Raqlet: Cross-Paradigm Compilation for Recursive Queries

Amir Shaikhha*
University of Edinburgh
Edinburgh, United Kingdom
amir.shaikhha@ed.ac.uk

Youning Xia*
University of Edinburgh
Edinburgh, United Kingdom
y.xia-32@sms.ed.ac.uk

Meisam Tarabkhah*
University of Edinburgh
Edinburgh, United Kingdom
m.tarabkhah@ed.ac.uk

Jazal Saleem
University of Edinburgh
Edinburgh, United Kingdom
j.saleem@sms.ed.ac.uk

Anna Herlihy
EPFL
Lausanne, Switzerland
anna.herlihy@epfl.ch

ABSTRACT

We introduce Raqlet, a source-to-source compilation framework that addresses the fragmentation of recursive querying engines spanning relational (recursive SQL), graph (Cypher, GQL), and deductive (Datalog) systems. Recent standards such as SQL:2023's SQL/PGQ and the GQL standard provide a common foundation for querying graph data within relational and graph databases; however, real-world support remains inconsistent across systems. Raqlet bridges this gap by translating recursive queries across paradigms through leveraging intermediate representations (IRs) grounded in well-defined semantics; it translates Cypher or SQL/PGQ to PGIR (inspired by Cypher), then into DLIR (inspired by Datalog), and finally to SQIR (inspired by recursive SQL). Raqlet provides a shared semantic basis that can serve as a golden reference implementation for language standards, while supporting static analysis and transformations (e.g., magic-set transformation) for performance tuning. Our vision is to make Raqlet a robust platform that enables rapid cross-paradigm prototyping, portable recursive queries, and formal reasoning about recursion even when targeting diverse query execution engines.

1 INTRODUCTION

Recursive queries are becoming increasingly important in various data-driven domains. Once considered of limited practical use [47], they are now seen as key enablers for expressing the complex logic required by modern applications. Cutting-edge AI-driven knowledge management systems emphasize recursion as a core requirement for expressing rule-based reasoning on graphs [5, 9]. In graph analytics and databases, recursion is fundamental for queries such as reachability and shortest paths. In the field of program analysis, deductive databases have emerged as a standard tool for building large-scale static analyzers to encode points-to analyses, dataflow frameworks, and other program analyses [46]. Recursive queries are also used in declarative networking to specify network protocols and distributed computations [2].

Despite this broad importance, support for recursive queries is fragmented across various data systems, each with distinct query languages, capabilities, and performance considerations. In relational databases, SQL has included recursive common table expressions (CTEs) since the SQL:1999 standard. However, recursive CTEs are limited in expressive power, and the query optimization support is restricted to non-recursive subsets. Graph databases offer a specific and limited form of recursion through variable-length path patterns, suitable for reachability and path enumeration. Deductive database systems utilize Datalog, but each Datalog implementation is essentially separate, with unique dialects and optimization strategies [30].

In addition, while relational, graph, and deductive systems all offer support for recursive queries, they do so in distinct and incompatible ways. Key distinctions include data models (tables vs. property graphs vs. logical facts), expressiveness (the types of recursion and logic allowed), and execution strategies (recursive SQL fixpoints vs. pointer-based graph traversal vs. bottom-up Datalog evaluation). These incompatibilities force practitioners to commit to a technology stack early in development, before application requirements are fully understood. Users must either accept the limitations of a high-level language, sacrifice the performance benefits of a specialized engine, or bear the significant cost of migrating between query engines and languages later in development.

Furthermore, despite ongoing efforts to standardize query languages [38], discrepancies remain in their implementations across different systems [34]. A "golden reference implementation" that is formally specified could help ensure consistency and serve as a common point of reference for various systems.

This paper proposes Raqlet, a novel compilation-based framework that aims to unify these disparate ecosystems for recursive queries. Rather than proposing yet another standalone query engine, Raqlet acts as a source-to-source compiler that translates recursive queries across different languages and systems.

Raqlet employs intermediate representations (IRs), including Property Graph IR (PGIR), SQL IR (SQIR), and Datalog IR (DLIR), to create a common ground for different query languages. For instance, a graph pattern query written in Cypher can be transformed into PGIR, capturing the essential graph traversal semantics. This representation can then be converted to DLIR, a logical rule-based format similar to Datalog with formal fixpoint semantics. Raqlet facilitates seamless bridging between relational, graph, and logic systems, enabling users to write recursive queries in one paradigm and execute them using the optimizations of another, all without the need for manual porting.

This compilation-oriented approach yields several key benefits.

*These authors contributed equally to this work.

 16th Annual Conference on Innovative Data Systems Research (CIDR '26). January 18-21, Chaminade, USA

Static Analysis (Section 4). Raqlet enables static analysis and reasoning of queries prior to execution. This includes the ability to identify whether recursion is linear or non-linear, verify stratification to prevent illegal negation or aggregation cycles, and prove properties like monotonicity, all of which facilitate more aggressive optimizations. Developers can receive early feedback (e.g., warnings about potential non-terminating recursions or suggestions for query rewrites to improve efficiency) and the system can automatically apply transformations.

Recursive Optimization (Section 5). Raqlet acts as an optimization catalyst, leveraging extensive query optimization techniques from relational, graph, and logic programming domains on the compiled IR. Methods such as Magic Sets rewriting and linearizing recursive rules can be incorporated within Raqlet's compilation pipeline, independent of the query's original language. Raqlet treats recursion as a general computation pattern that can be effectively optimized and executed across various systems.

Formal Semantics (Section 6). Finally, Raqlet establishes formal semantics for all input query languages. The DLIR abstraction is grounded in rigorous logical semantics; it inherits the well-defined semantics of Datalog, ensuring that recursive queries maintain clear and consistent meanings. This applies even to cases where input queries originate from other languages, such as Cypher or SQL.

2 RECURSIVE QUERY LANDSCAPE

Recursion is supported across a range of systems, each with distinct data models, query languages, and performance considerations. This diversity reflects a fragmented landscape in which no single system offers a definitive advantage across all dimensions of performance and expressiveness. In this section, we survey three major categories of recursive systems and discuss what classes of queries have empirically been shown to perform best on each system.

2.1 Graph Databases

Data models. Property Graphs (PG) represent graphs as nodes and directed edges. Resource Description Framework (RDF) models graphs as subject-predicate-object triples. PG supports both node and edge properties, which is nontrivial to model in RDF without data duplication, but unlike tables, neither model supports efficient representation of hypergraphs. GDBMS typically offer flexible, schema-optional design, though recent work such as Property Graph Schema (PG-Schema) [4] provides a formalism for specifying and enforcing strict schemas in PG systems.

Query languages. Neo4J [18] is a widely used PG database with its own query language, Cypher. Cypher has a concise and user-friendly syntax but lacks full expressivity, supporting only 15 of 30 queries in the popular LDBC SNB benchmark [45]. GQL is an ISO standard [26] that provides a unified language to facilitate interoperability across graph systems, yet it has been shown to be less expressive than recursive SQL [19]. SPARQL, the W3C standard language for RDF [23], is a key technology for the semantic web yet is limited in that generalized basic graph patterns (bgps) cannot be expressed in SPARQL [3], but can in Datalog.

Performance. Neo4J has been shown to outperform MySQL on certain complex queries [15]. SPARQL systems have been shown to outperform Neo4J for shortest-path and point-lookup queries, while PostgreSQL performs better than both SPARQL and Neo4J for point-lookup queries yet significantly worse for shortest path [35]. Benchmarking between GDBMS [48] ultimately shows that different systems excel at different queries.

2.2 Relational databases

Data model. Graphs can be modeled on RDBMS by relations representing nodes and joins between relations representing edges. As a result, *n*-ary relationships and hypergraphs are easy to model but heavily connected graphs result in many-to-many relations and complex, join-heavy queries [35].

Query languages. SQL's `WITH RECURSIVE` is based on a restricted form of Datalog, and cannot support non-linear or mutual recursion [24]. SQL/PGQ is a SQL extension for PG in RDBMS, where graphs are a view of a tabular schema [25] and uses the same pattern matching as GQL. Both recursive SQL and SQL/PGQ suffer from a lengthy and notoriously difficult-to-read ISO standard specification and lack of complete formal semantics [34].

Performance. RDBMS have been shown to outperform GDBMS under workloads that make heavy use of group by, sort, and aggregations [13]. DuckDB using `WITH RECURSIVE` has also been shown to outperform Datalog engines for linear queries without aggregation [21].

2.3 Deductive Systems

Data models. Deductive systems model nodes and edges as finite relations of "facts", typically with a rigid schema. Facts are stored in the *extensional* database (EDB) and graphs are modeled similarly to RDBMS.

Query languages. Queries are expressed using logic-based rules that define how to derive new facts from existing facts. The rules are stored in the *intensional* database (IDB). Deductive database query languages are more expressive than SQL or graph query languages, for example Datalog can express a broad class of queries including mutual and non-linear recursion, and many extensions to Datalog exist to express queries with negation and aggregation. While deductive databases lack the same wide usage or commercial support as GDBMS or RDBMS, Datalog has three well-established formal semantic models: model-theoretic, fixed-point, and proof-theoretic that serve as a rigorous foundation for formally reasoning about recursive query behavior.

Performance. Soufflé, a commercial Datalog engine, has been shown to outperform SQLite, PostgreSQL, and Neo4J for classic recursive queries like transitive closure [11].

2.4 Challenges

Deciding between database paradigms requires domain expertise, must be done relatively early in application development, and can be costly to revise. Each system makes trade-offs between data model, expressiveness, and performance, and clear comparisons between systems are rarely straightforward. For example, when comparing expressibility of query languages, SQL with recursion is Turing-complete and can therefore claim to simulate any computation, albeit in a very convoluted and inefficient way. It has been shown that these workarounds, possible in graph query languages as well, are not a replacement for native language support

as they can lead to exponential blowup of intermediate results and untenably slow performance [19]. Further, recent empirical results often contradict each other, and results depend heavily on query shape, data structure, and implementation details. Understanding which queries perform best on which system is sufficiently complex that researchers have applied machine learning to predict the runtime of a transitive closure query across GDBMS, RDBMS, and deductive systems [11]. While pair-wise translation schemes have been proposed between recursive query languages, for example, SPARQL-to-SQL or SQL-to-Cypher, no existing techniques capture the disparate features of recursive query languages under a single formal semantics. The fragmentation highlights the need for tools that can capture query functionality across systems and automatically translate between query languages to enable dynamic exploration of the recursive query landscape.

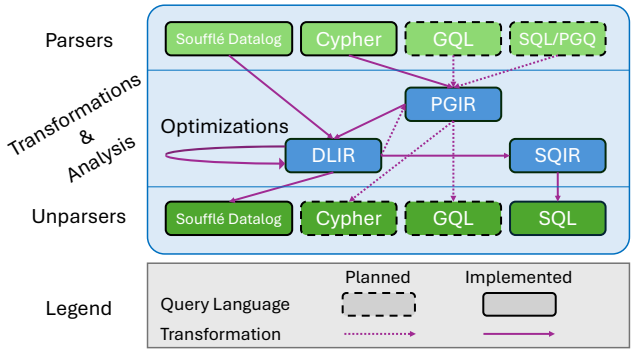


Figure 1: Raqlet architecture overview.

3 SYSTEM OVERVIEW

Figure 1 shows Raqlet's architecture. At a high level, Raqlet involves three main modules: (1) parsers as its frontend, (2) transformations and analysis as its middle end, and (3) unparsers as its backend. Raqlet's design enables the recursive query to be fully decoupled from its backend-specific representation and lays the foundation of portability across different GDBMSs, RDBMSs, and deductive engines. The key insight is to apply most transformations and analyses at the level of DLIR. Currently, Raqlet includes the implementation of a few query languages as input and target languages. We plan to implement the missing frontends and backends in the future.

We explain the compilation process of the framework through the following running example.

Running Example. We embed the LDBC Social Network Benchmark (SNB) interactive workload into our environment. Figure 2 presents its schema and Figure 3 shows a simplified version of short query 1 [16]. Given that most deductive databases utilize set semantics and lack certain features such as ordering and limiting the results, to achieve semantic equivalence in translated queries across different backends, we use `RETURN DISTINCT` instead and remove `ORDER BY` and `LIMIT` clauses in input Cypher queries.

Data Model Transformation. For Cypher queries, Raqlet takes a PG-Schema G as input and produces DL-Schema, inspired by the Datalog data model, which will be used for query translation. This is done by generating an EDB for every node and edge type in G. For instance, the node types `personType` and `cityType` in Figure 2a are translated to EDBs `Person` and `City`, respectively, in Figure 2b; the edge type `locationType` is translated to the EDB `Person_IS_LOCATED_IN_City`.

Cypher to PGIR Translation. The query translation begins by lowering an input Cypher query into PGIR (Property Graph IR), an intermediate language inspired by GPC [17] but extends to support core Cypher features required for LDBC SNB read workloads, including aggregation and shortest path finding. PGIR represents the query as a sequence of clause constructs such as `MATCH`, `WHERE`, and `RETURN`. The input query undergoes normalization and decomposition into PGIR expressions, such as patterns, filters, and aliasing, which are then mapped to their corresponding clause constructs.

In our example, the Cypher query (Figure 3a) is passed to the compiler and lowered into its PGIR version. Figure 3b illustrates the graphical representation of PGIR, where grey boxes denote clause constructs, dashed boxes indicate the contents of each clause and arrows imply the order of clauses. During lowering, the graph pattern is transformed into PGIR's edge pattern, which consists of edge label `IS_LOCATED_IN`, a unique identifier `x1` generated by the compiler, the type of edge (`directed` in this case), and its source and target nodes. Such nodes are represented in PGIR's node pattern which has a node label and an identifier. The transformed edge pattern is mapped to `MATCH` construct. Node condition `{id:42}` is extracted from the graph pattern and mapped to `WHERE` construct. Finally, the return statement in Cypher is lowered to `RETURN` construct which includes output items. This step simplifies the query representation and hence eases subsequent translations.

PGIR to DLIR Translation. Next, the query is translated from PGIR to DLIR, our core intermediate representation of recursive queries based on Datalog with negation and aggregation. DLIR represents a query as a sequence of rules, with a head specifying an IDB and a body implying how the view is computed. Figure 3c presents the graphical representation of the translated DLIR. Grey boxes denote the head of each DLIR rule, and dashed boxes indicate the body of the respective rules. The atoms that comprise the body are represented by the elements contained within the dashed boxes. This layer of Raqlet derives semantic-preserving translations for each PGIR construct and employs schema information in Figure 2b to deduce variable positions within atoms and infer their types.

In our example, each PGIR clause construct (i.e., `MATCH`, `WHERE`, and `RETURN`) is translated into a separate DLIR rule (i.e., `Match1`, `Where1`, and `Return`, respectively). The node and edge patterns in PGIR are mapped to their respective EDBs defined in DL-Schema with identifier variables placed at the appropriate position within atoms (note that node id is at the first position of EDB). The node condition `n.id = 42` is translated into the predicate `n = 42`, and the renaming operation `p.id AS cityId` is translated into the variable binding `p = cityId`.

DLIR to Datalog and SQL translation. This layer is responsible for converting DLIR to either Datalog or SQL. Generating Datalog from DLIR is relatively straightforward (Figure 3d). To produce SQL from DLIR, we transform each non-recursive DLIR rule into a Common Table Expression (CTE) and a recursive rule into a recursive CTE. The final SQL program consists of a sequence of CTEs followed by a `SELECT` statement from the last CTE.

```
CREATE GRAPH {
  ( personType:Person {id INT, firstName STRING, locationIP STRING}),
  ( cityType:City {id INT, name STRING}),
  (: personType )-[ locationType: isLocatedIn  {id INT} ]->(: cityType )
}
```

(a) PG-Schema.

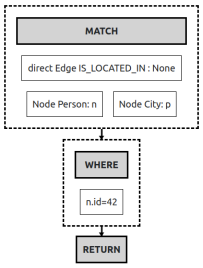
```
.decl Person(id: number, firstName: symbol,
             locationIP: symbol)
.decl City(id: number, name: symbol)
.decl Person_IS_LOCATED_IN_City(id1: number, id2: number,
                                id: number)
```

(b) DL-Schema.

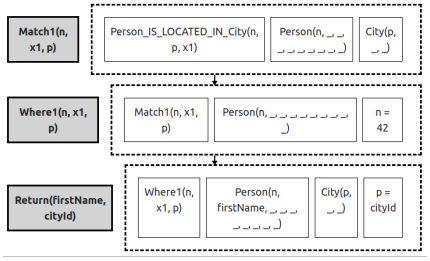
Figure 2: Schema transformation by Raqlet. The schema is simplified for presentation purposes.

```
MATCH
(n:Person
  {
    id:42
  }
)
  -[:IS_LOCATED_IN]->
(p:City)
RETURN DISTINCT
n.firstName AS firstName,
p.id AS cityId
```

(a) The input Cypher query.



(b) PGIR captures the core of Cypher.



(c) DLIR represents the query as a sequence of Datalog rules.

```
.decl Match1(n: number, x1: number, p: number)
Match1(n, x1, p) :- Person_IS_LOCATED_IN_City(n, p, x1),
    Person(n, _, _, _, _, _, _, _, _, _),
    City(p, _, _).
.decl Where1(n: number, x1: number, p: number)
Where1(n, x1, p) :- Match1(n, x1, p),
    Person(n, _, _, _, _, _, _, _, _, _),
    n = 42.
.decl Return(firstName: symbol, cityId: number)
Return(firstName, cityId) :- Where1(n, x1, p),
    Person(n, firstName, _, _, _, _, _, _, _, _),
    City(p, _, _),
    p = cityId.
.output Return
```

(d) The generated Datalog program using Soufflè's syntax.

```
WITH V1 AS (
 SELECT DISTINCT R1.id1 AS n, R1.id AS x1, R1.id2 AS p
 FROM Person_IS_LOCATED_IN_City AS R1, Person AS R2, City AS R3
 WHERE (R1.id1 = R2.id) AND (R1.id2 = R3.id)
), V2 AS (
 SELECT DISTINCT V1.n AS n, V1.x1 AS x1, V1.p AS p
 FROM V1, Person AS R1
 WHERE (V1.n = 42) AND (V1.n = R1.id)
), V3 AS (
 SELECT DISTINCT R1.firstName AS firstName, V2.p AS cityId
 FROM V2, Person AS R1, City AS R2
 WHERE (V2.n = R1.id) AND (V2.p = R2.id)
)
SELECT DISTINCT * FROM V3
```

(e) The generated SQL query.

Figure 3: Representations of the example query at different stages of Raqlet's translation pipeline.

In the running example, the rules `Match1`, `Where1`, and `Return` are translated into the CTEs `V1`, `V2`, and `V3`, respectively (Figure 3e). The logical conjunctions between relation atoms are translated into inner joins, and `SELECT DISTINCT` is used to keep set semantics. The final output is selected from the last CTE (`V3`).

4 STATIC ANALYSIS AND REASONING

Because recursive query engines vary widely in their expressiveness and evaluation strategies, Raqlet uses static analysis at the level of DLIR to enable early reasoning about query semantics. DLIR-level static analysis ensures that each analysis is implemented only once, independent of the source query language, and the logical rule-based structure of DLIR makes these analyses straightforward to express and execute.

The key goals of static analyses are (1) identifying unsupported queries by a backend due to semantic limitations, (2) uncovering optimization opportunities that can be expressed with query rewrites, and (3) providing backend-aware guidance to users to identify queries that may cause runtime problems such as non-termination.

Linearity. Linearity analysis identifies if a query contains linear recursion, which is represented in DLIR with rules that are defined by a single recursive predicate. Linearity is used to reject queries on backends that support only linear recursion, for example RDBMS. Raqlet can also apply well-known techniques [49] to rewrite non-linear queries into linear ones where possible, to eliminate self-joins and avoid costly materialization of intermediate results.

Mutual Recursion. Mutual recursion analysis identifies whether a query contains two or more recursive predicates that depend on each other in a cycle. This is used to reject queries on backends that do not support mutual recursion, such as RDBMS. Raqlet can also use this analysis to enable safe rewrites using existing rewrite techniques for mutually-recursive queries [31].

Monotonicity. Monotonicity analysis determines whether a recursive query is monotonic under set-inclusion, required by most recursive query engines to guarantee convergence. Non-monotonic constructs, such as negation or certain forms of aggregation, can prevent termination or be rejected at query compile-time by the target execution engine. Raqlet can use monotonicity analysis to reject unsupported queries and to enable rewrites that safely move aggregates inside recursion when monotonicity is preserved [20].

Termination Analysis. Termination analysis identifies whether a recursive query may lead to nontermination at runtime. This includes detecting properties such as interpreted functions over unbounded domains or the use of bag semantics, both of which can lead to infinite recursion [22]. Raqlet can use this analysis to

```
.decl Match1(n: number, x1: number, p: number)
Match1(n, x1, p) :- Person(n, _, _, _, _, _, _, _, _, _),
  City(p, _, _), Person_IS_LOCATED_IN_City(n, p, x1).
.decl Where1(n: number, x1: number, p: number)
Where1(n, x1, p) :- Person(n, _, _, _, _, _, _, _, _, _),
  City(p, _, _), Person_IS_LOCATED_IN_City(n, p, x1), n=42.
.decl Return(firstName: symbol, cityId: number)
Return(fn, ci) :- Person(n, fn, _, _, _, _, _, _, _, _),
  City(p, _, _), Person_IS_LOCATED_IN_City(n, p, x1), n=42, p=ci.
.output Return
```

(a) Optimization with inlining.

```
.decl Return(firstName: symbol, cityId: number)
Return(fn, ci) :- Person(n, fn, _, _, _, _, _, _, _, _), City(p, _,
      _), Person_IS_LOCATED_IN_City(n, p, x1), n=42, p=ci.
.output Return
```

(b) More optimization with dead rule elimination.

Figure 4: Examples of optimizations applied to a graph query represented in Datalog.

warn the user that their queries may not terminate under certain conditions, for example over cyclic data.

5 RECURSIVE QUERY OPTIMIZATION

Our framework provides an optimizer to produce more efficient DLIR programs that benefit from well-established as well as novel query optimization techniques.

Inlining. SQL and Datalog queries can involve intermediate views created through CTE clauses or intermediate rules (known as IDBs), respectively. Inlining the definition of such views can not only directly improve the performance, but also indirectly can open up opportunities for further optimizations, including removing self-joins on primary keys [39].

In our example, Figure 4a is an optimized version of Figure 3d where inlining has been applied. During inlining, rule atoms present in a body (which satisfy certain conditions, i.e. not involved in aggregation and negation) are replaced by their bodies. For instance, `Match1` in the body of `Where1` is inlined, and so is `Where1` in the body of `Return`. After inlining, since `Person` appears twice (due to a self-join) in `Where1`, the duplication is removed.

Dead Rule Elimination. In many cases, in particular after inlining rules, there will be intermediate rules that no longer contribute to the final result. In such cases, one can improve the performance by removing such unused rules.

Figure 4b further optimizes the previous example by applying dead rule elimination. Since the output rule `Return` is not dependent on rules `Match1` and `Where1`, these two rules are safe to be eliminated from the Datalog program. This will remove unnecessary intermediate results and hence reduce computation complexity.

Pushing Operators Past Recursion. In non-recursive queries, pushing operators (e.g., selection, projection, or aggregation) past the join is an essential technique to reduce the cost of joins. Similarly, the Datalog literature considers techniques for pushing operators past recursion. This includes optimizations such as the magic-set transformation [7] or FGH rule [50].

Semantic Join Optimizations. Semantic query optimization leverages the integrity constraints to optimize the queries further [12]. The database literature includes techniques for eliminating joins based on reasoning over integrity constraints [1, 32]. A novel optimization in the context of property graphs is to eliminate joins on keys that are disjoint, which can be inferred from the knowledge encoded in PG-Schema.

Extensibility and Portability. Unlike most industrial query optimizers, which are tailored to specific target database systems, our optimizer provides abstractions on the IR level, allowing various optimization options to be easily added or removed, thereby achieving extensibility and portability in query optimization.

Code Generation. Apart from generating code based on recursive query languages such as Datalog and SQL, Raqlet's allows for generating low-level code. This is achieved by introducing additional IRs that include more procedural information on how to run recursive queries [41, 44]. This way, Raqlet can benefit from the techniques developed for just-in-time query compilation [28, 33, 42, 43].

Preliminary Experimental Results. Finally, we evaluate the performance of the translated queries against the original Cypher query, using one of LDBC queries. We study unoptimized and fully optimized versions of the Datalog and SQL queries. Our experiments are conducted on a system featuring an AMD Ryzen 9 5950X 16-Core Processor operating at 3.4GHz, with 64GB of DDR3 RAM, running Ubuntu 24.04.1 OS. We use Neo4j 5.27.0 to run the original Cypher query and Soufflé 2.4.1 for the Datalog queries. SQL queries are executed using Tableau Hyper API 0.0.21200 and DuckDB 1.1.3.

Table 1 shows the execution times for LDBC short query 1 (SQ1) and complex query 2 (CQ2) for SF10. In most cases, translated Datalog and SQL queries have lower execution times compared to the original Cypher query. In all cases, the fully optimized versions have better performance compared to the unoptimized queries. We expect similar results for the rest of the queries, but leave a more detailed evaluation for future work.

Table 1: Execution time (ms) for each query.

Query	Optimized	Neo4j	Soufflé	DuckDB	HyPer
SQ1	✗	72.17	0.05	24.25	0.89
	✓	-	0.02	1.78	0.78
CQ2	✗	87.85	11.70	33.18	215.85
	✓	-	11.31	4.01	168.16

6 TOWARDS FORMAL SEMANTICS

The SQL standard has undergone several revisions since its inception through SQL:1999 (which introduced recursion) to the most recent SQL:2023 (when SQL/PGQ was introduced to support querying graphs). In parallel, the GQL standard [38] provides a formal declarative language for property graph databases that unifies query languages such as Cypher and GSQL. These efforts aim to ensure a consistent formal semantics regardless of the underlying system.

However, despite these standardization efforts, actual systems often deviate in both syntax and semantics. Existing SQL dialects are inconsistent in recursive semantics [22] and handling of NULLs [34]. Similarly, graph query languages such as Cypher and GSQL continue to evolve independently, introducing features that diverge from the GQL standard specifications. This is due to the lack of a "golden reference implementation" that resolves the inconsistencies between the standard specification and system implementations.

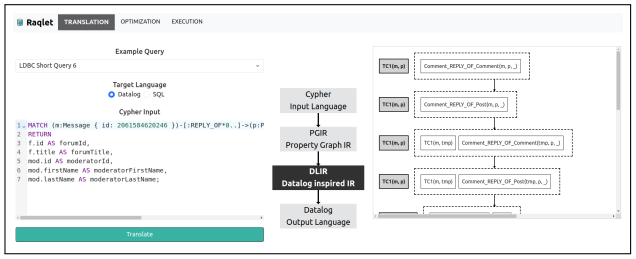


Figure 5: Scenario 1 demonstrates the translation process. Depending on the stage, Raqlet uses a graph visualization of the IR or a textual representation.

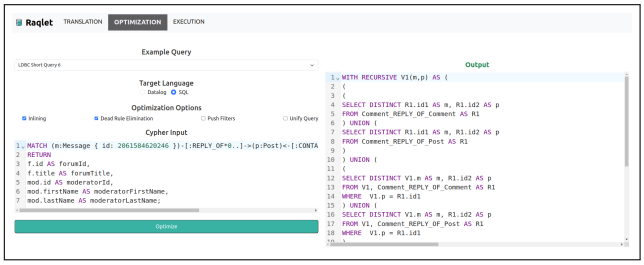


Figure 6: Scenario 2 optimizes the graph query depending on optimizations specified by the user.

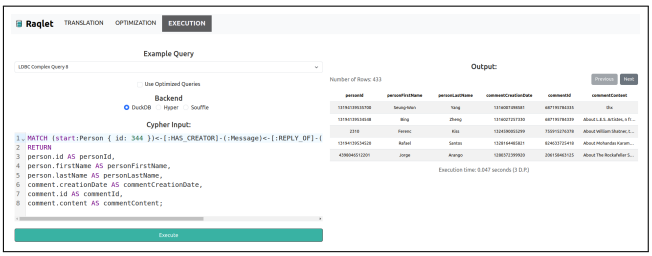


Figure 7: Scenario 3 executes the queries on an RDBMS (DuckDB or HyPer) or a Deductive DB engine (Soufflé).

Raqlet addresses this gap by leveraging its Datalog-inspired IR, DLIR, as a formally specified core. DLIR is grounded in the well-known least-fixed-point semantics of stratified Datalog and its extensions, such as Datalog^o [50]. When a Cypher or SQL/PGQ is compiled to DLIR, it can benefit from a precise logical semantics that follows fix-point logic. This way, DLIR serves as a golden reference implementation for the SQL:2023 and GQL standard.

To further improve this foundation, we plan to formalize DLIR semantics, Raqlet's translation pipeline, and DLIR optimizations using proof assistants such as Rocq [6, 8] (formerly Coq) or Lean [14]. The current Raqlet prototype is implemented in Scala to leverage functional programming's design patterns [40] and meta programming facilities [36], making it an appropriate starting point for rewriting in Rocq or Lean. Inspired by the prior efforts on formalizing SQL semantics to verify the correctness of query optimizers [6, 14], we aim to prove the semantic preservation of transformations from Cypher to SQL/PGQ to DLIR. This way, we ensure a machine-checked semantic core.

7 USER INTERFACE

In this section, we demonstrate how users interact with Raqlet using a web interface to enable easy, intuitive exploration and fast prototyping. Graph query languages are more intuitive for expressing queries over graphs than relational languages such as SQL [10]. Our web interface helps users to better understand the query translation process and its impact through three scenarios.

First, users interact with the translation tab to understand the query representation at each stage of the translation pipeline. Second, users experiment with query optimization techniques and observe their impact on the resulting query using the optimization tab. Finally, users execute queries on supported database backends and review query performance and results via the execution tab.

In all scenarios, users begin by choosing among a list of predefined Cypher queries or writing a custom one. We embed the LDBC SNB interactive benchmark into our demonstration environment and provide its 7 short queries and 12 complex queries as predefined workloads. The custom queries supplied by the user must adhere to the LDBC schema.

Scenario 1: Query Translation (Figure 5). In this scenario, the user interacts with the translation tab to get an overview of how the Cypher query is represented at each stage of the translation pipeline. The user can choose either Datalog or SQL as the target language. After clicking the Translate button, a translation pipeline is displayed and the user can click on one of the stages among PGIR, DLIR and SQL / Datalog depending on the target language. When the user chooses a particular stage, the respective graphical representation of the IR or textual representation of the code is shown on the right side. For example, in Figure 5 the user has clicked on DLIR, which triggers the graph visualization of the input LDBC short query 6 in DLIR.

Scenario 2: Query Optimization (Figure 6). To better understand how individual and combined optimization techniques impact the query, the user can interact with the optimization tab to observe changes in the result query by adding or removing predefined optimization options. Once the Cypher query is provided, the user first chooses the target language. The user can then tick the optimization options. After clicking the Optimize button, the optimized SQL query is displayed in the output field on the right side.

Scenario 3: Query Execution (Figure 7). In this scenario, the user investigates the query results and understands the query's run time on a database backend supported by Raqlet. This is done through the execution tab in the interface. Similar to the previous scenarios, the user first needs to provide an input Cypher query. Then, the user selects the backend engine to execute the translated query. Currently, Raqlet supports DuckDB [37] and HyPer [29] as the RDBMS backends, as well as Soufflé [27] as the deductive database backend. In addition, the user can decide whether to execute the unoptimized or fully optimized version of the target query. After clicking Execute, the query results and execution time are shown on the right side of the page. Figure 7 presents a case where the user executes the unoptimized LDBC complex query 8 on DuckDB.

8 CONCLUSION

This paper presented Raqlet, a framework that bridges the fragmented ecosystem of recursive query systems with cross-paradigm compilation. Raqlet enables the interoperability of relational, graph, and deductive systems by leveraging IRs grounded in well-defined semantics. This provides a shared semantic basis for system-agnostic formal reasoning and optimization. Raqlet aims to provide a golden reference implementation that can serve as a machine-checked semantic core for verifying optimizations and translation pipelines.

ACKNOWLEDGEMENT

This work was partly funded by a gift from RelationalAI and the Google Research Scholar Program.

REFERENCES

[1] Alfred V. Aho, Catriel Beeri, and Jeffrey D. Ullman. 1979. The Theory of Joins in Relational Databases. *ACM Trans. Database Syst.* 4, 3 (1979). https://doi.org/10.1145/320083.320091

[2] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. 2011. Dedalus: Datalog in Time and Space. In *Datalog Reloaded*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 262–281.

[3] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Aidan Hogan, Juan Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. arXiv:1610.06264 [cs.DB] https://arxiv.org/abs/1610.06264

[4] Renzo et al. Angles. 2023. PG-Schema: Schemas for Property Graphs. *Proc. ACM Manag. Data* 1, 2, Article 198 (jun 2023), 25 pages. https://doi.org/10.1145/3589778

[5] Molham Aref, Paolo Guagliardo, George Kastrinis, Leonid Libkin, Victor Marsault, Wim Martens, Mary McGrath, Filip Murlak, Nathaniel Nystrom, Liat Peterfreund, Allison Rogers, Cristina Sirangelo, Domagoj Vrgoc, David Zhao, and Abdul Zreika. 2025. Rel: A Programming Language for Relational Data. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*. ACM, 283–296. https://doi.org/10.1145/3722212.3724450

[6] Joshua S. Auerbach, Martin Hirzel, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. 2017. Prototyping a query compiler using Coq (experience report). *Proc. ACM Program. Lang.* 1, ICFP (2017), 9:1–9:15. https://doi.org/10.1145/3110253

[7] Francois Bancilhon et al. 1985. Magic Sets and Other Strange Ways to Implement Logic Programs (Extended Abstract) *(PODS '86)*. ACM, New York, NY, USA, 1–15.

[8] Pierre-Léo Bégay, Pierre Crégut, and Jean-François Monin. 2021. Developing and certifying Datalog optimizations in coq/mathcomp. In *CPP '21*, Catalin Hritcu and Andrei Popescu (Eds.). ACM, 163–177. https://doi.org/10.1145/3437992.3439913

[9] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. 2018. The Vadalog System: Datalog-based Reasoning for Knowledge Graphs. *Proc. VLDB Endow.* 11, 9 (2018), 975–987. https://doi.org/10.14778/3213880.3213888

[10] Sourav S Bhowmick, Byron Choi, and Chengkai Li. 2017. Graph querying meets HCI: State of the art and future directions. In *SIGMOD*. 1731–1736.

[11] Stefan Brass and Mario Wenzel. 2019. Performance Analysis and Comparison of Deductive Systems and SQL Databases. In *Datalog 2.0 (CEUR Workshop Proceedings)*, Mario Alviano and Andreas Pieris (Eds.), Vol. 2368. CEUR-WS.org, 27–38. https://ceur-ws.org/Vol-2368/paper3.pdf

[12] Upen S. Chakravarthy, John Grant, and Jack Minker. 1990. Logic-Based Approach to Semantic Query Optimization. *ACM Trans. Database Syst.* 15, 2 (1990), 162–207. https://doi.org/10.1145/78922.78924

[13] Yijian Cheng, Pengjie Ding, Tongtong Wang, Wei Lu, and Xiaoyong Du. 2019. Which Category Is Better: Benchmarking Relational and Graph Database Management Systems. *Data Science and Engineering* 4 (12 2019). https://doi.org/10.1007/s41019-019-00110-3

[14] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: proving query rewrites with univalent SQL semantics. In *PLDI 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 510–524. https://doi.org/10.1145/3062341.3062348

[15] Thi-Thu-Trang Do, Thai-Bao Mai-Hoang, Van-Quyet Nguyen, and Quyet-Thang Huynh. 2022. Query-based Performance Comparison of Graph Database and Relational Database *(SoICT '22)*. New York, NY, USA, 375–381. https://doi.org/10.1145/3568562.3568648

[16] Orri Erling et al. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD*. 619–630.

[17] Nadime Francis et al. 2023. GPC: A Pattern Calculus for Property Graphs *(PODS '23)*. New York, NY, USA, 241–250.

[18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *SIGMOD '18*. ACM, 1433–1445. https://doi.org/10.1145/3183713.3190657

[19] Amélie Gheerbrant, Leonid Libkin, Liat Peterfreund, and Alexandra Rogova. 2025. GQL and SQL/PGQ: Theoretical models and expressive power *(Proceedings of the VLDB Endowment)*. VLDB Endowment, 1798–1810. https://doi.org/10.14778/3725688.3725707

[20] Jiaqi Gu, Yugo H. Watanabe, William A. Mazza, Alexander Shkapsky, Mohan Yang, Ling Ding, and Carlo Zaniolo. 2019. RaSQL: Greater Power and Performance for Big Data Analytics with Recursive-aggregate-SQL on Spark *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 467–484. https://doi.org/10.1145/3299869.3324959

[21] Anna Herlihy, Anastasia Ailamaki, and Martin Odersky. 2025. Static Typing Meets Adaptive Optimization: A Unified Approach to Recursive Queries *(DBPL '25)*. Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3735106.3736533

[22] Anna Herlihy, Amir Shaikhha, Anastasia Ailamaki, and Martin Odersky. 2025. Language-Integrated Recursive Queries. arXiv:2504.02443 [cs.PL] https://arxiv.org/abs/2504.02443

[23] Aidan Hogan. 2020. *SPARQL Query Language.* Springer International Publishing, Cham, 323–448. https://doi.org/10.1007/978-3-030-51580-5_6

[24] ISO/IEC. 1999. ISO/IEC 9075:1999 — Information Technology — Database Languages — SQL. https://www.iso.org/standard/23493.html. International Organization for Standardization.

[25] ISO/IEC. 2023. ISO/IEC 9075-16:2023 — Information Technology — Database Languages — SQL — Part 16: Property Graph Queries (SQL/PGQ). https://www.iso.org/standard/76584.html. International Organization for Standardization.

[26] ISO/IEC. 2024. ISO/IEC 39075:2024 Information technology — Programming languages — GQL. https://www.iso.org/standard/78175.html. International Organization for Standardization.

[27] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *CAV*. Cham, 422–430.

[28] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an open framework for query optimization and compilation. *Proc. VLDB Endow.* 15, 11 (July 2022), 2389–2401. https://doi.org/10.14778/3551793.3551801

[29] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. *ICDE* (2011).

[30] David Klopp, Sebastian Erdweg, and André Pacak. 2024. A Typed Multi-level Datalog IR and Its Compiler Framework. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 327 (Oct. 2024), 29 pages. https://doi.org/10.1145/3689767

[31] Louisa Lambrecht, Torsten Grust, Altan Birler, and Thomas Neumann. 2025. Trampoline-Style Queries for SQL. In *Proc. CIDR*. Amsterdam, The Netherlands.

[32] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. 1979. Testing Implications of Data Dependencies. *ACM Trans. Database Syst.* 4, 4 (1979). https://doi.org/10.1145/320107.320115

[33] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

[34] Thomas Neumann and Viktor Leis. 2024. A Critique of Modern SQL and a Proposal Towards a Simple and Expressive Query Language. In *CIDR 2024*. www.cidrdb.org. https://www.cidrdb.org/cidr2024/papers/p48-neumann.pdf

[35] Anil Pacaci, Alice Zhou, Jimmy Lin, and M. Tamer Özsu. 2017. Do We Need Specialized Graph Databases? Benchmarking Real-Time Social Networking Applications *(GRADES'17)*. Association for Computing Machinery, New York, NY, USA, Article 12, 7 pages. https://doi.org/10.1145/3078447.3078459

[36] Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E Koch. 2017. Unifying analytic and statically-typed quasiquotes. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–33.

[37] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD*. ACM, 1981–1984.

[38] Alexandra Rogova et al. 2023. A Researcher's Digest of GQL. https://api.semanticscholar.org/CorpusID:260068940

[39] Hesam Shahrokhi, Amirali Kaboli, Mahdi Ghorbani, and Amir Shaikhha. 2024. PyTond: Efficient Python Data Science on the Shoulders of Databases. In *ICDE*. 423–435.

[40] Amir Shaikhha. 2024. Restaging Domain-Specific Languages: A Flexible Design Pattern for Rapid Development of Optimizing Compilers. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 80–93.

[41] Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional collection programming with semi-ring dictionaries. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–33. https://doi.org/10.1145/3527333

[42] Amir Shaikhha, Yannis Klonatos, and Christoph Koch. 2018. Building efficient query engines in a high-level language. *ACM Transactions on Database Systems (TODS)* 43, 1 (2018), 1–45.

[43] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In

SIGMOD'16, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1907–1922. https://doi.org/10.1145/2882903.2915244

[44] Amir Shaikhha, Dan Suciu, Maximilian Schleich, and Hung Q. Ngo. 2024. Optimizing Nested Recursive Queries. *Proc. ACM Manag. Data* 2, 1 (2024), 16:1–16:27. https://doi.org/10.1145/3639271

[45] Chandan Sharma, Pierre Genevès, Nils Gesbert, and Nabil Layaïda. 2025. Schema-Based Query Optimisation for Graph Databases. *Proc. ACM Manag. Data* 3, 1, Article 72 (Feb. 2025), 29 pages. https://doi.org/10.1145/3709722

[46] Yannis Smaragdakis and Martin Bravenboer. 2010. Using Datalog for Fast and Easy Program Analysis. In *Datalog*. https://api.semanticscholar.org/CorpusID:2014940

[47] Michael Stonebraker and Joseph M Hellerstein. 1998. *Readings in database systems*.

[48] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *Proc. VLDB Endow.* 16, 4 (Dec. 2022), 877–890. https://doi.org/10.14778/3574245.3574270

[49] D. J. Troy, C. T. Yu, and W. Zhang. 1989. Linearization of Nonlinear Recursive Rules. *IEEE Trans. Softw. Eng.* 15, 9 (Sept. 1989), 1109–1119. https://doi.org/10.1109/32.31368

[50] Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, and Dan Suciu. 2022. Optimizing Recursive Queries with Progam Synthesis. In *SIGMOD '22, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 79–93. https://doi.org/10.1145/3514221.3517827