SDQL Compiler

Examination: Part II

Year: 2026

Declaration of originality

I, the candidate for Part II of the Computer Science Tripos with Blind Grading Number 3961J, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this report, I adhered to the Department of Computer Science and Technology AI Policy. I am content for my report to be made available to the students and staff of the University.

Date May 15, 2026

Proforma

BGN: 3961J

Examination: Part II (2026)

Project Title: SDQL Compiler

Word-count: 11754 / 12000

Code line count: 7834

Project Originator: Amir Shaikhha

Day-to-Day supervisor: Neel Krishnaswami

Marking Supervisor: Neel Krishnaswami

Ethics Approval: None

Summary of Original Aims

The project aimed to build an independent Lean compiler for SDQL (functional database language grounded in semiring dictionaries [1]), achieving feature parity with the existing reference implementation. Beyond reimplementation, the project sought to extend SDQL with a closure operator and semiring multiplication, drawing on Tarjan's unified algebraic treatment of path problems [2], so graph algorithms could be expressed within the same dictionary formalism as relational queries. A further aim was to implement and empirically evaluate the algebraic optimisations described in the original SDQL specification.

Summary of Work Completed

I implemented `lean-sdql`, a complete SDQL compiler in Lean that generates Rust, exploiting Lean's dependent types and hygienic syntax macros to encode a indexed core IR; The compiler was validated against the reference implementation across the full TPC-H benchmark using differential testing. I extended SDQL beyond its specification with a closure operator, semiring multiplication, and a graph extension grounded in semirings. I implemented algebraic optimisations, and benchmarked lean-sdql against both the reference compiler and NetworkX, demonstrating competitive performance on relational queries and superior performance on graph workloads.

Summary of Special Difficulties

None recorded.

Table of Contents

Chapter 1: Introduction 1
1.1 Motivation 1
1.2 Intuition Behind Graph Theory/Relational Algebra/Linear Algebra 2
1.3 Background on SDQL 2
1.4 Previous Work on SDQL/Background 3
1.5 Background on Lean4 5
1.6 Key Contributions of this Dissertation 5
Chapter 2: Preparation 6
2.1 Software Engineering Methodology 6
2.1.1 Continuous Integration Pipeline Testing Strategy 8
2.1.2 Testing Methodology and Strategy 8
2.2 Research and Background reading 8
2.3 Starting Point 8
2.4 Proposal Refinement 8
2.5 Requirements Analysis 9
2.5.1 The Use of Lean 9
2.5.2 Contingency Plans/Risk Mitigation 9
2.5.3 Tradeoffs of the Compilation Target 10
2.6 Background on Kleene Algebras/Kleene Algorithm 10
2.6.1 Kleene Algebra 10
2.6.2 Kleene's Algorithm 11
2.6.3 Kleene's Algorithm and Graph Problems 11
Chapter 3: Implementation 12
3.1 Introduction to Lean and Dependent Type Theory 12
3.2 Modelling SDQL in Lean Data-Structures 12
3.3 Compilation Pipeline 13
3.3.1 High-Level Overview 13
3.3.2 Syntax Macro Elaboration 13
3.3.3 Intermediate Representations 15
3.3.3.1 Load Syntax 15
3.3.3.2 Untyped Syntax 15
3.3.3.3 Typed Named Syntax 16
3.3.3.4 Typed Unnamed Syntax 17
3.3.3.5 Abstract Rust Syntax 17
3.3.4 Elaboration-Time vs Runtime Split 18
3.3.5 The Use of `Unsafe` Lean 18
3.4 Graph Extension 18
3.5 Theory of Semi-Rings in Square Vector Spaces 20
3.6 Algebraic Optimisations 21
3.7 Implementation of Local Error Messages 22
3.8 Repository Overview 23
3.8.1 Compilation Files 24
3.8.2 Documentation Files 24
3.8.3 Testing Files 24
3.9 Runtime Tuning 24
Chapter 4: Evaluation 26
4.1 Experimental Setup 26
4.2 Effect of Dependent Types on Compiler Design 26

4.3 Correctness of Compilation Test-bench (TPC-H Benchmark) 27
4.4 Optimisation Performance tests 28
4.5 Reference Performance Comparison 31
4.6 The Effect of Data-structure selection 32
4.7 The Use of a Profiler 33
4.8 Evaluation of SDQL 34
4.9 Graph Database Comparison 36
Chapter 5: Conclusions 38
5.1 Lessons Learned 38
5.1.1 Reflections on Compiler Design 38
5.1.2 Reflections on SDQL 39
5.2 Future Work 39
Bibliography 40
Appendix 41
Project Proposal 48

Chapter 1: Introduction

1.1 Motivation

Data-scientists may construct pipelines that contain both data-queries and linear algebra. When a data-science workflow requires both relational queries and graph algorithms, practitioners must currently bridge incompatible tools (exporting from SQL to Neo4j to Python), losing both performance and correctness at each boundary. PL researchers have designed DSLs to capture individual parts of this workflow. Notable examples of DSLs that model this data-science pipeline include LaraDB and TACO, which will be discussed in Table 2. Relational Algebra DSLs model relational data, and linear algebra frameworks can efficiently compute over tensors. However, using different DSLs to model each stage introduces friction at the transition boundary of the data-science pipeline.

Consider the following motivating example: A data-analyst wants to compute the "most important actor". One way to do that might be to say that two actors co-star if they each play a role in the same film. That analyst then wants to compute the betweenness-centrality [3] of each actor to compute how crucial each actor is to the film industry. Betweenness-centrality is a reasonable measure of actor importance because an actor with a high betweenness-centrality would co-star with many other important actors. Throughout this running example, I will be comparing the industry-standard for data-science (SQL and Python) against SDQL, a novel database DSL based on the idea of semirings over dictionaries. Just as the fundamental operation in SQL is to `SELECT` from a `TABLE`, the fundamental aggregation operation in SDQL is to `sum(...)` over the entries of some dictionary `{K -> V}`.

The abstraction of "actors co-starring" can be elegantly represented by an SQL relational `JOIN`.

```
SELECT
    a.actor_id AS actor_1,
    b.actor_id AS actor_2
FROM film_cast a
JOIN film_cast b
ON  a.film_id  = b.film_id
```

SDQL can also represent joins naturally using its dictionary paradigm:

```
sum(role1 in actors)
  sum(role2 in actors)
    if role1.film == role2.film then
        {<actor1 = role1.actor, actor2 = role2.actor> -> true}
```

However, now if the analyst wants to compute the betweenness-centrality of the actors, SQL must export the result to `.csv`, so that the graph computations can be computed via Python. This gluing introduces friction. When the analyst must glue SQL output to a Python graph library, they waste time reformatting data. Conceptually, what could have been done in a single tool has instead been artificially split between three tools.

SDQL presents a more elegant solution for computing betweenness-centrality.

```
-- In SDQL (simplified):
let costar_graph = <join actors on shared films>
let shortest_paths = closure(costar_graph)  -- transitive closure!
let centrality = <aggregate over shortest paths>
```

The given example is pseudo-code. The full code for betweenness centrality in SDQL is available in the appendix Listing 3.

SDQL achieves expressivity by modelling data operations as semi-rings over dictionaries.

1.2 Intuition Behind Graph Theory/Relational Algebra/Linear Algebra

Before I even define the formalism, I will give intuition about why we should even expect there to be a correspondence between graph theory, relational algebra, and linear algebra, and why it seems reasonable that we ought to use a unified abstraction to express computations in all three domains.

First, given a relation $R(a, b)$, we may model this relation as a boolean matrix, M, where

$$M_{ab} = 1 \Leftrightarrow R(a, b).$$

Second, given a graph $G = (V, E)$, we may represent this graph as a boolean matrix $M : |V| \times |V|$, where

$$M_{ab} = 1 \Leftrightarrow (a, b) \in E.$$

This pattern does not only occur over the boolean-semiring, but also is relevant over the real-semiring. Consider the function $\text{countPaths} : \text{location} \times \text{location} \to \mathbb{R}$.

$$\text{countPaths}(A, C) \coloneqq \sum_{B \in \text{ locations}} \text{countPaths}(A, B) \times \text{countPaths}(B, C)$$

Once again, it's the same algebraic structure, instantiated over a different semiring.

In practice, data scientists experience friction converting/exporting between different representations, i.e. exporting from SQL query to Neo4j to Python, but this conversion is actually unnecessary. Semirings present an abstraction useful enough to model all three domains. Thus, a programming language that can model semirings can do hybrid data-science workloads without conversions costs, and have unified optimisation passes that straddle the boundaries between conversion stages.

Let us consider the example of forming a database join on a function $\text{birthDate} : \text{person} \times \text{date} \to \mathbb{B}$. In the boolean semiring, $+$ corresponds to $\vee$, $*$ corresponds to $\wedge$, and Σ corresponds to relational join. Thus, while finding people who share a database would typically be expressed in the notation of a relational join, it can also be expressed in the notation of a semiring

$$\text{coBirth}(p_1, p_2) \coloneqq \sum_{d \in \text{ date}} \text{birthDate}(p_1, d) \times \text{birthDate}(p_2, d)$$

Computationally, this correspondence is not merely a mathematical nicety, but a computational tool which allows us to compute over relations and graphs in a uniform and principled manner. Section 3.4 explains how well-known graph-algorithms can be represented as operations over adjacency matrices.

1.3 Background on SDQL

SDQL is a functional, immutable, strongly typed programming language used for databases. SDQL is based around the operations of a semiring, and collections over semirings (semimodules) [4].

The fundamental collection in SDQL is the dictionary `{K -> V}`, where the dictionary forms a semimodule when its elements form a semiring. Aggregation operations over the collection are performed by `sum(<k,v> in dict) ...ops...`, which aggregates via the semiring operation. For the syntax of SDQL, see Table 22.

$$\frac{\text{AddM } t \qquad \Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 + e_2 : t} \text{ T-Add}$$

$$\frac{\text{ScaleM } st_1 \qquad \text{ScaleM } st_2 \qquad \Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 * e_2 : t_1 \otimes t_2} \text{ T-Mul}$$

Figure 1: The rules `add` and `mul` demonstrate the SDQL is based on semimodules over a semiring

In Figure 1, I have highlighted just two typing rules from SDQL (`add` and `mul`), which show the semimodule structure of SDQL differing from typical addition/multiplication rules.

Aggregation operations in SDQL are performed via `sum(<k, v> in d) e`. It has the runtime behavior that

```
sum(<k, v> in {k1 -> v1, k2 -> v2, ..., kn -> vn}) e(k,v) = e(k1, v1) + e(k2, v2)
+ ... + e(kn, vn)
```

. For example, if the type `e(K, V) : bool`, then `sum` will have the semantics of an existential quantifier, or a "big or".

The key insight of aggregation via `sum` is that the + operator is not concrete, but + in SDQL operates generically over any semiring. Thus, the runtime behavior of `sum` depends on its type.

It's also worth explicitly explaining how multiplication works in SDQL: Multiplication does not have the type `* : T -> T -> T` (but I do introduce the new operator `*s : T -> T -> T` in this dissertation). Rather, multiplication has the type`* : A -> B -> A * B`. This makes * non-commutative.

```
{a->2, b->3} * {a->4, c->5}
= {a->{a->8, c->10 }, b->{a->12, c->15}}
```

but

```
{a->4, c->5} * {a->2, b->3}
= {a->4 * {a->2, b->3}, c->5 * {a->2, b->3}}
= {a->{a->8, b->12}, c->{a->10, b->15}}
```

Notice that the return type of multiplication corresponds to the tensor product of a vector space: $t_1 \otimes t_2$. The presentation only appears unconventional because dictionaries use named keys, instead of numbered places used in a vector. If we re-write `{a->4, c->5} * {a->2, b->3}` as `<4,5> * <2,3>`, the correspondence with the tensor $\langle 4, 5\rangle \otimes \langle 2, 3\rangle = \begin{pmatrix} 8 & 12 \\ 10 & 15 \end{pmatrix}$ becomes obvious.

For the full SDQL typing rules, see Figure 17.

1.4 Previous Work on SDQL/Background

SDQL was introduced by researchers at the University of Edinburgh [1]. Shaikhha et al. later gave added support for sparse tensor operations [5]. SDQL is based on the theory of semi-rings [6].

Paradigm	Theoretical Grounding	Language Implementation
Relational Algebra	multisets	SQL
dictionary-based	semimodules	SDQL

Table 1: Comparing SQL vs SDQL

DB	Theoretical foundation	Supports relational queries	Supports linear algebra	Supports graph algorithms	Compilation target	Algebraic optimisations
SQL	relational algebra	yes	no	no	C	yes
SDQL	semimodules over dictionaries	yes	yes	no	C++	no
SDQL + semiring + closure + optimisations	semimodules over dictionaries	yes	yes	yes	Rust	yes
LaraDB	associative tables over semirings	yes	yes	no	C++	yes
TACO/ GraphBLAS	index notation over tensors	no	yes	no	C	yes
Neo4J / Cypher	property graphs	partially	no	yes	Java (JVM bytecode)	yes
NetworkX	graphs (adjacency dict-of-dicts)	no	no	yes	interpreted Python	no

Table 2: Comparing SDQL against other DBs

SDQL is situated in the space of database languages. The dominant paradigm (relational algebra [7]) is based on multisets, whereas SDQL is based upon dictionaries of semirings. I compare the paradigms of SQL vs SDQL in Table 1.

Additionally, in the field, we can situate SDQL with respect to other attempts at a unified linear-algebra/ relational algebra database, including LaraDB [8], TACO [9] and GraphBLAS [10].

LaraDB represents collections as associative tables and supports both relational and linear algebra operations, but it lacks the semiring-polymorphic aggregation that gives SDQL its genericity and LaraDB's operations are fixed rather than parameterised by an arbitrary semiring. TACO is a compiler for sparse tensor algebra that uses index notation to generate efficient code for tensor contractions, but it operates exclusively over dense and sparse tensors and cannot express relational queries such as joins or selections, making it complementary to rather than competitive with SDQL's dictionary-based approach.

This project is unusual for using a dependently typed language for compilation. This has been done before: the Lean4 compiler is self-hosted in Lean, and the Idris compiler is self-hosted in Idris [11].

1.5 Background on Lean4

Lean4 is a dependently-typed functional programming language [12]. Lean4 is based on the theory of intuitionistic dependent type theory [13]. Lean's foundation in type-theory allows it to express complex types in its type-signature, while still remaining computable.

1.6 Key Contributions of this Dissertation

This dissertation implements and extends with new features SDQL, a DSL based on semi-rings which is flexible enough to serve as a hybrid DB and also support Linear Algebra workloads.

Thus, we are naturally motivated for the following research questions:

1. Can a dependently-typed language like Lean4 yield a cleaner compiler architecture for a language with rich algebraic structure?
2. Can language features of dependent types and hygienic syntax macros improve the ergonomics and extensibility of a compiler architecture?
3. Can SDQL be cleanly extended with semiring and closure properties, without diluting the original language?
4. Can closure properties of SDQL be applied performantly to graph problems, and be competitive against specialised graph databases?

Tarjan [2] demonstrates a unified technique for using semi-rings to solve graph-problems. This approach was referenced, but not implemented, in the original SDQL paper.

This dissertation contributes the following contributions:

1. Lean4 reimplementation,
2. closure operator implementation (which the original paper specified but didn't implement),
3. semiring multiplication `*s`,
4. graph extension via Kleene's algorithm
5. algebraic optimisations.

Contributions 2-5 are novel to this dissertation, and have not been implemented in any previous SDQL implementation.

This dissertation demonstrates that SDQL can be extended with Kleene-algebraic closure to express graph algorithms, and that a dependently-typed implementation language provides expressivity for compiler construction.

Chapter 2: Preparation

2.1 Software Engineering Methodology

I used the spiral methodology of software development [14]. This methodology was appropriate for this project because a compiler is a deep and staged process. Thus, it is valuable to gain partial feedback and partial progress early-on. I assessed spiral as being appropriate to quickly discover and iterate on risks/unknowns in the project. For example, using a dependently-typed AST and PHOAS were both unknowns at the preparation stage, so I iterated in a spiral to quickly discover the unknowns/challenges/painpoints. I have documented the iterations in my spiral in Table 3.

The use of the spiral methodology paid off for me because I was able to quickly assess and overcome unknowns of this project, including:

1. Haskell vs Lean
2. PHOAS vs DeBruijn representations
3. Statically-generated runtime vs dynamic runtime.

I also used TDD [15] because I set up the TPC-H testsuite before I had implemented the features necessary to run the testsuite. TDD was appropriate to use in this scenario because I gained quantifiable evidence of tests turning from red to green as I added more features to lean-sdql. Using TDD accelerated my development cycle, especially when adding the built-in functions to SDQL, because I could see which test-cases were still failing because of unimplemented built-ins.

Spiral Iteration	Date
Core AST	Oct 11 2025
Surface-Core translation	Oct 14 2025
Record types	Oct 17 2025
Add sum, addition evidence AddM	Oct 18 2025
Basic Rust codegen	Oct 23 2025
Package in Nix Flake	Oct 23 2025
Add testcases for open/closed terms	Oct 23 2025
Compare CI tests on stringified values	Oct 27 2025
Add Syntax Macro DSL	Oct 27 2025
Sort fields in record literal	Nov 3 2025
Add SDQLProg DSL to express programs with loads	Nov 4 2025
Add TPC-H benchmark to test-bench	Nov 9 2025
Add support for builtin functions used in TPC-H	Nov 10 2025
Link generated Rust with Rust runtime	Dec 8 2025
Refactor PHOAS into DeBruijn syntax	Dec 19 2025
Create Performance Comparison Tester	Jan 5 2026
Add closure and *s builtins	Jan 27 2026
Profile with flamegraphs	Jan 28 2026
Create graph test-bench	Feb 20 2026
Create graph performance tester	Feb 23 2026
Refactor optimisations to use combinators	Mar 9 2026
Add minplus semiring	Mar 17 2026
Change dict representation from BTreeMap to HashMap	Mar 20 2026

Table 3: Iterations in the Spiral Methodology

2.1.1 Continuous Integration Pipeline Testing Strategy

I used CI to check status of small, manageable commits. On each commit, I would run a CI job, which would run the test suite. By getting quick feedback on each commit, I was able to quickly catch regressions, and I maintained a small/modular commit-style. For example, when I introduced `closure(...)`, this lead to a subtle bug in string cloning through loops, but this bug was quickly caught by the CI.

From the beginning, I employed professional techniques, including continuous integration and testing of my code.

2.1.2 Testing Methodology and Strategy

I benefitted from having an existing implementation of SDQL available. I used a test suite of unit tests, and comparison tests against the reference implementation. In the literature, this is known as a differential testing strategy. Differential testing was core to my project, as one of my requirements was parity with the reference implementation. Differential testing (combined with TDD) provided a quick harness to catch bugs early in development.

2.2 Research and Background reading

I began my preparation by doing background reading. I had to teach myself the following concepts in preparation for this project:

1. The syntax and semantics of SDQL [1].
2. The linear algebra foundations and semantics of SDQL [5].
3. The algebraic foundations of graph theory [2] for background on my algebraic paths extension.

In particular, Tarjan [2] introduces new theory related to semi-rings. I taught myself this theory (of semi-rings, semi-modules, and closed semi-rings) in preparation for the project.

I did not have to teach myself Lean for this project, as I already had extensive experience using Lean.

2.3 Starting Point

I started off having access to the reference implementation of SDQL from earlier work. The reference implementation consists of `4169` lines of Scala. Note that the reference implementation does not include any implementations of semiring multiplication nor of Kleene star closure; these were genuinely novel additions to my implementation.

I did not copy the design nor the architecture of the reference implementation. The purpose of the reference implementation was as a "black box comparison". I employed the "Clean-Room technique" [16], only observing the behavior of the reference implementation but not looking into the architecture of the reference implementation.

I also used `tpch-dbgen`[1] to generate the TPC-H dataset. `tpch-dbgen` consists of `3560` lines of C/C++. `tpch-dbgen` produces tables in the `TPC-H` benchmark, at any given scale factor (SF). In Rust, I intentionally did not rely on 3rd-party Rust dependencies. In Lean, I did not depend on `mathlib`.

2.4 Proposal Refinement

I refined my proposal by consulting with my day-to-day supervisor, and by proactively reaching out to the authors of the original SDQL paper. I was sure to get several rounds of draft and review from my day-to-day supervisor in order to ensure a polished proposal. See Figure 16 for my email conversation with the original SDQL authors.

From corresponding with the authors, I learned that the `closure(e)` operator was defined, but not implemented in the original SDQL paper. This insight influenced how I planned my extensions.

[1] https://github.com/electrum/tpch-dbgen

2.5 Requirements Analysis

I was sure to plan out appropriate software engineering techniques fitting for this project.

I used the MoSCoW (must, should, could, won't) method [17], a formal system for defining requirements of a software project. I came up with the following list of requirements for my core:

1. The compiler MUST take in any valid string in SDQL, and generate Rust code for it. This is with respect to the given BNF and typing derivations for the SDQL language that I extracted from the SDQL papers.
2. The compiler MUST generate somewhat local error messages in the case of a type-error. This requirement was explicitly flagged by my supervisor, because it is an antipattern of many research compilers to say "error", but not give informative information to debug the error. The compiler MUST attach line and column numbers to AST nodes to surface syntax errors at the editor level.
3. The performance of the generated Rust code MUST be fast enough to be practically usable (within 1 order-of-magnitude of the reference implementation)
4. The test-bench MUST provide reasonable confidence that the Lean4 compiler has correct behavior. From reading the SDQL Paper, I noticed that SDQL was evaluated on the TPC-H benchmark (a comprehensive database benchmark). Thus, I reasoned it should be sufficient to compare the outputs of my implementation of SDQL against the reference implementation of SDQL on the TPC-H benchmark.
5. The performance comparison SHOULD give statistically significant evidence that the performance of the generated Rust code is sufficient.
6. The graph extension SHOULD be performant on known graph-algorithms.
7. The optimisation extension MUST preserve correct semantics of programs
8. The optimisation extension SHOULD improve runtime performance of optimised programs
9. The optimisation extension WON'T give any proofs of termination.

It's also important to explicitly state nonrequirements:

1. lean-sdql WON'T have proofs of correctness. Lean4 is a formal theorem prover, yet proving the correctness of a compiler is a significant undertaking [18]. Thus, I decided not to prove the correctness of the compiler.
2. lean-sdql WON'T make any guarantees about compilation speed, nor the size of the compiler output
3. The original SDQL paper contains a linear-algebra specific benchmark comparing against TACO; lean-sdql WON'T reimplement this benchmark nor to compare `lean-sdql` on this benchmark because linear algebra was not part of my central claims/research questions.

2.5.1 The Use of Lean

While not many compilers have been written in Lean4 yet, I chose it for this project based on the requirements of the project

1. Lean's features as a functional language make it easy to define transformations over an AST as functions.
2. Lean's dependent types allow ASTs to hold evidence which is used in synthesis.
3. Lean's advanced syntax-macro system allowed me to create an EDSL, embedding SDQL inside Lean. This allowed a tight editor-integration experience and for the clean display of local error information.

On the other hand, it would have been possible to use (for example) OCaml or Haskell or Scala for this project. I decided against them because I was specifically interested in how dependent types and syntax macros could be applied to compiler design.

2.5.2 Contingency Plans/Risk Mitigation

Before I began, I was aware of the risks of using Lean, and I made contingency plans accordingly.

I anticipated that it would be difficult to prove the termination of a compiler to the Lean4 termination checker, and it actually was more difficult than I imagined (at the type-level, in addition to at the term-level). See Implementation section.

In my proposal, I anticipated these risks, and I made a contingency plan to rewrite the compiler in Haskell if Lean proved unworkable. Since I was implementing the core AST as the first milestone, this problem would have been encountered quickly.

In my proposal, I proposed to implement the core syntax with PHOAS, thus automatically delegating all typechecking to Lean4's built-in typechecker. At the time, this appeared to be an elegant solution to get a typechecker for free. Initially, I planned to only have one intermediate representation.

2.5.3 Tradeoffs of the Compilation Target

I proposed to compile SDQL to Rust as my compilation target. My reasoning was that the SDQL reference implementation uses Scala to compile to C++, so I didn't want to compile to C++ (to avoid duplicating the reference implementation).

There's also the issue of Rust's pervasive borrow-checker, which was an issue when compiling to Rust.

2.6 Background on Kleene Algebras/Kleene Algorithm

The theory of Kleene Algebras is necessary to understand Section 3.4, so I will give a brief digression on Kleene Algebras and Kleene's Algorithm

2.6.1 Kleene Algebra

A semiring [6] is defined as a set with addition and multiplication $(S, +, 0, \cdot, 1)$, such that the operations of the semiring obey the following laws:

1. $(S, +, 0)$ forms a commutative monoid
2. $(S, \cdot, 1)$ forms a monoid
3. Distributivity:

$$a \cdot (b + c) = a \cdot b + a \cdot c$$

$$(b + c) \cdot a = b \cdot a + c \cdot a$$

A Kleene algebra [19] is defined as a semiring with a star operator a^* such that

$$a^* = 1 + a + a^2 + ...$$

Table 4 presents examples of the Kleene Star in well-known semirings.

Name	Type	add $(+)$	multiply $(\cdot)$	star (a^*)
Real sum-product	$\mathbb{R}$	$+$	$\times$	$\frac{1}{1-a}$ *(if* $\lvert a \rvert < 1$*)*
Min-product	$(0, \infty]$	min	$\times$	1 if $a \geq 1$; 0 if $a < 1$
Max-product	$[0, \infty)$	max	$\times$	1 if $a \leq 1$; ∞ if $a > 1$
Min-sum	$(-\infty, \infty]$	min	$+$	0 if $a \geq 0$; $-\infty$ if $a < 0$
Max-sum	$[-\infty, \infty)$	max	$+$	0 if $a \leq 0$; ∞ if $a > 0$
Max-min	$[-\infty, \infty]$	max	min	$+\infty$
Boolean	$\{T, F\}$	$\vee$	$\wedge$	T

Table 4: Examples of the Kleene Star

2.6.2 Kleene's Algorithm

If S is a Kleene algebra, then $S^{m \times m}$ (the set of all $m \times m$ matrices with entries in S) is also a Kleene algebra.

The star operator on matrices M^* is defined by Kleene's Algorithm. If a graph is represented by its adjacency matrix (i.e. M_{ij} represents the edge from vertex i to vertex j), then Kleene's algorithm can be used to compute graph-theoretic properties. This is explored in [2].

Partition the $n \times n$ matrix M into four sub-quadrants:

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

where:

- A is a $p \times p$ matrix (upper-left block),
- B is a $p \times q$ matrix (upper-right block),
- C is a $q \times p$ matrix (lower-left block),
- D is a $q \times q$ matrix (lower-right block),

with $p + q = n$.

The Kleene star of M is given by:

$$M^* = \begin{pmatrix} (A + BD^*C)^* & (A + BD^*C)^* BD^* \\ (D + CA^*B)^* CA^* & (D + CA^*B)^* \end{pmatrix}$$

When $M = [a]$ is a 1×1 matrix containing a single scalar element $a \in S$, the Kleene star is simply:

$$[a]^* = [a^*]$$

2.6.3 Kleene's Algorithm and Graph Problems

As was noted by [2], by defining custom semirings, many problems in graph-theory can be viewed as Kleene's Algorithm, but each viewed through the lens of a different semiring. Table 5 gives parallel examples of algorithms viewed both from the lens of graph-problems, and also from the lens of Kleene's Algorithm over some Kleene Algebra.

Semiring $(\oplus, \otimes)$	$\boldsymbol{M^*_{ij}}$ computes
(min, +) — the tropical semiring	Shortest path distance
(max, ×) — the Viterbi semiring	Most reliable path
(max, min) — the bottleneck semiring	Maximum-capacity path
(+, ×) — over reals	If it converges, total weight summed over all paths (related to matrix inverse $(I - M)^{-1}$)
$(\vee, \wedge)$ — Boolean	Transitive closure (reachability)

Table 5: Many well-known graph problems can be viewed as special cases of Kleene's Algorithm, when specialised to a specific case of a Kleene Algebra

Chapter 3: Implementation

3.1 Introduction to Lean and Dependent Type Theory

Lean4 is a functional programming language in the vein of OCaml or Haskell, but augmented with Dependent Type Theory [20].

I will not give a complete introduction to dependent type theory, as the formal definition of dependent type theory is too dense to present here, but I will proceed by example.

In Dependent Type Theory (DTT), types may depend on values. For example, the Lean4 type `Vec 4 Nat` represents vectors of length 4 holding `Nat`. Then, the type signature of `Vec.append` becomes `Vec.append : Vec n T -> vec m T -> Vec (n + m) T`. The key difference of dependent types is that the type-signature contains a term-level expression (because `n + m` is a term, not a type).

In OCaml, the programmer can specify certain shapes of trees using an inductive datatype. Lean also supports this capability, as shown in `Ty`:

```
inductive Ty : Type where
  | bool : Ty
  | real : Ty
  | dict : Ty → Ty → Ty
  | record : List Ty → Ty
  | string : Ty
  | int : Ty
```

However, Lean offers a unique feature (mimicked weakly by Haskell GADTs) to have the shape of inductive datatypes restricted by a run-time parameter. This trick is used in the definition of `Term` to restrict to only terms that obey the variable context of variables in-scope:

```
inductive Term2 : (ctx : List Ty) → Ty → Type where
    | var : Mem ty ctx → Term2 ctx ty
    | sum : (a : AddM ty) → TermLoc2 ctx (.dict D R)  → TermLoc2 (D :: R :: ctx) ty →
Term2 ctx ty
    | letin : TermLoc2 ctx ty₁  → TermLoc2 (ty₁ :: ctx) ty₂ → Term2 ctx ty₂
```

Relevantly for this project, we represent the core AST as a dependent type `Term2 : (ctx : List Ty) → Ty → Type .`

For example, the type `Term2 [.int, .dict .int .string] .int` represents a term in the context $\Gamma = [\mathbb{Z}, \{\mathbb{Z} \to \mathbb{S}\}]$, which returns a value of type $\mathbb{Z}$.

Dependent typing provides increased expressivity, but also increases the burden on the programmer. For example, to write a function `f : Term2 Γ .int → Term2 Γ .int` implicitly necessitates a proof that `f` does not change the context and that `f` preserves types.

3.2 Modelling SDQL in Lean Data-Structures

With the flexibility offered by dependent typing, I had to make careful design decisions about how to correctly model SDQL in Lean. In short, I came to the following decision

1. Use PHOAS for the surface-level representation
2. Use DeBruijn indexing by context and type for the core-representation, erasing named records into unnamed products.

`-- DeBruijn Indexing` `.lambda (.var 0)`
`-- PHOAS` `.lambda (fun x => .var x)`

Table 6: DeBruijn Indexing vs. PHOAS

PHOAS is defined as using the meta-level binders to model the object-level binders. For example, the SDQL code

```
let x = load(...)
sum(k in x) ...
```

is represented as

```
.letIn (.loadPath ...) fun x =>
    .sum (.var x) fun k => ...
```

In the core representation, terms are indexed by their context Γ and their return-type τ. For example, `.sum` is defined as

```
sum : (a : AddM ty) → TermLoc2 ctx (.dict dom range) → TermLoc2 (dom :: range :: ctx)
ty → Term2 ctx ty
```

Notice how `.sum` expands the context `ctx` with `dom` and `range`, meaning that any term inside the sum can refer to `dom` and `range`. By indexing the term by the context, we actually prevent an entire class of bugs. For example, it is impossible to write the term `sum(k in x) y + k` if `y` is out-of-scope in the context, and this will be a compile-time error.

This decision was pragmatically justified: for the surface-level representation, we can use PHOAS to avoid issues of variable-shadowing and renaming. On the other hand, type-checking within PHOAS is still an unresolved question. In DeBruijn syntax, we have to manually do variable reordering and weakening, but we get compile-time guarantees that terms will be well-typed and valid in the context.

3.3 Compilation Pipeline

3.3.1 High-Level Overview

The high-level compilation pipeline is depicted in Figure 2. At a high level, this compiler can be described in five main components

1. The elaboration macro system: Elaborating raw user syntax into a surface AST
2. The compilation pipeline: Compiling from a high-level representation to an internal representation
3. The evidence synthesizer: Synthesizing evidence needed for compilation.
4. The algebraic optimiser: Optimising the internal representation according to equational rewrite rules.
5. The rust backend: code-generating internal representation to Rust

Each of these major components will be described in detail in the following sections.

3.3.2 Syntax Macro Elaboration

Lean contains a unique language feature: the ability to extend the parser and elaborator in user-space. I used this feature to create an embedded DSL of SDQL inside Lean. Notice how this is against the typical compiler convention of having a parser and tokenizer as a program that inputs a file and outputs the parsed ASThe type-checking and evidence synthesis was done at elaboration time. Thus, the Lean embedding gains full IDE support for free, without having to implement the LSP.

For example, if we define the syntax rule

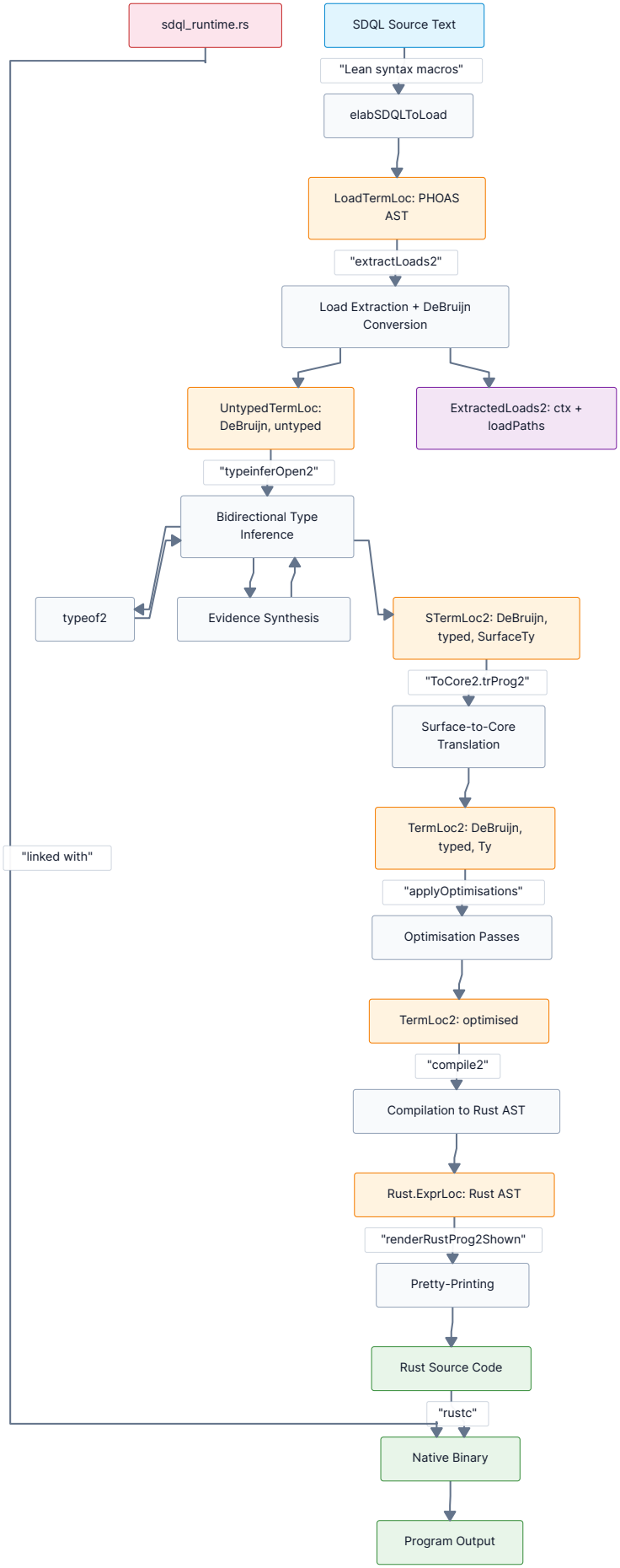


Figure 2: Compilation Pipeline

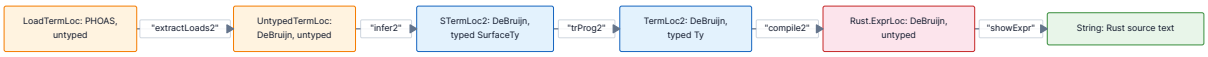


Figure 3: Intermediate representations

```
declare_syntax_cat sdql

syntax "sum" "(" "<" sdqlident "," sdqlident ">" "in" sdql ")" sdql : sdql
syntax "let" sdqlident "=" sdql "in" sdql : sdql

def elabSDQLToLoad (stx : TSyntax `sdql) : MacroM (TSyntax `term) :=
    match stx with
    | `(sdql| let $x:sdqlident = $e:sdql in $b:sdql) => do
        let ee ← elabSDQLToLoad e
        let bb ← elabSDQLToLoad b
        `(LoadTerm'.letin $ee (fun $xIdent => $bb))
    | `(sdql| sum(<$k:sdqlident, $v:sdqlident> in $d:sdql) $body:sdql) => do
        let dd ← elabSDQLToLoad d
        let bb ← elabSDQLToLoad body
        `(LoadTerm'.sum $dd (fun $kIdent => fun $vIdent => $bb))
```

then we can extend Lean with custom syntax.

Moreover, we can actually run arbitrary code at elaboration time, including throwing an elaboration error. This feature was used to implement local error messages.

3.3.3 Intermediate Representations

The compilation pipeline works in the following stages:

1. Syntax macro elaboration
2. Untyped syntax with `load(...)`
3. Untyped syntax withouts `load(...)`
4. Typed named record syntax
5. Typed unnamed tuple syntax
6. Abstract Rust AST
7. Concrete Rust String

I chose to have five intermediate representations (as opposed to 3 or 4) to keep the individual transformation functions small and modular. For example, going from IR 1 to IR 2 only involves removing loads, which means that each individual function can be simple and be responsible for only one task. The most involved transformation is going from IR 2 to IR 3, which involves typechecking and synthesizing evidence.

Each of these stages will be described in detail in the following sections. The stages are depicted in Figure 3.

3.3.3.1 Load Syntax

The syntax macros produce a simple, untyped AST very close to the user's input. This initial AST contains `Load[T]("path")` as a syntax node. This representation uses PHOAS.

I made this design decision so that the Lean frontend would handle issues of variable shadowing. PHOAS can be easily be converted to DeBruijn indexing.

At first, I attempted to use PHOAS for type inference. PHOAS is mathematically elegant, but opaque to inspect under the binder, so I switched to DeBruijn indexing. For the difference between PHOAS and DeBruijn indexing, see Table 6.

3.3.3.2 Untyped Syntax

The `loads` are all replaced with free variables. If a path is loaded twice, it will be replaced by the same free variable in each case. This step also produces a mapping `FV -> Path` which is used at Rust code-gen time.

The reason we replace `load(...)` with a free variable at this stage is it makes code-generation easier. If the initial code has

```
load(path1) + load(path2)
```

we can generate the following Rust:

```
let fv0 = crate::load_tbl_dicts!(path1;...);
let fv1 = crate::load_tbl_dicts!(path2;...);
fv0 + fv1;
```

PHOAS is transformed into an untyped DeBruijn representation.

3.3.3.3 Typed Named Syntax

We infer the types using a "bidirectional algorithm". Specifically, this algorithm is a variant of co-contextual bidirectional typechecking (see [21] for a reference on Bidirectional Typechecking).

In practice, this algorithm is implemented as a pair of functions:

```
def infer2 (ctx : List SurfaceTy) (expectedTy : SurfaceTy) (e : UntypedTermLoc
ctx.length) : Except TypeError (STermLoc2 ctx expectedTy)

def typeof2 (ctx : List SurfaceTy) (e : UntypedTermLoc ctx.length) : Except TypeError
SurfaceTy
```

Where `infer` checks if a term has a given type and `typeof` tries to find the type of an expression.

In this stage, we synthesize evidence. The following pieces of evidence are synthesized:

1. Add
2. Scale
3. Mul
4. Closure
5. hasField

These pieces of evidence are used at Rust code-gen time. If the synthesizer cannot synthesize any needed piece of evidence, it throws a type-error with local location information. For example, if the p rogrammer erroneously attempts to write the program `closure(<name="bob", age=10>)`, the elaborator will throw the following error at elaboration time, when it fails to synthesize the necessary evidence `hasClosure`: "Type Error: could not synthesize evidence that the type `<name: string, age: int>` is a Kleene Algebra".

The evidence which the typechecker synthesizes acts like a typeclass in Haskell, or a Trait in Rust. Since the runtime behavior of `... + ...` and `closure(...)` is type-polymorphic, the type-synthesizer remembers which instance is being used with evidence.

For example, at the time of code-generation, we split code-generation on what specific piece of evidence was synthesized:

```
def compileAdd (a : AddM ty) (lhs rhs : Rust.ExprLoc) : Rust.Expr :=
  match a with
  | .boolA => .binop .bitOr lhs rhs
  | .realA => .binop .add lhs rhs
  | .minPlusA => .callRuntimeFn .minPlusAdd [lhs, rhs]
  | .intA => .binop .add lhs rhs
  | .stringA => .binop .add lhs rhs
```

For example, I could've used Rust `structs` to represent SDQL records, but this would've required a custom runtime for each compilation. I wanted to have a single, static runtime for each compilation, so I compiled SDQL records into nameless Rust `tuples`. This decision necessitated that the code-generator remember

some kind of evidence about which name corresponds to which field, which I stored in the `hasField` evidence.

It was in this stage that I encountered two architectural hurdles that I overcame:
1. Use of PHOAS
2. Typing of Records

Initially, I used PHOAS for the core representation. This representation was clean and elegant, but I could not figure out how to elegantly do type-inference under a PHOAS representation. The problem is that the structure under a PHOAS binder is opaque, meaning one would have to feed the binder something to inspect it.

I also ran into difficulties with records. In Lean,

```
Term [] .record [("age", .int), ("name", .string)]
!= Term [] .record [("name", .string), ("age", .int)]
```

These terms are not definitionally equal. The problem does not go away if you use `FinSet`, because

```
Term [] .record {("age", .int), ("name", .string)}
!= Term [] .record {("name", .string), ("age", .int)}
```

(not definitionally equal).

I overcame this issue by sorting the fields in all record types alphabetically before typechecking. This canonicalisation presents a further subtlety, because SDQL loads input files with `let orders = load[<o_orderkey: {int -> int}, size: int>]("datasets/tpch/orders.tbl")`, so sorting these fields would actually change the semantics of the `load` builtin. I resolved this bug by having having a special case for `load`, such that `load` does not sort its fields. The `load` builtin loads a database from a `.tbl` file, where the order of the columns matters. At the boundary between how records are used in `load` (as-is order), and in the rest of the program (alphabetical sorting), we bridge the gap with the evidence `hasField`.

3.3.3.4 Typed Unnamed Syntax

The compiler forgets the names of records, just referring to them by positions.

This erasure has a subtlety that we can't re-order `Load[<t1, t2>](...)`, because this canonicalisation would change the semantics.

3.3.3.5 Abstract Rust Syntax

I created an AST to represent a simplified subset of Rust. We use this AST to code-generate to Rust.

I made the following design decisions in the code-generated Rust:
1. Represent `{K -> V}` as `HashMap<K,V>`
2. Represent each piece of evidence as a trait for clean code generation.

I chose the `HashMap` representation because, in profiling, dictionary lookup is the most common operation, and `HashMap` offers an $O(1)$ lookup (as opposed to a $O(\log n)$ lookup for `BTreeMap`).

Difficulties of Code-generating to Rust:
1. `float` isn't ordered in Rust. This lack of an ordering is due to IEEE 754 [22] ($\text{NaN} \neq \text{NaN}$). I overcame this difficult by writing a custom `OrderedFloat` type.
2. I used an inefficient workaround `x.clone()` to avoid the borrow checker, but this fix resulted in slower performance.

Builtin functions were implemented in `sdql_runtime.rs`, a Rust runtime. I created Rust macros to generate instances for tuples of length 1 - 8. Technically, this fix is incorrect, as I should have dynamically

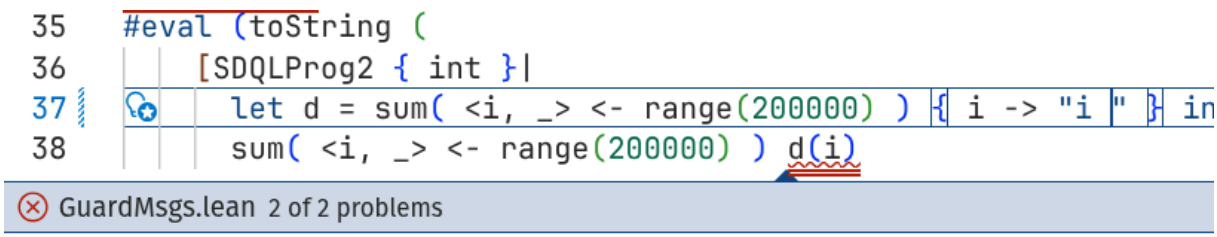


Figure 4: The syntax macros of lean-sdql cause errors to be thrown at editor-elaboration time, not at compile-time.

created the instances based on what lengths could occur in the program, but I found it difficult to generate a new runtime for each compilation, so that's a tradeoff.

The reference implementation allows for type-hints such as `@vec` which hint to the compiler to compile this dictionary to a vector instead of a hashtable. My implementation ignores these type hints, due to the simplicity of using a single representation. The choice of data-structure selection does have an effect on runtime performance, and I evaluate this tradeoff in Section 4.6.

3.3.4 Elaboration-Time vs Runtime Split

The elaboration-time vs runtime split is described in Figure 5.

An interesting design decision to point out is that type-checking and parsing happen at Lean elaboration time. Thus, the editor itself throws an error if there is a type-error in the SDQL expression, leading to an ergonomic developer experience. This embedding is a genuine ergonomics win. Observe that in Figure 4, the error is thrown, not at compile-time, but at editor-elaboration time.

3.3.5 The Use of `Unsafe` Lean

One atypical design decision that I should flag is that, because it was never a project goal to prove the correctness of my compiler in Lean, I used the `unsafe` keyword (which is very rarely used in Lean) to bypass Lean's termination checker.

When reasoning about complex folds over a giant AST, it's not always feasible to manually prove the termination of a program. In algebraic rewrites, the set of rewrites may not have a fixpoint (depending on what equational rewrites are used). Thus, I pragmatically used the `unsafe` keyword to bypass termination checking. Note that `unsafe` Lean still has typechecking. Moreover, I intentionally chose not to invest in termination proofs because they weren't part of my project goals. This concession does not affect claims of type-safety, as Lean still does type-checking in `unsafe` mode. I only used `unsafe` to avoid termination proofs, and I did not use `unsafeCast`, nor any operations that could lead to a runtime crash. Note that `unsafe` refers to the unsoundess of proofs (you can prove `False`), but not to the soundness of code-generation (which is still guaranteed not to crash at runtime). In practice, I did not observe any instances of non-termination, so this was not an issue.

3.4 Graph Extension

One of my extensions was to use Kleene's Algorithm to implement graph algorithms in SDQL – essentially turning SDQL into a graph database.

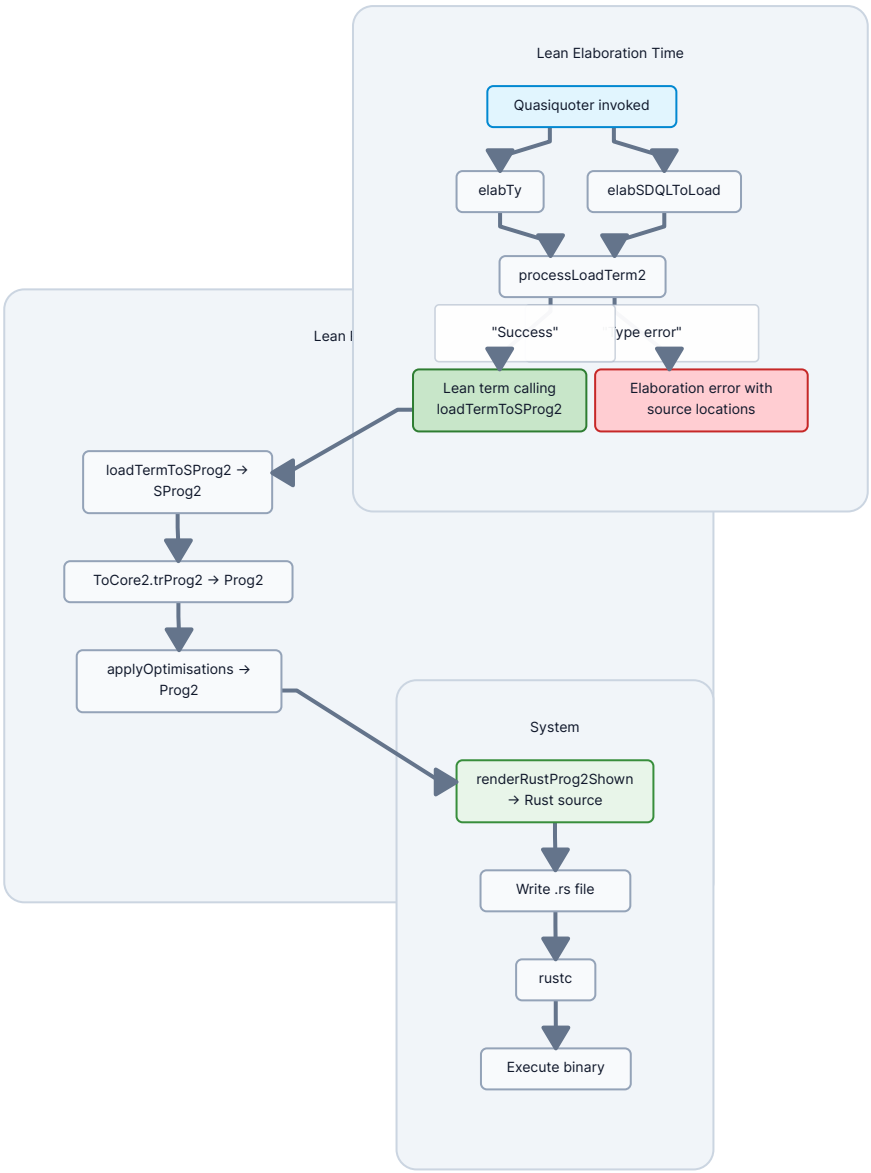


Figure 5: Elaboration vs runtime split

Graph Problem	Traditional Algorithm	Semi-Ring
all-pairs shortest path	Floyd-Warshall	$(\mathbb{N} \cup \infty, \min, +)$
reachability	Transitive Closure	$(\mathbb{B}, \wedge, \vee)$
Most Probable Path	Viterbi Algorithm	$([0,1], \max, *)$

Table 7: Examples of how Kleene's algorithm transforms any semi-ring into a corresponding graph problem

Tarjan [2] explores how different graph problems can be viewed as running Kleene's Algorithm over various semi-rings. There is an explicit bijection between semirings and certain classes of graph problems Table 7.

In general, all square vector spaces over a starred semi-ring form a starred semi-ring. For this project I only implemented Kleene's Algorithm over the type $\{K \to \{K \to V\}\}$ for V being a Kleene Algebra. (It would be feasible to implement Kleene's Algorithm over any $V \otimes V$, but I am only exploring the case of a graph for now).

This work extends the definition of the `closure(e)` operator to be defined on more types. The `closure(e)` operator was defined [1], but not implemented in the reference implementation.

3.5 Theory of Semi-Rings in Square Vector Spaces

For any vector space V, I will call the vector space $V \otimes V$ square.

Let V be a semi-module over k. Let $B : V \times V \to K$ be a bilinear form.

$V \otimes V$ is a semi-ring, where multiplication is given by

$$(a \otimes b) * (x \otimes y) = B(b, x) \cdot (a \otimes y).$$

Let $u = a \otimes b$ be an element of $V \otimes V$. The Kleene Star is given by:

$$(a \otimes b)^* = 1 + B(b, a)^* \cdot (a \otimes b)$$

This formula may be derived from $(a \otimes b)^n = B(b, a)^{n-1}(a \otimes b)$, because

$$(a \otimes b)^* = 1 + (a \otimes b) + (a \otimes b)^2 + ... = 1 + (a \otimes b) + B(b, a)(a \otimes b) + ... = 1 + B(b, a)^* \cdot (a \otimes b)$$

Note that all semi-modules in SDQL have a bilinear form. When V is finite-dimensional and B is non-degenerate, the semi-ring $(V \otimes V, +, *)$ is isomorphic to the matrix algebra $\mathrm{End}(V) \cong M_{n(k)}$.

This theory is used to implement semi-ring multiplication `*s : T -> T -> T` .

```
# pseudo code for *s : T -> T -> T
def semiring_multiply(a : V ⊗ V, b : V ⊗ V) -> V ⊗ V:
    acc = 0 : T
    for (al, ar) in decompose(a):
        for (bl, br) in decompose(b):
            acc = acc + <ar, bl> * (al ⊗ br)
    return acc
```

Note that under the specific example of multiplying an $n \times n$ matrix, this algorithm be exactly the same as the standard matrix-multiplication algorithm, and would use $O(n^3)$ multiplications.

However, the benefit of using this approach is that it doesn't incur a significant performance overhead, but it can also generically extend to any "square" semi-module in SDQL, including (for example):

1. `<a : int, b : int> ⊗ <a : int, b : int> = <a : <a : int, b : int>, b : <a : int, b : int>>`
2. `{int -> real} ⊗ {int -> real} = {int -> {int  -> real}}`

To implement this generalisation of multiplication requires an algorithm for taking the "square root" of a vector space.

```
def stensorRoot (T : SurfaceTy) : Option SurfaceTy :=  peelForStensorRoot T T

def peelForStensorRoot (T : SurfaceTy) (candidate : SurfaceTy) : Option SurfaceTy :=
  if candidate ⊗ candidate = T then
    some candidate
  else
    match candidate with
    | .dict _ V => peelForStensorRoot T V
    | .record σ =>
      match σ with
      | (_, v) :: _ => peelForStensorRoot T v
      | [] => none
    | _ => none
```

If the vector space V it not a square vector space, this algorithm returns `.none`.

Note that this theory is known in the literature, but is often presented $V \otimes V^*$ directly, with the evaluation pairing $\text{ev} : V^* \otimes V \to k$ [23].

3.6 Algebraic Optimisations

The code is optimised by optimisations as described in Figure 6.

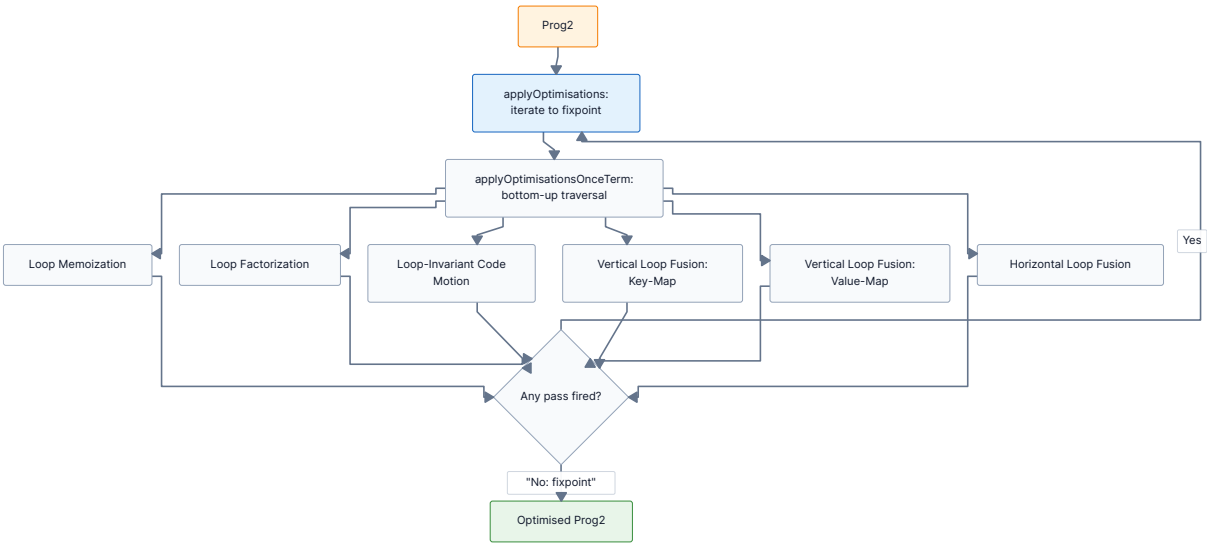


Figure 6: Optimisation Passes

The original SDQL paper mentions algebraic optimisations. I implemented these and implemented a test suite to test them. The optimisations are in Table 8. The optimisations in Table 8 are derived from the optimisations mentioned in the original SDQL paper. These optimisations all preserve the semantics of the original expression.

I applied these optimisations using a combinator/strategy architecture. I implemented functional patterns such as `OrElse` and `Inhabited`. This was inspired by the strategy pattern, as was demonstrated by Stratego [24].

For example,

```
partial def fixpointTerm (o : Optimisation) : Optimisation :=
    fun t => do
        let t' ← o t
        fixpointTerm o t' <|>  t'

def seqAnySubexpression : List Optimisation → Optimisation :=
    (·.foldr (β := Optimisation) (orElse ∘ anySubexpression) fail)

def optimiseUntilStable: List Optimisation →  Optimisation :=
  fixpoint ∘ seqAnySubexpression
```

Optimisation	Before	After
Vertical Loop Fusion 1	`let y = sum(<x, x_v> in e1) {f1(x) -> x_v} in sum(<x,x_v> in y) {f2(x) -> x_v}`	`sum(<x,x_v> in e1) { f2(f1(x)) -> x_v }`
Vertical Loop Fusion 2	`let y=sum(<x,x_v> in e1){x->f1(x_v)} in sum(<x,x_v> in y){x->f2(x_v)}`	`sum(<x,x_v> in e1) { x -> f2(f1(x_v)) }`
Horizontal Loop Fusion	`let y1=sum(x in e1) f1(x) in let y2=sum(x in e1) f2(x) in f3(y1, y2)`	`let tmp = sum(x in e1) <y1 = f1(x), y2 = f2(x)> in f3(tmp.y1, tmp.y2)`
Loop Factorization 1	`sum(x in e1) (e2 * f(x))`	`e2 * (sum(x in e1) f(x))`
Loop Factorization 2	`sum(x in e1) (f(x) * e2)`	`(sum(x in e1) f(x)) * e2`
Loop-Invariant Code Motion	`sum(x in e1) let y = e2 in f(x, y)`	`let y = e2 in sum(x in e1) f(x, y)`
Loop Memoization 1	`sum(x in e1) if(p(x) == e2) then g(x, e3)`	`let tmp = sum(<k,v> in e1) {p(x) -> {k -> v}} in sum(x in tmp(e2)) g(x, e3)`
Loop Memoization 2	`sum(x in e1) if(p(x) == e2) then f(x)`	`let tmp = sum(x in e1) {p(x) -> f(x)} in tmp(e2)`

Table 8: Table of optimisations in SDQL

In implementing the algebraic optimisations, I encountered a subtlety in my context-indexed representation: the algebraic optimisations change the context. For example: in the loop factorization `sum(<k,v> in d) (e2 * f(k,v)) ↦ e2 * sum(<k,v> in d) f(k,v)`, the term `e2` has `k,v` in the context on the left, but not on the right. So, it's not as simple as "simply re-arranging the terms". This is one place that indexing by context actually prevents a bug: the optimisation is correct iff `e2` does not depend on `k` or `v`. To address this, I created a utility

```
def extractInvariant2 : TermLoc2 (a :: b :: ctx) ty → Option (TermLoc2 ctx ty)
```

3.7 Implementation of Local Error Messages

I implemented local error messages using mutually-recursive inductive datatypes, demonstrating a concrete win of using Lean4.

```
  /-- A DeBruijn core term with source location -/
inductive TermLoc2 : (ctx : List Ty) → Ty → Type where
    | mk : {ctx : List Ty} → {ty : Ty} → (loc : SourceLocation) → Term2 ctx ty →
TermLoc2 ctx ty

inductive Term2 : (ctx : List Ty) → Ty → Type where
...
```

With the source location, the elaborator can throw an error at elaboration time, leading to a responsive developer feedback loop.

3.8 Repository Overview

```
docs
├── activeContext.md
├── grammar.md
├── optimisations.md
├── pipeline.md
├── productContext.md
├── progress.md
├── projectbrief.md
├── semi-ring.md
├── systemPatterns.md
└── techContext.md
Flamegraph.lean
GraphPerformance.lean
OptimisationPerformanceComparison.lean
PartIiProject
├── Bench
│   ├── Common.lean
│   └── Table.lean
├── CodegenRust.lean
├── Dict.lean
├── HList.lean
├── Mem.lean
├── Optimisations
│   ├── Apply.lean
│   ├── HorizontalLoopFusion.lean
│   ├── LoopFactorization.lean
│   ├── LoopInvariantCodeMotion.lean
│   ├── LoopMemoization.lean
│   ├── Term2Utils.lean
│   └── VerticalLoopFusion.lean
├── Optimisations.lean
├── Rust.lean
├── SurfaceCore.lean
├── SurfaceCore2.lean
├── SyntaxSDQL.lean
├── SyntaxSDQLProg.lean
├── Term.lean
├── Term2.lean
├── Untyped
│   ├── Errors.lean
│   ├── Evidence.lean
│   ├── ExtractLoads.lean
│   ├── Infer.lean
│   ├── LoadTerm.lean
│   ├── Pipeline.lean
│   ├── Tests.lean
│   ├── TypeOf.lean
│   ├── TypeUtil.lean
│   └── UntypedTerm.lean
└── untyped.lean
PartIiProject.lean
Performance.lean
scripts
├── compare_sdql_output.py
├── import_tpch_sdql.py
├── transitive_closure.py
└── viterbi.py
sdql_runtime.rs
test.lean
Tests
├── Cases.lean
├── GuardMsgs.lean
├── Main.lean
├── Optimisations
│   ├── HorizontalLoopFusion.lean
│   ├── LoopFactorization.lean
│   ├── LoopInvariantCodeMotion.lean
│   ├── LoopMemoization.lean
│   └── VerticalLoopFusion.lean
└── TPCH
    ├── Q01.lean
    ├── Q02.lean
    ├── Q03.lean
    ├── Q04.lean
    ├── Q05.lean
    ├── Q06.lean
    ├── Q07.lean
    ├── Q08.lean
    ├── Q09.lean
    ├── Q10.lean
    ├── Q11.lean
    ├── Q12.lean
    ├── Q13.lean
    ├── Q14.lean
    ├── Q15.lean
    ├── Q16.lean
    ├── Q17.lean
    ├── Q18.lean
    ├── Q19.lean
    ├── Q20.lean
    ├── Q21.lean
    └── Q22.lean
```

,

,

Listing 1: project tree

Figure 7: Key Source Files

The key source-files are described in Figure 7. See Listing 1 for the code structure of the repository. The tree for `tpch-dbgen` and the `sdql` reference implementation are omitted from Listing 1, because those are pre-existing tools that I relied upon but did not write. All other code was written by myself.

I'll discuss the files in chronological order of a compilation process, and then I'll discuss the benchmarking/testing scripts.

3.8.1 Compilation Files

1. `SyntaxSDQL.lean`: defines a syntax of SDQL expressions
2. `SyntaxSDQLProg.lean`: defines a syntax of SDQL programs
3. `Untyped/[...].lean`: defines the untyped syntax, and transformations from the untyped syntax to the typed syntax
4. `SurfaceCore2.lean`: Defines a transformation from the named syntax to the unnamed syntax
5. `Term2.lean`: Defines the unnamed core syntax
6. `Optimisations/[...].lean`: defines optimisations over the core syntax
7. `CodegenRust.lean`: Defines the transformation from the core syntax to the Rust syntax
8. `Rust.lean`: Defines a simplified subset of Rust syntax.

3.8.2 Documentation Files

1. `docs/[...].md`: I was sure to keep accurate and detailed documentation as I went along, so that I would have an easier time writing my dissertation, and so that I would not forget what I did. This practice demonstrates professional software practice.

3.8.3 Testing Files

1. `Flamegraph.lean`: generates flamegraphs using the `perf` profiler
2. `GraphPerformance.lean`: compares the performance of lean-sdql against a handwritten Python implementation of graph algorithms
3. `Tests/[...].lean` Tests correctness of compilation by, including on the TPC-H benchmark.

3.9 Runtime Tuning

I went through several rounds of iterative runtime optimisations, guided by quantitative benchmarking (see Table 9).

case	sdql-rs	lean-gen	lean/sdql
micro_sum_range_add	1ms	2ms	2.000×
micro_sum_range_dict_build	20ms	74ms	3.700×
micro_sum_range_lookup	7ms	8ms	1.142×
tpch_q01	18ms	4ms	0.222×
tpch_q02	17ms	4ms	0.235×
tpch_q03	17ms	3ms	0.176×
tpch_q04	15ms	3ms	0.200×
tpch_q05	15ms	3ms	0.200×
tpch_q06	16ms	2ms	0.125×
tpch_q07	19ms	3ms	0.157×
tpch_q09	25ms	1ms	0.040×
tpch_q10	16ms	3ms	0.187×
tpch_q11	16ms	3ms	0.187×
tpch_q12	4ms	2ms	0.500×
tpch_q13	16ms	3ms	0.187×
tpch_q14	15ms	3ms	0.200×
tpch_q16	20ms	3ms	0.150×
tpch_q17	19ms	4ms	0.210×
tpch_q18	15ms	3ms	0.200×
tpch_q19	13ms	4ms	0.307×
tpch_q20	4ms	3ms	0.750×
tpch_q21	616ms	2ms	0.003×
tpch_q22	20ms	3ms	0.150×
tpch_q01_sf001	166ms	501ms	3.018×
tpch_q02_sf001	44ms	28ms	0.636×
tpch_q03_sf001	201ms	574ms	2.855×
tpch_q04_sf001	155ms	616ms	3.974×
tpch_q05_sf001	172ms	587ms	3.412×
tpch_q06_sf001	137ms	394ms	2.875×
tpch_q07_sf001	170ms	774ms	4.552×
tpch_q09_sf001	192ms	810ms	4.218×
tpch_q10_sf001	171ms	890ms	5.204×
tpch_q11_sf001	18ms	46ms	2.555×
tpch_q12_sf001	183ms	765ms	4.180×
tpch_q13_sf001	39ms	108ms	2.769×
tpch_q14_sf001	159ms	529ms	3.327×
tpch_q15_sf001	171ms	513ms	3.000×
tpch_q16_sf001	21ms	37ms	1.761×
tpch_q17_sf001	151ms	480ms	3.178×
tpch_q18_sf001	171ms	594ms	3.473×
tpch_q19_sf001	144ms	393ms	2.729×
tpch_q20_sf001	157ms	606ms	3.859×
tpch_q21_sf001	800ms	126ms	0.157×
tpch_q22_sf001	47ms	91ms	1.936×
TOTAL	4413ms	9605ms	2.176×

Table 9: performance comparison

I experimented with the following changes:

1. switch `BtreeMap` implementation to using a `HashMap` implementation (for better cache locality)
2. switch hashing function from `SipHash` to `FxHash` (to prioritize speed over cryptographic security.)

These performance experiments paid off, and the final version after tuning was 2.3x faster than the initial prototype.

Chapter 4: Evaluation

My original aims are in Table 10.

Original Goal of Project	Did Accomplish	Section Evidencing Completion
compiler has correct semantics	✓	Section 4.3
compiler has optimisations that improve performance and are correct	✓	Section 4.4
compiler has graph database extensions that are correct and performant	✓	Section 4.9
lean-sdql has competitive performance to reference implementation	✓	Section 4.5

Table 10: Original Aims vs Evaluations

We may answer our original research questions in Table 11,

Original Research Question	Empirical Answer
Can a dependently-typed language like Lean4 yield a cleaner compiler architecture for a language with rich algebraic structure?	In terms of qualitative developer experience, yes, but in terms of lines-of-code, no
Can language features of dependent types and hygienic syntax macros improve the ergonomics and extensibility of a compiler architecture?	Yes
Can SDQL be cleanly extended with semiring and closure properties, without diluting the original language?	Yes
Can closure properties of SDQL be applied performantly to graph problems, and be competitive against specialised graph databases?	Yes

Table 11: Answers to the Original Research Questions

4.1 Experimental Setup

All experiments were done on an HP Pavilion Laptop 15 with an i5 CPU and 12GiB of memory. See Listing 5 for the specs of the machine that ran the experiments.

Experiments were done with

- Python 3.13.12
- rustc 1.94.0-nightly (7ecabfaaf 2026-01-03)
- elan 3.1.1

4.2 Effect of Dependent Types on Compiler Design

It is difficult to empirically compare the effect of choosing Scala vs Lean to write a compiler. I also acknowledge that this benchmark isn't an apples-to-apples comparison, as `lean-sdql` doesn't implement the same set of features as the reference implementation.

Nevertheless, I broke down a compiler into its basic parts (parser, typechecker, etc), and compared the number of lines of source-code in both using the `cloc` utility.

Counts use `cloc` code lines only. The reference counts exclude `sdql/backend/Interpreter.scala` and `sdql/storage/[...]`, since they are execution/runtime support rather than compiler stages. The Lean counts exclude `PartIiProject/Bench/[...]` and `PartIiProject/Untyped/Tests.lean`.

`PartIiProject/Term2.lean` is split at `namespace ToCore2`, so the core IR and the surface-to-core lowering pass are counted separately.

Stage	Reference (Scala LOC)	Lean LOC	Lean / Reference
Parsing / source frontend	261	478	1.83x
IR / semantic core	408	1287	3.15x
Load extraction / frontend normalisation	0	173	n/a
Type inference / checking	179	734	4.10x
Lowering / rewrites / optimisations	159	882	5.55x
Backend IR / code generation	459	787	1.71x
Pipeline glue / orchestration	57	55	0.96x
Total included compiler LOC	1523	4396	2.89x

We observe that the lean-sdql implementation is longer that the reference implementation. This is due to:
1. lean-sdql genuinely implements more features than the reference implementation (including algebraic rewrites, semiring operations, and Kleene-star closure).
2. Lean-sdql uses more intermediate representations (5 vs 3).

Even so, this comparison presents a genuine tradeoff: dependent types reveal more bugs at compile time, yet they impose a burden on the programmer of creating the correct abstractions and satisfying the typechecker.

Another tradeoff is the use of DeBruijn indexing. DeBruijn indexing ensures a proof-by-construction that all terms are well-scoped. However, using DeBruijn indexing necessitates "context plumbing" of carrying around the typing environment. One example of "context plumbing" was the earlier-mentioned `extractInvariant2 : TermLoc2 (a :: b :: ctx) ty → Option (TermLoc2 ctx ty)`.

Using dependent types is actually a spectrum , as shown in Table 12. `lean-sdql` lands in the third row, which provides qualitative benefits over the standard approach (second row).

At the level I was using dependent types, Lean caught the following errors:
1. Changing the context of a variable
2. Expressions referring to variables not in the context
3. Optimisations changing the type of expressions

Typing Discipline	Errors eliminated
`unsafeCast`/Lean as Python	Any type-errors could happen
simply typed Lean/Lean as OCaml	Runtime type errors are prevented (standard approach to compilers)
Typed AST	All programs are valid, all optimisations preserve program types + contexts (this is where `lean-sdql` lands)
Fully verified program/CompCert	All errors are eliminated

Table 12: Using dependent types is a spectrum

4.3 Correctness of Compilation Test-bench (TPC-H Benchmark)

The reference implementation of SDQL implements the TPC-H benchmark (see [25]).

I implemented a harness to run both the reference implementation and the Lean4 implementation on the TPC-H benchmark at scale factor $SF = 0.05$.

In this benchmark are also micro-tests of simple language constructs to allow for fast iteration and prototyping.

The output of the differential test suite demonstrates that all tests pass. The full output is available in the appendix: Listing 6. The verification of the entire testsuite provides strong evidence that lean-sdql has correct compilation semantics.

The output shows that the lean4 implementation is approximately 3-7x slower than the reference implementation at $SF = 0.05$.

The test-bench gives us reasonable confidence in the correctness of `lean-sdql`, because it covers all the relational features of `sdql`. I also have graph-related tests in the testsuite to make sure we have 100% coverage. The TPC-H benchmark doesn't test `closure(e)` nor `*s`, so these were tested by unit tests (and by the graph test-bench).

It was a conscious design decision to choose $SF = 0.05$ because larger scale-factors would lead to crashes on the testing hardware.

Even though compile times and memory usage were explicitly not requirements of my core, I still measured them on the TPC-H benchmark to make sure they were tolerable/practicable for the end-user.

- Mean Lean-SDQL codegen: 152.7 ms
- Mean rustc: 30.652 s
- Mean end-to-end compile phase: 30.826 s
- Mean Lean peak RSS/HWM: 93,171 KiB (91.0 MiB)
- Mean largest rustc max RSS: 199,601 KiB (194.9 MiB)
- Largest observed combined parent Lean RSS + child rustc RSS: about 291,684 KiB (284.8 MiB)

4.4 Optimisation Performance tests

I wrote a script to benchmark pre-optimised code against post-optimised code in terms of runtime performance.

Benchmark	Unoptimised (95% CI)	Optimised (95% CI)	Opt / unopt	p-value
vertical_loop_fusion_key_map	51.40 ± 1.35 ms	45.90 ± 2.10 ms	0.893x	8.00e-05
vertical_loop_fusion_value_map	49.90 ± 0.80 ms	38.75 ± 1.45 ms	0.777x	1.00e-05
horizontal_loop_fusion	8.15 ± 0.35 ms	8.05 ± 0.45 ms	0.988x	0.4311
loop_factorization_left	7.40 ± 0.50 ms	7.50 ± 0.50 ms	1.014x	0.6543
loop_factorization_right	7.75 ± 0.40 ms	8.00 ± 0.35 ms	1.032x	0.8569
loop_invariant_code_motion	7.85 ± 0.40 ms	7.75 ± 0.50 ms	0.987x	0.4395
loop_memoization_lookup	85.60 ± 1.95 ms	8.25 ± 0.50 ms	0.096x	1.00e-05
loop_memoization_partition	93.25 ± 3.00 ms	15.65 ± 1.05 ms	0.168x	1.00e-05
tpch_q02_memo	32.70 ± 1.25 ms	31.95 ± 1.50 ms	0.977x	0.2434
tpch_q03_memo	599.60 ± 5.10 ms	604.45 ± 5.35 ms	1.008x	0.8918
tpch_q05_memo	607.75 ± 7.65 ms	608.45 ± 5.45 ms	1.001x	0.5627
tpch_q11_memo	41.80 ± 1.25 ms	41.95 ± 0.95 ms	1.004x	0.5924
tpch_q15_memo	409.75 ± 5.90 ms	400.45 ± 4.40 ms	0.977x	0.0086
tpch_q20_memo	561.45 ± 5.10 ms	574.30 ± 18.10 ms	1.023x	0.9433
tpch_q21_memo	115.15 ± 3.70 ms	140.50 ± 1.65 ms	1.220x	1.0000

Table 13: Optimisation performance comparison. Runtimes are mean wall-clock milliseconds with 95% bootstrap confidence intervals. p-values use a one-sided two-sample permutation test for the optimised runtime being lower than the unoptimised runtime.

Benchmark	Unoptimised (95% CI)	Optimised (95% CI)	Opt / unopt	p-value
vertical_loop_fusion_key_map	419.60 ± 4.70 ms	281.80 ± 2.80 ms	0.672x	5.41e-06
vertical_loop_fusion_value_map	454.00 ± 6.00 ms	165.30 ± 4.50 ms	0.364x	5.41e-06
horizontal_loop_fusion	53.00 ± 2.30 ms	52.00 ± 4.80 ms	0.981x	0.3732
loop_factorization_left	42.90 ± 0.60 ms	42.90 ± 1.60 ms	1.000x	0.5445
loop_factorization_right	43.30 ± 1.60 ms	43.30 ± 2.00 ms	1.000x	0.5294
loop_invariant_code_motion	44.20 ± 1.50 ms	43.90 ± 2.50 ms	0.993x	0.4485
loop_memoization_lookup	1212.10 ± 14.20 ms	80.00 ± 2.80 ms	0.066x	5.41e-06
loop_memoization_partition	1211.50 ± 14.00 ms	105.90 ± 4.30 ms	0.087x	5.41e-06
tpch_q02_memo	138.50 ± 5.40 ms	145.80 ± 6.50 ms	1.053x	0.9497
tpch_q03_memo	2063.60 ± 57.20 ms	1979.10 ± 47.70 ms	0.959x	0.0223
tpch_q05_memo	2120.60 ± 50.50 ms	2202.30 ± 48.60 ms	1.039x	0.9892
tpch_q11_memo	94.90 ± 3.00 ms	100.80 ± 3.60 ms	1.062x	0.9892
tpch_q15_memo	1516.40 ± 37.30 ms	1474.00 ± 9.70 ms	0.972x	0.0276
tpch_q20_memo	1867.00 ± 39.30 ms	1841.30 ± 8.80 ms	0.986x	0.0567
tpch_q21_memo	497.20 ± 3.00 ms	525.20 ± 5.60 ms	1.056x	1.0000

Table 14: Optimisation Performance Comparison (Rust Optimisations Manually Disabled)

Loop factorization produces no speedup, because the LLVM compiler that Rust uses already performs this optimisation. We can validate this explanation by manually disabling Rust optimisations in Table 14. Once

we manually disable Rust + LLVM optimisations, we see that Loop factorization still produces no speedup, because it only optimised one multiplication, whose cost is dominated by iterating over a HashMap.

We can compare the output of the pre-optimised vs the post-optimised code. See Figure 8. We observe that optimisation provides a statistically significant improvement in the runtime (at $\alpha = 0.05$ using a one-sided permutation test) for the following testcases:

1. vertical loop fusion key map ($p = 2.71 \times 10^{-5}$)
2. vertical loop fusion value map ($p = 5.41 \times 10^{-6}$)
3. loop memoization lookup ($p = 5.41 \times 10^{-6}$)
4. loop memoization partiton ($p = 5.41 \times 10^{-6}$)

Let me walk through `loop_memoization_lookup` (Table 8), because that case offers the largest speedup in the test-bench.

```
-- Memoization lookup + hoisting: without hoisting this is not a win; include code motion.
unsafe def p_loop_memoization_lookup : SProg2 :=
  [SDQLProg2 { int }|
    let memoN = 100000 in
    let memoM = 1000 in
    let d = sum(i <- range(memoN)) { i -> i } in
    sum(i <- range(memoM))
        sum(<k, v> <- d)
            if k == i then v
  ]
```

```
// unoptimized code:
    let mut x3 = 0;
    for x5 in 0..(x1) {
      x3 = (x3) + ({
        let mut x6 = 0;
        for (x8, x7) in (x2).iter()
          x6 = (x6) + (if ext_eq((x8, x5))
            x7
          else
            0
          );
        x6
      });
    }
    x3
  ;
  println!("{}", SDQLShow::show(&result));
```

```
// optimised code:
 let x3 = {
    let mut x3 = HashMap::<i64, i64>::default();
    for (x5, x4) in (x2).iter()
      dict_insert_add(&mut (x3), x5, x4);
    x3
  };
    let mut x4 = 0;
    for x6 in 0..(x1) {
      x4 = (x4) + (lookup_or_default(&x3, x6, 0));
    }
    x4
  println!("{}", SDQLShow::show(&result));
```

Listing 2: Comparison of pre-optimised vs post-optimised code

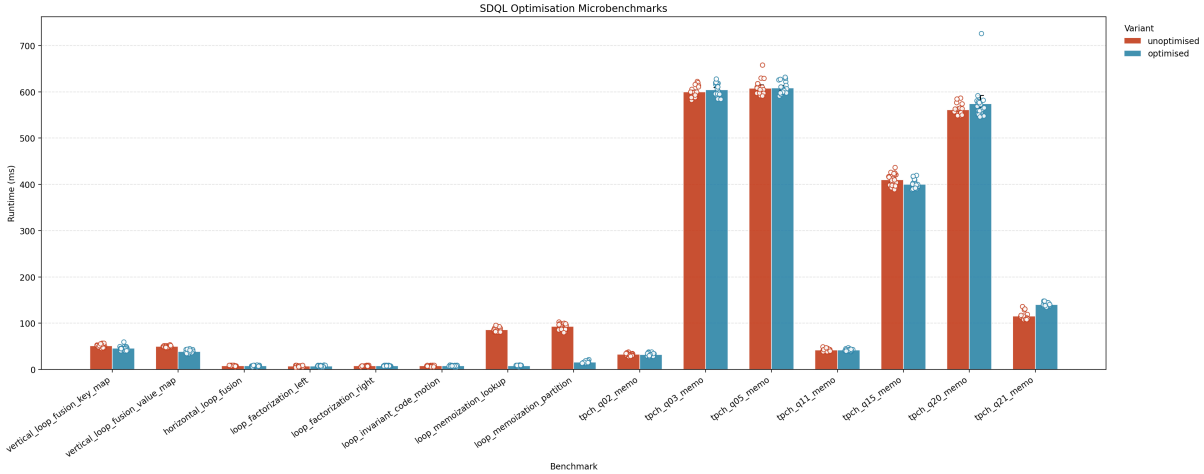


Figure 8: The Effect of Optimisation on Runtime Performance

On the TPC-H benchmark, the algebraic optimiser fires on the following testcases: 2,3,5,11,15,20,21.

On these cases, loop memoization does not lead to a significant speedup, because, in the rule

```
sum(x in e1) if(p(x) == e2) then f(x)
==> let tmp = sum(x in e1) {p(x) -> f(x)} in tmp(e2)
```

But when `e2` is a constant or a single scalar computed once, the dictionary is built at cost O(n) and probed exactly once. The break-even requires:

$$\text{cost}(\text{build partition}) + \text{cost}(\text{one lookup}) < \text{cost}(\text{one linear scan with filter})$$

This inequality almost never holds because:

1. Hashing every key during partition construction is at least as expensive as the equality comparisons during filtering
2. Allocating inner dictionaries for every distinct key adds memory pressure and cache misses
3. A single probe means zero amortisation of the build cost

It makes sense as to why other optimisations did not fire on the TPC-H benchmark, as the authors of the SDQL paper confirmed that they did the loop fusion by hand on the TPC-H benchmark.

4.5 Reference Performance Comparison

I compared the performance of Lean SDQL against the reference implementation of SDQL on a set benchmark.

See Table 9 for a table comparing the performance of `lean-sdql` vs the reference implementation across a varied benchmark, which includes TPCH.

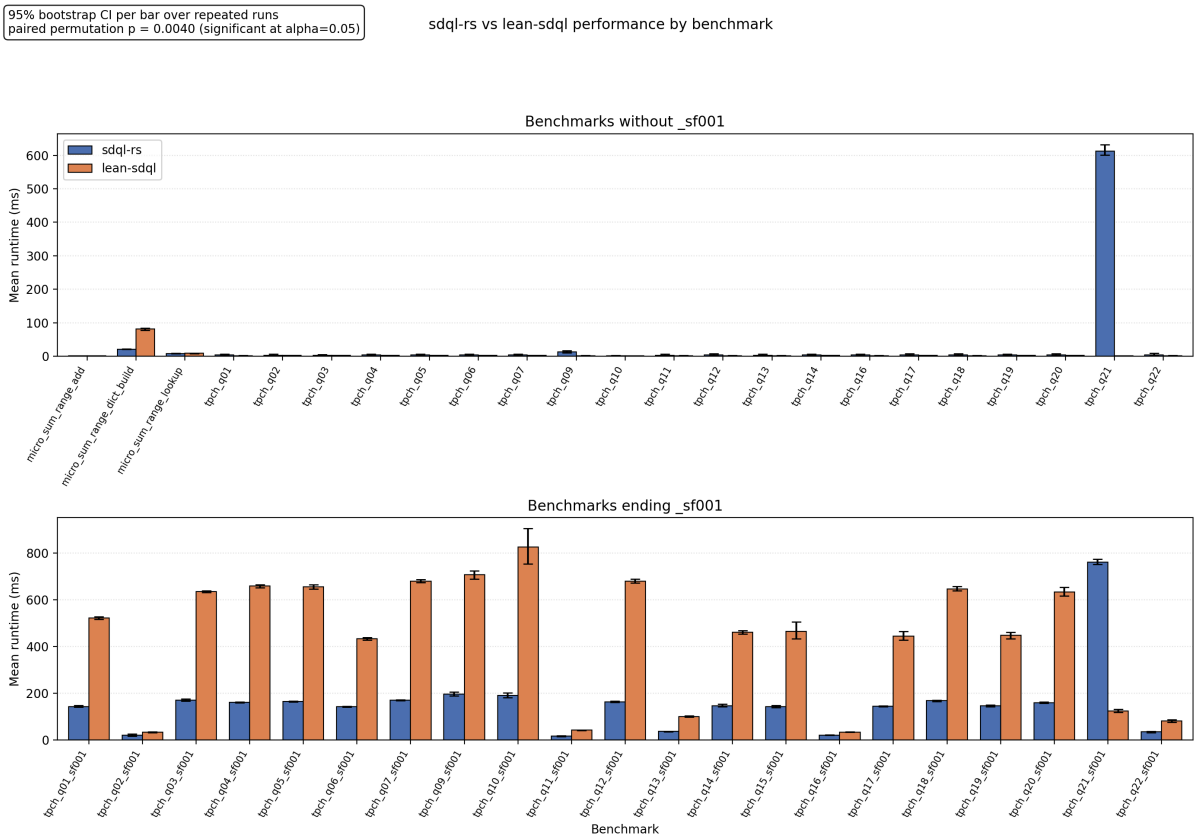


Figure 9: performance comparison

The barchart can be see in Figure 9. The barchart compares the performance of `lean-sdql` vs the reference implementation across a varied benchmark, which includes TPCH.

I did a paired permutation test, specifically, a sign-flip variant, with $N = 10$ runs and calculated $p = 0.0045$, which is significant at $\alpha = 0.05$. Thus, we can say that the reference implementation is statistically significantly faster than lean-sdql. A sign-flip test is appropriate to use in this scenario because we cannot assume that runs have a normal distribution.

My implementation achieves a 3x-5x slowdown on the TPC-H at $SF = 0.05$. Note that `lean-sdql` outperforms the reference implementation on the tiny database due to startups costs, but is slower on the larger database.

This performance can be analysed by the use of a profiler.

4.6 The Effect of Data-structure selection

My initial implementation used `BTreeMap`. I then refactored to used `HashMap`.

The reference implementation defaults to `HashMap`, but allows used-supplied compilation type-hints to compile to `VecDict`. I do not do this approach.

To empirically quantify the extent to which datastructure selection affects runtime performance, I designed a custom benchmark just to test datastructure selection.

The SDQL reference implementation allows for two user-supplied compiler hints/type annotations: `@vec` and `@smallvecdict`.

See Figure 10. We observe that `@vec` and `@smallvecdict` lead to a 1-10x speedup.

While doing this investigation, I uncovered a semantic incorrectness in the reference implementation. The reference implementation calculates the `size` builtin as the size of the underlying datastructure, whereas I calculate `size` as the number of unique keys.

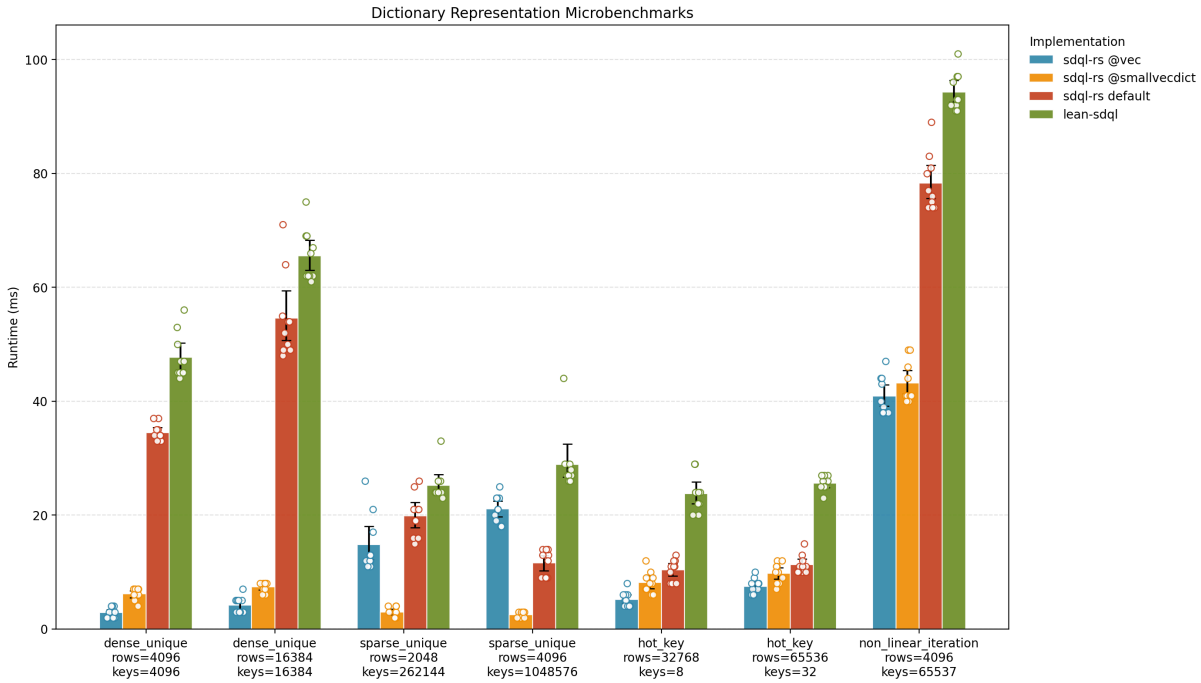


Figure 10: The effect of datastructure selection on runtime performance

I did an experiment where I manually removed the `@vec(...)` annotations from the TPC-H tests and re-ran the TPC-H performance comparison in Section 4.5. I found that datastructure selection accounts for 1.69% of the difference in performance between `lean-sdql` and the reference implementation (see Table 16). Interestingly, in `q21` at the `SF = 0.05` level, removing the hints improved the runtime, suggesting that the manual typehint of `@vec(600001)` was actively bad.

case	baseline sdql-rs	no-hints sdql-rs	lean-sdql	delta	lean/no-hints
tpch_q04	1.4ms	1.3ms	1.7ms	-10.0%	1.35x
tpch_q09	10.5ms	1.0ms	1.4ms	-90.4%	1.39x
tpch_q21	617.1ms	1.0ms	1.7ms	-99.8%	1.73x
tpch_q22	1.8ms	1.3ms	1.7ms	-28.6%	1.32x
tpch_q04_sf001	159.5ms	171.6ms	671.4ms	+7.6%	3.91x
tpch_q09_sf001	185.4ms	187.4ms	671.9ms	+1.1%	3.58x
tpch_q21_sf001	727.4ms	42.3ms	122.8ms	-94.2%	2.90x
tpch_q22_sf001	30.0ms	33.6ms	69.9ms	+11.9%	2.08x

Table 16: The effect of removing typehints from the reference implementation

case	baseline sdql-rs	adjusted string sdql-rs	lean-gen	gap closed
tpch_q03_sf001	170.0ms	293.9ms	634ms	26.7%
tpch_q06_sf001	142.0ms	209.0ms	432ms	23.1%
tpch_q11_sf001	17.0ms	19.2ms	42ms	8.8%
tpch_q13_sf001	36.0ms	49.9ms	101ms	21.4%
tpch_q14_sf001	147.0ms	210.6ms	461ms	20.3%
tpch_q17_sf001	145.0ms	207.4ms	444ms	20.9%

Table 17: The effect of forcing `string` instead of `varchar(...)` on reference implementation

Also, the reference implementation uses `varchar(...)` instead of `string`. If we force the reference implementation to use `string` instead, this optimisation accounts for `22.86%` of the difference (see Table 17).

I argue that `@vec` selection is not merely a drop-in replacement, because the annotation must be given a static upper bound on the size of the vector `@vec(....)`. If this upper bound is violated at run time, the reference implementation gives an out-of-bounds error.

In `load(...)`, I changed the implementation of `lean-sdql` to use the datastructure representation `Vec<T>` instead of `HashMap<i64,T>`. The results are in Table 18. We observe that using `Vec<T>` instead of `HashMap<i64, T>` provides approximately a `2x` speedup.

case	old lean	vec-load lean	speedup	sdql-rs now	lean/sdql now
tpch_q04_sf001	616ms	241ms	2.56x	166ms	1.45x
tpch_q09_sf001	810ms	289ms	2.80x	188ms	1.54x
tpch_q21_sf001	126ms	58ms	2.17x	722ms	0.08x
tpch_q22_sf001	91ms	42ms	2.17x	29ms	1.45x
TOTAL	9605ms	3942ms	2.44x	3956ms	0.996x

Table 18: The effect of loading lean-sdql input columns as vectors

4.7 The Use of a Profiler

When doing serious performance analysis, it is important to use a profiler, rather than merely timing the runtime.

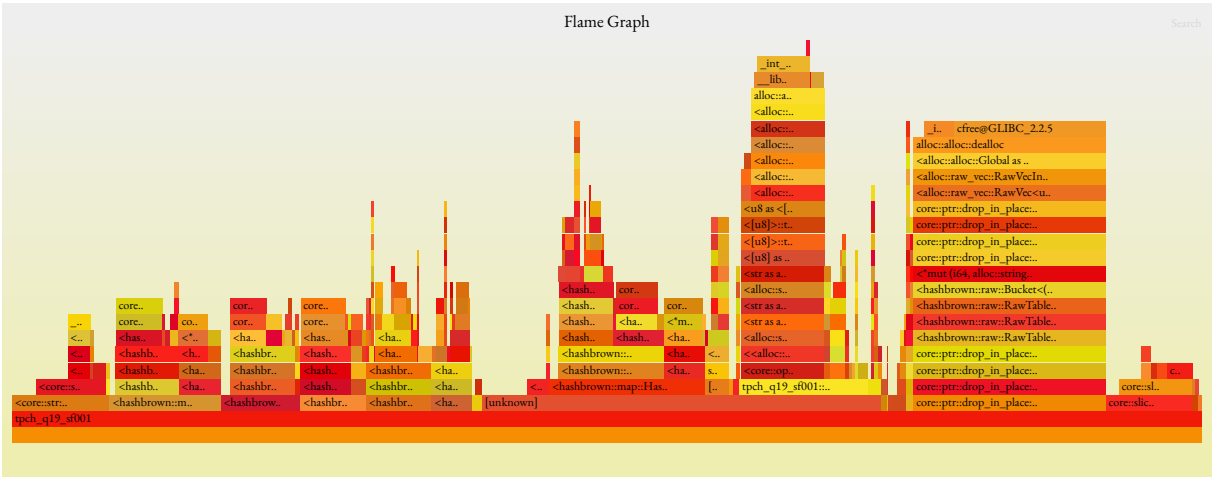


Figure 11: The flamegraphs from profiling `lean-sdql` on Q19

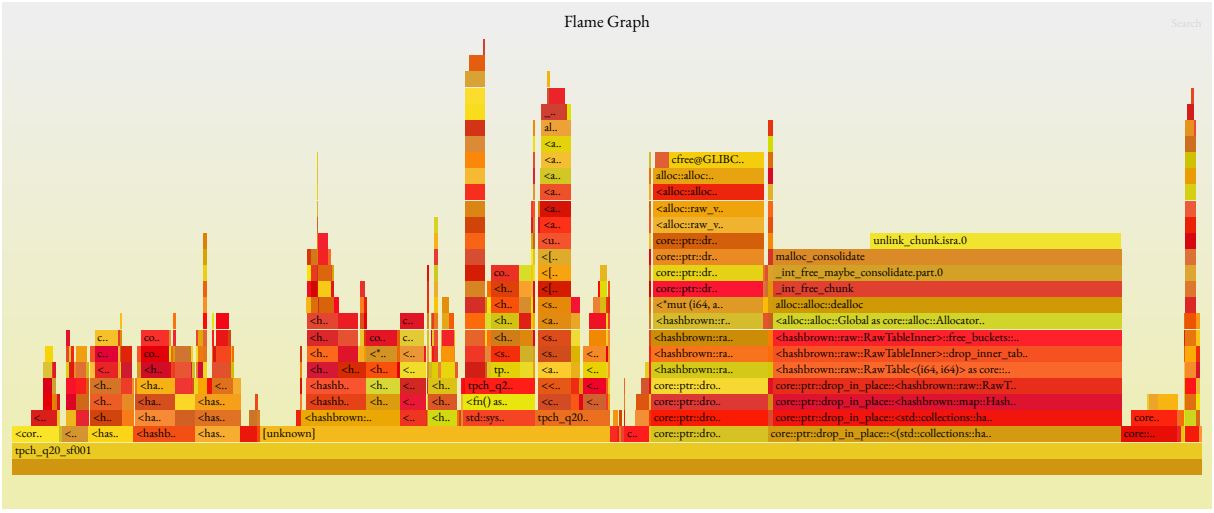


Figure 12: The flamegraphs from profiling `lean-sdql` on Q20

I have selected an excerpt of the flamegraphs to analyse in Figure 11 Figure 12. For the sake of brevity, the remaining flamegraphs may be viewed in the appendix: Figure 18. Figure 11 reveals that 38% of Q19 runtime is spent in `HashMap` resize operations, suggesting that pre-sizing the result dictionary would yield a substantial improvement I made flamegraphs for testcase at the $SF = 0.05$ level.

I tried many things to try and reduce the runtime of `lean-sdql` (see Section 3.9).

The reference implementation is still faster than `lean-sdql` for two main reasons:

1. Input representation mismatch: `lean-sdql` stores dense columns as `HashMap<row_id, value>`, but the reference implementation uses arrays/vectors. For example, the reference implementation can compile `@vec(...) {int -> string}` to `Vec<string>`, whereas `lean-sdql` will always compile that datatype to `HashMap<int, string>`. See Section 4.6 for a discussion of this tradeoff.
2. Excessive intermediate materialization: `lean-sdql` builds many temporary dictionaries, but the reference implementation uses a design-pattern of streaming through folds.

4.8 Evaluation of SDQL

I argue that, having implemented an SDQL compiler, I am in a unique position to evaluate the SDQL language and `SDQL + closure + semirings + optimisations`.

To evaluate these languages, I will use the already-established CDNs (Cognitive Dimensions of Notation) framework [26], which evaluates the user experience of a language along several dimensions. Although any

design evaluation is inherently subjective, (due to the evaluator), the CDNs framework defines a structured and well-accepted methodology for evaluating the design of programming languages. I have visualised the evaluation in Figure 13, and given my justification for each rating in Table 19.

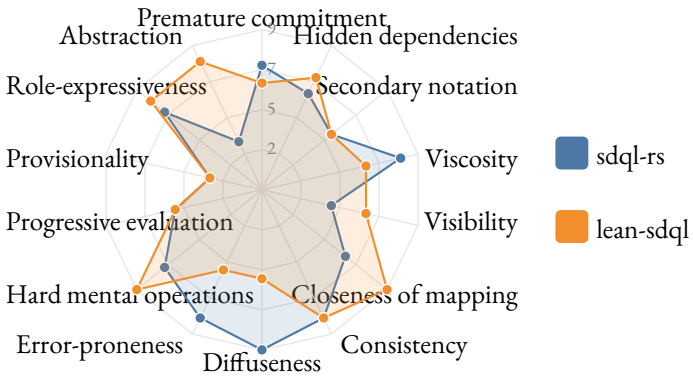


Figure 13: CDNs Evaluation: sdql-rs vs lean-sdql

Dimension	Reference Justification	lean-sdql Justification
Closeness of Mapping	Relational joins expressed through nested `sum` comprehensions and `dom()` checks. Forces translation of declarative SQL concepts into operational dictionary lookups.	`*s` (semiring multiplication) and `closure` operators allow expression of transitive closures and tensor operations exactly as they exist in graph theory and linear algebra.
Abstraction	Highly rigid; confined to `let` bindings and macro-level functions like `ext()`. No user-defined abstraction mechanisms.	Embedded within Lean 4, inheriting a superior abstraction manager. Algebraic optimisations allow high-level intent (e.g. `a *s b`) without manually defining nested loops.
Role-Expressiveness	Keywords like `sum`, `unique`, and `promote` clearly broadcast intent. Nested comprehensions require reading through layers to deduce operations like matrix multiplication.	Operators like `*s` immediately signal "semiring multiplication" to anyone with domain knowledge. Slightly more self-documenting at the operator level.
Diffuseness	Exhaustive, repetitive schema definitions: `<l_orderkey: @vec {int -> int}, ...>`. Highly verbose for common patterns.	Compresses dramatically. A 4D tensor multiplication taking dozens of lines of nested loops is expressed as `(a * b) *s (x * y)`.
Viscosity	Adding a single field (e.g. `c_phone`) to a downstream projection requires updating the initial dictionary, intermediate dictionaries, and the final `unique(...)` reduction. Highly viscous.	Higher-order operations pass complex types implicitly through semiring operations, reducing the number of sites requiring coordinated edits.
Premature Commitment	Lacks a query optimiser that reorders joins; programmer must manually determine efficient join order and write `let` bindings in that exact sequence.	Algebraic optimisations alleviate some burden, allowing focus on the *what* rather than the *how*. Some ordering constraints remain.
Hard Mental Operations	Tracking a standard hash-join via `dom()` is conceptually simpler, though tedious. Lower semantic density per token.	`p_betweenness_centrality_diamond` requires mentally tracking nested sums over transitive closures while holding types like `pair_sv.count * pair_vt.count` in working memory. High semantic density.
Hidden Dependencies	Dependencies hidden by dictionary keys and tuple indices. `o_h(key)(0)` creates fragile, invisible coupling to tuple structure.	`closure` hides a massive recursive execution graph behind a single word. Algebraic optimisations introduce invisible rewrite dependencies.
Visibility	Deep nesting of `if/then` inside `sum` comprehensions creates a visual "wall of text" obscuring core logic.	Algebraic operators flatten operations, improving visibility. Complex record types (e.g. `<_1: <_1: int, _2: real>, ...>`) can still clutter the visual field.
Error-Proneness	Invites "off-by-one" tuple index errors (e.g. `c_h(orders.o_custkey(i))(5)`). Hardcoded indices silently break if schema changes.	type system and named records (`<_1: int, _2: real>`) catch structural mismatches at compile time, reducing runtime slips.

4.9 Graph Database Comparison

The graph extension of SDQL (see Section 3.4) turns SDQL into a graph database.

I wrote a bespoke implementation of transitive closure and Viterbi's algorithm in Python, comparing that implementation against the auto-generated SDQL implementation.

The first implementation achieved 0.02x the speed of Python, due to cloning of BTrees. The next implementation was faster than Python, by avoiding cloning.

I compared the two implementations on a chain topology, on a star topology, and on a cycle topology. I ran the transitive closure algorithm and the viterbi algorithm. See Listing 4 for a table which compares the performance of `lean-sdql` against handwritten Python algorithms on a benchmark which includes a variety of graph topologies.

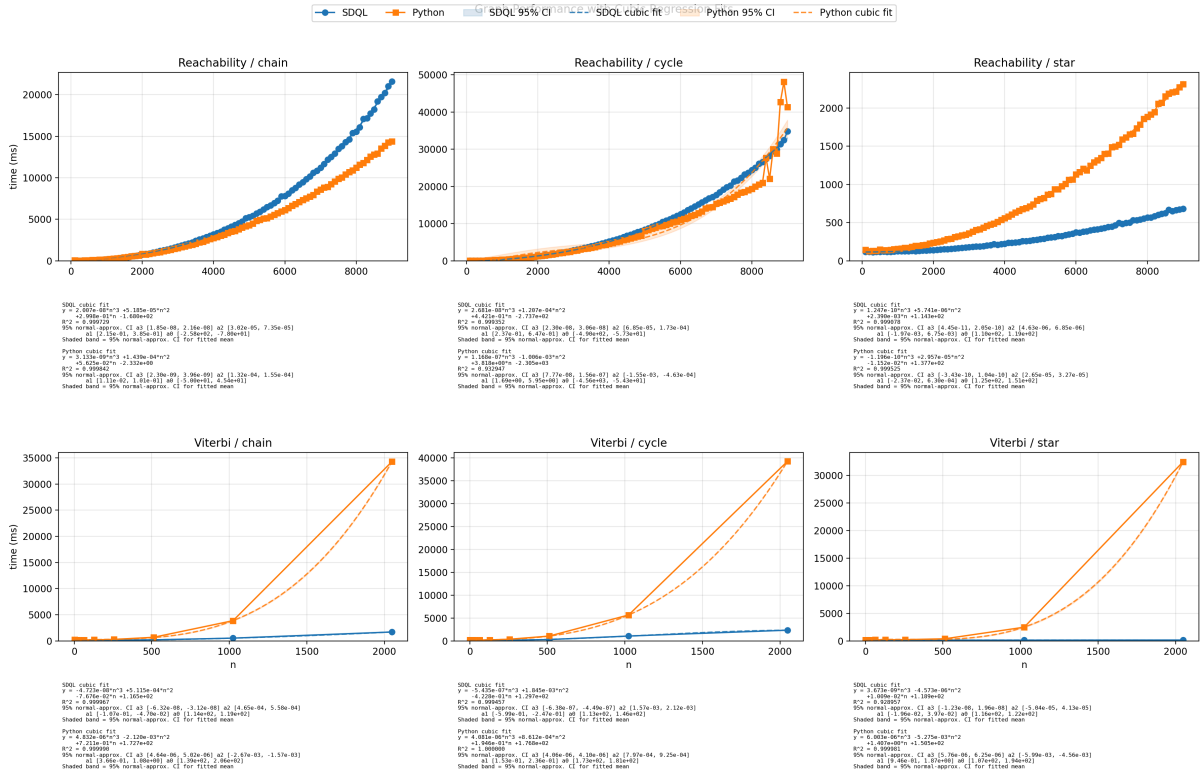


Figure 14: We benchmark the performance of SDQL vs hand-written Python graph algorithms

The performance graphs comparison are in Figure 14.

Note that from theory, we expect the runtime to be cubic $O(n^3)$. From the cubic regression, we see that we have R-values from `0.92-1.0`, which indicates that the experimental results do correspond to theoretical predictions about cubic scaling.

However, merely comparing SDQL against handwritten Python would not be sufficient. To create a stronger comparison, I compared SDQL against NetworkX [27] an industrial-strength graph library for Python.

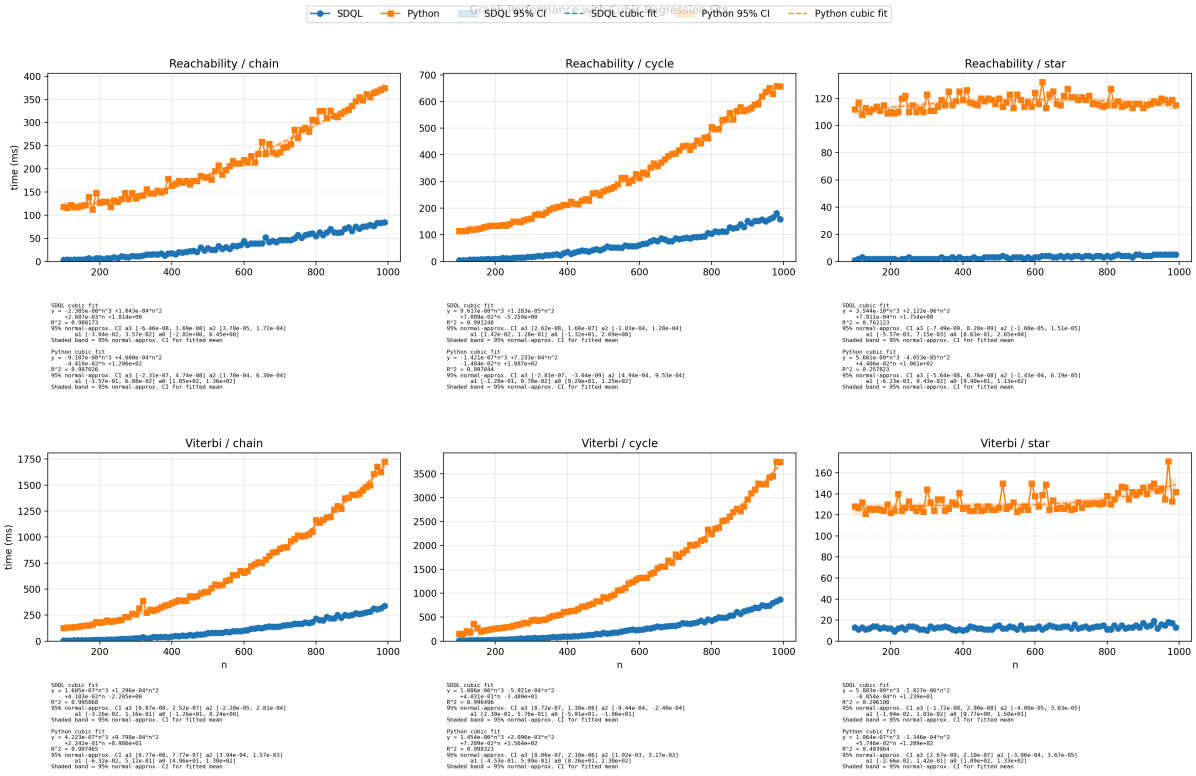


Figure 15: Performance of SDQL vs NetworkX

By analysing the comparison in Figure 15, we see that SDQL outperforms NetworkX in all 3 cases. Furthermore, all three cases have a strong cubic fit. The case of the star graph does not have a cubic fit because $A^* = I + A$.

That means the closure has only $2n - 1$ nonzero entries: the n identity entries plus the $n - 1$ star edges. This sparsity is true for both (reachability and Viterbi). By contrast:

1. chain: closure has about $\frac{n(n+1)}{2}$ entries
2. cycle: closure has n^2 entries
3. star: closure has $2n - 1$ entries

Chapter 5: Conclusions

Using Lean4 on this project was a significant undertaking. The overhead of satisfying the Lean typechecker sometimes overshadowed the gains of having a typed AST.

It's plausible that using a functional language like Haskell might have been more practical.

Reflecting, it was tedious to have to manually implement each semiring. For a commercial/industrial language, SDQL should expose some way for the user to define user-specified custom semirings. This lack of extensibility is a felt pain-point in the language that I observed but did not address.

Additionally, the language design of SDQL contains a typecast `promote[T](e)`, which casts `e` to type `T`. This implementation was tedious to implement because I implemented directly as a `N^2` different casting functions in the rust runtime (where `N` is the number of types). In the future, it might be more elegant to implement it as a pathfinding synthesis at the synthesis stage. Additionally, this approach might have allowed for covariant casting, such as `promote[{string -> real}](e : {string -> int})`. This painpoint was especially felt when implementing the `<dist: real, count: int>` for betweenness-centrality (see Listing 3).

While PHOAS is elegant (giving substitution, weakening for free), I couldn't figure out how to write a type-checker in PHOAS representation. I suspect that Lean4 isn't set up to handle parametricity. This limitation is in contrast to other theorem provers, such as Beluga [28]. Future work is needed to investigate whether it is possible to write a type-checker in PHOAS, and what the correct representation should be.

If I were to do this project again, I would have done it in Haskell. I still would have the AST carry evidence (e.g. which instance of addition is being instantiated), but this evidence would not be indexed by the type.

From this investigation, we have observed that a dependently-typed language such as Lean, which was typically relegated to only tasks of theoretical and formal proofs, can additionally handle the practical task of a real-world compiler. We have seen that dependent types can be richly applied to the AST and to the transformations. We have observed how the unique features of Lean's syntax macro system lead to a tight feedback loop with the editor, leading to a productive developer experience. Furthermore, we have demonstrated that, although SDQL was originally designed to express hybrid relational-tensor workloads, it can cleanly be extended to graph problems. When we extend SDQL with these new capability, we are motivated in a principled way by the mathematical structure of Kleene Algebras, and consequently, the resulting implementation of SDQL is performant and cohesive.

However, there still exist open questions from this investigation:

1. How can programmers minimise the proof-burden imposed when they use a dependently-typed language?
2. What is the correct way to model record types in a dependent embedding?
3. Can PHOAS be realistically used in a typechecker? Or does it inherently require specialised theories such as Contextual Modal Type Theory?

5.1 Lessons Learned

5.1.1 Reflections on Compiler Design

Compilers have been developed that use either none of the features of dependent types (GCC), or all of the features of dependent types (CompCert).

This project was situated in the middle space: using some (but not all) of the feature of dependent types. This is an underexplored area, and it remains open what the canonical way to do this is. The main area of research is how to structure and model the internal core representation of a programming language in dependent types. Expressive languages , including Lean, offer a panoply of choices. Once the core representation is chosen, the rest of the implementation writes itself for free.

5.1.2 Reflections on SDQL

SDQL is a promising language, for databases (similar to SQL). However, it is not Turing-complete. While all of the TPC-H queries can be successfully converted to SDQL, it is possible that a future query could be outside of scope of SDQL, requiring full recursion. The original reference for SDQL includes a definition of an extension for full recursion, but this extension was not implemented.

This limitation presents a genuine tension: SDQL with recursion would be more difficult to compile, but SDQL without recursion risks re-introducing the language-conversion friction that we sought to eliminate all along. How to navigate this tension is an unexplored question that could be investigated with further research.

5.2 Future Work

The following are projects for future work to further explore the space of SDQL and dependent types:

1. A run-time query optimiser to be competitive with SQL
2. A way of batching to work with datasets which do not fit in memory. In the current implementation of TPC-H, all dictionaries are pre-materialised in memory before the query begins, resulting in a memory bottleneck for larger scale-factors. Future work could explore on-demand materialisation and streaming of dictionaries to stress-test at-scale database sizes.
3. An implementation of SDQL[prod] and SDQL[rec]. SDQL[prod] is an extension to SDQL that enables "forall" `prod` queries (the dual to "exists" `sum` queries in SDQL), which could be relevant for ensuring database validity. SDQL[rec] is an extension to SDQL that enables full recursion and Turing-completeness. Both of these extensions could be useful in unlocking entirely novel classes of database operations, but raise their own implementation challenges that have not yet been explored.
4. SDQL deeply-embedded in Lean: this research direction would involve keeping SDQL as a DSL, but allowing an escape hatch to any functionality of Lean. This investigation would involve hooking into the Lean code-generator (instead of Rust code-generation), and formally verifying the correctness of the conversion. This would allow SDQL to be used as a DSL to specify semi-ring and graph computations within Lean proofs.

Bibliography

[1] Shaikhha, Huot, Smith, and Olteanu, "Functional collection programming with semi-ring dictionaries," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 1–33, 2022.

[2] Tarjan, "A Unified Approach to Path Problems," *Journal of the ACM*, vol. 28, no. 3, pp. 577–593, July 1981, doi: 10.1145/322261.322272.

[3] Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, no. 1, p. 35, Mar. 1977.

[4] Golan, "Semimodules over Semirings," in *Semirings and their Applications*, Dordrecht: Springer Netherlands, 1999, pp. 149–161. doi: 10.1007/978-94-015-9333-5_14.

[5] Shaikhha *et al.*, "A semi-ring dictionary query language for data science," WorkingPaper, July 2024.

[6] nLab authors, "semiring." Mar. 2026.

[7] Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, June 1970, doi: 10.1145/362384.362685.

[8] Hutchison, Howe, and Suciu, "LaraDB: A Minimalist Kernel for Linear and Relational Algebra Computation," 2017, doi: 10.48550/ARXIV.1703.07342.

[9] Kjolstad, Kamil, Chou, Lugato, and Amarasinghe, "The Tensor Algebra Compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 77:1–77:29, Oct. 2017, doi: 10.1145/3133901.

[10] Kepner *et al.*, "Mathematical foundations of the GraphBLAS," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, 2016, pp. 1–9.

[11] Brady, "IDRIS— systems programming meets full dependent types," in *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, 2011, pp. 43–54.

[12] d. Moura and Ullrich, "The lean 4 theorem prover and programming language," in *International Conference on Automated Deduction*, 2021, pp. 625–635.

[13] Martin-Löf and Sambin, *Intuitionistic type theory*, vol. 9. Bibliopolis Naples, 1984.

[14] Boehm, "A spiral model of software development and enhancement," *ACM SIGSOFT Software Engineering Notes*, vol. 11, no. 4, pp. 14–24, Aug. 1986, doi: 10.1145/12944.12948.

[15] Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[16] Zeidman, "A Gold Standard for Clean Room Development to Protect from Intellectual Property Infections," *Int'l. In-House Counsel J.*, vol. 16, p. 8773, 2023.

[17] Kravchenko, Bogdanova, and Shevgunov, "Ranking requirements using MoSCoW methodology in practice," in *Computer science on-line conference*, 2022, pp. 188–199.

[18] Leroy, Blazy, Kästner, Schommer, Pister, and Ferdinand, "CompCert-a formally verified optimizing compiler," in *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[19] Conway, *Regular algebra and finite machines*. Courier Corporation, 2012.

[20] d. Moura and Ullrich, "The Lean 4 Theorem Prover and Programming Language," in *Automated Deduction – CADE 28*, Springer International Publishing, 2021, pp. 625–635. doi: 10.1007/978-3-030-79876-5_37.

[21] Dunfield and Krishnaswami, "Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism." [Online]. Available: https://arxiv.org/abs/1306.6032

[22] "IEEE Standard for Floating-Point Arithmetic," 2008, doi: 10.1109/ieeestd.2008.4610935.

[23] Lang, *Algebra*. Springer Science & Business Media, 2012.

[24] Visser, "Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5," in *International Conference on Rewriting Techniques and Applications*, 2001, pp. 357–361.

[25] Boncz, Neumann, and Erling, "TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark," in *Performance Characterization and Benchmarking*, Nambiar and Poess, Eds., Cham: Springer International Publishing, 2014, pp. 61–76.

[26] Green, "Cognitive dimensions of notations," *People and computers V*, pp. 443–460, 1989.

[27] Hagberg and Conway, "Networkx: Network analysis with python," *URL: https://networkx. github. io*, vol. 1031, 2020.

[28] Pientka and Cave, "Inductive beluga: Programming proofs," in *International Conference on Automated Deduction*, 2015, pp. 272–281.

Appendix

```
-- outputs  "{2 -> 0.5, 3 -> 0.5, }"
unsafe def p_betweenness_centrality_diamond : SProg2 :=
  [SDQLProg2 { { int -> real } }|
    let g =
      { 1 -> { 2 -> 1.0, 3 -> 1.0 }
      , 2 -> { 4 -> 1.0 }
      , 3 -> { 4 -> 1.0 }
      } in
    let e_spc =
      sum( <u, edges> in g )
        { u ->
          sum( <v, weight> in edges )
            { v -> promote[sp_count](<count = 1.0, dist = weight>) }
        } in
    let closure_result = closure(e_spc) in
    let vertices = dom(closure_result) in
    sum( <s, row_s> in closure_result )
      sum( <t, pair_st> in row_s )
        let pair_st_rec = promote[<count: real, dist: real>](pair_st) in
        sum( <v, _> in vertices )
          if  (not (s == v)) && (not (v== t)) then
            let pair_sv = promote[<count: real, dist: real>](row_s(v)) in
            let pair_vt = promote[<count: real, dist: real>](closure_result(v)(t)) in
            if pair_sv.dist + pair_vt.dist == pair_st_rec.dist then
              { v -> (pair_sv.count * pair_vt.count) / pair_st_rec.count }
  ]
```

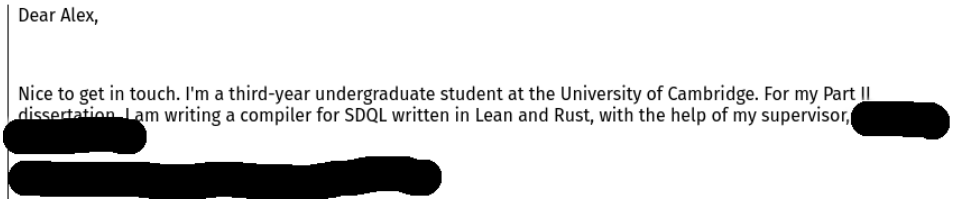
Listing 3: Implementation of Betweenness Centrality in SDQL

```
============================================================

Graph Reachability: SDQL vs Python (wall-clock ms)
case                                SDQL        Python   SDQL/Python
--------------------------------------------------------------------
reach_chain_2 (n=2)                  2ms          23ms       11.500×
reach_chain_4 (n=4)                  2ms          23ms       11.500×
reach_chain_8 (n=8)                  2ms          21ms       10.500×
reach_chain_16 (n=16)                2ms          23ms       11.500×
reach_chain_32 (n=32)                3ms          26ms        8.666×
reach_chain_64 (n=64)                3ms          29ms        9.666×
reach_chain_128 (n=128)              6ms          26ms        4.333×
reach_chain_256 (n=256)             16ms          40ms        2.500×
reach_chain_512 (n=512)             45ms          81ms        1.800×
reach_chain_1024 (n=1024)          172ms         203ms        1.180×
reach_chain_2048 (n=2048)          825ms         813ms        0.985×
reach_cycle_2 (n=2)                  2ms          24ms       12.000×
reach_cycle_4 (n=4)                  2ms          23ms       11.500×
reach_cycle_8 (n=8)                  2ms          26ms       13.000×
reach_cycle_16 (n=16)                2ms          26ms       13.000×
reach_cycle_32 (n=32)                2ms          24ms       12.000×
reach_cycle_64 (n=64)                3ms          27ms        9.000×
reach_cycle_128 (n=128)              6ms          28ms        4.666×
reach_cycle_256 (n=256)             25ms          43ms        1.720×
reach_cycle_512 (n=512)             70ms          99ms        1.414×
reach_cycle_1024 (n=1024)          285ms         323ms        1.133×
reach_cycle_2048 (n=2048)         1289ms        1335ms        1.035×
reach_star_2 (n=2)                   2ms          24ms       12.000×
reach_star_4 (n=4)                   3ms          26ms        8.666×
reach_star_8 (n=8)                   2ms          22ms       11.000×
reach_star_16 (n=16)                 2ms          25ms       12.500×
reach_star_32 (n=32)                 3ms          25ms        8.333×
reach_star_64 (n=64)                 2ms          22ms       11.000×
reach_star_128 (n=128)               3ms          26ms        8.666×
reach_star_256 (n=256)               4ms          26ms        6.500×
reach_star_512 (n=512)               6ms          32ms        5.333×
reach_star_1024 (n=1024)            16ms          49ms        3.062×
reach_star_2048 (n=2048)            39ms         129ms        3.307×
--------------------------------------------------------------------
TOTAL                              2848ms        3692ms        1.296×

Viterbi (max-product closure): SDQL vs Python (wall-clock ms)
case                                  SDQL        Python   SDQL/Python
----------------------------------------------------------------------
viterbi_chain_2 (n=2)                  2ms          77ms       38.500×
viterbi_chain_4 (n=4)                  2ms          76ms       38.000×
viterbi_chain_8 (n=8)                  2ms          79ms       39.500×
viterbi_chain_16 (n=16)                3ms          83ms       27.666×
viterbi_chain_32 (n=32)                3ms          76ms       25.333×
viterbi_chain_64 (n=64)                3ms          95ms       31.666×
viterbi_chain_128 (n=128)              7ms         112ms       16.000×
viterbi_chain_256 (n=256)             25ms         183ms        7.320×
viterbi_chain_512 (n=512)             94ms         584ms        6.212×
viterbi_chain_1024 (n=1024)          453ms        3938ms        8.693×
viterbi_chain_2048 (n=2048)         1669ms       34547ms       20.699×
viterbi_cycle_2 (n=2)                  2ms          85ms       42.500×
viterbi_cycle_4 (n=4)                  3ms          76ms       25.333×
viterbi_cycle_8 (n=8)                  3ms          79ms       26.333×
viterbi_cycle_16 (n=16)                3ms          79ms       26.333×
viterbi_cycle_32 (n=32)                4ms          80ms       20.000×
viterbi_cycle_64 (n=64)                7ms          92ms       13.142×
viterbi_cycle_128 (n=128)             19ms         121ms        6.368×
viterbi_cycle_256 (n=256)             44ms         244ms        5.545×
viterbi_cycle_512 (n=512)            188ms         936ms        4.978×
viterbi_cycle_1024 (n=1024)          953ms        5656ms        5.934×
viterbi_cycle_2048 (n=2048)         2417ms       38177ms       15.795×
viterbi_star_2 (n=2)                   5ms          84ms       16.800×
viterbi_star_4 (n=4)                   6ms          80ms       13.333×
viterbi_star_8 (n=8)                   7ms          97ms       13.857×
viterbi_star_16 (n=16)                 7ms          81ms       11.571×
viterbi_star_32 (n=32)                 6ms          78ms       13.000×
viterbi_star_64 (n=64)                 7ms          82ms       11.714×
viterbi_star_128 (n=128)               8ms          83ms       10.375×
viterbi_star_256 (n=256)              10ms         111ms       11.100×
viterbi_star_512 (n=512)              17ms         260ms       15.294×
viterbi_star_1024 (n=1024)            20ms        2725ms      136.250×
viterbi_star_2048 (n=2048)            47ms       35289ms      750.829×
----------------------------------------------------------------------
```

Listing 4: Performance of lean-sdql against Python on graph problems


Dear Alex,

Nice to get in touch. I'm a third-year undergraduate student at the University of Cambridge. For my Part II dissertation, I am writing a compiler for SDQL written in Lean and Rust, with the help of my supervisor, [redacted]

[redacted]

Figure 16: Email to authors of the original SDQL implementation

```
nixos
    description: Notebook
    product: HP Pavilion Laptop 15-eg2xxx (643L4UA#ABA)
    vendor: HP
    width: 64 bits
  *-core
       description: Motherboard
       vendor: HP
       version: 42.05
     *-memory
          description: System Memory
          size: 12GiB
     *-cpu
          description: CPU
          product: 12th Gen Intel(R) Core(TM) i5-1235U
          vendor: Intel Corp.
          physical id: 24
          bus info: cpu@0
          version: 6.154.4
          size: 400MHz
          capacity: 4400MHz
          width: 64 bits
          clock: 100MHz
          configuration: cores=10 enabledcores=10 microcode=1077 threads=12

OS: NixOS 24.05.20241230.b134951 (Uakari) x86_64
Host: HP 89F6
Kernel: 6.12.7
Packages: 1212 (nix-system), 7690 (nix-user), 12 (flatpak)
Shell: bash 5.2.32
CPU: 12th Gen Intel i5-1235U (12) @ 4.400GHz
GPU: Intel Alder Lake-UP3 GT2 [Iris Xe Graphics]
Memory: 7007MiB / 11634MiB
```

Listing 5: Specs of the machine used for benchmarking

```
[PASS] add_int
[PASS] dict_insert
[PASS] lookup_hit
[PASS] lookup_miss
[PASS] sum_vals
[PASS] underscore_ident
[PASS] if_then_true
[PASS] if_then_false
[PASS] semiring_mul_real
[PASS] semiring_mul_matrix_2x2
[PASS] semiring_mul_record_tensor
[PASS] p_id_matrix
[PASS] semiring_mul_4d_tensor
[PASS] closure_bool
[PASS] closure_real
[PASS] closure_matrix_bool
[PASS] closure_matrix_bool_linear
[PASS] closure_matrix_real
[PASS] closure_graph_self_loop
[PASS] closure_graph_chain_3
[PASS] closure_graph_cycle_3
[PASS] closure_graph_star
[PASS] closure_graph_diamond
[PASS] closure_graph_disconnected
[PASS] closure_graph_reverse_chain
[PASS] closure_graph_weighted_real
[PASS] vertical_loop_fusion_key_map (optimised matches unoptimised)
[PASS] vertical_loop_fusion_value_map (optimised matches unoptimised)
[PASS] horizontal_loop_fusion (optimised matches unoptimised)
[PASS] loop_factorization_left (optimised matches unoptimised)
[PASS] loop_factorization_right (optimised matches unoptimised)
[PASS] loop_invariant_code_motion (optimised matches unoptimised)
[PASS] loop_memoization_lookup (optimised matches unoptimised)
[PASS] loop_memoization_partition (optimised matches unoptimised)
[PASS] tpch_q01 (matches sdql-rs/target/release/tpch_q01_tiny)
[PASS] tpch_q02 (matches sdql-rs/target/release/tpch_q02_tiny)
[PASS] tpch_q03 (matches sdql-rs/target/release/tpch_q03_tiny)
[PASS] tpch_q04 (matches sdql-rs/target/release/tpch_q04_tiny)
[PASS] tpch_q05 (matches sdql-rs/target/release/tpch_q05_tiny)
[PASS] tpch_q06 (matches sdql-rs/target/release/tpch_q06_tiny)
[PASS] tpch_q07 (matches sdql-rs/target/release/tpch_q07_tiny)
[PASS] tpch_q09 (matches sdql-rs/target/release/tpch_q09_tiny)
[PASS] tpch_q10 (matches sdql-rs/target/release/tpch_q10_tiny)
[PASS] tpch_q11 (matches sdql-rs/target/release/tpch_q11_tiny)
[PASS] tpch_q12 (matches sdql-rs/target/release/tpch_q12_tiny)
[PASS] tpch_q13 (matches sdql-rs/target/release/tpch_q13_tiny)
[PASS] tpch_q14 (matches sdql-rs/target/release/tpch_q14_tiny)
[PASS] tpch_q16 (matches sdql-rs/target/release/tpch_q16_tiny)
[PASS] tpch_q17 (matches sdql-rs/target/release/tpch_q17_tiny)
[PASS] tpch_q18 (matches sdql-rs/target/release/tpch_q18_tiny)
[PASS] tpch_q19 (matches sdql-rs/target/release/tpch_q19_tiny)
[PASS] tpch_q20 (matches sdql-rs/target/release/tpch_q20_tiny)
[PASS] tpch_q21 (matches sdql-rs/target/release/tpch_q21_tiny)
[PASS] tpch_q22 (matches sdql-rs/target/release/tpch_q22_tiny)
[PASS] tpch_q01_sf001 (matches sdql-rs/target/release/tpch_q01_tiny)
[PASS] tpch_q02_sf001 (matches sdql-rs/target/release/tpch_q02_tiny)
[PASS] tpch_q03_sf001 (matches sdql-rs/target/release/tpch_q03_tiny)
[PASS] tpch_q04_sf001 (matches sdql-rs/target/release/tpch_q04_tiny)
[PASS] tpch_q05_sf001 (matches sdql-rs/target/release/tpch_q05_tiny)
[PASS] tpch_q06_sf001 (matches sdql-rs/target/release/tpch_q06_tiny)
[PASS] tpch_q07_sf001 (matches sdql-rs/target/release/tpch_q07_tiny)
[PASS] tpch_q09_sf001 (matches sdql-rs/target/release/tpch_q09_tiny)
[PASS] tpch_q10_sf001 (matches sdql-rs/target/release/tpch_q10_tiny)
[PASS] tpch_q11_sf001 (matches sdql-rs/target/release/tpch_q11_tiny)
[PASS] tpch_q12_sf001 (matches sdql-rs/target/release/tpch_q12_tiny)
[PASS] tpch_q13_sf001 (matches sdql-rs/target/release/tpch_q13_tiny)
[PASS] tpch_q14_sf001 (matches sdql-rs/target/release/tpch_q14_tiny)
[PASS] tpch_q15_sf001 (matches sdql-rs/target/release/tpch_q15_tiny)
[PASS] tpch_q16_sf001 (matches sdql-rs/target/release/tpch_q16_tiny)
[PASS] tpch_q17_sf001 (matches sdql-rs/target/release/tpch_q17_tiny)
[PASS] tpch_q18_sf001 (matches sdql-rs/target/release/tpch_q18_tiny)
[PASS] tpch_q19_sf001 (matches sdql-rs/target/release/tpch_q19_tiny)
[PASS] tpch_q20_sf001 (matches sdql-rs/target/release/tpch_q20_tiny)
[PASS] tpch_q21_sf001 (matches sdql-rs/target/release/tpch_q21_tiny)
[PASS] tpch_q22_sf001 (matches sdql-rs/target/release/tpch_q22_tiny)
```

Listing 6: TPC-H run output

$$\frac{A \in \Gamma}{\Gamma \vdash x : A}\ \text{T-Var} \qquad \frac{}{\Gamma \vdash i : \texttt{int}}\ \text{T-Int} \qquad \frac{}{\Gamma \vdash r : \mathbb{R}}\ \text{T-Real} \qquad \frac{}{\Gamma \vdash b : \mathbb{B}}\ \text{T-BB}$$

$$\frac{}{\Gamma \vdash s : \text{string}}\ \text{T-String} \qquad \frac{\Gamma \vdash e_1 : t_1 \quad ... \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash <e_1, ..., e_n> : <t_1, ..., t_n>}\ \text{T-Record}$$

$$\frac{}{\Gamma \vdash \{\} : \{t_1 \to t_2\}}\ \text{T-EmptyDict} \qquad \frac{\Gamma \vdash k : t_1 \quad \Gamma \vdash v : t_2 \quad \Gamma \vdash d : \{t_1 \to t_2\}}{\Gamma \vdash \{k \to v\} + + d : \{t_1 \to t_2\}}\ \text{T-DictInsert}$$

$$\frac{\text{AddM } t_2 \quad \Gamma \vdash d : \{t_1 \to t_2\} \quad \Gamma \vdash k : t_1}{\Gamma \vdash d(k) : t_2}\ \text{T-Lookup} \qquad \frac{\Gamma \vdash e : \mathbb{B}}{\Gamma \vdash \text{not}(e) : \mathbb{B}}\ \text{T-Not}$$

$$\frac{\Gamma \vdash e_1 : \mathbb{B} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t}\ \text{T-If} \qquad \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, t_1 \vdash e_2 : t_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : t_2}\ \text{T-Let}$$

$$\frac{\text{AddM } t \quad \Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 + e_2 : t}\ \text{T-Add}$$

$$\frac{\text{ScaleM } st_1 \quad \text{ScaleM } st_2 \quad \Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 * e_2 : t_1 \otimes t_2}\ \text{T-Mul}$$

$$\frac{\text{HasMul } t \quad \Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 * se_2 : t}\ \text{T-SemiringMul} \qquad \frac{\text{HasClosure } t \quad \Gamma \vdash e : t}{\Gamma \vdash \text{closure}(e) : t}\ \text{T-Closure}$$

$$\frac{\Gamma \vdash e : t_1}{\Gamma \vdash \text{promote}[t_2](e) : t_2}\ \text{T-Promote} \qquad \frac{\text{AddM } t \quad \Gamma \vdash d : \{t_1 \to t_2\} \quad \Gamma, t_1, t_2 \vdash e : t}{\Gamma \vdash \text{sum}(<k, v> \text{ in } d)e : t}\ \text{T-Sum}$$

$$\frac{\text{has\_proj} < t_1, ..., t_n > it \quad \Gamma \vdash r :< t_1, ..., t_n >}{\Gamma \vdash r.i : t}\ \text{T-Proj} \qquad \frac{\Gamma \vdash e_1 : \mathbb{B} \quad \Gamma \vdash e_2 : \mathbb{B}}{\Gamma \vdash e_1 \parallel e_2 : \mathbb{B}}\ \text{T-Or}$$

$$\frac{\Gamma \vdash e_1 : \mathbb{B} \quad \Gamma \vdash e_2 : \mathbb{B}}{\Gamma \vdash e_1 e_2 : \mathbb{B}}\ \text{T-And} \qquad \frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 = e_2 : \mathbb{B}}\ \text{T-Eq}$$

$$\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 \leq e_2 : \mathbb{B}}\ \text{T-Leq} \qquad \frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 < e_2 : \mathbb{B}}\ \text{T-Lt}$$

$$\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 - e_2 : t}\ \text{T-Sub} \qquad \frac{\Gamma \vdash e_1 : \mathbb{R} \quad \Gamma \vdash e_2 : \mathbb{R}}{\Gamma \vdash e_1 / e_2 : \mathbb{R}}\ \text{T-Div}$$

$$\frac{\Gamma \vdash e_1 : \text{string} \quad \Gamma \vdash e_2 : \text{string}}{\Gamma \vdash \text{endsWith}(e_1, e_2) : \mathbb{B}}\ \text{T-EndsWith} \qquad \frac{\Gamma \vdash e_1 : \text{string} \quad \Gamma \vdash e_2 : \text{string}}{\Gamma \vdash \text{startsWith}(e_1, e_2) : \mathbb{B}}\ \text{T-StartsWith}$$

$$\frac{\Gamma \vdash e_1 : \text{string} \quad \Gamma \vdash e_2 : \text{string}}{\Gamma \vdash \text{contains}(e_1, e_2) : \mathbb{B}}\ \text{T-Contains} \qquad \frac{\Gamma \vdash e_1 : \text{string} \quad \Gamma \vdash e_2 : \text{string}}{\Gamma \vdash \text{firstIndex}(e_1, e_2) : \mathbb{Z}}\ \text{T-FirstIndex}$$

$$\frac{\Gamma \vdash e_1 : \text{string} \quad \Gamma \vdash e_2 : \text{string}}{\Gamma \vdash \text{lastIndex}(e_1, e_2) : \mathbb{Z}}\ \text{T-LastIndex}$$

Figure 17: SDQL2 Typing Rules

$$\frac{\Gamma \vdash e : \text{string} \quad \Gamma \vdash i : \mathbb{Z} \quad \Gamma \vdash j : \mathbb{Z}}{\Gamma \vdash \text{subString}(e, i, j) : \text{string}}\ \text{T-SubString} \qquad \frac{\Gamma \vdash d : \{t_1 \to t_2\}}{\Gamma \vdash \text{dom}(d) : \{t_1 \to \mathbb{B}\}}\ \text{T-Dom}$$

builtin function	b	::=	|| *or*
		|	&& *and*
		|	= *equals*
		|	<= *less than or equal to*
		|	< *less than*
		|	- *subtraction*
		|	/ *division*
		|	endsWith *string ends with*
		|	startsWith *string starts with*
		|	contains *string contains*
		|	firstIndex *first index in string*
		|	lastIndex *last index in string*
		|	subString *substring of a string*
		|	dom *domain of a dictionary*
		|	range *The range* $1...n$
		|	size *The size of a dictionary*
		|	date *a date literal*
		|	year *the year of a date*
		|	concat *concatenate two records*

Type	t	::=	BB *Boolean*
		|	real *real number*
		|	date *date*
		|	$\{t_1 \to t_2\}$ *dictionary*
		|	$<n_1 : t_1, n_2 : t_2, ..., n_m : t_m>$ *record*
		|	string *string*
		|	int *int*
		|	maxProduct *max product*

Expr	e	::=	x *variable*
		|	i *integer literal*
		|	r *real literal*
		|	b *Boolean literal*
		|	s *string literal*
		|	$< n_1 = e_1, n_2 = e_2, ..., n_m = e_m >$ *record literal*
		|	$\{n_1 = e_1, n_2 = e_2, ..., n_m = e_m\}$ *dictionary literal*
		|	$e_1(e_2)$ *dictionary lookup*
		|	not(e) *not*
		|	if e_1 then e_2 else e_3 *if-then-else*
		|	let x = e_1 in e_2 *let-in*
		|	$e_1 + e_2$ *addition*
		|	$e_1 * e_2$ *multiplication*
		|	$e_1 *_s e_2$ *semi-ring multiplication*
		|	closure(e) *closure*
		|	promote[T](e) *type promotion*
		|	sum(<k,v> in e_1) e_2 *sum*
		|	$e.i$ *projection*
		|	$b(e)$ *builtin function*

Table 22: BNF of the syntax of SDQL

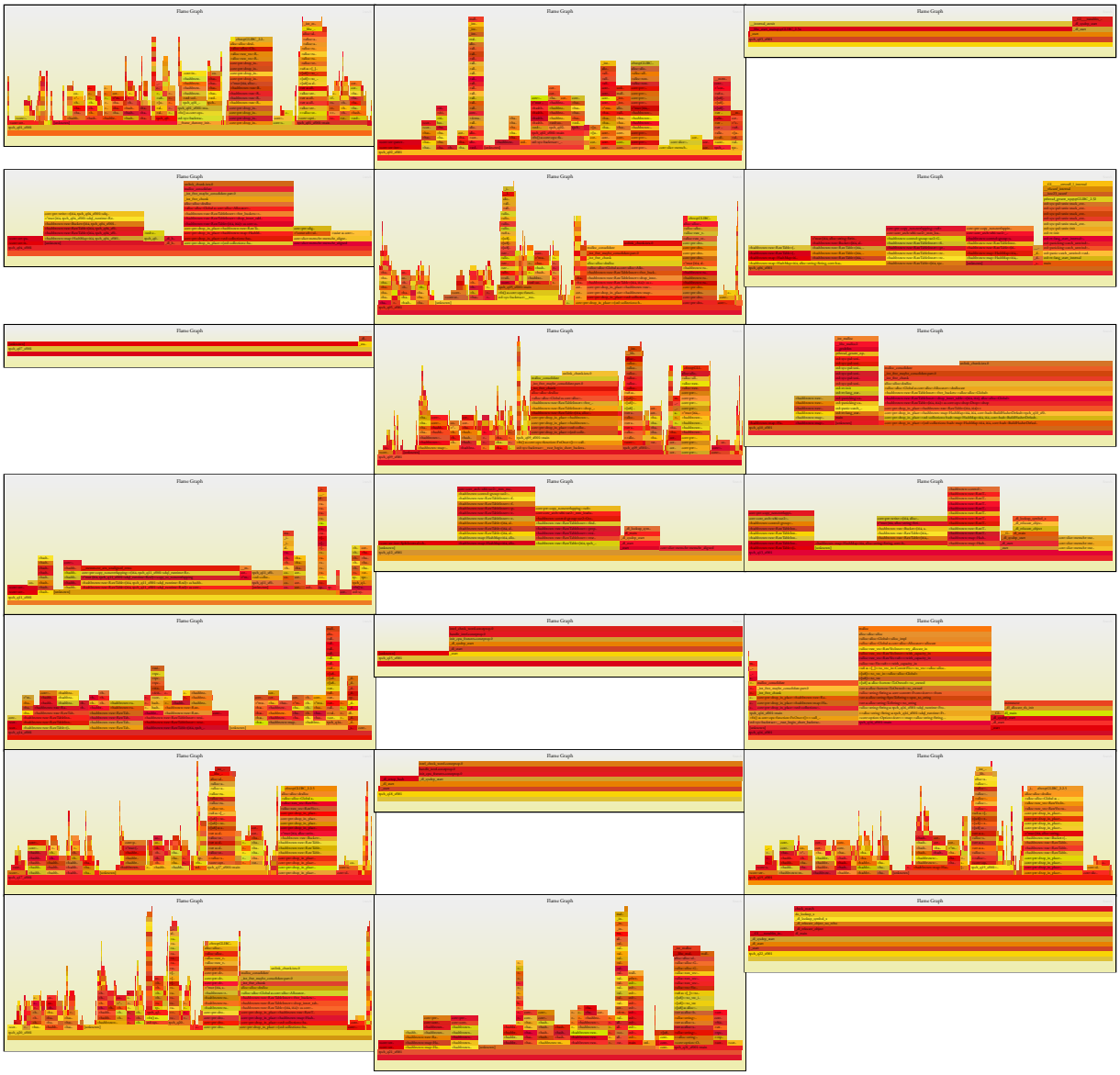


Figure 18: The flamegraphs from profiling `lean-sdql`

Project Proposal

Part II Project Proposal 2

October 10, 2025

Contents

1 Introduction and description **2**
- 1.1 Background . . . 2
- 1.2 Objectives . . . 2
  - 1.2.1 SDQL Lexer/Parser . . . 2
  - 1.2.2 Algebraic Optimisations . . . 2
  - 1.2.3 Interpreter . . . 2
  - 1.2.4 Code Generator . . . 3
  - 1.2.5 Path Primitives . . . 3

2 Risk Management and Fallback **3**

3 Starting point **3**

4 Success criteria and evaluation plan **3**
- 4.1 Success Criteria . . . 3
- 4.2 Evaluation Plan . . . 4
  - 4.2.1 Quantitative Benchmark Evaluation . . . 4
  - 4.2.2 CDNs Evaluation . . . 4

5 Work plan **4**

6 Resource declaration **5**

7 Works Cited **6**

1 Introduction and description

1.1 Background

SDQL is a research language developed by researchers (Amir Shaikhha, Alex Mascolo, Amirali Kaboli) at the university of Edinburgh (Shaikhha, Amir and Huot, Mathieu and Smith, Jaclyn and Olteanu, Dan, 2022). It is a statically-typed language with functional collections designed for database and data-science tasks. An implementation of SDQL already exists, which uses Scala to generate C++.

The goal of this project is to create a compiler for SDQL using the Lean 4 programming language. While Lean might be a niche language, the Lean compiler is already self-hosted in Lean, demonstrating that using a functional language to create a compiler is a feasible strategy.

1.2 Objectives

1.2.1 SDQL Lexer/Parser

The goal is to create a parser for SDQL into the Lean abstract syntax tree (AST). This will be achieved by using Lean's elaboration and syntax macros to create an embedded domain-specific language (EDSL) inside Lean. This is already-established technique. The front-end will make use of Algebraic Data Types (ADTs), well-supported in Lean, to concisely represent the AST of SDQL.

Semirings can be conveniently abstracted by the notion of a type-class in Lean, which are natively supported.

1.2.2 Algebraic Optimisations

The SDQL paper lists algebraic optimisations that can be performed over SDQL. The paper lists six optimisations.

As Lean is a functional language, these optimisations can be implemented as functional re-writes over the SDQL AST.

1.2.3 Interpreter

Before beginning the work on the code generation, there will be a simple interpreter of SDQL. This interpreter will be simple, not necessarily performant, but only with the goal of quickly checking the correctness of SDQL semantics.

1.2.4 Code Generator

A code generator will be implemented using Rust. Rust is an industrially-used programming language which can compile to native code, making it an ideal code-generation target.

1.2.5 Path Primitives

As an extension to the project, I will extend the SDQL language with primitives for reasoning about paths. Path problems can be reinterpreted as problems in the context of semi-ring matrices (Tarjan, Robert Endre, 1981). In other words, this will add a new primitive "fixed point" operator to the language over dictionaries, which will allow the user to emulate graph-theoretic algorithms (such as Floyd-Warshall) using the method of Tarjan.

2 Risk Management and Fallback

Lean and Rust are novel languages with innovative features; indeed that is one reason why they were chosen for this project. If the inherent limitations of using a young language ever block this project from progress, a fallback is proposed to Haskell and C++. Haskell is a more mature functional language than Lean, and C++ is a more mature systems-level language than Rust.

3 Starting point

As of the starting point of the project (October 10, 2025) no code exists yet for this project.

4 Success criteria and evaluation plan

4.1 Success Criteria

This project has a clear success criterion: feature-parity with the reference implementation of the existing SDQL compiler. In other words: for each input program satisfying the SDQL specification, if the reference implementation supports that program, then my implementation will also support that program and have semantics that comply with the original SDQL specification.

Compiler performance (in terms of runtime/binary size/memory footprint) is also a goal, but a secondary goal to be achieved after correctness.

4.2 Evaluation Plan

4.2.1 Quantitative Benchmark Evaluation

The correctness and performance of my implementation will be tested using a benchmark of SDQL programs. The SDQL paper already contains a set of SDQL programs. I will extend this benchmark.

For each SDQL program in the benchmark, I will run that program on both the reference implementation and my implementation. This comparison will test the following properties:

1. Semantic correctness
2. Compile time
3. Binary Size
4. Execution Time

4.2.2 CDNs Evaluation

To evaluate a programming language from a human usability standpoint, a well-established framework from the literature is the Cognitive Dimensions of Notation (CDNs) framework.

The CDN evaluation will be done by an expert evaluator (myself), so I do not need any ethical approval.

5 Work plan

Date	Days	Task
[2025-10-07 Tue]	14	Project bootstrapping: choose host language (Lean4), repo/CI setup, fina
[2025-10-21 Tue]	14	Grammar and AST: parser, pretty-printer, golden tests
[2025-11-04 Tue]	14	Type system: semiring/semimodule abstractions via type classes; well-typ
[2025-11-18 Tue]	14	Core evaluator/IR: dictionary ops, joins, aggregations; naïve execution pi
[2025-12-02 Tue]	14	Core parity pass: replicate SDQL features; end-to-end MVP on small dat
[2025-12-16 Tue]	21	Robustness: error handling, REPL/CLI, test coverage, micro-bench harn
[2026-01-06 Tue]	14	Kleene-star extension: starred semirings; Warshall reachability
[2026-01-20 Tue]	14	Floyd–Warshall (APSP): matrix semiring implementation; correctness tes
[2026-02-03 Tue]	14	Optimisations I: vertical/horizontal loop fusion; IR rewrites; invariants
[2026-02-17 Tue]	14	Optimisations II: memoisation, loop factorisation, code motion
[2026-03-03 Tue]	14	Extra semirings: formal languages, set/intersection; integration tests
[2026-03-17 Tue]	14	Benchmark suite v1: graph, relational, linear algebra; Nix for reproducibi
[2026-03-31 Tue]	14	Run benchmarks + profiling: SDQL vs. reference
[2026-04-14 Tue]	7	Feature freeze; Evaluation round 1 analysis; CDN study design and pilot
[2026-04-21 Tue]	14	Writing sprint I: background, design, and implementation chapters
[2026-05-05 Tue]	6	Writing sprint II: evaluation chapter, figures, tables; re-run final benchma
[2026-05-11 Mon]	5	Full draft to supervisor; revisions; polish; compliance checks (word count,
[2026-05-16 Sat]	2	Final formatting, archive reproducibility artifacts, submission packaging
[2026-05-17 Sun]	0	Submission

6 Resource declaration

I plan on using my own personal computer for this project. The effect of hardware failure can be mitigated by making regular commits on the version control software git, in combination with a remote git repository, such as GitHub. In the event of catastrophic hardware failure, it might be possible to use university computers.

I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

Resources Declaration

- Hardware: personal laptop HP Pavilion, Intel I5, NixOs.
  - Software (all open-source): Lean 4 (stable), Rust (stable toolchain via rustup), Nix (for reproducible builds), Git/GitHub, GNU coreutils. No proprietary tools required.

7 Works Cited

Shaikhha, Amir and Huot, Mathieu and Smith, Jaclyn and Olteanu, Dan (2022). *Functional collection programming with semi-ring dictionaries*, ACM New York, NY, USA.

Tarjan, Robert Endre (1981). *A unified approach to path problems*, ACM New York, NY, USA.