froGQL: Worst-Case Optimal Joins and Type-Driven Optimization for Lightweight GQL

Felipe Avendaño[1(✉)], Jean Paul Duchens[1], Sebastián Ferrada[1,2,3,4], and Matías Toro[1,3]

[1] DCC, Universidad de Chile, Santiago, Chile
{favendan,jduchens,sferrada,mtoro}@dcc.uchile.cl
[2] IDIA, Universidad de Chile, Santiago, Chile
[3] Millennium Institute for Foundational Research on Data, Santiago, Chile
[4] National Center for Artificial Intelligence Research, Santiago, Chile

Abstract. We present froGQL, an open-source graph database that keeps a property graph, its schema, and its indexes in a single file and runs inside the host application, in the manner of SQLite, while implementing the ISO GQL standard. froGQL makes two contributions. First, its engine evaluates multi-pattern queries with Leapfrog Triejoin, a worst-case optimal algorithm whose cost is bounded by the largest result the query could produce, applied directly over a sorted-adjacency layout so that no intermediate results are built. On queries that start from one node identified by an indexed property and then follow several edges, froGQL is the fastest of the three systems we measure on the LDBC Social Network Benchmark: 13 ms on IC2 against 19 ms for Kùzu and 21 ms for GraphQLite, and 1.1 ms on IC11 against Kùzu's 4.8 ms. Resolving the equality filter on the starting node before the search begins earns that margin: without it, IC11 takes 4.6 s. Second, a static type system rejects queries that cannot return a result before any part of the graph is read. The check costs 0.1 ms on a four-hop query the engine would otherwise spend 132 s enumerating, and stays below 1% of query time in 26 of the 36 valid cases measured; it does not depend on froGQL's runtime and can be placed in front of any conforming GQL engine. froGQL is MIT-licensed and available at https://github.com/pleiad/frogql, with packages for Rust, Python, Node.js, and the browser.

1 Introduction

Knowledge graphs have become a cornerstone of modern data management, from enterprise knowledge bases to scientific data integration and AI-powered pipelines [20]. Querying them efficiently, however, has long required substantial infrastructure: production graph database systems offer excellent performance but demand non-trivial installation, configuration, and operational overhead that is often disproportionate to the needs of a research prototype or a lightweight

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2027
M. Koubarakis et al. (Eds.): ISWC 2026, LNCS 17128, pp. 104–122, 2027.
https://doi.org/10.1007/978-3-032-42029-9_6

application. At the other end of the spectrum, simulating graphs on top of relational engines such as SQLite is accessible, yet sacrifices expressiveness and join performance [1,6]. We present froGQL, a self-contained, single-file property graph engine designed to occupy the gap between these extremes and make GQL graph pattern matching fast, portable, and easy to deploy.

The motivation for froGQL comes from two converging needs. First, there is practical value in fast prototyping with a lightweight graph engine that requires no server, no configuration, and no installation beyond a single binary, yet supports a standard query language and handles complex graph patterns efficiently—whether to ship a graph dataset alongside a paper as a portable artifact or to explore graph-structured data without committing to a full database stack. Second, worst-case optimal (WCO) join algorithms have demonstrated dramatic improvements over pairwise strategies for multi-way join evaluation [29,34], yet their benefits have largely been confined to heavyweight or research-only systems. froGQL explores whether that algorithmic machinery can be delivered in a lightweight, embeddable package. By applying the Leapfrog Triejoin (LTJ) algorithm [34] directly over a sorted-adjacency storage layout, froGQL is the fastest of the three systems we measure on queries that start from one node identified by an indexed property and then follow several edges (LDBC IC2, IC11) [3], and trails the fastest system by factors of 2 to 25 on the remaining four . Resolving those equality filters before the search begins—*index folding*—accounts for most of that margin: disabling it costs a factor of 4,200 on IC11.

A second design axis is the integration of a static graph type system as a query optimizer. froGQL supports GQL-style graph type declarations: a user can assert that nodes carry certain labels and properties, and the engine statically checks incoming queries against that schema before execution. When a query pattern is provably unsatisfiable, froGQL rejects it immediately without touching the graph. This is particularly useful for applications that generate queries programmatically, such as AI agent pipelines with structured relational memory over a knowledge graph [16,24,32,39], where a malformed query should fail fast rather than trigger a full graph scan. The type system is runtime-agnostic and can be layered over any conforming GQL engine as a static pre-pass.

We envision froGQL being useful in at least three settings: *(i)* reproducible graph query research, where a single `.gdb` file bundles data, schema, and indexes into a portable artifact; *(ii)* rapid prototyping, where the absence of a server lets a developer go from raw data to graph queries in minutes; and *(iii)* structured memory for AI agents, where the typed schema acts as a contract against malformed graph state.

We evaluate froGQL on the LDBC Social Network Benchmark [3] interactive workload against Kùzu and GraphQLite, and on a suite of unsatisfiable and valid query patterns that measures the type checker's overhead and the time it saves; savings reach 132 s on queries the runtime would otherwise fully enumerate.

Our contributions are threefold. FROGQL is a single-file ISO GQL property graph engine written in Rust, with two storage backends, an interactive shell, and Python, Node.js, and browser WebAssembly bindings. Its query engine adapts Leapfrog TrieJoin to GQL path patterns over a sorted-adjacency index, achieving worst-case optimal multi-pattern join evaluation within an embeddable binary. A static type system detects provably empty query patterns before any graph access, with formal soundness guarantees; it is independent of FROGQL's runtime and portable to other GQL engines.

The remainder of the paper addresses preliminary concepts and related work (Sect. 2); gives an overview of FROGQL (Sect. 3) and its storage (Sect. 4); describes our optimizations (Sects. 5 and 6); and ends with our experimental results (Sect. 7) and conclusions (Sect. 8).

2 Background and Related Work

2.1 Property Graphs and GQL

A *property graph* is a labeled, directed multigraph in which both nodes and edges may carry sets of labels and key-value properties. We assume countably infinite, pairwise disjoint sets VID and EID of node and edge identifiers, Labels of node and edge labels, Props of property names, and Vals of literal values. Following Angles et al. [4], we define it formally as follows.

Definition 1. (Property Graph). *A* property graph $G = (V, E, \rho, \delta, \lambda, \sigma)$ *consists of finite sets of nodes* $V \subset \mathsf{VID}$ *and edges* $E \subset \mathsf{EID}$*; a function* $\rho : E \to V \times V$ *giving each edge its endpoints; a* directionality *function* $\delta : E \to \{\rightarrow, \leftarrow, —\}$ *; a labelling function* $\lambda : V \cup E \to 2^{\mathsf{Labels}}$ *assigning a set of labels to each element; and a partial function* $\sigma : (V \cup E) \times \mathsf{Props} \mapsto 2^{\mathsf{Vals}}$ *associating property names with values to both nodes and edges.*

GQL [15,22] is the first ISO standard for property graph query languages, building on the foundations of Cypher [19]. Its core mechanism is *graph pattern matching*: a `MATCH` clause specifies a path pattern, optionally annotated with label constraints and property filters, whose evaluation against a graph produces a *table of variable bindings*, one row per match. From that point, the remaining clauses operate relationally: rows are filtered (`WHERE`), extended (`OPTIONAL MATCH`), tested for existence (`(NOT) EXISTS`), sorted (`ORDER BY`), limited (`LIMIT`), and projected (`RETURN`), in direct analogy with relational algebra and SQL. FROGQL implements this fragment; we refer the reader to Deutsch et al. [15] for the full language specification and to Sect. 3 for the precise coverage of FROGQL.

2.2 Worst-Case Optimal Join Evaluation

Evaluating a multi-pattern GQL query requires finding all combinations of variable bindings that simultaneously satisfy every pattern: a multi-way join problem. Pairwise strategies evaluate two patterns at a time, producing intermediate

results that can grow quadratically before the remaining patterns are applied. For dense patterns such as cliques or cycles, this *intermediate blowup* makes pairwise strategies impractical regardless of the final output size.

The AGM bound [9] establishes the maximum output size of a join as a function of the input relation sizes. An algorithm is *worst-case optimal* (WCO) if it runs in $\tilde{O}(Q^*)$ time, where Q^* is the AGM bound of query Q; pairwise join algorithms are generally *not* WCO.

Leapfrog TrieJoin (LTJ) [34] is a WCO algorithm for any conjunctive query. Rather than joining two relations at a time, LTJ binds variables one at a time and intersects all relevant constraints simultaneously, using sorted iterators and binary-search-based *leapfrog* steps to skip non-matching values without materializing intermediate results. LTJ naturally applies to graph pattern matching by treating each directed edge as a triple $(src, label, tgt)$ and each pattern variable as a join variable across triples [34]. This lineage runs through the Semantic Web: Jena-LTJ [21] brought LTJ to SPARQL via Apache Jena, and MillenniumDB [35] deploys it in a multi-model engine. Those systems apply LTJ to data that is natively triples. froGQL reverses the transfer: it decomposes property-graph edges into (s, p, o) triples indexed in six orderings, so a property-graph engine reuses the join machinery developed for RDF. Section 5 describes how froGQL adapts LTJ to GQL path patterns over its sorted-adjacency storage layout.

2.3 Type Systems for Graph Queries

The GQL standard [22] defines value types but lacks a formal type system for query validation; three prior lines of work address this gap.

Schema Languages for Property Graphs. PG-Schema [5] introduces a schema language for property graphs, specifying the labels and property types nodes and edges may have; its purpose is to validate stored data, not to constrain queries. The closest RDF analogs are ShEx [31] and SHACL [25]. froGQL adopts PG-Schema in its graph-type catalog (Sect. 6) and lifts it to queries, meeting declared types against each pattern variable.

Typed Pattern Calculi. The Graph Pattern Calculus (GPC) [18] provides a formal model of GQL path patterns. Its type system targets coarse syntactic-type errors (rejecting, for instance, a variable used as both a node and an edge) and does not consider emptiness arising from label or property mismatches, as labels and property types of individual variables are not tracked. The Flexible Path Pattern Calculus (FPPC) [40] extends GPC with property-based filtering and a gradual type lattice: a Boolean algebra over labels, open and closed property records, and an unknown type $\star$ for imprecision. FPPC proves emptiness, type safety, and a gradual guarantee. froGQL implements and extends this lattice and repurposes it as a static optimizer: when the meet of a pattern descriptor with the active graph type collapses to $\bot$, the sub-pattern is pruned before evaluation.

Static Analysis of Graph Patterns. Parallel work on regular path queries addresses the complexity of pattern evaluation and rewriting [12,27,33]; it tar-

gets tractability and expressive power rather than schema-driven validation, orthogonal to FROGQL's type-driven optimization.

2.4 Graph Data Management Systems

Graph data management systems span a wide deployment spectrum. Production systems such as Neo4j [28], Amazon Neptune [2], and JanusGraph [23] offer ACID guarantees and horizontal scalability at the cost of substantial infrastructure. Embeddable systems such as Kùzu [17] reduce operational overhead and support WCO joins, but adopt a full-columnar DBMS architecture and use openCypher rather than GQL. At the lightweight end, DuckDB's SQL/PGQ [38] and several SQLite extensions [13] layer property graphs over relational tables, inheriting pairwise join strategies with no graph-native indexing or schema support.

Three open-source systems occupy the same lightweight native-graph niche as FROGQL (GraphLite [14], webbery/gqlite [37], auksys/gqlite [10]); all three were evaluated and excluded after failing to load or query the full LDBC SNB dataset (Sect. 7). WCO joins have been deployed in Jena-LTJ [21] and MillenniumDB [35,36]; FROGQL brings the same algorithm to an embedded engine. In the RDF space, Oxigraph [30] and COTTAS [7] address a similar portability need but target SPARQL rather than property graphs or GQL.

Table 1 positions FROGQL against comparable systems. No other maintained embedded engine combines GQL, WCO joins, and a static type system.

Table 1. Positioning against related systems. "WCO" marks worst-case-optimal joins; "SCT" marks schema-driven detection of empty queries before execution.

System	Language	Deployment	WCO	SCT	Status
Neo4j [28]	Cypher	server	–	–	active
Neptune [2]	openCypher/SPARQL	managed cloud	–	–	active
Kùzu [17]	openCypher	embedded columnar	✓	–	archived
DuckDB PGQ [38]	SQL/PGQ	embedded relational	–	–	active
GraphQLite [13]	Cypher	embedded (SQLite)	–	–	active
MillenniumDB [35]	MQL/SPARQL	server	✓	–	active
Jena-LTJ [21]	SPARQL	server (Jena)	✓	–	research
Oxigraph [30]	SPARQL	embedded	–	–	active
COTTAS [7]	SPARQL	embedded columnar	–	–	active
FROGQL	ISO GQL	embedded single-file	✓	✓	active

3 System Overview

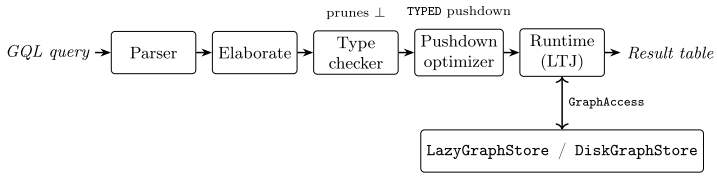

Fig. 1. froGQL query pipeline. Static stages (type checker, pushdown optimizer) eliminate work before graph access. Runtime is generic over the storage backend.

3.1 Architecture and Pipeline

Figure 1 shows the froGQL pipeline. The **parser** produces a `PathPattern` AST, which passes through **elaboration**: inline value filters such as `(x:L k: v)` are hoisted into `WHERE x.k=v` conditions. The AST then reaches the **type checker** (Sect. 6), which checks each descriptor against the schema; a provably unsatisfiable descriptor short-circuits the query to an empty result, without graph scan.

Surviving patterns pass to the **pushdown optimizer**. Besides value-predicate pushdown and index-folding, froGQL pushes down *type predicates*: a top-level `WHERE` conjunct of the form `x.a TYPED T` folds into the descriptor of `x` (e.g., `MATCH (x)WHERE x.age TYPED INT` becomes `MATCH (x: age INT)`), turning a runtime check into a static constraint that the type checker can exploit (Sect. 6).

Finally, the optimized plan is evaluated by the **runtime**, which applies the LTJ algorithm to join and concatenation patterns (Sect. 5) — including any-direction edges via a mirrored index, and bounded repetitions unrolled into LTJ-compatible unions — and falls back to pairwise hash-join for the remaining shapes (unions and non-unrollable repetitions). The runtime is generic over the `GraphAccess` trait, which abstracts the two persistent storage backends (Sect. 4), allowing the same query code to operate over either without modification.

3.2 Query Language Coverage

froGQL implements the core GQL pattern-matching fragment: node and edge patterns with label conjunctions, disjunctions, and negation; directed, reverse, and undirected edges; multi-pattern comma-joins; quantified patterns `,mn,m`; `WHERE` filters (arithmetic, boolean, and type tests); `OPTIONAL MATCH`; `WHERE (NOT) EXISTS` subqueries; `ORDER BY` and `LIMIT`; aggregation functions in `RETURN` (`COUNT`, `SUM`, `AVG`, etc.); and `RETURN` with projection and `DISTINCT`. Data modification (`INSERT`, `SET`, etc.) and graph type DDL (`CREATE`, `USE`, `VALIDATE`) are also fully supported. Beyond that core, froGQL supports unbounded repetition (`*`, `+`)

under the ISO §16.6 path-pattern prefixes that make it finite (`SHORTEST`, `TRAIL`, `SIMPLE`, `ACYCLIC`); named path variables bind first-class path values for the ISO §20.16 path functions; and correlated `VALUE` scalar subqueries, `GROUP BY`, and record constructors round out the value language. Together, these cover 13 of the 14 LDBC SNB Interactive Complex query templates.

Data Loading. Bulk ingestion is available via `import_csv` and `import_json`. Multi-DML chains and concurrent writing are not yet supported; a single-writer/multi-reader model and full WAL are left to future work.

3.3 Interfaces

FROGQL runs as a native binary: launched with a `.gdb` file, it opens an interactive shell through which users issue queries, inspect schema, and manage graph types.

For programmatic access, FROGQL provides a Python library with an intentionally minimal API: `frogql.open(path)` returns a connection whose `conn.execute()` yields results as dictionaries keyed by the `RETURN` aliases, plus `conn.schema()` and element counts. A Node.js binding (five prebuilt platforms) mirrors that API, a browser WebAssembly build runs the engine over an in-memory store, and the library is available as a Rust crate. The core stays server-free, following SQLite's single-user embedded model; an HTTP wrapper takes a few lines in any of the host languages.

Availability: The source code of FROGQL is maintained on GitHub[1] and released under the MIT License, which permits both academic and commercial reuse. A single release tag publishes, through an automated pipeline, the Python package on PyPI[2], the Node.js[3] and browser WebAssembly[4] packages on npm, and the Rust crate[5]; each release is archived at Zenodo [11]. The resource is newly released, so we make no adoption claims. Its reuse potential rests on how it is distributed: the engine needs no server and no configuration, and it is reachable from Python, Node.js, Rust, and the browser through standard package managers. The repository ships the quick-start guides, example databases, and reproducible benchmark harnesses described in this paper.

Sustainability and Maintenance: FROGQL is university research software, maintained by the authors' group as the engine underpinning our ongoing graph-query research; a multi-year grant application will put that maintenance on a dedicated footing. The design keeps the overhead low regardless of funding: releases are automated, the MIT license and per-release Zenodo archival keep the project forkable, and the on-disk format is documented independently of the engine and read with backward compatibility, so data stays readable even if development stops.

[1] https://github.com/pleiad/frogql.
[2] https://pypi.org/project/frogql.
[3] https://www.npmjs.com/package/frogql.
[4] https://www.npmjs.com/package/frogql-wasm.
[5] https://crates.io/crates/frogql.

4 Storage Architecture

froGQL stores everything in a single `.gdb` file organized as 4 KB slotted pages managed by an LRU-cached `Pager` (default 2000 pages, ≈8 MB resident). Page 0 is the file header; the remaining pages hold a string table where all node and edge names, label names, property keys, and string values are interned and assigned a `u32` identifier, node and edge records, label and adjacency indexes, and a graph-type catalog (Sect. 6). Figure 2 shows the page layout.

Region	Contents
Header (page 0)	Magic, element counts, fixed-offset root pointers to every region, active graph-type name.
String table	Deduplicated labels, property names, and string values; each interned as a `u32` identifier (SID).
Node/edge records	Slotted pages of variable-length cells: SIDs, labels, and inline properties; edge cells add endpoints and direction.
Record-location indexes	Node ID → (*page*, *cell*) array and per-edge topology; read sequentially at open.
Label indexes	Label SID → sorted element IDs, per element kind.
Adjacency	CSR layout (offsets + flat edge-ID arrays), stored separately per direction; loaded in $O(N + E)$.
Graph-type catalog	Page chain serializing the catalog (§6) as JSON.

Fig. 2. In-file layout of a `.gdb` database. Every region is reachable from page 0 via fixed-offset pointers. The six sorted edge orderings used by LTJ (Sect. 5) are not persisted and are rebuilt at open in $O(N + E)$.

4.1 Storage Backends

The runtime is generic over a common backend interface, `GraphAccess`. Two backends serve a `.gdb` database with memory/latency trade-offs; the query engine is unaware of which is in use. `LazyGraphStore` is the default: the froGQL shell and the Python binding `frogql.open` both open every `.gdb` through it. `DiskGraphStore` is opt-in for graphs whose metadata exceeds the memory budget. (A third backend, `Graph`, ingests JSON for tests and small interactive examples; it does not load `.gdb` files, so it is excluded from our comparison.)

`LazyGraphStore` keeps the edge topology arrays (`src`, `tgt`, `directed`), the label index, and the adjacency index in memory, and fetches labels and properties through the page cache. `DiskGraphStore` keeps only the topology resident and pushes all other accesses through the cache, incurring one cache miss per non-topological lookup. Both backends share the same on-disk format, so the same `.gdb` opens unchanged through either.

4.2 On-Disk Indexes

Two indexes sit on disk and are used by the query runtime (Sect. 5).

Label Index. For every label SID, a sorted page chain stores the identifiers of the nodes (and, separately, the edges) carrying that label. A query that filters on a single label fetches the chain root from the header in $O(1)$ and scans the chain sequentially; queries with compound labels such as `A & B` pick the smallest indexed set first (label-index selection) and intersect.

Adjacency Index. Incident edges are stored in a CSR layout across six parallel arrays: `out_offsets`/`out_flat`, `in_offsets`/`in_flat`, and `und_offsets`/`und_flat` for outgoing, incoming, and undirected senses, respectively. Each flat array holds packed *(edge_id, other_node)* pairs; the offset arrays index into them per node. The runtime uses this index for single-edge traversals and as the seed for the in-memory `TripleIndex` constructed at database load (Sect. 5).

String Interning. Every label, property name, and string-typed property value is deduplicated through the string table, so labels in indexes and record cells are 4-byte SIDs rather than copies of their textual form, making index keys integer-comparable and page chains cheap to binary search.

Updates and Index Lifecycle. Data modification follows SQLite's explicit-commit model without a write-ahead log: mutations stage in an in-memory overlay with per-statement atomicity, and an explicit save rewrites the file atomically, meaning that there is no in-place page allocation to manage. After a mutation, the triple index is rebuilt lazily on the next query ($\sim$0.5 s at SF0.1, $\sim$1.6 s at SF0.3). It is deliberately not persisted: rebuilding takes under two seconds at our scale, while persisting it would increase the file size by roughly 12%. Incremental index maintenance under the overlay is on the roadmap.

5 LTJ-Based Pattern Matching

A GQL comma-join such as `(a)-[]->(b), (b)-[]->(c), (c)-[]->(a)` is a multi-way join over three binary relations from the adjacency index. We encode each pattern edge as a sorted iterator over a triple-structured representation of the graph and evaluate the join with LTJ [34], which is WCO for this class of queries.

From Patterns to Triples. frogql models every directed edge as a triple (s, p, o) of node and edge-label identifiers. A query pattern is decomposed by sliding a $(Node, Edge, Node)$ window over each `Concat` chain; anonymous nodes get fresh variables and label constants are pre-bound—e.g., `(a)->(b)->(c)->(d)` yields $\{(a, p_1, b), (b, p_2, c), (c, p_3, d)\}$.

Triple Index. Triples are stored in six orderings (`SPO`, `SOP`, `POS`, `PSO`, `OSP`, `OPS`); each entry carries the original edge identifier for result reconstruction. Given any prefix of constants, binary search runs in $O(\log n)$ time into the matching ordering.

Iterators and Leapfrog Seek. Every query triple gets an `LtjIterator` over the ordering that matches its already-fixed positions, exposing the classical `leap`/`down`/`up` primitives. To bind a variable shared by k triples, the runner rotates round-robin through the k iterators, leaping each to the maximum last-seen value until all agree—the leapfrog step of [34].

Variable Elimination Order (Join Ordering). A static VEO binds *non-lonely* variables (appearing in more than one triple) before *lonely* ones, which appear in a single triple and become inner loops over remaining matches. Node-label filters and pushed-down property-type constraints are evaluated as soon as all their variables are bound, pruning entire sub-trees before descent rather than filtering rows post-join. Join ordering in froGQL thus lives inside LTJ rather than in a plan enumerator: the VEO fixes the variable order statically, and equality predicates that hit a secondary index are folded to constants *before* the search begins, removing those variables from the order entirely (the index-fold ablation in Sect. 7.2 measures this). Cost-based, adaptive variable ordering is future work.

Scope and Fallback. The decomposition handles chains and clique joins of directed, reverse, and undirected edges with optional label constraints. Reverse edges are normalized by swapping endpoints . Any-direction edges (`-[e]-`) also run through LTJ, via a mirrored triple index that stores every edge in both senses; bounded repetitions `{n,m}` over a single edge, unroll into LTJ-compatible unions. The remaining shapes—explicit unions and non-unrollable repetitions — fall back to pairwise hash-join. Retaining that path is deliberate: hash-join is the strategy froGQL used before LTJ was added, so introducing LTJ cannot make any pattern shape slower than it was.

Complexity. The core LTJ search is worst-case optimal with respect to the AGM bound, producing results in $\tilde{O}(Q^* + |out|)$ time. End-to-end cost adds index construction: building all six orderings is $O(E \log E)$, amortized across a session, as `TripleIndex` is built at open and shared across every query in that connection.

6 Type-Driven Query Optimization

6.1 The Graph Type System

froGQL maintains a persistent catalog of named *graph types*. A graph type constrains which labels and properties nodes and edges may carry; the type checker consults the active entry to validate every query against it.

Descriptors. The unit of typing is the *descriptor*: a pair (ℓ, π) of a label constraint and a property constraint describing a single node or edge. Pattern variables and schema entries are both typed by descriptors, compared via lattice operations.

Label Types. Label constraints form a Boolean algebra: conjunction $A \,\&\, B$ (the node carries both labels), disjunction $A|B$, negation $!A$, the top element $\star$ (any label) and the bottom element $\perp$ (no label, no value can inhabit).

Property Types. A property constraint maps attribute names to *simple types*: `INT`, `FLOAT`, `BOOL`, `STRING`, lists, nested records, the unknown type $\star$, and $\bot$. *Open* records $\{a\ T_1, b\ T_2, \star\}$ require at least the listed fields; *closed* records $\{\{a\ T_1, b\ T_2\}\}$ require exactly those. Both forms are inherited from FPPC, allowing a graph type to accept partial schemas without losing precision on spelled-out fields.

Catalog. A graph type is a set of node and edge descriptors persisted in the `.gdb` catalog: a reserved `DEFAULT` entry auto-derived from the data at import time, plus user-defined entries (`CREATE GRAPH TYPE`). The active entry is selected with `USE`; `SHOW` and `VALIDATE` round out the DDL surface.

Meet, Join, and $\bot$. Each layer of the type system (simple types, property records, label constraints, descriptors) forms a bounded lattice with meet ($\sqcap$) and join ($\sqcup$); descriptor operations act componentwise. The bottom element $\bot$ represents the empty type: a descriptor with $\bot$ on either component is $\bot$, an edge type is $\bot$ when its descriptor or either endpoint is, and a union is $\bot$ only when both branches are—so unsatisfiability detection (Sect. 6.2) short-circuits a query only when no alternative branch can rescue it.

FROGQL implements and extends the type system of FPPC [40], lifting it from a calculus to a full graph engine with ahead-of-time query pruning.

6.2 Static Unsatisfiability Detection

The type checker walks the pattern, builds a descriptor for every node and edge variable, and takes the meet across positions where the same variable appears. Each descriptor is then *refined* against the active graph type: if no schema entry is a subtype of the variable's descriptor, the meet collapses to $\bot$, which propagates outward to the path type. When the path type is $\bot$, the compiler emits an empty plan, and the runtime immediately returns no rows. Figure 3 presents the judgment in pseudocode. Consider graph type `(:Person)-[:KNOWS]->(:Person)` for the following examples.

```
MATCH (x: Company)-[:KNOWS]->(y)
-- Verdict: refine (x: Company & Person) against schema
--          finds no subtype → ⊥ → empty result, no scan
```

The first example is unsatisfiable through the label component: the descriptor for `x` is `Company & Person`, no schema entry meets it compatibly, so refinement returns the empty join $= \bot$ and the query is pruned.

```
MATCH (x: {name INT})-[:KNOWS]->(x: {name STRING})
-- Verdict: x binds to (:Person {name INT ⊓ STRING}) =
--          (:Person {name ⊥}) = ⊥ → empty result, no scan
```

The second example is unsatisfiable per the property component, via variable repetition: x occurs in two positions, their descriptors meet before refinement, and `ame INT ⊓ STRINGname INT⊓STRING` $=$ `name` $\bot$ empties the descriptor, pruning the query.

```
function CHECK(pattern P, schema S) → PathType
  Γ ← ∅                          -- var → descriptor (ℓ, π)
  for each occurrence of variable x with descriptor d in P:
    Γ[x] ← REFINE(Γ[x] ⊓ d, S) -- meet across repeats
  return PATHTYPE(P, Γ)

function REFINE(descriptor d, schema S) → descriptor
  -- join of the compatible schema entries: ⨆{ s ⊓ d | s ∈ S, s ⊓ d ≠ ⊥ }
  return ⨆_{s∈S}(s ⊓ d) -- empty join = ⊥

function PATHTYPE(P, Γ) → PathType -- ⊥-propagation
  match P: case Node(x) / Edge(x): return Γ[x]
    case Concat(P1,P2), Join(P1,P2), Filter(P1):
      ⊥ if any sub-PathType is ⊥, else composite
    case Union(P1,P2): ⊥ only if both branches are ⊥
    case Repeat(P1,n,m): ⊥ if P1 is ⊥ and n > 0
-- caller: PathType = ⊥ ⇒ emit empty plan (no graph scan);
--         EXISTS body = ⊥ ⇒ fold to false / NOT EXISTS to true
```

Fig. 3. The static unsatisfiability judgment. Descriptors of repeated variables meet ($\sqcap$) before refinement; REFINE joins the meets with every compatible schema entry, so an empty join collapses to $\bot$; $\bot$ propagates through concatenation, join, and filter, but a union survives while any branch does.

Folding `EXISTS` *and* `NOT EXISTS`. The same emptiness verdict folds existential subqueries to literals: when the inner pattern's path type collapses to $\bot$, `WHERE EXISTS ...` rewrites to `WHERE false` and `WHERE NOT EXISTS ...` to `WHERE true`.

```
MATCH (x: Person) WHERE NOT EXISTS { MATCH (x)-[:KNOWS]->(y: Company) }
RETURN x
-- Verdict: inner y refines to ⊥ → body empty →
--          NOT EXISTS folds to true → WHERE drops out,
--          query reduces to MATCH (x: Person) RETURN x
```

7 Evaluation

We address two questions: does the static type system prune unsatisfiable patterns at negligible cost on valid queries, and is the engine competitive against established baselines on a standard graph workload? The study splits into an *internal* benchmark (froGQL in isolation, two storage backends, two scale factors) and a *cross-system* comparison against Kùzu [17] and GraphQLite [13]. Both run on the LDBC Social Network Benchmark [3] (SNB) at scale factors 0.1 (∼327k nodes, 1.5M edges) and 0.3 (∼908k nodes, 4.6M edges). All measurements were taken on a desktop-class machine: AMD Ryzen 9 9900X3D (12 cores/24 threads), 128 GB RAM, NVMe SSD, Debian GNU/Linux.

7.1 Internal Benchmark

Our 25-case set is split into three categories: nine valid cases (including LDBC IC2 and IC8 with the sorting and projection clauses the benchmark specifies, a variable-length match, an aggregate, and a query that is valid but empty on this dataset); ten statically unsatisfiable cases; and six invalid cases with free

variables or parse errors. Each query runs through the compile pipeline with and without the type checker and executes five times per backend; we report medians. Each case is measured on both backends at both scale factors, giving 36 measurements for the valid cases, which we refer to as *cells*. The full query set is in the repository.

Backend Scaling. Figure 4 compares Lazy (`LazyGraphStore`: topology in RAM, properties read on demand via LRU page cache) and Disk (`DiskGraphStore`: everything paged on demand) on the nine valid cases at both scale factors. On NVMe storage the cost of full disk residency is moderate on the anchored joins: IC2 takes 46 ms at SF0.1 and 367 ms at SF0.3 on Disk, against Lazy's 14 ms and 217 ms (∼3.4× and ∼1.7× slower); the widest gap is IC8's ∼80× at SF0.1 (475 ms vs. 6 ms). Both backends grow super-linearly with data size on the join-heavy queries (IC2: ∼16× Lazy, ∼8× Disk for ∼3× more data) and sublinearly on the queries that a single index lookup answers.

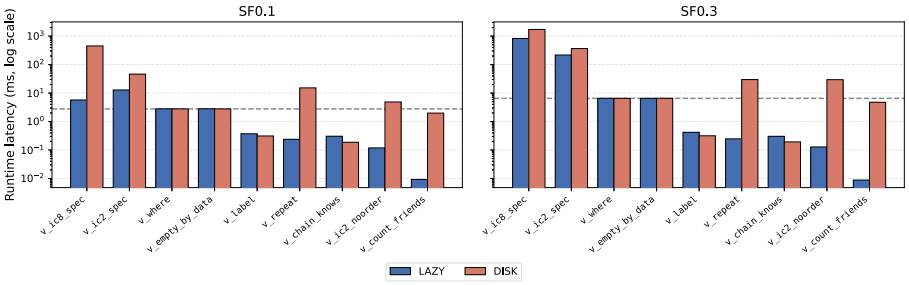

Fig. 4. Query latency for the nine queries on Lazy and Disk at SF0.1 and SF0.3.

Static Typing Efficacy. Figure 5 contrasts the type checker's rejection latency against the unchecked runtime on statically unsatisfiable cases. For deep-join chain contradictions, the asymmetry is stark: the type checker rejects a four-hop chain with an incompatible leaf in 67 μs, while the unchecked runtime spends 18 s at SF0.1—and 132 s at SF0.3—enumerating intermediate states before discarding every row. Time savings across the full empty-by-typing suite range from negligible on simple single-edge mismatches, where the runtime's label index (∼10 µs) marginally beats the type checker's static traversal (18–45 µs), up to the 132 s above, a ∼1,400,000× speedup measured to completion.

Type-Checker Overhead on Valid Queries. On valid queries, type-checker overhead is below 1% in 26 of 36 measured cells, and on *every* cell whose unchecked runtime exceeds 1 ms. The remaining ten cells are sub-millisecond queries with relative overheads up to 130%, whose absolute compile cost stays under 48 μs; the largest absolute compile cost on any valid cell is 113 μs (the full IC2 at SF0.1). No cell required aborting at a time budget: the formerly pathological unchecked case (`i_unbound_in_where_chain`, SF0.3 Disk) now completes in 8.1 s.

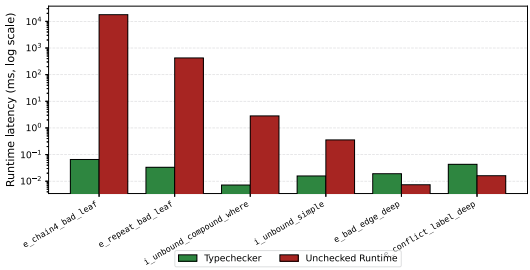

Fig. 5. Type checker vs. unchecked runtime on unsatisfiable queries (SF0.1, Lazy).

Memory Footprint. On SF0.3, Lazy opens at 841 MiB and peaks at 7,636 MiB during the query loop; Disk at 873 MiB / 7,968 MiB (SF0.1: 367 / 1,685 and 360 / 1,775 MiB). Per-case RSS attribution separates two different costs behind those peaks. The engine's own footprint—topology, triple index, secondary indexes—plateaus at 1.2 GiB (Lazy) / 1.6 GiB (Disk) across the entire SF0.3 case set; a compact CLTJ index (six LOUDS succinct tries [8]) can shrink its share ∼2.9× at a 1.4–2.1× latency cost. The remaining ∼6.2 GiB comes from a *single* case: executing, with the checker disabled, a query the checker rejects, which builds multi-gigabyte intermediate sets before discarding all rows. With the checker enabled, neither that query's 8 s of runtime nor its 6 GiB of memory usage occurs.

7.2 Cross-System Benchmark

Systems Considered. Two external graph databases serve as baselines. **Kùzu** [17] (v0.11.3, its final release before the project was archived in October 2025) is a vectorized columnar engine with an openCypher dialect, run in its only mode: embedded, disk-based storage with a buffer-managed page cache, bulk-loaded via `COPY FROM`. **GraphQLite** [13] (v0.4.4) is a Cypher [19] dialect over a SQLite extension whose embeddable single-file storage model mirrors froGQL.

We compare froGQL against both systems on the six LDBC IC queries (IC2, IC5, IC6, IC8, IC9, IC11) at SF0.1, with three timed iterations after one warm-up per parameter row. The froGQL column is split into *baseline* (all optimizations enabled) and *no-index-fold* (the LTJ index-folding pre-pass disabled, all other optimizations retained), and is reported for *both* storage backends—Lazy and Disk, which reads everything from disk, so the comparison against the disk-based Kùzu can be read with both systems keeping the same amount of data in RAM—plus a compact-CLTJ index build, the low-memory configuration.

Bulk-Load Cost. Building froGQL's single `.gdb` from the LDBC SF0.1 CSV files takes 9.5 s (142 MB file); GraphQLite ingests the same data in 10.7 s (335 MB) and Kùzu, whose columnar `COPY FROM` is the fastest loader, in 1.1 s (94 MB). Every later session reopens the file in ∼1.2 s, so the ingest cost is paid once and amortized across all subsequent sessions and queries.

Latency. Figure 6 reports per-IC median latency. On IC2 (a multi-hop join anchored on a single `Person.id` lookup), froGQL Lazy leads at 13 ms, against

19 ms for Kùzu and 21 ms for GraphQLite; this is the canonical shape that static index-folding targets, collapsing an equality predicate on an indexed property to a constant before execution. On IC11 froGQL Lazy is again the fastest at 1.1 ms vs. 4.8 ms for Kùzu, benefiting from the same optimization. With froGQL also reading everything from disk, the results reverse: froGQL Disk runs IC2 in 47 ms and IC11 in 4.6 s, since the Disk backend does not yet integrate the secondary indexes that index folding relies on—froGQL's edge on the anchored shapes comes from its RAM-resident indexes, not from the join algorithm alone. On IC5, IC6, and IC9 Kùzu leads (38 ms, 14 ms, and 61 ms) against froGQL Lazy's 164 ms, 123 ms, and 1,543 ms; all three involve significant post-join sorting or aggregation that froGQL's current execution model does not yet optimize. On IC8, GraphQLite leads (13 ms), with Kùzu at 21 ms and froGQL at 27 ms. The compact-CLTJ build tracks the array baseline at 1.3–2.0× across the ICs and is marginally *faster* on IC11 (0.8 ms).

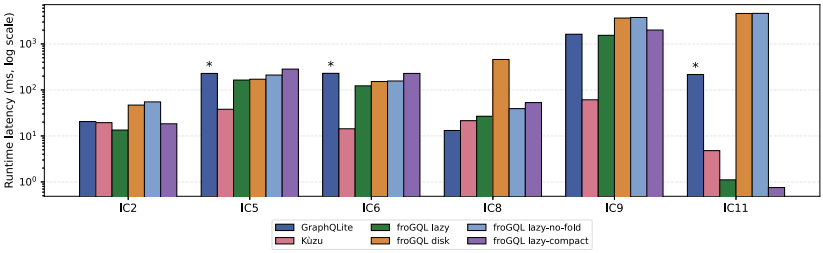

Fig. 6. Cross-system per-IC median latency on LDBC SF0.1 (* = divergent GraphQLite row count).

Result-Content Equivalence. froGQL (all four configurations) and Kùzu agree on row counts across all six queries and all parameter rows. GraphQLite diverges on IC5, IC6, and IC11, returning fewer rows than both other systems on a subset of parameter rows (shown with * in Fig. 6); on those cells, GraphQLite performs strictly less work, and its latency is not directly comparable. These divergences trace to multiple known limitations in GraphQLite 0.4.4: planner differences for unsupported graph-shape patterns on IC5/IC6, and a runtime issue with undirected variable-length expansion on IC11.

Memory Footprint. Peak resident-set size (RSS) during the query loop is 519–730 MiB for froGQL Lazy, 500–640 MiB for the compact build, 373–614 MiB for Disk, 380–480 MiB for Kùzu, and 27–33 MiB for GraphQLite; the last figure understates GraphQLite's true working set, as its data resides in OS-managed memory-mapped files outside process RSS. froGQL Lazy thus carries a ∼1.4× RSS premium over Kùzu for its RAM-resident indexes, while Disk overlaps Kùzu's range when both read everything from disk.

Ablation. Disabling the static index-folding pre-pass has sharply query-dependent effects. On IC11, the regression is pronounced: baseline completes

in 1.1 ms while the same query without the fold takes 4,629 ms (≈ 4,200×), since IC11 filters its starting node on an indexed property; IC2 regresses 4.1× for the same reason, and IC8, IC6, and IC5 only 1.3–1.5×. On IC9, froGQL's worst cell (1,543 ms), disabling repetition unrolling costs only 1.2×—the dominant cost is not the `[:knows*1..2]` expansion but the post-expansion pipeline (deduplication and `ORDER BY` over a large candidate set), which runs outside LTJ and is the primary remaining performance target.

Methodology Note. We translated LDBC SNB Cypher reference [26] queries into each engine's dialect, preserving patterns, conditions, and modifiers. We check agreement with row-content hashes; timing covers compilation and execution.

7.3 Discussion

Where LTJ Works. LTJ [34] is most effective when a query starts from a variable that an index lookup fixes to a single value. IC2 and IC11 fit this template, and froGQL leads both competitors there in its default Lazy configuration; the Disk column shows that this edge rests on RAM-resident secondary indexes, since without them index folding does not apply and Kùzu's buffer-managed design wins when both systems read from disk. Queries that defy the template expose the remaining bottlenecks: post-join deduplication and sorting scale with candidate set size, and grouping aggregation is not yet optimized, which account for the IC5, IC6, IC8, and IC9 gaps, as the IC9 ablation confirms.

What The Static Type System Buys. On valid queries, the type checker is essentially free: its overhead is below 1 % on 26 of the 36 valid-case cells measured, and on every query whose unchecked runtime exceeds 1 ms. On queries with a schema-level contradiction, the type checker rejects in tens of microseconds for simple clashes and ∼100 μs for the deepest patterns tested (Fig. 5), producing speed-ups from 30× on simple mismatches up to ∼1,400,000× (132 s saved on a single query) on the most severe chain enumeration we measured to completion. On two structurally trivial empty patterns, the runtime's index-driven detection is marginally faster; both paths are sub-millisecond.

Limitations. The ∼1.4× peak-RSS premium over Kùzu is the price of froGQL's RAM-resident indexes; the compact CLTJ build and the Disk backend each trade part of it back, the first for 1.3–2.0× latency, the second for index folding. Four-out-of-six tested ICs show that engine-level work beyond WCO joins remains. Compile-time checking is not yet uniformly cheap: one adversarial rejected case shows compile times of around 100 μs that scale with the inferred schema, a cost we are investigating; it does not affect the valid-query overhead figures above.

8 Conclusion

froGQL is an embeddable GQL graph database written in Rust, built around a static type system that rejects provably-empty queries before execution. The

type system is runtime-agnostic and can be deployed as a static pre-pass over any conforming GQL engine. On the LDBC Social Network Benchmark, the type checker adds negligible overhead on valid queries (below 1 % on 26 of 36 measured cells) and saves up to 132 s of computing time on queries the runtime would otherwise enumerate to exhaustion. On the queries that start from an indexed node and follow several edges (IC2 and IC11), FROGQL's combination of LTJ and static index folding in its default LAZY configuration outperforms both baselines; on the remaining ICs, post-join processing (deduplication, sorting, and aggregation) is the primary gap relative to Kùzu. FROGQL is MIT-licensed, with packages for Rust, Python, Node.js, and the browser, and ships all benchmark harnesses for reproducibility. Future work includes scaling to SF1 and beyond, optimizing the post-join pipelines, adaptive variable ordering for LTJ, an RDF/SPARQL front-end over the six-ordering triple core, and comparing against server-class systems such as Amazon Neptune [2], JanusGraph [23], and MillenniumDB [35].

Acknowledgments. This work was funded by ANID–Millennium Science Initiative Program–Code ICN17_002. S. Ferrada was supported by ANID Fondecyt Iniciación 11251322; by CENIA FB210017, Basal ANID; and by the Vicerrectoría de Investigación y Desarrollo (VID) of U. de Chile, project code UI-016/24. M. Toro was supported by ANID Fondecyt Iniciación 11250054.

Declaration of Use of Generative AI. The authors used LLMs (Claude Opus 4.7) during the preparation of this manuscript. Specifically, LLM tools assisted with code generation, prose drafting, and paragraph restructuring. All technical decisions, experimental design, and results are entirely the work of the authors. The authors reviewed, tested, and edited all LLM-assisted output and take full responsibility for the accuracy and integrity of the final manuscript.

References

1. Aberger, C.R., Lamb, A., Tu, S., Nötzli, A., Olukotun, K., Ré, C.: Emptyheaded: a relational engine for graph processing. ACM Trans. Database Syst. **42**(4) (2017). https://doi.org/10.1145/3129246
2. Amazon Web Services: Amazon Neptune: Fast, reliable graph database built for the cloud (2025). https://aws.amazon.com/neptune/. Accessed 06 May 2026
3. Angles, R., et al.: The LDBC Social Network Benchmark (2020). https://doi.org/10.48550/ARXIV.2001.02299. https://arxiv.org/abs/2001.02299
4. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., Vrgoč, D.: Foundations of modern query languages for graph databases. ACM Comput. Surv. **50**(5), 68:1–68:40 (2017). https://doi.org/10.1145/3104031
5. Angles, R., et al.: Pg-schema: schemas for property graphs. Proc. ACM Manag. Data **1**(2) (2023). https://doi.org/10.1145/3589778
6. Angles, R., Gutierrez, C.: Survey of graph database models. ACM Comput. Surv. **40**(1), 1–39 (2008). https://doi.org/10.1145/1322432.1322433. https://dl.acm.org/doi/10.1145/1322432.1322433

7. Arenas-Guerrero, J., Ferrada, S.: COTTAS: columnar triple table storage for efficient and compressed RDF management. In: The Semantic Web–ISWC 2025, pp. 313–331. Springer Nature Switzerland, Cham (2026)
8. Arroyuelo, D., Gómez-Brandón, A., Hogan, A., Navarro, G., Rojas-Ledesma, J.: Compact data structures meet worst-case-optimal joins. VLDB J. (2025). compactLTJ
9. Atserias, A., Grohe, M., Marx, D.: Size bounds and query plans for relational joins. SIAM J. Comput. **42**(4), 1737–1767 (2013)
10. AUKSYS: GQLite: A small, self-contained graph query database engine on SQLite (2025). https://gitlab.com/auksys/gqlite. Accessed 06 May 2026
11. Avendaño, F., Duchens, J.P., Ferrada, S., Toro, M.: froGQL (2026).https://doi.org/10.5281/zenodo.20079184
12. Barceló, P., Libkin, L., Reutter, J.L.: Querying regular graph patterns. J. ACM **61**(1) (2014). https://doi.org/10.1145/2559905
13. Colliery Software: GraphQLite: A SQLite extension that adds graph database capabilities with Cypher query language support (2025). https://github.com/colliery-io/graphqlite. Accessed 06 May 2026
14. DeepGraph AI: GraphLite: An embeddable graph database with ISO graph query language support (2025). https://github.com/GraphLite-AI/GraphLite. Accessed 06 May 2026
15. Deutsch, A., et al.: Graph pattern matching in GQL and SQL/PGQ. In: Proceedings of the 2022 International Conference on Management of Data, pp. 2246–2258. ACM, Philadelphia PA USA (2022). https://doi.org/10.1145/3514221.3526057. https://dl.acm.org/doi/10.1145/3514221.3526057
16. Edge, D., et al.: From local to global: a Graph RAG approach to query-focused summarization. arXiv preprint arXiv:2404.16130 (2024). https://arxiv.org/abs/2404.16130
17. Feng, X., Jin, G., Chen, Z., Liu, C., Salihoglu, S.: Kùzu graph database management system. In: Proceedings of the Conference on Innovative Data Systems Research (CIDR), vol. 7, pp. 25–35 (2023)
18. Francis, N., et al.: GPC: a pattern calculus for property graphs. In: Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, pp. 241–250. PODS 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3584372.3588662
19. Francis, N., et al.: Cypher: an evolving query language for property graphs. In: Proceedings of the 2018 International Conference on Management of Data, pp. 1433–1445. ACM, Houston TX USA (2018). https://doi.org/10.1145/3183713.3190657. https://dl.acm.org/doi/10.1145/3183713.3190657
20. Hogan, A., et al.: Knowledge graphs. ACM Comput. Surv. **54**(4), 1–37 (2022). https://doi.org/10.1145/3447772. https://dl.acm.org/doi/10.1145/3447772
21. Hogan, A., Riveros, C., Rojas, C., Soto, A.: A worst-case optimal join algorithm for SPARQL. In: Ghidini, C., et al., (eds.) ISWC 2019. LNCS, vol. 11778, pp. 258–275. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30793-6_15
22. ISO/IEC: ISO/IEC 39075:2024 Information technology – Database languages – GQL (2024). https://www.iso.org/standard/76120.html. Accessed 29 Apr 2025
23. JanusGraph Contributors: JanusGraph: An open-source, distributed graph database (2024). https://doi.org/10.5281/zenodo.14807604. https://janusgraph.org/
24. Jia, R., Wu, M., Ding, Y., Lu, J., Zhang, Y.: Agent-enhanced heterogeneous graph rag for academic question answering. In: Proceedings of the ACM Web Conference

2026, pp. 8765–8768. WWW 2026, Association for Computing Machinery, New York, NY, USA (2026). https://doi.org/10.1145/3774904.3792960
25. Knublauch, H., Kontokostas, D.: Shapes constraint language (SHACL). W3C recommendation, World Wide Web Consortium (W3C) (2017). https://www.w3.org/TR/shacl/
26. LDBC Council: LDBC SNB Interactive v1 Reference Cypher Queries (2023). https://github.com/ldbc/ldbc_snb_interactive_v1_impls/tree/main/cypher/queries. Accessed 2026
27. Libkin, L., Martens, W., Vrgoc, D.: Querying graphs with data. J. ACM **63**(2) (2016). https://doi.org/10.1145/2850413
28. Neo4j, Inc.: Neo4j Graph Database (2025). https://neo4j.com
29. Ngo, H.Q.: Worst-case optimal join algorithms: techniques, results, and open problems. In: Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, pp. 111–124. PODS 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3196959.3196990
30. Pellissier Tanon, T.: Oxigraph (2025). https://doi.org/10.5281/zenodo.7408022. https://github.com/oxigraph/oxigraph
31. Prud'hommeaux, E., Labra Gayo, J.E., Solbrig, H.: Shape expressions: an RDF validation and transformation language. In: Proceedings of the 10th International Conference on Semantic Systems (SEMANTiCS 2014), pp. 32–40. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2660517.2660523
32. Rasmussen, P., Paliychuk, P., Beauvais, T., Ryabinin, J., Chalef, D.: Zep: a temporal knowledge graph architecture for agent memory. arXiv preprint arXiv:2501.13956 (2025). https://arxiv.org/abs/2501.13956
33. Reutter, J.L., Romero, M., Vardi, M.Y.: Regular queries on graph databases. Theory Comput. Syst. **61**(1), 31–83 (2017). https://doi.org/10.1007/S00224-016-9676-2
34. Veldhuizen, T.L.: Leapfrog triejoin: a simple, worst-case optimal join algorithm. In: Proceedings of the 17th International Conference on Database Theory (ICDT), pp. 96–106 (2014). https://doi.org/10.5441/002/icdt.2014.13
35. Vrgoč, D., et al.: MillenniumDB: an open-source graph database system. Data Intell. **5**(3), 560–610 (2023). https://doi.org/10.1162/dint_a_00229. https://direct.mit.edu/dint/article/5/3/560/117375/MillenniumDB-An-Open-Source-Graph-Database-System
36. Vrgoč, D., et al.: MillenniumDB: a multi-modal, multi-model graph database. In: Companion of the 2024 International Conference on Management of Data, pp. 496–499. SIGMOD 2024, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3626246.3654757
37. Webbery: GQLite: Graph database lite (2024). https://github.com/webbery/gqlite. Accessed 06 May 2026
38. Wolde, D.t., Szárnyas, G., Boncz, P.: DuckPGQ: bringing SQL/PGQ to DuckDB. Proc. VLDB Endow. **16**(12), 4034–4037 (2023). https://doi.org/10.14778/3611540.3611614
39. Xu, G., Corter, J.: Graph RAG for automated short answer grading with feedback: bridging pedagogical needs and technical capabilities. In: Proceedings of the AAAI Conference on Artificial Intelligence vol. 40, no. 48, pp. 40916–40924 (2026). https://doi.org/10.1609/aaai.v40i48.42125. https://ojs.aaai.org/index.php/AAAI/article/view/42125
40. Ye, W., et al.: Flexible and expressive typed path patterns for GQL. Proc. ACM Program. Lang. **9**(OOPSLA2) (2025). https://doi.org/10.1145/3763061