phase 2: walk key chain
    Find src_cidr_v4 key for 185.234.72.0/24
    sub_bitmap = 0b00000100
    candidate_mask = 0xFF & 0b00000100 = 0b00000100
  Phase 3: descend into child 2

Child 2 (LOD-7, 2 TB window):
  Phase 1: no changes
  Phase 2: walk key chain
    Find src_cidr_v4 key for 185.234.72.0/24
    sub_bitmap = 0b00100000
    candidate_mask = 0b00100000
  Phase 3: descend into child 5
```

```
...continues for 7 more levels, each single-child descent...

LOD-0 leaf (1 MB window):
  Walk the key chain to confirm the exact /32 match
  Read flow_token_refs to jump into .events stream
  Scan matching records

Total: 9 nodes visited (one per level).
Total raw packet bytes read: only the bytes in the confirmed flow.
```

4.5 PROFILE output from a real query

The effect becomes concrete in the PROFILE output from a real query against 2.1 TB of capture:

```
quarry> PROFILE FIND packets WHERE ip.src == 185.234.72.19

  QAT nodes visited     847
  Packet data read      412 KB  (of 2.1 TB total)
  Data fraction         0.000019%
  Query time            38 ms
```

847 nodes visited. 412 KB of packet data read. 2.1 TB skipped entirely. 38 milliseconds.

The same query against a 32 EB single-tree capture would visit a similar number of nodes, because the match set defines node count, not the capture size. The tree is deeper — 16 levels instead of 9 — which means a few more levels of navigation at the top, but each of those upper-level traversals is a single node read with the same two-phase filter. The cost of those extra levels is negligible compared to the cost of touching any meaningful fraction of the underlying data. Captures beyond 32 EB scale through multi-file and distributed mode described in the spec: the format is identical, the navigation algorithm is identical, and each shard's subtree query executes independently.

> *Query performance scales with the specificity of the question, not the size of the capture. The same query against 32 EB visits approximately the same number of QAT nodes as the same query against 1 GB.*

4.6 The sub-bitmap AND compounds across keys

The real power of the algorithm is in multi-predicate queries. Each additional predicate contributes a further AND that narrows the candidate_mask:

```
Query: WHERE ip.src IN 10.x AND tcp.port == 443 AND proto == TLS
```

```
At a branch node:
  src_cidr key       sub_bitmap = 0b10110101
  dst_port key       sub_bitmap = 0b11010111
  app_proto key      sub_bitmap = 0b00110101

  candidate_mask     = 0b10110101
                     & 0b11010111
                     & 0b00110101
                     = 0b00010101

Descend into children 0, 2, 4 only.
Five of eight children pruned at this level alone.
```

Over 9 levels of tree for a 2.1 TB capture — or 16 levels for 32 EB — compound pruning eliminates virtually all irrelevant data before any raw packet byte is touched. A more specific query prunes harder. A less specific query prunes less but still prunes usefully. The worst case — a query for something that matches every window — is simply a linear scan through the leaves, which is as fast as the underlying file system permits.

5. Append-Only and Capture-Time Writes

The second design principle that makes the QAT work is that **the index is written as a byproduct of capture**, not assembled afterward. This has consequences for latency, for concurrency, and for what's possible during live analysis.

5.1 Sequential append

The QAT file is append-only except for a small, well-defined set of backward writes. The only `pwrite()` targets in the entire format are:

- The block header's `root_node_offset` and `root_lod`, updated on root promotion (at most once per LOD crossed).
- The block header's `blob_length`, updated continuously as the blob grows.
- A parent node's `child_offsets[i]`, updated once when a child is appended.
- A key chain's tail `next_key` field, updated when a new key is appended.
- The PCAPNG wrapper's `block_total_length`, updated once at section close.

Everything else is pure sequential write. The write pattern is what a modern NVMe drive and filesystem are optimized for — large sequential I/O at the speed of the underlying hardware.

5.2 The QAT writer

The writer lives in the Vantage SDK (vantage-sdk), delivered as part of the Vantage Platform and also bundled into the jNetWorks SDK. It runs alongside the capture pipeline. For a single 800 Gbps capture with multiple analyzers processing packets in parallel, the token stream flowing into the writer is still about one-thousandth the rate of the packet stream — significant, but sequential and manageable.

The writer tracks the state of the windows at each LOD in a small in-memory structure. For a tree up to LOD-15 (32 EB), the total RAM cost is $15 \times 8 \times 8$ bytes = 960 bytes. The writer builds keys for each level as tokens arrive, accumulates them in the current window, and commits them to the .index file sequentially when a window fills.

Per 1 MB LOD-0 window, the writer:

- Accumulates pruning keys from the observations in the window.
- Applies coarsening if any key-type chain exceeds its cap.
- Appends the LOD-0 leaf node to the blob.
- Appends the key chain entries.
- Updates the in-memory rolling buffer of pending children at LOD-1.

When 8 LOD-0 leaves complete, a LOD-1 branch node is written — seq_table filled from tracked timestamps and packet counts, child_offsets populated from the rolling buffer. The LOD-1 buffer is cleared, the LOD-1 node is added to the LOD-2 pending buffer, and so on. Cascading upward happens naturally as each level fills.

5.3 Parallel shard analysis

The append-only design enables trivial parallelism. Multiple shards of a capture can be analyzed concurrently, each writing its own subtree into the same .index blob at distinct offsets. Because pwrite() targets are non-overlapping — each shard owns its slot in its parent node — no coordination is needed between shard analysis jobs.

```
Shard A → appends at offset X → pwrite() root.child_offsets[A] = X
Shard B → appends at offset Y → pwrite() root.child_offsets[B] = Y
Shard C → appends at offset Z → pwrite() root.child_offsets[C] = Z
```

Only root promotion is serialized, and root promotion fires at most once per LOD level crossed — extremely rare during normal operation.

5.4 Post-emission keys

The format supports appending new keys to any existing node at any time. A second-pass analyzer — Sentinel's post-capture correlation engine, a user-defined Vantage Query rule, a SILO that completes a correlation hours after the packet was captured — can attach a new key to any node in the tree.

```
Old tail key.next_key = 0  (was the end of the chain)

Writer appends new key at current blob end.
pwrite() old_tail.next_key = new_key_offset
pwrite() node.last_key_offset = new_key_offset
```

Two pwrite() calls. No reshuffling, no compaction, no format change. The chain continues to function. A reader walking the chain sees the new key as naturally as any other. Queries that run after the append see the updated pruning information. Queries that ran before the append saw the previous state.

This is what makes retrospective correlation possible — the mechanism described in the Token Stream whitepaper, where an IDS alert arriving days after its packet can still bind to the original window via a carrier token. The QAT and the .events stream both support post-emission writes through identical mechanisms.

> *Every query is bounded. Every write is sequential. No pass ever blocks another.*

6. PCAPNG Compatibility

The .index file is not a proprietary container. It is a single **PCAPNG copyable custom block**, type 0x00000BAD, with the Sly Technologies Private Enterprise Number (PEN) as the custom block's identifier.

The same is true of the .events token stream. And of the SUM\0 section summary block that jNetWorks writes inline in every capture.

6.1 What this means

A standard PCAPNG reader — tcpdump, Wireshark, Zeek, or any tool built on libpcap or pcapng — encounters these blocks and skips them. Custom blocks are defined by the PCAPNG specification to be skippable by readers that don't recognize their PEN. The reader walks past, reads the next block, and continues. The capture file remains fully valid and fully readable.

A Vantage-aware tool — Lynx, the Quarry virtual filesystem, the Vantage Query engine — recognizes the PEN, unwraps the block, and gets the QAT, the events stream, or the summary metadata.

This means the .index file can live inside the capture file itself, as embedded blocks within the PCAPNG stream, or as a separate sidecar file alongside the capture. Either way, the format is a valid PCAPNG. Either way, every existing PCAP-aware tool reads the capture without issue. Either way, Vantage-aware tools get the index.

6.2 Why this matters

The network industry has invested decades in PCAP tooling. Suricata reads PCAP. Zeek reads PCAP. Every forensic analyst's laptop has Wireshark on it. Every CI pipeline in network security uses `tshark` or `tcpdump` somewhere. A proprietary capture format — no matter how technically superior — asks customers to give up that entire tooling investment.

Vantage doesn't ask. The `.pcapng` files Vantage produces are valid PCAPNG. You can open them in Wireshark. You can run Suricata against them. You can pipe them through whatever pipeline you already built. The QAT is additive — tools that understand it get the fast path; tools that don't get the same capture data they've always been able to read.

This is architecturally the same posture Vantage takes with the Token Stream: we produce a format, you consume it wherever you already work. For the `.index` file, the PCAPNG wrapper means "wherever you already work" includes every PCAP-aware tool on earth.

6.3 The SUM\0 block

A smaller but related case: `SUM\0` is a PCAPNG custom block written by jNetWorks immediately after each Section Header Block in every capture. It's a lightweight summary — 164 bytes — carrying:

- First and last packet index in the section
- First and last timestamp
- Packet count and byte count
- RMON counters (broadcast/multicast, CRC errors, fragments)
- Protocol distribution histogram
- Packet size histogram
- Flow count and new-flow count
- Section flags (complete/in-progress, has QAT, has events, has metrics)

This is written **regardless of whether full analysis runs**. A jNetWorks capture with zero analyzers attached still gets `SUM\0` blocks, because RMON and basic counters are tracked in the capture path itself. The cost is negligible and the payoff is that even a minimally-analyzed capture supports instant sequential anchors and coarse-grained queries.

A reader can answer "how many packets in the last hour" from `SUM\0` blocks alone, without ever opening the `.index` file. The QAT fills in the structure and precision; `SUM\0` provides the floor.

7. SILO Correlation Through One Namespace

The QAT handles packet data efficiently. But packet data is never the whole picture. A realistic investigation requires correlating packets with IDS alerts from Suricata or Zeek, flow metadata from NetFlow or IPFIX, and threat intelligence from MISP or VirusTotal.

In most environments, this correlation happens manually. An analyst queries Wireshark for the packet, pivots to the SIEM for the Suricata alert, cross-references the flow record in the NetFlow tool, and looks up the IP in VirusTotal. Four tools. Four namespaces. Four query languages.

Vantage handles this through **SILOs** — Source Integration Layer Overlays. A SILO is a declaration of intent: it describes where a data source lives, what format it uses, and how it relates to other sources in the namespace.

7.1 The four SILO archetypes

Capture SILO. The anchor. Owns the .index file, the .events token stream, and the raw PCAP data. When Sentinel captures a packet, it writes the PCAP bytes and simultaneously commits keys to the QAT and structured events to the .events stream — TCP state transitions, HTTP request/response pairs, DNS query/answer cycles, TLS handshake outcomes.

Analysis SILO. Suricata, Zeek, Snort. An Analysis SILO does not sit alongside the Capture SILO — it merges into it. Alert keys from Suricata rule matches are committed directly into the Capture SILO's QAT at the time they fire. When an analyst queries "show packets where suricata.alert is present," they are navigating the same QAT that holds the packet keys. IDS alerts and packets share a namespace. No pivot required.

Telemetry SILO. NetFlow, IPFIX, cloud flow logs. A Telemetry SILO maintains its own QAT, cross-referenced by 5-tuple at query time. When a query asks for traffic involving a specific IP, the engine navigates both the Capture QAT and the Telemetry QAT in parallel and returns a unified result. The analyst sees one answer, not two.

Enrichment SILO. MISP, VirusTotal, MaxMind, internal threat intel. Enrichment SILOs append threat scores, geographic annotations, and campaign tags to matching flows at query time. An IP that appears in a VirusTotal feed as a known C2 infrastructure host will surface that tag automatically when it appears in query results.

7.2 The correlation token

When an external SILO event binds to a specific packet flow, the SILO correlation bridge emits a **token** into the Token Stream — the same stream the QAT indexes. Because every token is automatically a potential QAT key, the correlation is queryable the moment it's written. A Suricata alert that binds to a flow at 14:23 becomes searchable via the QAT at 14:23.

The Token Stream whitepaper covers this in detail: the per-worker correlation cache keyed by timestamp/frame/tuple, the confidence lifecycle (FULL, PARTIAL, INFERRED, UNKNOWN) for correlations that can't complete synchronously, and the retrospective correlation that refines earlier emissions as stored data becomes available. The QAT picks up every one of those tokens as a key, which means the confidence state — whether a correlation is verified, probable, or speculative — is itself queryable.

7.3 What this looks like in practice

A single Vantage Query can span all four SILOs:

```
FIND flows
  WHERE suricata.alert IS PRESENT
    AND tls.anomaly IS PRESENT
    AND netflow.bytes > 1000000
    AND threat_intel.score > 75
  DURING LAST 24h
  FROM captures, suricata-silo, netflow-silo, misp-silo
```

One query. Four data sources. No JOIN syntax. No tool switching. The correlation is declared once at SILO definition time and resolved transparently at query time. Because every SILO writes into the same QAT namespace, the query engine traverses a single tree with a single algorithm, regardless of how many sources contribute predicates.

> *One query. Four data sources. The analyst does not need to know which SILO holds which data. They ask a question in plain English. The engine figures out where to look.*

8. Scaling to Exabytes: The Storage Math

Storage overhead is 0.021% of the underlying capture size. This is not a tuning target or a measurement for a specific workload. It is a structural property of the format — the geometric series of the tree's branching factor.

8.1 The math

A 1 MB LOD-0 leaf holds the pruning keys for its 1 MB window. Each level above contains one node per 8× larger window, so the total node count is:

```
total_nodes = LOD-0 nodes × (1 + 1/8 + 1/64 + 1/512 + ...)
            = LOD-0 nodes × 1.143
```

The geometric series sums to 8/7 ≈ 1.143. Higher-LOD nodes add a fixed 14.3% over LOD-0 count regardless of scale.

Concrete numbers from the spec, assuming normal traffic (800 B average packet, 62 flows/MB):

Capture	LOD-0 beacons	Total beacons	QAT size	Overhead
1 GB	1,024	1,173	~225 KB	0.021%
1 TB	1,048,576	1,198,373	~225 MB	0.021%
1 PB	~1.07 B	~1.23 B	~225 GB	0.021%

```
32 EB     ~34.4 T        ~39.3 T       ~7.0 PB      0.021%
```

The overhead ratio is identical at every scale. At 32 EB, the QAT is 7 PB — large in absolute terms, trivially small in relative terms. The .events token stream adds further overhead (covered in the Token Stream whitepaper) and the combined metadata still stays under 1% of the underlying capture.

8.2 What this enables

The combined .index + .events metadata cost is under 1% of the packet data it describes. This is the number that makes **permanent metadata retention** economically viable.

The full implications are covered in the *Stop Rolling Over Your Evidence* whitepaper, but the short version: retaining the QAT and the .events stream permanently — even after the raw packets are released under a rolling buffer policy — preserves full query capability for everything the analyzers recorded. Flow summaries, protocol events, IDS correlations, anomaly detections, behavioral metrics. Indefinitely. At negligible storage cost.

A tier-4 retention policy that keeps metadata forever and packets for 30 days costs less than 1% of what full-packet retention for 30 days alone costs. The QAT is what makes that math work.

8.3 mmap strategy

A practical note on query performance at scale. The QAT is designed for memory mapping. The spine of the tree — the root and the LOD-5+ branch nodes — is small enough to always fit in RAM. For a 1 PB capture, the spine is a few MB.

Active subtrees below LOD-5 are mmap'd on demand. Each LOD-4 subtree is at most 4 GB — a trivially mmappable window size. The OS pages in what the query needs and pages out cold subtrees automatically. Working set for a typical petabyte query is the spine plus one or two candidate LOD-4 subtrees, total a few MB to a few GB. The OS handles the rest.

This is why query performance stays flat across scales. The working set is not "the whole tree" — it's the spine plus the active candidate region. The spine is constant. The candidate region depends on query selectivity, not on capture size.

9. One More Time: This Is Not a Database

I want to be specific about this because the confusion has real consequences for how teams evaluate and deploy packet intelligence infrastructure.

```
Database index    Built after data arrives
QAT               Built as packets arrive — zero additional latency

Database query    Scans rows, applies filters — reads data
```

```
Vantage Query     Navigates tree, prunes branches — eliminates data

Database schema   Defines indexable fields before data arrives
QAT               Accepts any analyzer output — no schema change needed

Database store    Structured records
QAT               Keys derived from raw packets in native PCAP format

Database index    Separate structure maintained alongside data
QAT               Same PCAPNG custom-block format as the capture itself

Database scaling  Horizontal sharding with coordination overhead
QAT               Append-only, non-overlapping pwrites, parallel trivially

Database overhead 2–5× source data for rich indexing
QAT               0.021% structural overhead
```

> *The QAT is not a faster database. It is a different answer to a different question: not "how do we store packet data in a queryable form" but "how do we add queryability to packet data without changing what it is."*

A database index is built *about* the data. The QAT is built *of* the data — its keys are the same tokens analyzers emit during decoding, the same format the Token Stream uses, living in the same PCAPNG container as the packets themselves. There is no translation layer. There is no schema to drift. There is no separate service to operate.

If you're evaluating the QAT and you keep finding yourself asking "how does this compare to ClickHouse" or "could we build this on Elasticsearch," the answer is: probably yes with enormous engineering effort, almost certainly at far higher storage and operational cost, and with an ingest pipeline that can't keep up. The right question is "what would a packet-native index look like," and the QAT is the answer.

10. Getting Started

The QAT ships with the Vantage Platform. Every capture — live or offline — produces a `.index` file automatically, alongside the `.events` token stream. The virtual `.tokens` projection, the `/dev/vantage/{capture-name-or-id}/analysis` device, and the Vantage Query engine are available on every Vantage deployment.

For teams building their own capture infrastructure at the protocol level, the **jNetWorks SDK v3** is available under internal and OEM license. The SDK bundle includes the QAT writer, the Token Stream implementation, the `.index`/`.events` file storage layer, and the Vantage Query

engine — everything an SDK customer needs to produce and consume Vantage-compatible output from their own capture pipeline. Full Vantage Platform adoption unlocks SILO correlation, the carrier-token confidence lifecycle, and the analytical tooling built on top.

Request a demonstration at slytechs.com/contact. We will walk through the QAT with your traffic, on your infrastructure, and show the PROFILE output of real queries at whatever scale your environment produces.

Related reading

- *AI Defense Needs a Feature Stream, Not Another Model* — the jNetWorks Token Stream and Vantage Token Stream architecture. Covers how tokens are produced, labeled at emission, and composed through SILO correlation. The tokens that become QAT keys are described there.
- *When the Breach Happens, Is Your Data Already There?* — the 24-minute forensic readiness workflow, worked end-to-end. Demonstrates the query performance described in this whitepaper against a real incident-response scenario.
- *Stop Rolling Over Your Evidence* — the intelligent retention architecture that keeps the QAT and .events stream indefinitely at 0.59% combined metadata overhead, enabling full query capability long after packet data is released.

Mark Bednarczyk is the founder and CEO of Sly Technologies Inc., a network packet capture and analysis company based in the Greater Tampa Bay area of Florida. Sly Technologies has been building packet infrastructure since 2005 and ships the Vantage Platform and the jNetWorks SDK. slytechs.com