Filter-as-Query: Filtered Approximate Nearest-Neighbour Search over SPARQL, where the Filter is an Exact Basic Graph Pattern over the Engine's Own Dictionary Ids

Jesse Wright · the sparq project

A systems-integration contribution. All evidence in this document is deterministic and machine-independent (asserted equivalence invariants, recall floors, a cost-model constant); no wall-clock latency or throughput is claimed — the performance evaluation is specified in Section 5.5. and deferred to the canonical runner.

Abstract

Knowledge graphs increasingly carry dense vector embeddings next to their symbolic triples, and queries increasingly mix the two: *return the nearest neighbours of this embedding, but only those entities that satisfy this graph pattern.* Filtered approximate nearest-neighbour (ANN) systems support such constraints in two ways: by mirroring per-vector attributes onto the index and evaluating a flat predicate over the copy, or — in recent engine-integrated designs — by evaluating the predicate in a host database engine. We describe an RDF-native instance of the second family, implemented inside a dictionary-encoded SPARQL engine: the constraint on a vector-neighbour variable is the join-connected sub-pattern of the query's own Basic Graph Pattern (BGP), evaluated *exactly* by the host engine over its permutation indexes, and materialised as an id-set (`IdMask`) over the *same dictionary-id space* the vectors are keyed on. Relative to per-vector attribute schemes, the filter language widens to the full BGP join — including multi-hop (transitive) and cyclic sub-patterns no per-vector tag expresses. Relative to engine-integrated systems such as VBASE and NaviX, which already evaluate joins in the host engine, the delta is narrower and architectural: one shared key space (no metadata mirroring, no id translation at the boundary) and an enforced answer contract. That contract is scoped precisely. The SPARQL-level filtered path described here is *answer-exact*: both of its physical strategies rank the complete mask-admitted pool, so the filtered top-k is identical to post-filtering the unfiltered result, unconditionally and by construction. Two machine-checked invariants pin that argument at different layers: end-to-end equality of the ordered result ids across single-pattern, transitive, and cyclic constraint shapes, and — at the layer that chooses between the strategies — equality of the two branches' hits, scores included. The engine also carries an *approximate* filtered traversal, used for the recall measurement reported here but not wired into the SPARQL surface; for that path we claim no pre≡post equivalence at all. We are explicit about what this paper is and is not: the ANN traversal is unmodified prior art, the constraint pushdown is a static instantiation of sideways-information-passing, the answer-safety property is mathematically simple, and no performance numbers are reported — the contribution is the integration architecture, its enforced correctness envelope, and a specified evaluation design against the current filtered-ANN state of the art.

1. Introduction

Two representations of the same entities now routinely coexist: a knowledge graph stores what is *asserted* about an entity, and a learned embedding stores what is *similar* to it [1]. Query workloads follow: retrieval-augmented pipelines and hybrid search applications ask for the nearest neighbours of a query vector *subject to* symbolic constraints — the nearest `:Vehicles`, the nearest products still in stock, the nearest papers by authors at a given institution.

The database community has converged on *filtered ANN* as the abstraction for this workload: an approximate nearest-neighbour index whose traversal is constrained to vectors satisfying a predicate [2], [3], [4]. Deployed designs fall into two families. In the first, the predicate is evaluated over per-

vector *metadata* — a scalar tag, label set, or attribute column stored with (or mirrored onto) the vector index [2], [3], [5]. That architecture has three consequences that matter in an RDF setting:

1. **Mirroring and drift.** The attributes the filter can see must be copied out of the primary store into the vector index's metadata columns and kept in sync under updates — a second copy of the data with its own consistency obligations.
2. **A flat filter language.** A per-vector tag can express `colour = red` but not a join — *vehicles whose owner is based in Berlin* — let alone a multi-hop or cyclic graph constraint. The filter language is strictly weaker than the query language of the host system.
3. **An unspecified correctness contract.** Whether the filtered result equals what post-filtering the unfiltered result would have produced is typically a property of the traversal heuristics, observed empirically rather than enforced.

In the second, more recent family, the predicate is evaluated by a host database engine and the index consumes the resulting admissible set: VBASE [6] runs SQL — predicates *and joins* — through PostgreSQL's executor behind a unified index iterator, and NaviX [4] evaluates an ad-hoc selection sub-query inside the Kùzu graph DBMS before the traversal. These systems already escape the flat filter language; what they leave open in an RDF setting is *which key space the admissible set lives in* and *what answer contract the integration enforces*.

This paper describes *Filter-as-Query*, the filtered-ANN integration of the sparq RDF engine. Relative to the per-vector-attribute family it dissolves all three consequences at the architecture level; relative to the engine-integrated family it makes two narrow, deliberate moves. The engine dictionary-encodes RDF terms to integer ids and keys its vectors by those same ids. A vector-neighbour variable in a SPARQL query is constrained by the ordinary triple patterns it joins with; the engine evaluates that sub-BGP — *exactly*, over its existing permutation indexes [7] — and projects the admissible ids into an `IdMask` that the ANN traversal consumes directly. The filter *is* a query: there is no mirrored metadata and no id translation at the boundary, the filter language is the BGP join language itself (including transitive and cyclic sub-patterns; Section 4.), and the narrow-never-widen property is enforced as deterministic, machine-checked invariants — end-to-end across every constraint shape, and, at the layer that chooses between them, across both physical execution strategies.

1.1. Contributions and non-contributions

We claim the following, each forward-referencing its evidence:

- **C1 — Filter compilation into the engine's own id space** (Section 4.). The constraint on a neighbour variable `?n` is the connected component of the BGP join-graph containing `?n`, extracted by a terminating fixpoint (Algorithm 1) that handles multi-hop and cyclic sub-patterns, evaluated exactly by the host engine, and projected as an `IdMask` over the dictionary ids the vectors are already keyed on. No metadata is mirrored; ids are never re-encoded at the boundary. The extraction itself is standard graph reachability and the pushdown it performs is a static sideways-information-passing decision (Section 2.); what we claim is the compilation target — the engine's own id space — not the strategy. The SPARQL surface is a magic-predicate family (`vec:nearest` and `vec:search`); Section 4.2. gives the end-to-end pipeline on a worked query.
- **C2 — An enforced answer-safety envelope** (Section 4.4.). Pre-filtering narrows the candidate set and never changes the answer: the filtered top-k is identical to post-filtering the unfiltered result by the same constraint, asserted end-to-end as equality of the ordered result ids over single-pattern, transitive, and cyclic constraint shapes — each fixture exercising the pre-filter plan its mask selectivity elects under the default cost model — with strategy-agnosticism asserted separately, and with scores, at the cost layer. The property is scoped precisely to the path that carries it: the SPARQL-level filtered path is answer-*exact* (both physical strategies rank the complete admitted pool), so

the equivalence holds by construction and unconditionally there. The crate's *approximate* filtered traversal is a separate path, not reached from the SPARQL surface, and we claim no equivalence for it (Section 4.4.). C2′s value is that it is *enforced* — end-to-end from the SPARQL surface, and across the physical pre/post decision of C3 — not that it is deep (Section 4.5.): as Section 2. notes, narrowing-without-falsification is the defining property of a semi-join, and we present it as an enforcement result rather than a discovery.

- **C3 — A deterministic pre/post-filter decision rule with a confined failure mode** (Section 4.6.). A one-constant cost model chooses between scanning the mask and scanning the store; both branches provably return the identical answer, so a mis-estimate costs throughput, never correctness.
- **C4 — An honest evaluation protocol** (Section 5.). Every number in this paper is injected at build time from an evidence ledger whose records are labelled by environment; headline tables can only cite deterministic, machine-independent records (the build fails otherwise). Because the canonical performance runner is not yet available, we report *no* latency or throughput and instead specify the evaluation design — baselines, datasets, metrics, and the results that would falsify the approach. We say *specified* and not *pre-registered*: no public, timestamped registry deposit exists, and we do not claim the epistemic warrant that pre-registration would carry.

Equally important is what we do *not* claim. The ANN index and its traversal (an HNSW-style graph [8]) are unmodified prior art; we contribute no new traversal, pruning, or termination strategy, and systems whose contribution is a better constrained traversal [2], [3], [4] are complementary rather than competitors: our mask could feed any of them. The BGP-join filter language is a widening only relative to per-vector attribute schemes — engine-integrated systems [4], [6] express joins too, and Section 2. states that comparison carefully. The mask-caching layer that memoises BGP⟶`IdMask` derivations is an engineering optimisation, not a contribution. And this paper contains no performance evaluation: we state that gap plainly in Section 6. and specify the evaluation design in Section 5.5. rather than let two correctness tables masquerade as one.

2. Related work

Graph-based ANN indexes. HNSW [8] and DiskANN [9] are the dominant graph-traversal ANN families; Faiss [10] is the standard library baseline. sparq's vector index is an HNSW-style graph; nothing in this paper modifies it.

Filtered ANN over per-vector attributes. Filtered-DiskANN [2] constrains a DiskANN graph by per-vector *labels*, building label-aware graph edges; filters are conjunctions over a bounded label vocabulary attached to each vector. ACORN [3] is *predicate-agnostic*: it accepts an arbitrary per-query predicate set (in the limit, a bitset over vector ids) and searches a denser HNSW variant under it, explicitly decoupling the filter's semantics from the index. SeRF [5] specialises to range filters. A 2025–2026 wave extends the attribute-filtered space along three axes. PathFinder [11] supports *conjunctions and disjunctions* over multiple attributes by selectively building attribute-specific indexes and planning across them (including borrowing an index built for one attribute to filter on another) with a cost-based optimizer. EMA [12] targets *general* (mixed numeric and categorical) attribute filtering *with dynamic updates*, attaching compact per-edge summaries ("markers") that guide the traversal without false negatives. E2E [13] learns a lightweight per-query cost predictor from early-probe statistics (attribute distributions, intermediate distances) and *adaptively terminates* the filtered traversal per query. A unified benchmark [14] standardises evaluation across the filter-then-search / search-then-filter / hybrid taxonomy with selectivity- and difficulty-stratified analysis. All of these operate over per-vector attributes; the question this paper addresses — *who derives the admissible set, over which key space, and under what enforced contract* — is upstream of, and complementary to, each of them.

Engine-integrated filtered search — the critical comparison. The closest prior systems evaluate the filter in a host engine rather than over mirrored tags, and against them the expressivity contrast

above *disappears*, so we state the comparison carefully. NaviX [4], inside the Kùzu graph DBMS, answers filtered queries in which the admissible subset is defined by an *ad-hoc selection sub-query* in the DBMS's query language — joins expressible — evaluated by the query processor *before* the traversal (prefiltering), with an adaptive choice of search heuristic per node driven by local selectivity. VBASE [6] integrates vector indexes into PostgreSQL behind a unified iterator; its filter language is SQL — predicates and joins — evaluated by the relational executor, and its early termination rests on a *relaxed monotonicity* property under which it argues result equivalence with top-k-based execution. AnalyticDB-V [15] and Milvus [16] combine relational predicates over attribute columns with vector search, including pre-/post-filter strategy switching. Relative to this family our delta is narrow, and we tabulate it as such (Table 1): (i) an *RDF instantiation* in which the constraint is a BGP with standard SPARQL semantics and the admissible set is materialised in the engine's term-dictionary id space, which is *identically* the vector table's key space — no attribute columns co-located with the index, nothing mirrored, no id translation at the boundary; (ii) an answer contract that is *enforced* (build-failing deterministic equivalence assertions across constraint shapes end-to-end, and across both physical plans at the layer that selects between them) rather than argued from a traversal property or measured; and (iii) a deliberately minimal one-constant strategy rule whose mis-selection is proven answer-invariant (Section 4.6.). Whether the shared-id-space design yields a measurable memory or freshness advantage over these systems is an empirical question this paper defers to the specified evaluation (Section 5.5.); we do not claim it here.

System	Filter language	Filter evaluated by	Vector key space	Answer contract
Filtered-DiskANN [2]	label conjunctions	index (label-aware edges)	index ids + mirrored labels	empirical recall
ACORN [3]	arbitrary per-query predicate / bitset (caller-computed)	caller supplies; index traverses under it	index ids + mirrored attributes	empirical recall
NaviX [4]	ad-hoc selection sub-query in the DBMS query language (joins expressible)	Kùzu query processor, before the traversal (prefiltering); adaptive per-node search heuristics	DBMS-internal node ids (vectors native to the DBMS)	empirical recall, evaluated for robustness across selectivity and correlation
VBASE [6]	SQL — predicates and joins	PostgreSQL executor, through a unified index iterator	attribute columns co-located with the vector index in the host store	result equivalence to top-k-based execution, argued under a relaxed-monotonicity assumption on the traversal
Milvus / AnalyticDB-V [15], [16]	relational predicates over attribute columns	relational executor over co-located / mirrored columns	index ids + attribute columns	empirical / engine-specific
Filter-as-Query (this work)	full BGP join incl. transitive + cyclic sub-patterns	host SPARQL engine, exactly, over permutation indexes	the engine's dictionary ids (identically the vector key space; nothing mirrored)	enforced pre≡post equivalence (deterministic asserts; unconditional on the SPARQL-level path, which is answer-exact; nothing claimed for the approximate traversal)

Table 1: Qualitative delta against the nearest prior art, from the cited papers' published descriptions. Join-expressive filters are *not* unique to this work: NaviX and VBASE evaluate them in their host engines — the expressivity contrast holds only against the per-vector-attribute rows. The traversal machinery of the attribute-filtered systems is a contribution we do not compete with (our mask could feed any predicate-agnostic index). No performance ordering is implied by this table.

Constraint pushdown and semi-joins — the lineage we inherit. The mechanism by which the constraint reaches the index is not new, and positioning it as new would be an overclaim. Deriving the exact projection of one sub-query and using it to narrow another is the semi-join [17], and computing such a narrowing set *before* evaluating the operation it constrains is sideways-information-passing, the strategy the magic-sets rewriting family formalised for recursive programs [18]. The idea is native

to RDF engines in particular: RDF-3X passes filters sideways between the joins of a BGP to prune scans [19]. Our Algorithm 1 is, in these terms, a *static* sideways-information-passing decision — take the join-connected component, evaluate it exactly, pass the projection on — and its termination and correctness arguments are the ordinary ones for that family rather than new results. Two consequences we accept rather than paper over. First, the answer-safety of Section 4.4. is a semi-join property, not a discovery of ours; what we contribute is that it is machine-enforced end-to-end (Section 4.5.). Second, the classical failure mode of sideways-information-passing applies unchanged here: the derived set can cost more to compute than the operation it narrows. That is precisely the mask-derivation cost we flag as unmeasured in Section 4.3. and Section 6.. The transfer to filtered ANN — the narrowed operation being a vector traversal rather than another relational join — is the part we claim, together with the shared key space that makes the transfer free.

Vector search in RDF and graph stores. Property-graph and RDF stores increasingly bundle vector indexes (NaviX in Kùzu is the research frontier [4]; several production stores ship vector plugins). To our knowledge no prior RDF engine compiles the SPARQL BGP itself into the ANN filter over a shared dictionary-id space; dictionary encoding with permutation indexes is, however, entirely standard RDF-engine architecture [7] — our point is precisely that the standard architecture already contains the right key space for filtered ANN, if the integration is done inside the engine rather than beside it.

3. Preliminaries

Let I, B, L be the pairwise-disjoint sets of IRIs, blank nodes, and literals; an RDF graph G is a finite set of triples (s, p, o). A *dictionary-encoded* engine assigns each term a unique integer id via a bijective dictionary $\mathrm{dict} : I \cup B \cup L \to \mathbb{N}$ and stores G as integer triples in several sort orders (permutation indexes) [7]. A basic graph pattern (BGP) P is a set of triple patterns over terms and variables; its semantics $[\![P]\!]_G$ is the set of solution mappings from the variables of P to terms, per Pérez et al. [20] and the SPARQL 1.1 recommendation [21]. For a variable x, $\pi_x([\![P]\!]_G)$ denotes the projection of the solutions onto x.

The engine additionally stores a vector table $V : \mathrm{dom(V)} \to \mathbb{R}^d$ *keyed by dictionary id*, with $\mathrm{dom(V)} \subseteq \mathrm{range(dict)}$: an embedding is attached to an RDF term by id, not by a separate vector-store key. A k-NN query is (q, k) with $q \in \mathbb{R}^d$; the exact answer over a candidate set $C \subseteq \mathrm{dom(V)}$ is the k ids in C minimising the distance to q (deterministic id-order tie-break). An ANN index answers approximately; its quality is measured by recall@k against the exact answer. A *filtered* k-NN query additionally supplies an admissible set $M \subseteq \mathrm{dom(V)}$ (here: an `IdMask`), and must answer over $C = \mathrm{dom(V)} \cap M$.

4. Filter-as-Query

4.1. The constraint is the join-connected sub-BGP

Consider a query whose BGP contains a vector-neighbour variable `?n`, declared by the engine's kNN operator (Section 4.2. gives the concrete SPARQL surface). Ordinary triple patterns mentioning `?n` — directly (`?n a :Vehicle`) or through intermediate variables (`?n :ownedBy ?p . ?p :basedIn :Berlin`) — constrain which ids `?n` may bind to. The constraining sub-pattern is *not* just the patterns that mention `?n`: it is the connected component of the BGP's join-graph (patterns as nodes, edges between patterns sharing a variable) that contains `?n`. A pattern with no shared-variable path to `?n` cannot constrain it and is excluded — its bindings join in later, unaffected by the vector search.

```
extract(P: BGP, ?n: neighbour variable) -> C(P, ?n):
  V <- { ?n }          // reached variables
  C <- {}              // constraining sub-BGP
  repeat until no change:
    for each t in P \ C:
      if vars(t) ∩ V ≠ {}:
        C <- C ∪ {t};  V <- V ∪ vars(t)
  return C
```

Algorithm 1: Connected-component constraint extraction. The worklist fixpoint terminates because C grows monotonically and is bounded by P; cyclic sub-BGPs (a back-edge to `?n`) add no complication because membership, not path enumeration, drives the loop.

The engine then evaluates $C(P, \text{`?}n\text{`})$ — exactly, with its ordinary BGP machinery over the permutation indexes — and materialises the mask $M = \text{dict-ids}(\pi_{\text{`?}n\text{`}}([\![C(P, \text{`?}n\text{`})]\!]_G))$. Because $[\![\cdot]\!]_G$ is the engine's own exact semantics, M contains *precisely* the ids consistent with the constraint: no approximation enters through the filter, only through the ANN traversal it feeds. A cyclic component such as `{ ?n :owns ?x . ?x :ownedBy ?n }` is handled identically — Algorithm 1 terminates on membership, and the exact evaluation of the cyclic join is the engine's job, not the index's.

4.2. The query surface, end to end

The kNN operator is surfaced in plain SPARQL as a magic-predicate family in the `vec:` namespace: `?n vec:nearest ( q k )` binds `?n` to the k nearest neighbours of q (a seed IRI whose stored vector is used, or a numeric vector literal), and `( ?n ?score ) vec:search ( q k )` additionally binds the cosine score. The argument lists are ordinary SPARQL RDF collections; malformed shapes (non-variable neighbour positions, non-constant q or k) are hard query errors rather than silent mismatches. The following worked query drives the whole pipeline of Figure 1:

```
PREFIX vec: <http://sparq.dev/vec#>
PREFIX ex:  <http://example.org/>
SELECT ?n ?owner WHERE {
  ?n vec:nearest ( "0.12,0.94,..." 10 ) .  # bind ?n to the 10 nearest
  ?n ex:ownedBy ?owner .                    # constrains ?n directly
  ?owner ex:basedIn ex:Berlin .             # constrains ?n via ?owner
  ?d ex:caption ?c .                        # disconnected: joins later
}
```

The rewrite proceeds in five stages. (1) The query is parsed to the SPARQL algebra and the `vec:` pattern identifies the neighbour variable `?n` and its arguments. (2) Algorithm 1 extracts the connected component of `?n` — here `{ ?n ex:ownedBy ?owner . ?owner ex:basedIn ex:Berlin }`; the caption pattern shares no variable path to `?n` and is set aside. (3) The engine evaluates that sub-BGP exactly and projects `?n`, yielding the `IdMask` of dictionary ids of entities owned by Berlin-based owners. (4) The decision rule of Section 4.6. picks the pre- or post-filter physical plan, and the filtered k-NN runs over the vector table — which is keyed by the same dictionary ids the mask contains, so the mask is consumed as-is. (5) The k resulting (id, score) pairs are inlined into the algebra as a `VALUES` table binding `?n` (and the score variable for `vec:search`), and the engine evaluates the full query normally: every remaining pattern — including the disconnected one — joins against the `VALUES`-bound `?n` like any other bindings. `VALUES` is unordered, so best-first order is recovered by `ORDER BY` on the bound score when it matters. The engine's parser, planner, and executor are otherwise unchanged: the integration is a query rewrite plus the mask seam into the vector subsystem. Each `vec:` request in a

BGP derives its own component mask independently; a neighbour variable constrained by no pattern falls back to the plain unfiltered search.

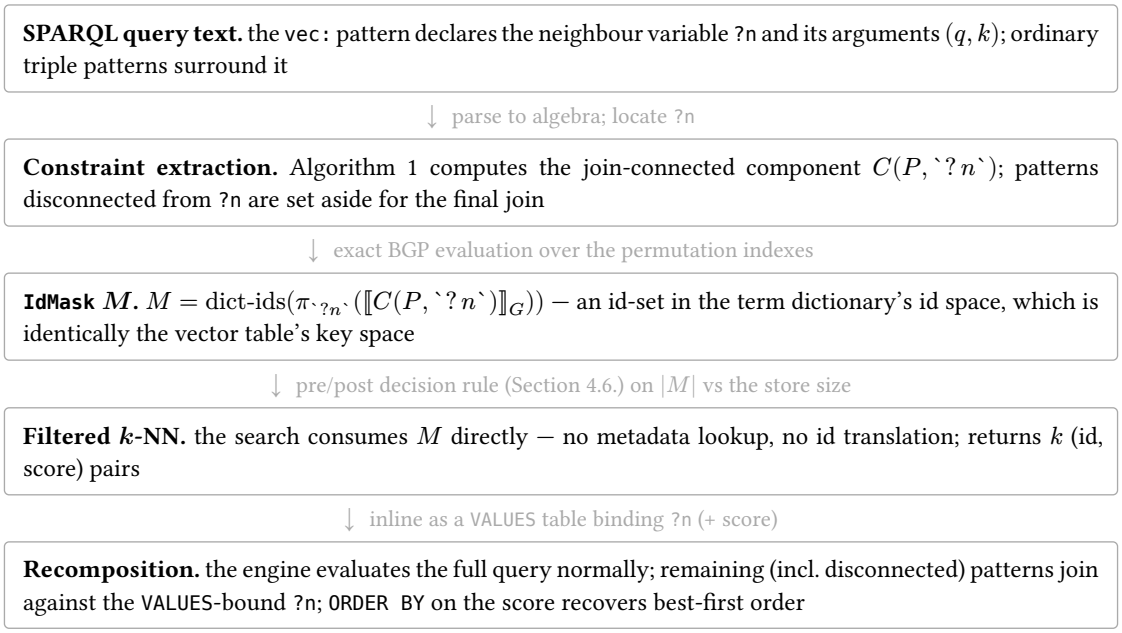

Figure 1: The Filter-as-Query pipeline. The `IdMask` is derived *before* the ANN traversal by the engine's own exact BGP evaluation and consumed by the traversal in the same dictionary-id space; the vector search's bindings re-enter the query as an ordinary `VALUES` table, so the engine's planner and executor are unchanged.

4.3. Why the shared id space matters

The mask's elements are dictionary ids, and the vector table is keyed by dictionary ids. Three practical properties follow. *No mirroring:* there is no per-vector metadata column to extract, denormalise, or keep consistent — the constraint reads the primary triple indexes. *No boundary translation:* the mask flows into the traversal as-is; there is no join between a vector-store id space and an engine id space at query time. *Full filter expressivity for free:* any future improvement to the engine's BGP evaluation (new join algorithms, better cardinality estimation) is automatically an improvement to the filter, because the filter *is* a query.

The honest converse: deriving M costs an exact BGP evaluation, which for an expensive constraint (a large transitive component over a big graph) may dominate the vector search itself. We treat mask-derivation cost as a first-class open measurement in the deferred evaluation (Section 5.), not as a solved problem; a memoising mask cache (keyed by the constraining sub-BGP and a graph fingerprint, so any genuine graph change invalidates it) exists in the implementation but is an optimisation, not a contribution. We also note that engine-integrated systems achieve closely related properties inside their own stores — NaviX keys vectors by DBMS-internal node ids [4] — so the claim here is the RDF instantiation and its enforced contract, not exclusivity of the idea; whether the shared dictionary-id design buys measurable memory or freshness advantages is a question the specified evaluation (Section 5.5.) exists to answer.

4.4. Answer-safety as an enforced invariant

The correctness contract is *narrow-never-widen*: constraining the search must only remove candidates, never change what is returned for the candidates that remain.

Proposition (exact path). For the exact evaluation path, the filtered top-k over $\text{dom}(\text{V}) \cap M$ equals the result of ranking all of $\text{dom}(\text{V})$, deleting non-members of M, and truncating to k; in particular it is a

subset of any unfiltered ranking prefix that contains at least k members of M. *Proof sketch:* deleting non-members commutes with ranking by distance under a deterministic total order (distance, then id); both sides denote the k minimal admissible elements. Because the exact path ranks the *entire* store, no under-fill boundary exists: if M admits fewer than k stored vectors, both sides are equally short. ■

Which path carries the proposition. The precondition of that proposition — that the ranking is over the entire admitted pool — is met by the SPARQL-level filtered path described in Section 4.2., and this is the load-bearing scoping fact of the paper. Every filtered `vec:` request is served by the cost-model seam of Section 4.6., and *both* of its strategies produce a complete admitted ranking: the pre-filter branch ranks all of M, and the post-filter branch ranks all of $\mathrm{dom}(\mathrm{V})$ and then drops non-members. Neither branch is a bounded traversal, so neither can under-fill, and the equivalence is unconditional on the path a SPARQL user reaches.

Why we claim nothing for the approximate path. The crate also implements an approximate filtered traversal (the graph index's own mask-aware search), and it is that traversal — not the `vec:` path above — which the filtered recall floor of Table 3 measures. For it we claim no pre≡post equivalence, and we want to be precise about why, because a weaker statement would be easy to write and would be wrong. It is tempting to say the equivalence holds whenever at least k mask-admissible candidates are surfaced within the traversal's explored frontier. That condition is *necessary but not sufficient*: pre-filtered and post-filtered traversals explore *different* frontiers, so surfacing k admissible candidates does not make them the same k that post-filtering the unfiltered traversal would have yielded. A sufficient condition would have to quantify over the true admitted top-k, which is exactly what an approximate index does not guarantee it visits. We therefore state the honest contract for that path as a measured recall floor (Table 3) and nothing stronger.

The engine does carry an iterative over-fetch loop — size the initial fetch by the mask's selectivity, post-filter, and grow the fetch until k admissible survivors are found or the backend is exhausted — behind a pluggable backend seam, and it repairs *under-fill* (returning fewer than k when k admissible vectors exist) rather than recall. We flag plainly that this seam is exercised by its own tests and is *not* wired into the `vec:` rewrite: it is provision for a future approximate filtered path, not a mitigation shipping underneath the results reported here (Section 6.).

4.5. Scope: the invariant is shallow by design

For an *exact* mask the proposition above is near-immediate, and we do not present it as mathematically deep. Answer-safety deserves space for its *enforcement surface*, not its depth. In deployed filtered-ANN systems the analogous property spans several moving parts — the mirrored metadata's freshness, the traversal's pruning heuristics, the pre/post strategy switch — and typically holds empirically rather than by contract. Here it is pinned as a set of deterministic, machine-checked equivalences (fixed fixtures, fixed seeds) at two layers: end-to-end from the SPARQL surface, across every constraint shape the compiler accepts — single-pattern, transitive, and cyclic, each compared as the ordered list of result ids — *and*, on the decision rule below, across *both physical plans*, forced in turn on one fixture and compared with their scores. The two are separate assertions rather than a cross-product: the constraint-shape fixtures run whichever plan their mask selectivity elects (at their 50% selectivity, the pre-filter branch), and it is the cost layer that establishes the plan choice is not observable in the answer. A regression in either layer fails the build rather than skewing an experiment. The evidence table is in Section 5.; its scope (unconditional on the SPARQL-level filtered path; nothing claimed for the approximate traversal) is stated in the proposition above. The property itself is not ours to claim as novel: narrowing a search by the exact projection of a sub-query, without falsifying its result, is the defining property of a semi-join (Section 2.). What we contribute is that the property is *held* across the whole integration, by construction and under CI, rather than argued or observed.

4.6. A deterministic pre/post-filter decision rule

Given M, the engine chooses *how* to apply it: visit only masked ids (pre-filter), or run the unfiltered scan and drop non-members (post-filter). With $m = |M|$, $n = |\mathrm{dom(V)}|$, and a single modelled constant `scatter_penalty` pricing a scattered masked-row access against a sequential row, pre-filter is chosen iff $m \cdot \mathrm{scatter\_penalty} \leq n$. With the default penalty the crossover sits at a mask admitting at most 0.5 of the store.

For completeness — because a reader comparing the implementation against this section would otherwise find a discrepancy — the engine carries *two* such decisions, at different layers and with deliberately different constants. The rule just given governs the exact SPARQL-level path this paper describes. The approximate filtered traversal (Section 4.4.) has its own, far more conservative line: it pre-filters only below roughly a hundredth of the store or a small absolute floor of nodes, because above that line an exact scan of the mask stops paying against a beam-widened traversal. Only the first rule bears on any claim made here; we name the second so that the two constants are not mistaken for one.

This is a heuristic over an estimate, and we make no claim that the constant is optimal on any hardware — that is exactly the kind of claim this paper refuses to make without the canonical runner. The property we *do* claim is architectural: both branches return the identical top-k — asserted at this layer by forcing each strategy in turn on one fixture and comparing the returned hits and scores — so the decision rule's failure mode is confined to throughput. A mis-estimated crossover can make a query slower; it cannot make it wrong. This separation — correctness pinned by invariant, performance left to honest measurement — is the design stance of the whole integration.

5. Evaluation

5.1. Methodology and evidence discipline

Every number in this paper is injected at build time from a versioned evidence ledger; the build fails if a headline table cites any record not labelled *canonical* — deterministic, machine-independent, asserted in CI (fixed seeds, fixed fixtures, `assert!` floors). Non- canonical (work-box, "indicative") measurements are barred from headline use by the same gate. This discipline is why the present section contains correctness evidence only: *no canonical performance environment exists yet*, and we prefer an honest gap to an unreproducible number. The provenance stamp on the published page records the evidence commit.

5.2. What is established: answer-safety across constraint shapes

Constraint shape	pre-filter ≡ post-filter	evidence
single-pattern (`?n a :Vehicle`)	holds	`filtered_bgp.rs`
transitive (2-hop join)	holds	`filtered_bgp_transitive.rs`
cyclic sub-BGP	holds	`filtered_bgp_cyclic.rs`

Table 2: Enforced answer-safety across constraint shapes. Each cell is a deterministic equivalence proven by assertion over a fixed fixture — the filtered top-k, as an ordered list of result ids, equals post-filtering the unfiltered result by the same constraint, and is a subset of it. Scope: these fixtures exercise the SPARQL-level filtered path, which is answer-exact, so the equivalence is unconditional there (Section 4.4.); each runs the plan its mask selectivity elects under the default cost model (here the pre-filter branch), the plan choice itself being pinned as unobservable by the separate cost-layer assertion of Section 4.6.; and no equivalence is claimed, or tested, for the approximate traversal of Table 3. A cell cannot silently render empty: a canonical record that ceased to be true fails the build, which is the property Section 4.5. claims. Per Section 4.5., read this as an *enforcement* result (the invariant is CI-pinned end-to-end), not a deep one.

5.3. What is established: the approximation budget

The mask is exact; the only approximation is the ANN traversal it feeds. The relevant sanity question is whether the *filtered* traversal preserves the index's recall against the *exact-filtered* ground truth. On deterministic fixtures it does, with asserted floors. One attribution matters and is easy to get wrong: the second row below measures the graph index's own mask-aware traversal against exact-filtered ground truth. That traversal is *not* the path a `vec:` query takes (Section 4.4.) — the floors bound the approximation the integration would inherit were an approximate backend wired in, and they carry no implication for the answer-exact path whose invariants Table 2 pins.

Setting	vectors × dim	recall@10 floor	queries
unfiltered HNSW vs exact brute force	50,000 × 32	0.95	100
approximate filtered traversal vs exact-filtered (≈50% mask; not the `vec:` path)	20,000 × 32	0.9	100

Table 3: Deterministic recall floors (asserted lower bounds over fixed seeds). These are correctness sanity checks on an unmodified HNSW-style index under masking — they bound the approximation the integration *would* inherit from an approximate backend. They are *not* a recall-versus-latency evaluation and we do not present them as one: a single broad selectivity (a mask admitting about half the store), synthetic vectors, and low dimensionality (see Section 6.). The second row measures the graph index's mask-aware traversal, which the SPARQL surface does not reach; neither row characterises the answer-exact path of Table 2, whose recall is by construction 1.0.

evidence: `crates/sparq-vectors/tests/recall.rs::hnsw_recall_at_10_vs_brute_force_on_50k` (environment: canonical)

5.4. What is established: the decision constant

The pre/post crossover of Section 4.6. is pinned as a canonical constant — a mask admitting at most 0.5 of the store selects the pre-filter branch under the default scatter penalty — together with the branch-equivalence assertions of Table 2. No claim is made about where the crossover *should* sit on real hardware; that is a measurement, and it is deferred.

5.5. What is not established, and the specified design to establish it

This paper reports no latency, no throughput, no recall-versus-latency Pareto, no selectivity sweep, and no comparison measurement against any external system: it is a systems description whose empirical

case is still owed. To make the deferred evaluation falsifiable rather than aspirational, we specify its design:

- **Baselines.** ACORN [3] (predicate-agnostic, accepts our mask directly — the critical comparison, since it isolates the value of *in-engine* mask derivation), NaviX [4] (the closest engine-integrated system), post-filtered unmodified HNSW, and exact filtered scan. Where the unified filtered-ANN benchmark [14] applies, its harness and datasets take precedence over bespoke ones for comparability.
- **Workloads.** (i) Standard filtered-ANN benchmark suites for the flat-predicate regime — the regime where we expect *no* advantage and must show we are not worse; (ii) a KG-native workload — entity embeddings over a public knowledge graph with BGP constraints of graded join depth (1–3 hops, with and without cycles) and graded selectivity — the regime flat-attribute systems cannot express natively, which exists to measure whether the expressivity delta carries real workloads rather than to flatter the system. (NaviX can express joins; for it the KG-native workload measures the shared-id-space and contract deltas at equal expressivity, not an expressivity gap.)
- **Metrics.** Recall@10 versus queries-per-second Pareto frontiers per selectivity band; mask-derivation time reported *separately* and end-to-end (the integration's honest overhead, amortised and unamortised — the mask cache disabled and enabled); index memory including any metadata mirroring the baselines require and we avoid — the direct test of whether the shared-id-space delta has measurable value.
- **Selectivity sweep.** Mask fractions from highly selective through the cost-model crossover to near-unfiltered, run against an approximate backend wired into the `vec:` path, so that the under-fill regime Section 4.4. leaves unclaimed is actually exercised.
- **Environment.** The canonical bare-metal runner, with every number environment-labelled under the discipline of Section 5.1.; nothing measured on shared development hardware will be reported as evidence.
- **Falsification criteria.** The integration thesis *fails* if (a) mask derivation dominates end-to-end latency at realistic selectivities on the KG-native workload even amortised, or (b) the shared-id-space design shows no measurable cost advantage (memory or freshness/update overhead) over mirrored-metadata baselines at equal recall, or (c) filtered-traversal recall collapses at selective masks with no viable fallback crossover. Publishing the evaluation commits to reporting these outcomes if they occur.

6. Limitations

- **No performance evidence.** The largest limitation is stated throughout: this paper contains no wall-clock measurement of any kind. The evaluation design of Section 5.5. is specified but unexecuted, blocked on the canonical runner. Until it runs, the claim inventory is architectural and correctness-only.
- **Answer-safety is shallow, and inherited.** Per Section 4.5., the headline invariant is near-immediate for an exact mask, and per Section 2. it is the semi-join property rather than a new one; its value is enforcement breadth, not depth. Readers seeking a theoretical contribution will not find one here.
- **The pushdown mechanism is not novel.** The constraint extraction of Algorithm 1 is a static instantiation of sideways-information-passing (Section 2.). We claim its transfer to filtered ANN over a shared dictionary-id space, not the strategy.
- **Recall floors are sanity checks.** Table 3 covers one broad selectivity, synthetic low-dimensional vectors, and modest scale — floors chosen to be deterministic in CI, not to characterise the index. They bound nothing about behaviour at high selectivity or at realistic embedding dimensionality.

- **The approximate filtered path is unclaimed, and its over-fetch is unwired.** The answer-safety result covers the SPARQL-level path only, which is answer-exact. The crate's approximate filtered traversal has no equivalence claim (Section 4.4.) and no equivalence fixture. The iterative over-fetch that would repair under-fill for such a path exists behind the backend seam but has no caller outside its own tests: it is not a mitigation operating beneath any result reported here, and we would be overclaiming to present it as one. Wiring an approximate backend into `vec:` — and pinning its selective-mask behaviour — is future work, probed by the selectivity sweep of Section 5.5..
- **Mask-derivation cost is unmeasured.** An expensive constraint component may cost more than the search it narrows; the crossover between "derive the mask" and "post-filter without one" is precisely the kind of question only the deferred evaluation can answer.
- **Updates and concurrency.** The mask cache is invalidated soundly (keyed by a graph fingerprint, so any genuine graph change forces re-derivation), but the intra-query snapshot semantics of a derived `IdMask` under concurrent writes — the mask versus the store state the traversal reads — are not formally treated in this paper.
- **Single node.** The integration is single-node; interaction with federation is out of scope.
- **Related-work basis.** The feature table (Table 1) is qualitative and compiled from the cited papers' published descriptions; we have not reproduced any baseline's artifact. No claim in this paper depends on their internals.

7. Conclusion

Filter-as-Query is a small architectural thesis executed inside a real RDF engine: if the engine dictionary-encodes its terms and keys its vectors by the same ids, then the right filtered-ANN filter is not a mirrored attribute but *the query itself* — the join-connected sub-BGP, evaluated exactly by the machinery that already exists, projected into the id space the index already speaks. The filter language becomes the BGP join language, transitive and cyclic constraints included — a strict widening over per-vector attribute schemes, and parity with engine-integrated evaluators; metadata mirroring disappears; and the narrow-never-widen contract — the semi-join property this design inherits rather than invents — is enforced end-to-end as build-failing invariants rather than observed empirically, unconditionally on the answer-exact path the SPARQL surface reaches, with strategy mis-selection confined, provably, to throughput. What remains — and what we have specified rather than performed — is the honest performance case against ACORN-class and engine-integrated filtered search. The architecture stands or falls on that measurement, and this paper is written so that either outcome is reportable.

8. Artifact availability

The engine is open source, and every value this paper reports in a headline table — the measured floors and constants, and the boolean answer-safety records those tables render — is a build-time citation of a committed evidence record rather than a transcribed number, so those values cannot drift from the tests that produced them without failing the build. That guarantee is scoped to exactly them: the paper's prose, architectural, and novelty claims — including the scoping of the answer contract in Section 4.4. — are ordinary prose, checked by review rather than bound to the ledger, and can drift as any paper's can. The three answer-safety invariants of Table 2 are the assertions in `filtered_bgp.rs`, `filtered_bgp_transitive.rs`, and `filtered_bgp_cyclic.rs`; the recall floors of Table 3 are asserted in `recall.rs` and `filtered.rs`; the decision constant of Section 5.4. is pinned by `cost_model.rs` against the model in `cost.rs` — all under `crates/sparq-vectors`. Constraint extraction (Algorithm 1) and the `VALUES` recomposition of Section 4.2. live in the same crate's query-rewrite module, and the pluggable backend seam that Section 6. describes as unwired is `backend.rs`. Each record cited here carries its own source path in the evidence ledger, and the published page stamps the evidence commit, so a reader can resolve any figure in this paper to the test that produced it. No baseline system's

artifact was reproduced (Section 6.), and there is nothing further to release for the deferred evaluation of Section 5.5., which has not run.

References

[1] A. Bordes, N. Usunier, A. Garcia-Durán, J. Weston, and O. Yakhnenko, "Translating Embeddings for Modeling Multi-relational Data," in *Advances in Neural Information Processing Systems 26 (NIPS)*, 2013.

[2] S. Gollapudi, N. Karia, V. Sivashankar, R. Krishnaswamy, N. Begwani, and H. V. Simhadri, "Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters," in *Proceedings of the ACM Web Conference (WWW)*, 2023.

[3] L. Patel, P. Kraft, C. Guestrin, and M. Zaharia, "ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data," *Proceedings of the ACM on Management of Data (SIGMOD)*, vol. 2, no. 3, 2024.

[4] G. Sehgal and S. Salihoglu, "NaviX: A Native Vector Index Design for Graph DBMSs with Robust Predicate-Agnostic Search Performance," 2025. [Online]. Available: https://arxiv.org/abs/2506.23397

[5] C. Zuo, M. Qiao, W. Zhou, F. Li, and D. Deng, "SeRF: Segment Graph for Range-Filtering Approximate Nearest Neighbor Search," *Proceedings of the ACM on Management of Data (SIGMOD)*, vol. 2, no. 1, 2024.

[6] Q. Zhang *et al.*, "VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[7] T. Neumann and G. Weikum, "RDF-3X: a RISC-style engine for RDF," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, 2008.

[8] Y. A. Malkov and D. A. Yashunin, "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, 2020.

[9] S. Jayaram Subramanya, F. Devvrit, R. Kadekodi, R. Krishnaswamy, and H. V. Simhadri, "DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node," in *Advances in Neural Information Processing Systems 32 (NeurIPS)*, 2019.

[10] J. Johnson, M. Douze, and H. Jégou, "Billion-Scale Similarity Search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, 2019.

[11] T. Wu and D. Tang, "PathFinder: Efficiently Supporting Conjunctions and Disjunctions for Filtered Approximate Nearest Neighbor Search," 2025. [Online]. Available: https://arxiv.org/abs/2511.00995

[12] M. Li, B. Lu, J. Cheng, and C. Ma, "EMA: Approximate Nearest Neighbor Search with General Attribute Filtering and Dynamic Updates," 2026. [Online]. Available: https://arxiv.org/abs/2606.00734

[13] W. Xia, M. Yang, W. Li, and W. Wang, "E2E: Efficient Filtered AKNN Search via Adaptive Termination," 2026. [Online]. Available: https://arxiv.org/abs/2602.06721

[14] J. Shi, Y. Cai, and W. Zheng, "Filtered Approximate Nearest Neighbor Search: A Unified Benchmark and Systematic Experimental Study," 2025. [Online]. Available: https://arxiv.org/abs/2509.07789

[15] C. Wei *et al.*, "AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, 2020.

[16] J. Wang *et al.*, "Milvus: A Purpose-Built Vector Data Management System," in *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, 2021.

[17] P. A. Bernstein and D.-M. W. Chiu, "Using Semi-Joins to Solve Relational Queries," *Journal of the ACM*, vol. 28, no. 1, 1981.

[18] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman, "Magic Sets and Other Strange Ways to Implement Logic Programs," in *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*, 1986.

[19] T. Neumann and G. Weikum, "Scalable Join Processing on Very Large RDF Graphs," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, 2009.

[20] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and Complexity of SPARQL," *ACM Transactions on Database Systems*, vol. 34, no. 3, 2009.

[21] S. Harris and A. Seaborne, "SPARQL 1.1 Query Language, W3C Recommendation." 2013. [Online]. Available: https://www.w3.org/TR/sparql11-query/
