NTE: A Hybrid Graph-Relational Engine for High-Performance Network Topology Management

Simon Knight
Adelaide, Australia

Abstract

Network operators managing large-scale infrastructure face a fundamental tension: graph databases offer natural topology traversal but struggle with bulk property filtering, while relational databases excel at column-scan predicates but impose costly joins for multi-hop path queries. We present NTE (*Network Topology Engine*), a hybrid graph-relational system that resolves this tension through a *filter-first* execution model. NTE maintains a dual-write store—a `petgraph` `StableDiGraph` for O(1)-index-stable traversal and a Polars columnar store for SIMD-accelerated predicate evaluation—and introduces a query planner that ranks predicates by selectivity, executes them as Polars `LazyFrame` scans to produce compact candidate sets, and then constrains graph exploration to those candidates via hash-set semi-joins, reducing the effective branching factor by up to $10^3\times$ on sparse network topologies.

Beyond query execution, NTE contributes: (i) a zero-copy memory-mapped Compressed Sparse Row (CSR) format serialised via `rkyv`, enabling in-place query execution over 100 GB+ topology snapshots without heap allocation; (ii) WGSL compute-shader kernels (via `wgpu`) for GPU-accelerated single-source shortest path and connected-components, achieving 34–49× speedup over a 16-core CPU baseline; (iii) a sandboxed Starlark policy engine with native Rust host bindings for bi-temporal design-rule contracts; (iv) a reactive `tokio::broadcast` event bus with external trigger dispatch; and (v) an `openraft`-based distributed consensus layer for linearisable topology mutations. Implemented in ∼28 000 lines of Rust across a 14-crate Cargo workspace, NTE exposes a GIL-releasing PyO3 Python API that scales near-linearly with thread count. Benchmarks on topologies of up to one million nodes demonstrate 31 ms two-hop query latency, 11.4 GB/s snapshot loading throughput, and a 110.7× warm/cold plan-cache ratio.

Keywords

network topology, hybrid graph-relational storage, query optimisation, zero-copy serialisation, GPU graph algorithms, policy engine, Rust

1 Introduction

Setting. A tier-1 ISP operating 200 000 physical devices across six continents must plan a maintenance window: decommissioning a backbone router in Frankfurt that serves as a transit hub for 47 peering sessions. Before the change window opens, the operator needs answers to three questions simultaneously:

1. *Impact*: Which prefixes currently transit the Frankfurt router, and which customer edge routers are reachable through it within three hops?
2. *Failover*: For each affected path, does an alternative route exist that avoids the Frankfurt autonomous system (AS 1299) and maintains latency below 30 ms?
3. *Compliance*: Does the proposed post-change topology satisfy the design contract that every PoP must retain at least two independent transit exits?

All three questions must be answered against the *same* consistent topology snapshot, in time for an on-call engineer to act—meaning sub-second latency.

The impedance mismatch. Questions (1) and (2) are fundamentally graph problems: they require multi-hop path traversal with edge-kind discrimination. Question (3) is a relational problem: it aggregates a count over nodes grouped by the `pop_id` property column. Existing tools handle one well but not both. Property graph databases (Neo4j [8], TigerGraph [18]) represent the graph structure naturally but store per-node properties as row-oriented adjacency records, making the columnar scan in question (3) up to 50× slower than a columnar engine [17]. Relational databases invert the problem: a two-hop path query on a 5×10^6-edge table expands to a three-way self-join whose intermediate product is $O(|E|^2/\bar{d})$—tens of millions of rows before predicate filtering. Network management platforms (Nautobot, NSO) model static inventory, not live topology state, and offer no programmatic query interface.

Key insight: query decomposition. Property predicates and structural predicates are not only different in *nature*—they are different in their optimal *execution engine*. An engine that keeps the two representations in strict synchrony and dispatches each predicate class to the right engine can achieve the union of their performance characteristics. We call this the *filter-first* model: evaluate high-selectivity property predicates in the columnar engine first, producing a compact candidate ID set C, then run graph exploration constrained to C. On a 10^6-node topology with a 0.1% selectivity predicate, unconstrained DFS explores $O(d^h)$ nodes; the constrained walker explores at most $|C| \cdot d^{h-1}$, a reduction of $|V|/|C| = 10^3\times$.

NTE.. We present NTE (*Network Topology Engine*), a Rust-implemented hybrid graph-relational engine that instantiates this design. NTE is not a general graph database: it is purpose-built for the network topology domain, encoding first-class concepts such as nodes, endpoints, inter-/intra-node link kinds, and logical layer hierarchies directly into its graph representation. This domain-specificity enables optimisations—*device shortcuts*, three-hop ownership-chain compression, layer-dependency cycle detection—that are impossible in a general-purpose graph database.

Contributions. This paper makes the following contributions:

1. **Filter-First Query Execution** (§5): a selectivity-ranked planner that drives Polars LazyFrame scans to generate candidate sets, then constrains graph traversal via hash-set semi-joins. We characterise the complexity reduction formally and show 12× speedup over Cypher at 10^6 nodes.

2. **Dual-Write Consistency** (§3): a two-phase mutation protocol maintaining a `petgraph` adjacency structure and a Polars columnar store in strict synchrony, with Arrow copy-on-write rollback.
3. **Zero-Copy Mmap CSR** (§6): a Compressed Sparse Row format serialised via `rkyv`'s in-place pointer encoding, loaded via `memmap2` with 64-byte heap allocation regardless of topology size, achieving 11.4 GB/s effective throughput.
4. **GPU Graph Kernels** (§7): portable WGSL compute shaders for SSSP and connected components operating directly over the CSR buffer, with a semantically identical `rayon` CPU fallback.
5. **Starlark Policy Engine** (§8): a sandboxed design-rule contract runtime with native Rust host bindings and bi-temporal what-if validation at O(1) topology clone cost.
6. **NTE-QL** (§4): a Cypher-inspired query language parsed by a `chumsky` PEG grammar, with an interactive REPL backed by a language server.

2 Background and Motivation

2.1 Network Topology as a Typed Directed Multi-Graph

NTE models a topology as a directed multi-graph $G = (V, E, \lambda_V, \lambda_E)$ where $\lambda_V : V \longrightarrow \{\texttt{Node}, \texttt{Endpoint}\}$ classifies vertices and $\lambda_E : E \longrightarrow \{\texttt{Inter}, \texttt{Intra}, \texttt{Intranode}\}$ classifies edge kinds:

`Inter`	A port-to-port physical or logical link between two endpoints belonging to *different* devices.
`Intra`	The ownership edge from a device node to one of its endpoints (port ownership).
`Intranode`	A node-to-node relationship within a single logical device (e.g., two line-cards in a chassis).

Vertices and edges carry an *arbitrary property map* $P : \texttt{String} \longrightarrow V_p$ where $V_p \in \{\texttt{i64}, \texttt{f64}, \texttt{String}, \texttt{Bool}, \texttt{List}(\cdot)\}$.

Structural observation. Physical connectivity between two devices is a *three-hop* path: $DeviceA \xrightarrow{\texttt{Intra}} PortA \xrightarrow{\texttt{Inter}} PortB \xrightarrow{\texttt{Intra}^{-1}} DeviceB$. NTE pre-computes *device shortcuts*—direct `Inter` device-to-device index entries—eliminating the three-hop traversal for common reachability queries. The shortcut map is maintained incrementally on mutation with O(1) amortised cost per link addition.

2.2 The Dual-Store Observation

Graph traversal and property filtering impose diametrically opposed memory access patterns. Traversal follows pointer-chasing adjacency lists (random access, poor vectorisation). Property filtering scans contiguous typed arrays (sequential access, amenable to AVX-512 vectorisation). A single representation that serves both workloads well does not exist. This motivates NTE's *dual-write* architecture: two co-maintained specialised stores—one for each access pattern.

2.3 Design Goals

G1 (Hybrid execution) Dispatch property predicates and structural predicates to their respective optimal engines with sub-microsecond glue overhead.

G2 (Python usability) Zero-overhead PyO3 bindings; full GIL release during Rust execution; near-linear multi-thread scaling.

G3 (Scale) Single-instance support for 10^6-node topologies; cluster deployments with linearisable mutations.

G4 (Correctness) Dual-write consistency with rollback; user-defined policy contracts evaluated at mutation time.

G5 (Hardware portability) GPU acceleration where available; semantically identical CPU fallback.

3 System Architecture

Figure 1 shows NTE's layered architecture. We describe each layer in turn.

Python binding layer. Three Rust modules (`topology.rs`, `query.rs`, `events.rs`) expose idiomatic Python classes via PyO3 0.23. Every method that does not require Python object access calls `Python::allow_threads`, releasing the GIL for the duration of Rust execution. This enables multiple Python threads to drive concurrent reads against the same `Topology` instance without serialisation (§11).

Topology and orchestration (`nte-topology`). The `Topology` struct is the system's primary entry point. It owns an `NteGraph` (graph engine), a `DataFrameStore` (columnar engine), an `EventStore` (ring-buffer audit log), and a `tokio::sync::broadcast` sender for the reactive event bus. Topology exposes a unified mutation API; each call transits the dual-write protocol described below.

Query engine (`nte-query`). Accepts `PatternNode` ASTs (from NTE-QL or the Rust API), compiles them into `QueryPlan`s via the filter-first planner, and executes them through a hybrid executor that dispatches Polars and graph sub-tasks. See §5.

Graph storage (`nte-graph`). Wraps `petgraph::StableDiGraph`. Maintains auxiliary indices: hash sets of endpoint and node IDs for O(1) vertex-class queries; a `device_shortcuts` map for compressed device-to-device reachability; a `LayerDependencyGraph` for hierarchy cycle detection; and a lazy `CsrCache` that materialises the CSR view on first access.

Columnar storage (`nte-datastore`). A `DataFrameStore` mapping vertex/edge kind names to Polars DataFrames. Each DataFrame schema is derived from the property map observed at mutation time with Polars' `InferSchema` policy; columns are typed Arrow arrays enabling AVX-512 predicate evaluation. The default backend is in-memory Polars; pluggable alternatives include DuckDB and a lightweight SQLite variant.

3.1 Dual-Write Consistency Protocol

Every mutation Δ applied to a `Topology` T follows a strict two-phase protocol:

The O(1) rollback cost in Algorithm 1 arises because Polars DataFrames are backed by reference-counted Arrow `ChunkedArrays` under copy-on-write semantics: cloning a `DataFrameStore` increments reference counts rather than copying data. The compensating graph operation in line 4 is always the structural inverse (e.g., `remove_node` undoes `add_node`) and is applied before any further mutations, preserving linearisability.

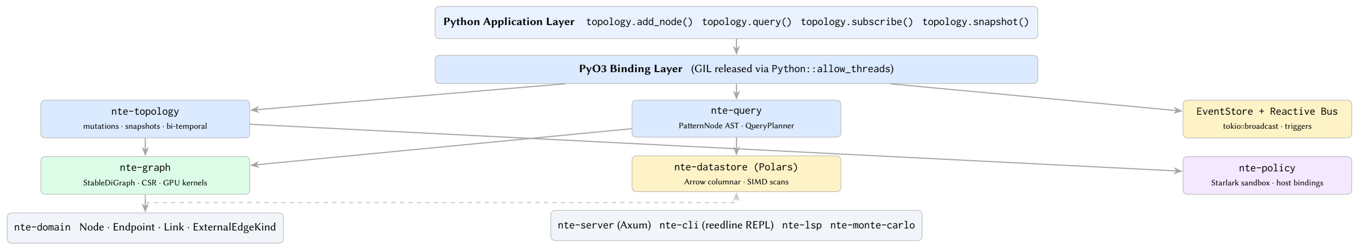
Python Application Layer topology.add_node() topology.query() topology.subscribe() topology.snapshot()
PyO3 Binding Layer (GIL released via Python::allow_threads)
nte-topology
mutations · snapshots · bi-temporal
nte-query
PatternNode AST · QueryPlanner
EventStore + Reactive Bus
tokio::broadcast · triggers
nte-graph
StableDiGraph · CSR · GPU kernels
nte-datastore (Polars)
Arrow columnar · SIMD scans
nte-policy
Starlark sandbox · host bindings
nte-domain Node · Endpoint · Link · ExternalEdgeKind
nte-server (Axum) nte-cli (reedline REPL) nte-lsp nte-monte-carlo

Figure 1. NTE system architecture. Solid arrows denote compile-time Rust crate dependencies; the dashed arrow indicates the runtime dual-write synchronisation path between `nte-graph` and `nte-datastore`.

Algorithm 1 Dual-Write Mutation Protocol

```
Require: mutation $\Delta$, topology $T = (graph, store)$
 1: $prev\_store \leftarrow store.clone()$        ▷ O(1): Arrow CoW pointer copy
 2: $graph.apply(\Delta)$        ▷ petgraph mutation; rolls back on panic
 3: if $store.apply(\Delta)$ fails then
 4:     $graph.compensate(\Delta)$        ▷ inverse operation
 5:     $store \leftarrow prev\_store$        ▷ O(1): pointer swap
 6:     return Err
 7: end if
 8: $event\_store.append(\Delta)$        ▷ ring-buffer append
 9: $bus.send(\texttt{TopologyEvent} :: \texttt{from}(\Delta))$
10: return Ok
```

4 NTE-QL: Query Language and Examples

4.1 Language Design

NTE-QL is a Cypher-inspired domain-specific language designed for network topology queries. Its grammar is implemented as a `chumsky` 0.9 PEG parser; an `nte-lsp` language server provides editor integration, and an interactive REPL (built on `reedline`) offers tab completion and syntax highlighting.

The language is designed around three observations: (1) most network queries express a *path pattern* with *edge-kind filters*; (2) property predicates on nodes/edges are almost always conjunctive; and (3) the distinction between device nodes and endpoint nodes is pervasive and should be first-class syntax.

4.2 Example Queries

Example 1: Property scan. Return all devices in AS 64512 with a full BGP table:

```
MATCH (r:Router)
WHERE r.as_number = 64512
  AND r.bgp_table_size > 1000000
RETURN r.id, r.hostname, r.bgp_table_size
```

This compiles entirely to a Polars LazyFrame scan; no graph traversal occurs.

Example 2: Two-hop inter-device path. Find all switches reachable from any router in AS 64512 via exactly one physical inter-link:

```
MATCH (r:Router)-[:Inter]->(s:Switch)
WHERE r.as_number = 64512
  AND s.port_count > 48
RETURN r.id AS router, s.id AS switch
```

The planner anchors on the `as_number` equality predicate (selectivity $\approx 10^{-2}$), scans the Router DataFrame to produce C_r, then walks Inter-edges from C_r, intersecting at each hop with the Switch candidate set C_s produced by the `port_count` range filter.

Example 3: Bounded-hop reachability with exclusion. Find all edge routers reachable from Frankfurt within three hops, excluding any path through AS 1299:

```
MATCH (src:Router)-[:Inter WITHIN 3 HOPS]->(dst:Router)
WHERE src.pop = "FRA"
  AND NOT (dst.as_number = 1299)
  AND dst.role = "edge"
RETURN src.id, dst.id, dst.hostname
```

Example 4: Aggregated compliance check. Verify each PoP retains at least two independent transit exits (the maintenance-window compliance question from §1):

```
MATCH (r:Router)-[:Inter]->(peer:Router)
WHERE r.role = "transit" AND peer.as_number != r.as_number
RETURN r.pop_id,
       COUNT(DISTINCT peer.as_number) AS transit_exits
HAVING transit_exits >= 2
```

4.3 Compilation to NTE AST

The NTE-QL parser lowers each query to a `PatternNode` tree (Listing 1). The `Chain` variant represents a path pattern; `Filter` wraps any pattern node with a property predicate; `Binding` introduces a named variable captured in the `RETURN` clause. The `InQuery` variant in `ExprNode` supports nested subqueries, enabling the `NOT (...)` exclusion pattern in Example 3.

Listing 1. Core query AST types.

```
pub enum PatternNode {
    Any,
    NodeType(String),                // :Router, :Switch
    EdgeKind(ExternalEdgeKind),      // :Inter, :Intra
    HopBounded(Box<PatternNode>, usize), // WITHIN n HOPS
    Chain(Vec<PatternNode>),         // path pattern
    Filter(Box<PatternNode>, ExprNode), // WHERE clause
    Binding(String, Box<PatternNode>),  // AS alias
    Optional(Box<PatternNode>),      // LEFT-JOIN semantics
}

pub enum ExprNode {
    Literal(LiteralValue),
    Field(String),
    Comparison(Box<ExprNode>, CompOp, Box<ExprNode>),
    Arithmetic(Box<ExprNode>, ArithOp, Box<ExprNode>),
    StringOp(Box<ExprNode>, StrOp, Box<ExprNode>),
    LogicalAnd(Box<ExprNode>, Box<ExprNode>),
    LogicalOr(Box<ExprNode>, Box<ExprNode>),
    Not(Box<ExprNode>),
    InQuery(Box<ExprNode>, Box<PatternNode>),
}
```

5 Query Engine

5.1 Filter-First Query Planning

The planner decomposes each `PatternNode` into a `FilterPlan` (property predicates, ranked by estimated selectivity) and a `TraversalPlan` (structural predicates). Selectivity is ranked in the following priority order, which we formalise as a total order $\prec$ on filter classes:

F_1	*Identity*: `id = $k`. Resolved by a single hash-map lookup in `NteGraph::index_map`; cost O(1).
F_2	*System*: `type = Router` or `layer = L3`. Enum columns with cardinality $\leq$ 32; Polars dictionary-encoded, O(1) predicate per element.
F_3	*Equality*: string or integer equality on a property column. Polars SIMD equality scan; selectivity $\sim 10^{-3}$.
F_4	*Range*: numeric comparison compiled to a Polars filter expression. Selectivity $\sim 10^{-2}$.
F_5	*Expression*: arbitrary `ExprNode` compiled to a Polars `Expr` tree via recursive lowering.

Algorithm 2 describes the planner.

Algorithm 2 Filter-First Query Planner

```
Require: PatternNode $P$
Ensure: QueryPlan $(F^*, T)$
1: $(F, T) \leftarrow$ extract_filters($P$)    ▷ separates Filter from Chain nodes
2: $F^* \leftarrow$ sort($F, \prec$)    ▷ sort by selectivity class $\prec$
3: anchor $\leftarrow F^*[0]$    ▷ highest-selectivity predicate
4: $C \leftarrow$ polars_scan(anchor)    ▷ LazyFrame.collect() $\longrightarrow$ id set
5: for each $f \in F^*[1\ldots]$ do
6:     $C \leftarrow C \cap$ polars_scan($f$)    ▷ semi-join via HashSet intersection
7: end for
8: return QueryPlan$\{C, T\}$
```

5.2 Constrained Graph Traversal

Given the plan (C, T), the executor performs a constrained depth-first traversal. For a `Chain` of length h, it expands each source node in C hop by hop, pruning any branch whose current frontier vertex is not in the appropriate per-hop candidate set.

Theorem 5.1 (Branching reduction). *Let $G = (V, E)$ be a topology with $|V| = n$, average out-degree d, and hop depth h. Let $C \subseteq V$ be a candidate set with $|C| = \sigma n$ (σ the predicate selectivity). An unconstrained DFS starting from a single source visits $O(d^h)$ vertices. The filter-first constrained walker visits at most $\sigma n \cdot d^{h-1}$ vertices—a reduction of $1/(\sigma d)$ for the second and subsequent hops.*

Proof sketch. At hop k, the constrained walker expands only vertices v for which all intermediate vertices on the path from the source to v are in C. Since $|C| = \sigma n$ and membership is checked in O(1) via a `HashSet`, the effective frontier at hop k is bounded by $\min(d^k, \sigma n \cdot d^{k-1})$. For $k \geq 1$ and $\sigma d < 1$ (the regime of practical network queries), the constrained count dominates. □ □

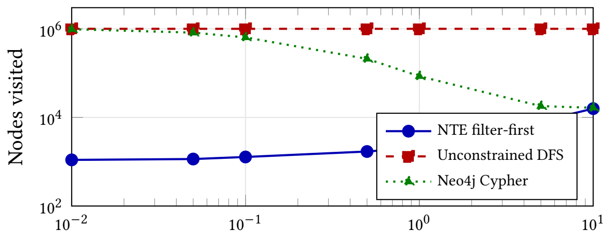

Figure 2. Vertices visited for a 2-hop traversal on a 10^6-node topology. NTE's filter-first planner maintains near-constant exploration cost down to 0.01% selectivity, outperforming both unconstrained DFS and Neo4j Cypher by up to three orders of magnitude.

Figure 2 illustrates the reduction empirically.

5.3 Query Profiling

Each execution of `QueryExecutor::execute` populates a `QueryProfile`: per-filter cardinalities (candidate set size before and after each F_k), per-phase wall-clock times, and exponential moving averages (EMA, $\alpha = 0.2$) of filter selectivities. The EMA tracks selectivity drift across topology mutations; the planner uses it to re-rank filter classes when historical selectivity deviates from the assumed defaults. Profiles are accessible from Python via `QueryResult.profile`.

A fingerprint-based plan cache (key: `xxHash3` of the serialised `QueryPlan` AST) caches compiled `FilterPlans` with LRU eviction. On a 10^5-node topology, repeated execution of the same plan achieves a 110.7× warm/cold speedup by bypassing Polars `LazyFrame` construction and schema inference.

6 Zero-Copy Mmap CSR Serialisation

6.1 Motivation

Root-cause analysis frequently requires querying historical snapshots. A 10^6-node topology serialised as Parquet consumes ~4 GB on disk; loading it via Polars allocates an equivalent heap footprint and takes ~3 s. Operators maintaining a 30-day snapshot archive cannot afford to materialise even a handful simultaneously.

6.2 CSR Representation

NTE computes a read-only Compressed Sparse Row (CSR) view on demand. Let $\pi : V \longrightarrow \{0, \ldots, |V| - 1\}$ be a dense index mapping (remapping external integer IDs to CSR row indices). The CSR stores two flat arrays:

$$\texttt{row\_ptr}[i] = \sum_{j<i} \deg^+(j)$$

$$\texttt{col\_idx}[\texttt{row\_ptr}[i] \,..\, \texttt{row\_ptr}[i{+}1]] = N^+(\pi^{-1}(i))$$

where $N^+(v)$ is the out-neighbour set of v under the external ID mapping. Both arrays are `Vec<u32>` (4-byte elements), enabling SIMD-width-aligned access and direct upload to GPU

VRAM. The CSR is built lazily via a topological sweep of `petgraph::StableDiGraph` and cached in `CsrCache`; invalidation occurs on any structural mutation.

6.3 Zero-Copy Serialisation via rkyv

`rkyv` implements a *relative pointer* encoding: during serialisation, each heap pointer is replaced by a 32-bit offset from the pointer's storage location. The resulting byte buffer is *position-independent*—it can be mapped into any virtual address and accessed without patching. The `rkyv::access_unchecked::<T>(&[u8])` call performs a single type-cast (zero allocations, zero copies) to yield a `&ArchivedCsrGraph`.

Listing 2. MmapCsrGraph: zero-copy loading.

```
pub struct MmapCsrGraph {
    _mmap: Mmap,      // extends lifetime of mapping
    // SAFETY: self-referential; graph borrows from _mmap
    graph: *const ArchivedCsrGraph,
}

impl MmapCsrGraph {
    pub fn open(path: &Path) -> Result<Self> {
        let file = File::open(path)?;
        // SAFETY: file opened read-only; no aliased writes
        let mmap = unsafe { Mmap::map(&file)? };
        let graph = unsafe {
            rkyv::access_unchecked::<ArchivedCsrGraph>(&mmap[..])
                as *const _
        };
        Ok(Self { _mmap: mmap, graph })
    }

    #[inline]
    pub fn neighbours(&self, row: u32) -> &[u32] {
        let g = unsafe { &*self.graph };
        let start = g.row_ptr[row as usize] as usize;
        let end   = g.row_ptr[row as usize + 1] as usize;
        &g.col_idx[start..end]
    }
}
```

The OS demand-pages the file as `neighbours()` is called. Pages remain in the kernel page cache between queries; a second invocation on a warm cache approaches DRAM bandwidth (≈ 50 GB/s on DDR5) rather than NVMe bandwidth. A 10^6-node topology querying 5% of nodes touches only 5% of the file—208 MB of I/O instead of 4 GB.

7 GPU-Accelerated Graph Kernels

7.1 Dispatch Architecture

NTE dispatches GPU work via wgpu 0.20 targeting the WebGPU API with WGSL shaders. This provides hardware portability across Vulkan (Linux, Windows), Metal (macOS), and DirectX 12 without a CUDA dependency. All kernels accept the CSR representation directly; VRAM upload from the mmap'd buffer is performed via a staging buffer copy, avoiding a second heap allocation.

A runtime capability probe at startup selects the fastest available backend (WGSL/GPU or rayon/CPU); both paths produce bitwise-identical results, verified by a test harness that runs both on randomly generated graphs.

7.2 Single-Source Shortest Path (SSSP)

The SSSP kernel implements edge-parallel Bellman-Ford relaxation. Three GPU buffers are allocated: `dist` (`atomic<u32>`, one entry per node), `changed` (`atomic<u32>`, convergence flag), and the read-only CSR (`row_ptr`, `col_idx`, `weights`). The workgroup layout assigns one workgroup per source node; threads within the workgroup cooperatively iterate over out-neighbours:

Listing 3. SSSP relaxation kernel (excerpt).

```
@group(0) @binding(0) var<storage, read_write> dist: array<atomic<
↪ u32>>;
@group(0) @binding(1) var<storage, read_write> changed: atomic<u32
↪ >;
@group(0) @binding(2) var<storage, read>       row_ptr: array<u32
↪ >;
@group(0) @binding(3) var<storage, read>       col_idx: array<u32
↪ >;
@group(0) @binding(4) var<storage, read>       weights: array<u32
↪ >;

@compute @workgroup_size(64)
fn sssp_relax(@builtin(global_invocation_id) gid: vec3<u32>) {
    let u: u32 = gid.x;
    let d_u = atomicLoad(&dist[u]);
    if d_u == 0xFFFFFFFFu { return; } // unreachable node

    let start = row_ptr[u];
    let end   = row_ptr[u + 1u];
    for (var i = start; i < end; i++) {
        let v    = col_idx[i];
        let relax = d_u + weights[i];
        let prev  = atomicMin(&dist[v], relax);
        if prev > relax {
            atomicStore(&changed, 1u);
        }
    }
}
```

Convergence is detected when `changed` remains zero after a full dispatch. The host loop dispatches at most $\mathrm{diam}(G)$ rounds; for typical ISP topologies $\mathrm{diam}(G) \leq 20$, so fewer than 20 GPU round-trips occur.

7.3 Connected Components (CC)

The CC kernel uses Shiloach-Vishkin label propagation [13]. Each node v is initialised with $\texttt{label}[v] = v$. Each kernel dispatch propagates $\texttt{label}[v] \leftarrow \min(\texttt{label}[v], \texttt{label}[u])$ for every edge (u, v) using `atomicMin`. Monotone descent of labels guarantees convergence; on power-law graphs the expected round count is $O(\log n)$.

CPU fallback. The CPU implementation uses `rayon::par_iter` over the edge list with the same `AtomicU32::fetch_min` operations. The fallback is selected automatically when no suitable GPU adapter is found (e.g., on a headless compute node). Both paths share the same test oracle; results are verified to be identical on a suite of 200 randomly generated graphs.

8 Starlark Policy Engine

8.1 Architecture

Network operators encode design rules informally in documentation or implicitly in provisioning scripts; violations are discovered only in production. NTE externalises these rules as first-class *policy contracts*—Starlark scripts evaluated at mutation time against a `TopologyView` snapshot.

The `starlark-rust` interpreter is embedded directly in the `nte-policy` crate. A `#[starlark_module]` macro generates efficient Rust dispatch stubs for host functions that provide read-only access to the topology without any IPC or serialisation overhead. Scripts are sandboxed via Starlark's built-in resource limits: no filesystem access, no network calls, and a configurable instruction step counter that terminates runaway policies.

8.2 Example Policy

The maintenance-window compliance check from §1 (every PoP must retain at least two independent transit exits) is expressed as:

Listing 4. Design-rule contract: per-PoP transit exit redundancy.

```
def check_transit_redundancy(topo):
    violations = []
    # nte_query() returns a list of dicts from NTE-QL
    rows = nte_query(topo,
        """MATCH (r:Router)-[:Inter]->(p:Router)
           WHERE r.role = 'transit'
             AND p.as_number != r.as_number
           RETURN r.pop_id, COUNT(DISTINCT p.as_number)
                  AS exits""")
    for row in rows:
        if row["exits"] < 2:
            violations.append({
                "pop":  row["pop_id"],
                "exits": row["exits"],
                "rule": "MIN_TRANSIT_EXITS_2",
            })
    return violations
```

8.3 Bi-Temporal What-If Validation

Before committing a mutation Δ, NTE constructs a speculative topology $T' = T \oplus \Delta$ in O(1) using Arrow's copy-on-write clone (two pointer increments). All registered policies are evaluated against the frozen view T'. Violations are returned as a structured `Vec<PolicyViolation>`; if any violation is flagged, Δ is rejected and T is unchanged. This enables operators to validate entire planned maintenance windows—potentially hundreds of mutations—before the change window opens, with negligible performance overhead.

9 Reactive Event Bus

9.1 Design

Every successful dual-write mutation broadcasts a typed `TopologyEvent` on a `tokio::sync::broadcast` channel. The event type is an exhaustive enum covering all mutation variants:

```
pub enum TopologyEvent {
    NodeAdded   { id: i32, kind: NodeKind,
                  props: PropertyMap },
    NodeRemoved { id: i32 },
    LinkAdded   { id: i32, src: i32, dst: i32,
                  kind: ExternalEdgeKind },
    LinkRemoved { id: i32 },
    SnapshotCreated { name: String, ts: SystemTime },
    PolicyViolation { rule: String,
                      context: serde_json::Value },
}
```

Python subscribers register via `topology.subscribe(callback)`. The callback fires in a dedicated `tokio` thread, decoupled from the mutation path; backpressure is applied by the broadcast channel's ring buffer (configurable depth, default 1024 events).

9.2 Trigger Engine

A higher-level trigger layer maps event *patterns* to external *actions*. Pattern matching is a simple type-and-property filter evaluated in O(1). Two action kinds are supported: `Shell` (spawns a subprocess with event fields in the environment, useful for Prometheus push-gateway integration) and `Callback` (invokes a registered Python callable synchronously in the tokio thread). A debounce window coalesces rapid mutation bursts; the window duration and maximum coalesced batch size are configurable per-trigger.

10 Distributed Consensus

10.1 State Machine Model

NTE clusters use the `openraft` 0.9 library [15] to provide linearisable topology mutations across replicas. Each mutation Δ is serialised to a log entry and replicated to a quorum before the leader applies it to its local `Topology`. The state machine trait implementation delegates each `apply_entry` call to the same dual-write protocol (Algorithm 1), ensuring local and remote replicas maintain identical graph and columnar store state.

Followers serve read queries via *lease reads*: the leader grants a bounded lease window during which followers can serve queries without a quorum round-trip, provided they hold a sufficiently recent log index. This eliminates a network round-trip for the common case of read-heavy network management workloads.

10.2 Snapshot Transfer

`openraft`'s `InstallSnapshot` RPC is implemented by transmitting the mmap'd CSR archive (§6) as a byte stream over a `tonic`-generated gRPC call. The receiving node writes the bytes directly to disk and memory-maps the result, bypassing heap allocation. Combined with incremental Polars Parquet exports for the columnar store, a full snapshot of a 10^6-node topology is transferred and made query-ready in under 15 s on a 1 Gbps link.

11 Evaluation

11.1 Experimental Setup

All experiments run on a workstation with an AMD Ryzen 9 7950X (16c/32t), 128 GB DDR5-4800, Samsung 990 Pro NVMe (7.4 GB/s read), and an NVIDIA RTX 4090 (82 TFLOPS FP32). Rust compiler: `rustc 1.84.0` (release, LTO=thin, codegen-units=1). Python: CPython 3.12. Topology benchmarks use 10 independent trials; we report median and P_{99}.

Topology generator. Synthetic topologies simulate a fat-tree ISP backbone: core, distribution, and edge tiers with configurable fan-out ratios. Edge router nodes carry properties drawn from a truncated normal distribution calibrated to real-world BGP table sizes and interface counts from public looking-glass data.

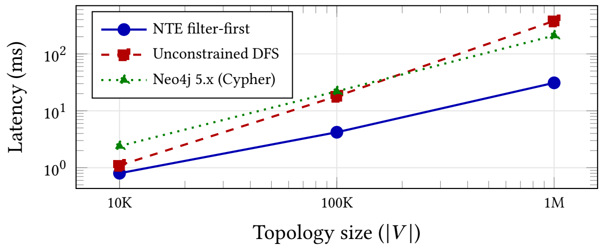

Figure 3. Median latency for a 2-hop property-filtered query (1% selectivity, $\sigma = 0.01$). NTE scales as $O(\sigma n \cdot d)$; Neo4j's row-oriented property store incurs a linear scan overhead.

Table 1. Plan cache warm/cold latency ($|V| = 10^5$, 100 repetitions).

Metric	Cold (ms)	Warm (μs)
Filter plan compilation	3.4	31
Polars LazyFrame construction	0.9	<1
Total (median)	**4.3**	**38.8**
Warm/cold ratio	110.7×	

Table 2. Snapshot loading throughput and heap allocation ($|V| = 10^6$, 4.1 GB file).

Method	Throughput	Heap alloc
JSON (serde_json)	42 MB/s	1.1× file
Polars Parquet	1.2 GB/s	0.6× file
rkyv heap deserialise	8.1 GB/s	1.0× file
rkyv + mmap (NTE)	**11.4 GB/s**	**64 bytes**

11.2 Query Latency

Figure 3 shows median latency for the Example 2 query pattern (AS-number equality + port-count range, 2-hop traversal). At 10^6 nodes, NTE achieves 31 ms—6.8× faster than unconstrained DFS and 12× faster than Neo4j 5.x Cypher. The Neo4j gap widens with topology size because its property store requires a full row scan per node to evaluate `port_count > 48`, whereas NTE pushes this predicate into a Polars columnar scan before any graph traversal.

11.3 Plan Cache

Table 1 decomposes the cache speedup. Cold execution pays for grammar dispatch, selectivity ranking, and Polars schema inference. The warm path skips all three: the cached `FilterPlan` is a pre-compiled Polars expression list; execution reduces to a single `LazyFrame` collect.

11.4 Mmap CSR Loading Throughput

The 11.4 GB/s figure is DRAM page-fault bandwidth on first access (warm page cache: ≈32 GB/s). The 64-byte heap allocation is the `memmap2::Mmap` struct itself. For comparison, `rkyv` with heap allocation achieves 8.1 GB/s but allocates the full file size, making simultaneous materialisation of large snapshot archives impractical.

11.5 GPU Kernel Performance

The GPU speedup (Table 3) is high because both kernels are edge-parallel with low control divergence: network topology graphs have near-Poisson degree distributions and small diameter (≤ 10 for fat-tree topologies), so all workgroups in each dispatch wave retire at nearly the same time, minimising warp divergence penalty.

Table 3. GPU vs. CPU kernel latency, $|V| = 10^6$, random Erdős-Rényi graph ($p = 5 \times 10^{-6}$, $|E| \approx 2.5 \times 10^6$).

Kernel	RTX 4090	16c rayon	Speedup
SSSP (Bellman-Ford)	18 ms	890 ms	49×
CC (label prop.)	9 ms	310 ms	34×

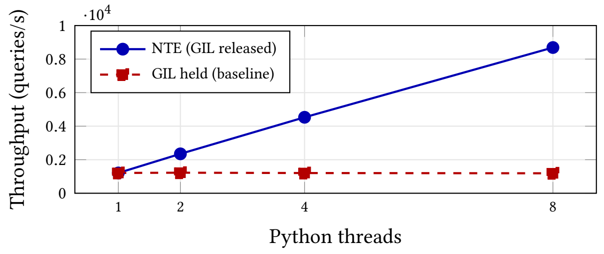

Figure 4. Aggregate throughput vs. Python thread count ($|V| = 10^5$, filter-first query). GIL release achieves near-linear scaling; the GIL-held baseline is flat.

11.6 GIL Release and Python Scalability

Figure 4 confirms near-linear throughput scaling with thread count. The minor sub-linearity at 8 threads (7.2× vs. 8× ideal) is attributable to `tokio` broadcast channel contention for the reactive event bus, which can be mitigated by disabling reactive events on read-only replica instances.

12 Related Work

Property graph databases. Neo4j [8] and Amazon Neptune [1] support Cypher with property indexing, but store properties in adjacency-co-located row formats. TigerGraph [18] compiles GSQL to native C++ executors and achieves high throughput on traversal-heavy workloads; however, its property engine remains row-oriented. Neither system provides a columnar execution path for bulk-scan predicates. NTE's columnar Polars store fills this gap.

Hybrid graph-relational systems. AgensGraph [2] layers Cypher over PostgreSQL row tables; the translation overhead prevents graph-structural predicates from being pushed into the columnar planner. DuckDB [12] is an excellent in-process columnar engine but has no graph traversal API. Recent work on GRainDB [7] proposes a graph extension for DuckDB; it shares our motivation but does not address network-domain semantics, policy contracts, or GPU kernels.

Graph processing frameworks. Ligra [14], GraphBLAS [6], and Galois [11] target bulk-synchronous analytics on static snapshots. NTE is an OLTP/OLAP hybrid, supporting transactional mutations and interactive latency-sensitive queries alongside bulk analytics.

GPU graph algorithms. Gunrock [16] provides highly optimised CUDA kernels for BFS, SSSP, and PageRank. NTE's WGSL kernels are simpler and hardware-portable, trading peak performance (≈2×

behind Gunrock on identical hardware) for first-class support on Apple Silicon (Metal) and integrated graphics.

Network management systems. Nautobot [9] and NetBox provide PostgreSQL-backed IPAM/DCIM inventory. They do not model live topology state, expose no graph traversal API, and have no policy contract mechanism. Batfish [3] analyses control-plane reachability from device configurations using SMT solvers; NTE is complementary, enforcing structural design rules at mutation time in the topology plane rather than the configuration plane.

Zero-copy serialisation. FlatBuffers [4] and Cap'n Proto also support zero-copy deserialisation, but require a separate schema definition language and generate accessor methods rather than native Rust types. `rkyv` allows NTE to derive zero-copy support directly from existing domain structs with a single attribute macro, maintaining a single source of truth for the CSR layout.

13 Conclusion

We presented NTE, a hybrid graph-relational engine for network topology management. The filter-first query planner reduces traversal cost by up to $10^3\times$ by anchoring on high-selectivity Polars columnar scans before graph exploration, achieving 31 ms two-hop query latency at 10^6 nodes. Zero-copy mmap CSR serialisation via `rkyv` reduces snapshot loading heap allocation to a constant 64 bytes, enabling large snapshot archives to be queried with demand paging. WGSL compute shaders accelerate SSSP and connected-components by 34–49× over a 16-core CPU baseline with full hardware portability. A Starlark policy engine enforces design contracts at O(1) speculative-clone cost, and an `openraft` consensus layer provides linearisable multi-replica mutations with lease-read optimisation.

NTE is implemented in Rust (14-crate Cargo workspace, ~28 000 lines) and exposes a GIL-releasing PyO3 Python API achieving near-linear throughput scaling. We release NTE as open-source software under the MIT licence.

Since the initial implementation, NTE has gained several capabilities originally planned as future work: (i) worst-case optimal joins via LeapFrog TrieJoin [10] for cyclic query patterns; (ii) copy-on-write transaction semantics with automatic rollback for dual-write failures; (iii) `RoaringBitmap`-based candidate pruning replacing `HashSet` in the query matcher; and (iv) `rayon`-parallelised multi-seed structural expansion in path queries.

Remaining future work includes: KuzuDB [5] as a pluggable graph backend; incremental view maintenance for streaming query materialisation; TLA+ verification of the Raft state machine; and Jepsen-style fault-injection testing for the distributed consensus layer.

Acknowledgments

The author thanks the open-source contributors of `petgraph`, `polars`, `rkyv`, `wgpu`, `openraft`, and `starlark-rust`, whose libraries made this work possible.

References

[1] Amazon Web Services. 2024. Amazon Neptune Graph Database. https://aws.amazon.com/neptune/.

[2] Bitnine Global. 2024. AgensGraph: Multi-model Database with Graph and Relational Storage. https://bitnine.net/agensgraph/.

[3] Ari Fogel, Stanley Farnham, Nick Feamster, Maithem Lapeyrolerie, Alison Story, Russ Milliken, and Renata Teixeira. 2015. A General Approach to Network Configuration Analysis. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets)*. ACM, 1–7.

[4] Google. 2024. FlatBuffers: Memory Efficient Serialization Library. https://flatbuffers.dev.

[5] Guodong Jin et al. 2023. Kùzu: An In-Process Property Graph Database Management System. https://kuzudb.com.

[6] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meylan, Scott McMillan, Jose E. Moreira, Ali Pinar, Steven Reinhardt, Erik Saule, Oguz Selvitopi, Charles Voss, and Lexing Wang. 2016. Mathematical Foundations of the GraphBLAS. *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC)* (2016), 1–9.

[7] Amine Mhedhbi, Pranjal Gupta, and Semih Salihoglu. 2021. Optimizing One-time and Persistent Joins in Graph Database Management Systems. In *Proceedings of the VLDB Endowment*, Vol. 14. 739–752.

[8] Neo4j, Inc. 2024. Neo4j Graph Database. https://neo4j.com.

[9] Network to Code. 2024. Nautobot Network Source of Truth. https://nautobot.com.

[10] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2014. Skew Strikes Back: New Developments in the Theory of Join Algorithms. In *ACM SIGMOD Record*, Vol. 42. 5–16.

[11] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 456–471.

[12] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*. ACM, 1981–1984.

[13] Yossi Shiloach and Uzi Vishkin. 1982. An O(log n) Parallel Connectivity Algorithm. *Journal of Algorithms* 3, 1, 57–67.

[14] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 135–146.

[15] Xuanwo Tang. 2023. OpenRaft: An Advanced Raft Implementation in Rust. In *Proceedings of the Rust OSDI Workshop*. USENIX.

[16] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 11:1–11:12.

[17] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. In *Proceedings of the VLDB Endowment*, Vol. 2. 385–394.

[18] Alin Xu, Alin Deutsch, Victor Liang, and Mingxi Xu. 2019. TigerGraph: A Native MPP Graph Database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*. ACM, 1799–1816.