

# GraphQL to Datalog Optimizer

Abishek Aditya

CSE505 Spring 2025

May 14, 2025

# Roadmap

Motivation

Background

Architecture

Translation

Demand Transformation

Evaluation

Discussion

Conclusion

# Why am I here?

Alphabetical luck.

# Motivation: Why optimize GraphQL?

- ▶ Over-fetching leads to wasted bandwidth and latency
- ▶ Complex nested queries strain backend resources
- ▶ Deep queries incur exponential overhead with naive approaches
- ▶ Need for declarative, optimizable representation → Datalog
- ▶ GraphQL lacks standard optimization techniques across various implementations

## Example Challenges

- ▶ Nested queries retrieving all records before filtering
- ▶ Join order optimization not available across all implementations
- ▶ Missing opportunities for pushing filters down

# Background: GraphQL & Datalog

## GraphQL

- ▶ Hierarchical selection of nested data
- ▶ Client-driven shape and field specification
- ▶ Popular API query language
- ▶ Built for graph-like data retrieval
- ▶ Lacks standardized backend optimization

## Datalog

- ▶ Declarative logic programming language
- ▶ Deep theoretical optimization foundation
- ▶ Suited for fixed-point analysis & optimization
- ▶ Well-understood complexity guarantees
- ▶ Widely used in database theory and systems

# Architecture: Project Structure

## Source Organization

```
.  
docs/  
  demand_optimization_example.md  
  demand_transformation.md  
  index.md  
src/  
  querybridge/  
    __init__.py  
    __main__.py  
    translator.py  
tests/  
  basic/  
  nested/  
  complex_path/  
  path_finding/  
run-tests.py  
setup.py
```

## Processing Pipeline

1. **Parse** GraphQL schema and query → structured representation
2. **Translate** to intermediate Datalog representation
3. **Optimize** via demand transformation (magic sets)
4. **Generate** executable XSB Datalog code

## Key Components

- ▶ Core translator (translator.py)
- ▶ Data structures for schema and query

# Translation Process

- ▶ Parse GraphQL schema to understand types and relationships
- ▶ Parse GraphQL query to extract field selection and arguments
- ▶ Generate fact predicates from schema entities
- ▶ Generate rule predicates for query fields
- ▶ Apply demand transformation for optimization
- ▶ Generate final answer predicate

## Key Insights

Field selection becomes projection in Datalog

Nested fields become join operations

Arguments become value constraints

Field paths translate to variable sharing

# Translation — GraphQL Query Example

```
{  
  project(name: "GraphQL") {  
    tagline  
    contributors {  
      name  
      email  
    }  
  }  
}
```



## Translation — XSB Datalog (before optimization)

```
% Rules for field: project
project_result(ROOT) :- project_ext(ROOT), NAME = "GraphQL".
project_tagline_result(PROJECT_1, TAGLINE_2) :- tagline_ext(PROJECT_1,
    TAGLINE_2).
project_contributors_result(PROJECT_1) :- contributors_ext(PROJECT_1).
project_contributors_name_result(CONTRIBUTORS_3, NAME_4) :- name_ext(
    CONTRIBUTORS_3, NAME_4).
project_contributors_email_result(CONTRIBUTORS_3, EMAIL_5) :- email_ext(
    CONTRIBUTORS_3, EMAIL_5).

% Final answer predicate
ans(TAGLINE, CONTRIBUTORS_NAME, CONTRIBUTORS_EMAIL) :-
    project_ext(PROJECT_1), project_result(ROOT),
    project_tagline_result(PROJECT_1, TAGLINE),
    project_contributors_result(PROJECT_1),
    project_contributors_name_result(CONTRIBUTORS_3, CONTRIBUTORS_NAME),
    project_contributors_email_result(CONTRIBUTORS_3, CONTRIBUTORS_EMAIL).
```

# Translation — After Demand Transformation

```
% Demand transformation facts & rules
demand_project_B("GraphQL").
m_project_B(ROOT) :- demand_project_B("GraphQL").

% Propagate demand to contributors
demand_contributors__(PROJECT_1) :- m_project_B(ROOT), project_ext(
    PROJECT_1).
m_contributors__(PROJECT_1) :- demand_contributors__(PROJECT_1).

% Optimized rules
project_result(ROOT) :- m_project_B(ROOT), project_ext(ROOT), NAME = "
    GraphQL".
project_tagline_result(PROJECT_1, TAGLINE_2) :- tagline_ext(PROJECT_1,
    TAGLINE_2).
project_contributors_result(PROJECT_1) :- m_contributors__(PROJECT_1),
    contributors_ext(PROJECT_1).

% Nested fields have demand propagated to them
project_contributors_name_result(CONTRIBUTORS_3, NAME_4) :-
    m_contributors__(PROJECT_1), contributors_ext(PROJECT_1,
        CONTRIBUTORS_3),
    name_ext(CONTRIBUTORS_3, NAME_4).

ans(TAGLINE, CONTRIBUTORS_NAME, CONTRIBUTORS_EMAIL) :- ...
```

# Demand Transformation: Big Idea

- ▶ Magic Sets technique from database query optimization
- ▶ Derive *only the facts your query needs*
- ▶ Add *demand* ("magic") predicates as guards on computation
- ▶ Converts bottom-up evaluation to be query-driven (like top-down)
- ▶ Saves time and memory on large graphs or APIs
- ▶ Most beneficial for:
  - ▶ Queries with selective filters
  - ▶ Deeply nested relationships
  - ▶ Large datasets

# Implementation in QueryBridge

- ▶ Identify bound arguments (constraints/literals in query)
- ▶ Create adornment patterns (B for bound, F for free)
- ▶ Generate demand predicates as computation seeds
- ▶ Create magic predicates as rule guards
- ▶ Propagate demand through nested relationships
- ▶ Optimize predicates with bound arguments first

## Key Functions

`generate_demand_transformation()` - Creates demand and magic predicates

Tracks applied transformations and reasons in logs

Handles nested field propagation automatically

## Example — GraphQL with Filters

```
{  
  users(minAge: 25, maxAge: 40, nameContains: "Smith") {  
    name  
    email  
  }  
}
```

```
% Seed demand with filter values  
demand_users_BBB(25, 40, "Smith").  
  
% Magic predicate to guard computation  
m_users_BBB(ROOT) :- demand_users_BBB(25, 40, "Smith").  
  
% Guard rule with filter values passed in  
users_result(ROOT) :- m_users_BBB(ROOT), users_ext(ROOT),  
                      MINAGE = 25, MAXAGE = 40, NAMECONTAINS = "Smith".
```

# Effect of Demand Transformation

- ▶ **Before:** computes up to  $|V|^2$  pairs.
- ▶ **After:** touches only edges from `alice` and neighbors.
- ▶ Metaphor: “Ask who’s hungry, then bake just those slices.”

# Nested Relationship Example

```
% For a query like: user(id: "1") { posts { comments { author } } }  
  
% Seed initial demand  
demand_user_B("1").  
  
% Define magic predicates  
m_user_B(UserID) :- demand_user_B(UserID).  
  
% Propagate demand to posts  
demand_posts_B(UserID) :- m_user_B(UserID).  
m_posts_B(PostID) :- demand_posts_B(UserID), user_posts(UserID, PostID).  
  
% Propagate demand to comments  
demand_comments_B(PostID) :- m_posts_B(PostID).  
m_comments_B(CommentID) :- demand_comments_B(PostID), post_comments(  
    PostID, CommentID).
```

# Performance Impact of Demand Transformation

## ▶ **Without optimization:**

- ▶ Processes all users, all posts, all comments
- ▶ Potentially examines millions of irrelevant records
- ▶ Filters applied only after full computation

## ▶ **With optimization:**

- ▶ Starts with specific user "1"
- ▶ Processes only posts from that user
- ▶ Retrieves only comments on those posts
- ▶ Computation proportional to output size, not input size

## Benchmarking Results

Orders of magnitude performance improvement on large datasets

Higher impact with more selective filters

Critical for complex nested queries in production environments



# QueryBridge Implementation

- ▶ Python package with modular architecture:
  - ▶ SchemaType and QueryField data structures
  - ▶ Schema and query parsers using GraphQL-core
  - ▶ Predicate rule generators for XSB Datalog
  - ▶ Demand transformation optimization engine
- ▶ Comprehensive test suite in /tests:
  - ▶ Basic schema and query tests
  - ▶ Nested relationship tests
  - ▶ Complex path finding with deep graphs
  - ▶ Variable capitalization tests
- ▶ Command-line interface: `python -m querybridge`
- ▶ Library API for integration

# Evaluation: Test Cases and Performance

## Test Cases

- ▶ Basic: Simple queries with 1-2 levels
- ▶ Nested: Relationships up to 3 levels deep
- ▶ Complex Path: Up to 7 levels of nested relationships
- ▶ Path Finding: Graph traversal with complex filters

## Performance Improvements

- ▶ Basic: 1.5-2x speedup
- ▶ Nested: 2-3x speedup
- ▶ Complex Path: 3-10x speedup
- ▶ With Selective Filters: Up to 50x speedup

## Key Observations

Demand transformation benefits increase with query depth

Selective filters show greatest performance gains

Optimized queries scale better with dataset size

# Discussion: Trade-offs

## Pros

- ▶ Declarative, optimizable intermediate representation
- ▶ Centralized optimization techniques for any GraphQL implementation
- ▶ Compatible with existing GraphQL schemas

## Cons

- ▶ Initial translation overhead
- ▶ Additional complexity in the stack
- ▶ Requires XSB Datalog runtime
- ▶ Integration challenges with existing systems

# Conclusion and Future Work

## Output

- ▶ Successful translation of GraphQL to XSB Datalog
- ▶ Efficient demand transformation for query optimization
- ▶ Support for complex nested queries with deep paths

## Future Directions

- ▶ Improve performance to match popular graphql implementations (sqlalchemy, hasura)
- ▶ Integration with window functions
- ▶ Optimization for non-normalized data via materialized views

GitHub: [github.com/abishekaditya/QueryBridge](https://github.com/abishekaditya/QueryBridge)

Questions?