QueryBridge: GraphQL-to-Datalog Optimizer

Translate GraphQL queries into Datalog programs with demand-transformation optimisation

Abishek Aditya
Stony Brook University
116551308

May 14, 2025

1 Problem and Plan

1.1 Problem Description

Objective. QueryBridge converts *GraphQL* queries into *XSB Datalog* and rewrites the resulting program with demand transformation—a magic-sets–style optimisation that aggressively pushes bindings downward, eliminating unnecessary intermediate relations. The tool accepts:

1. a GraphQL schema file,
2. one or more GraphQL query files.

It emits an `.P` file whose top-level predicate is `ans/N`. Demand transformation follows Liu *et al.* [1].

Interface.

- **CLI** `python -m querybridge schema.gql query.gql [--demand]`
- **API** `translate_graphql_to_xsb(schema, query, apply_demand)`→string

Requirements.

1. Parse GraphQL schemas and queries.
2. Generate semantically equivalent Datalog.
3. Apply demand transformation on request.
4. Support nested selections, arguments and fragments.
5. Translate typical queries in under two seconds.

2 Overview of the Tool

QueryBridge is organised as a three-stage pipeline:

1. GraphQL front-end builds typed ASTs.
2. Rule generator emits naïve Datalog.
3. Optimiser rewrites with demand transformation, then pretty-prints.

3 Features

- GraphQL schema and query parsing.
- Generation of XSB-compatible Datalog.
- Demand transformation optimiser [1].
- Support for deeply nested queries and argument filters.
- Clean, extensible Python code base.

4 Methodology: Demand Transformation and SIP Strategies

- **Demand Transformation** [1]: Refines classic magic-set rewriting by dynamically pruning irrelevant facts during evaluation, slashing intermediate materialisation cost.
- **SIP (Sideways Information Passing) Strategies** [2]: Control how variable bindings propagate between sub-goals, letting the optimiser choose an evaluation order that minimises search space in Datalog.

These two techniques steer both translation and optimisation, enabling QueryBridge to handle deeply nested and recursive GraphQL queries efficiently.

5 Prior Research

- **Hasura GraphQL Engine** [3]: Generates optimised SQL from GraphQL but hides the translation inside a server process.
- **XSB Datalog Engine** [4]: Mature logic-programming runtime with tabling and indexing; used here as the execution back-end.
- **Magic-Set Optimisation Slides** [2]: Overview of magic sets, semi-naïve evaluation and SIP strategy selection.
- **Deductive DBMS CORAL** [5]: Demonstrates magic-set style optimisations in a real database setting.

6 Installation

1. Clone the repository:

```
git clone <repo-url>
cd QueryBridge
```

2. Create a virtual environment and install:

```
python -m venv venv
source venv/bin/activate          # Windows: venv\Scripts\activate
pip install -e .
```

7 Usage

7.1 Command-Line

```
python -m querybridge                    # default demo
python -m querybridge schema.gql query.gql --demand
```

7.2 Library

```
from querybridge import translate_graphql_to_xsb

code = translate_graphql_to_xsb("schema.gql", "query.gql",
                                apply_demand=True)
print(code)
```

7.3 Testing

Run the consolidated end-to-end test suite:

```
python tests.py
```

The script prints its results as a Python docstring, for example:

```
=== Running test in basic ===
  Generating XSB (no demand)... done.
  Generating XSB (with demand)... done.
  Running XSB (no demand)... done.
  Running XSB (with demand)... done.
  PASS: outputs match

=== Running test in nested ===
  Generating XSB (no demand)... done.
  Generating XSB (with demand)... done.
  Running XSB (no demand)... done.
  Running XSB (with demand)... done.
  PASS: outputs match

Summary: 2/2 tests passed.
```

The script exits with code 0 if all tests pass, or non-zero otherwise.

7.4 Benchmark

Micro-benchmark GraphQL/SQLite vs. XSB. For each ‘schema.graphql‘ + ‘query.graphql‘ in the given test folder, this script:

1. Runs GraphQL via Ariadne + SQLite and measures timing.
2. Generates XSB code (with and without demand) via `translate_graphql_to_xsb`, writes Prolog drivers, runs them in XSB, and measures timing.
3. Verifies that the outputs match between both XSB variants and compares counts with the SQLite path.

```
pip install ariadne graphql-core sqlalchemy

python benchmark.py tests --runs 5 --xsb-path /usr/local/bin/xsb
```

Use `--runs` to override the default number of iterations, and `--xsb-path` to point to your XSB executable.

8 Core Query Generation Process

QueryBridge's translator is organised around three tightly-coupled code-generation routines. Together they turn a typed GraphQL AST into an XSB program that is both semantically correct and, when requested, fully optimised with demand transformation.

8.1 `generate_predicate_rules(ast_node)` → `List[Rule]`

- Accepts a field-selection, fragment, or inline fragment node.
- Emits naïve Datalog rules whose *head* predicate corresponds to the GraphQL field name.
- For nested selections the function recurses, stitching together child predicates with join conditions determined from the schema's type relationships (e.g. `@oneToMany`, interfaces, unions).
- Attaches argument constraints as *equality literals* in the rule body so that simple filters are pushed as far down the join tree as possible even before demand optimisation.

8.2 `generate_answer_predicate(root_field, binds)` → `Rule`

- Builds the `ans/`N predicate that serves as the query answer returned to the user.
- Injects constant bindings gathered from GraphQL arguments (`id`, `name`, ...) directly into the goal list, ensuring evaluation can start with those keys bound.
- The resulting rule is the *only* entry point for evaluation, which simplifies later correctness proofs and unit checks.

8.3 `generate_demand_transformation(rules)` → `List[Rule]`

1. **Create demand predicates.** For every original predicate p/k a fresh demand version p_d/k is generated, whose first arguments encode the bound keys.
2. **Propagate bindings via SIP.** Sideways-information-passing rules are synthesized to move key bindings from parent to child predicates exactly as prescribed by the SIP strategy chosen for each join (hash-join, indexed lookup, depth-first, etc.).
3. **Rewrite rule bodies.** Occurrences of the original predicates are replaced by their demand-aware counterparts, guarded by the newly created demand relations. This step is semantics-preserving [1] yet typically shrinks the ground program by an order of magnitude in our benchmarks.

Together, these three passes reduce a complex GraphQL query to a compact, table-driven XSB program that executes with far fewer intermediate tuples—especially when deep nesting or recursion is involved.

Project	Capabilities
Hasura/PostGraphile	Full GraphQL servers in front of Postgres—planning, auth, caching, live queries—yet the SQL generator is buried inside the server; no "give me SQL" API.
graphene,strawberry-sqlalchemy	Maps SQLAlchemy models ⇒ GraphQL types and runs through the ORM; never produces raw SQL text.
sgqlc-build-sqla (prototype)	Creates SQLAlchemy models from a schema and helps construct queries, but still demands manual AST walking; not production-ready.
joernio/graphql-to-sql (2019 demo)	Translator for a tiny GraphQL subset (no fragments, variables, nesting); unmaintained and outputs only naïve `SELECT`s.

Table 1: Why existing work cannot serve as a direct GraphQL-to-SQL translator.

9 State of the Art

10 Implementation Status

QueryBridge will take a GraphQL schema + query pair, compile it to naïve XSB Datalog, and then rewrite the program with classic Magic-Set demand transformation so that only facts relevant to the bound arguments reach the join—yielding visibly smaller rule sets and faster XSB execution in our unit tests. The unified test runner (tests.py) now discovers every subfolder under tests/, generates two Datalog files (with and without demand), executes both under XSB, and asserts that the ans/... predicates are identical, confirming that demand transformation preserves semantics even as it prunes intermediate relations. Likewise, the benchmark script (benchmark.py) has been fleshed out to time each variant over configurable runs, report throughput and rule-count statistics, and compare naïve vs. demand-transformed performance.

11 Future Work

- Integrate a cost model to decide when demand transformation is profitable.
- Emit SQL for non-recursive fragments to enable hybrid SQL + Datalog back-ends.
- Extend recursion support once a suitable GraphQL-to-SQL translator emerges.

Acknowledgements

Thanks to Prof. Yanhong Annie Liu for guidance and the GraphQL Foundation for `graphql-core`.

References

[1] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Graph queries through Datalog optimizations. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010. `https://doi.org/10.1145/1836089.1836093`.

[2] P. Brass. *Magic Sets, SIP Strategies and Optimising Logic Programs.* Lecture slides, 2022. `https://users.informatik.uni-halle.de/~brass/lp22/print/cd_magic.pdf`.

[3] Hasura Inc. Hasura GraphQL engine. `https://hasura.io/graphql/`. Accessed 2025-04-18.

[4] The XSB Project. The XSB logic programming and deductive database system. `https://xsb.sourceforge.io`. Accessed 2025-04-18.

[5] R. Ramakrishnan, A. Silberschatz, and D. Stuckey. The CORAL deductive database system. *The VLDB Journal*, 1993.