Program Optimization With Collection Annotations and Database-Style Term Rewriting

by

Xin Zhang

B.A., University of Oxford (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2026

Authored by: Xin Zhang
Department of Electrical Engineering and Computer Science
May 7, 2026

Certified by: Adam Chlipala
Arthur J. Conner (1888) Professor of Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Program Optimization With Collection Annotations and Database-Style Term Rewriting

by

Xin Zhang

Submitted to the Department of Electrical Engineering and Computer Science
on May 7, 2026 in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

ABSTRACT

Since the 1970s, database engines have optimized queries by representing them in core calculi and repeatedly applying algebraic rewrite rules. Larger applications often treat database engines as important external data structures, benefiting from their internal optimizations. However, those applications are typically optimized with more conventional compiler techniques like dataflow analysis. We present here our experiences fusing application code with database logic and jointly optimizing them, now with algebraic rewrites for both, not just queries. That is, database-style data structures and algorithms are optimized with ahead-of-time knowledge of not just access patterns but also how query results are used in general-purpose code. The top new enabling idea is keeping track of contexts around queries using coercions between different consistency levels (list, bag, set). Our method is embodied in a library within the Rocq Prover, allowing rapid assembly of custom compilation pipelines with mechanized proof of correctness. We parameterized our compiler over optimization hints and conducted experiments using a range of example programs. The compiler was formally verified to produce semantically preserving pipelines when instantiated with arbitrary optimization hints. It achieved the expected asymptotic optimizations using hints generated by a large-language model, and the concrete runtime of each optimized program was comparable to that of the corresponding program written in Python with access to a SQLite database.

Thesis supervisor: Adam Chlipala
Title: Arthur J. Conner (1888) Professor of Computer Science

Acknowledgments

Much of the content of this thesis is drawn from a joint conference submission co-authored with Dustin Jamner, Jack Feser, and Prof. Adam Chlipala. I would like to acknowledge Christian Teshome for his contributions to the codebase of the project associated with this thesis.

This thesis is based upon work supported by the Agency for Science, Technology and Research (A*STAR), Singapore, under the National Science Scholarship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Research in databases has led to intensive optimizations in query planners for relational-database-management languages, with SQL being a prominent example of such languages. Programmers can write declarative queries about the underlying database, and the query planner applies algebraic rewrite rules to determine the algorithmic details. We observe that, while many important applications are built on top of databases, typically the logic around database queries is written in a rather different style, and compilers optimize it in a rather different way. Is this divergence fundamental? Are there benefits to be gained by unifying the treatment of declarative queries and e.g. imperative loops over the results? We present one new approach in that direction, for programs that combine features of general-purpose languages with database-inspired comprehension code. Essentially, the compilers we build give programs custom in-memory databases, and we implement the whole framework in Rocq and generate compiler-correctness proofs, an objective that is simplified by phrasing all optimization in terms of proved rewrite rules and orchestrating strategies for applying them.

Data-intensive applications nowadays often involve more complex logic than typical database queries, and it is unclear how to generalize the database-style term-rewriting framework to optimize such application code. Although procedural extensions to SQL have existed for a long time, writing complex application logic entirely in these extended SQL languages incurs significant performance cost, and many common programming-language features remain missing from the extensions [1]. It is therefore impractical to develop data-intensive applications entirely in SQL, so programmers naturally write code in an imperative rather than declarative style to handle the data returned from queries. As a result, there is need for explicit communication between application and database, and the misalignment between them causes the impedance-mismatch problem [2]. The impedance mismatch causes overhead in both programming and execution due to the information flow across the two languages. Moreover, co-optimizations between the database representation and the application code should be possible if the programs that will run on the data are known at compile time [3], but the impedance mismatch makes it difficult to implement the optimizations correctly. In particular, SQL's default bag semantics can result in nondeterministic query outputs when a query retrieves multiple rows without specifying a total order [4]. Depending on how the query output is used, the application code may or may not behave nondeterministically. Hence, it becomes even harder to reason about the semantic preservation of optimizations on the application code. Prior works such as Links have explored integration of database

management and application programming languages, but many choose to compile a subset of their languages to SQL [5]. That choice inherently ties the language semantics to SQL's semantics, but the SQL standard is ambiguous, with different valid implementations of SQL potentially producing different results for the same query [4]. Furthermore, database-style optimizations are often nontrivial to implement, but most prior works provide no machine-checked proofs that their optimizations preserve program semantics.

This thesis proposes to narrow the gaps through a framework built around the Conquord language, a language designed for data-intensive application programming, under the assumption that the queries that will be executed on the database are available at compile time. Conquord has language primitives common to imperative programming languages useful for writing application code, and it also has primitives that help express the interpretations of database tables as lists, bags, or sets. The source-to-source compiler for Conquord is capable of database-style optimizations such as applying relational-algebra rewrite rules. The compiler can also create auxiliary data structures known as indexes for a table and transform the application code to utilize the indexes for better runtime performance. Most notably, it relaxes the interpretation of a table where possible, e.g. from a list to a bag, and that enables the use of indexes that operate on the alternative interpretation. To reason about the soundness of the optimizations, we have a deep embedding of Conquord in the Rocq theorem prover, and Conquord programs have deterministic semantics. All the transformations in the compiler are formally verified to preserve the semantics of the programs. The source code for Conquord is available at https://github.com/mit-plv/fiat2. To evaluate Conquord's design, we parameterized our compiler over optimization hints and optimized a range of example programs using the compiler, with the respective optimization hints generated by a large-language model. The compiler was formally verified to preserve program semantics when instantiated with arbitrary hints. We translated the Conquord programs before and after the optimizations to Python code for execution and found that the optimized Conquord programs achieved the expected asymptotic speedups compared to the input programs. The optimized Conquord programs were also compared with Python programs that access SQLite databases with similar database optimizations. The concrete runtime of each optimized program was comparable to that of the corresponding Python-SQL program.

One notable precursor of this work is Fiat [6], a Rocq framework for derivation of programs from specifications by stepwise refinement. SQL-style queries are one natural form of specification, which can be handled in a unified framework based on a nondeterminism monad. As one proceeds translating a specification into a performant implementation, nondeterminism progressively collapses, for instance fixing the order of result rows coming out of a query to match what is easy to compute from the underlying data structure. On the one hand, working in a special monad adds complexity to a framework and brings compile-time costs in usability and performance, indicating the appeal of a framework that uses rewrites based on simple equality. On the other hand, when we consider information security, there are serious downsides of refinement in general.

In fact, it is widely known that refinement does not interact naturally with *information-flow* restrictions. As a concrete example, imagine that our data-intensive application is an email server, written out initially with queries and updates over tables representing different users' mailboxes. Let us also imagine a query to show a user's five most recent email messages, which is inherently ambiguous, given that a flood of messages might arrive at the same time.

It is dangerous to give the compiler flexibility to resolve that ambiguity in any way compatible with refinement of nondeterministic programs. For instance, the choice of messages to show or the order to show them in might depend on consulting other users' mailboxes – e.g., Alice sees an order dependent on whether Bob has received messages from HotCRP about reviewing Alice's paper. That example is contrived and unlikely to arise in an honestly developed compiler, but we can imagine optimizations like sharing storage when multiple users receive the same message, which could lead to orders leaking information about email to other users. The Conquord approach does not suffer from the same weakness because programs are deterministic. That is, they have unique correct computation results, and the compiler has no leeway to tinker with them, so no transformations applied by the compiler can cause the resulting program to leak secrets.

The systematic solution adopted by Conquord is to commit to deterministic semantics of source programs, preserved by all compiler phases, but achieving good performance via algebraic rewrites becomes nonobvious. All prior approaches we are aware of either apply only to SQL-like code or are phrased using nondeterminism. We had to develop an important new design pattern for tracking *context* around queries, noting whether downstream code has the ability to distinguish differences in order or multiplicity. When downstream code is order-agnostic, we can return query results in the order most natural given the underlying data structure; while given more discriminating downstream code, we should sort the results. An alternative approach to drive such decisions would have been to rely on static analysis, perhaps with a type system, to track context. However, we found it aesthetically pleasing, and simplifying of our implementation, to track context using *casts* between consistency levels (list, bag, set) and *pushdown* rules for casts, already familiar from classic database optimization.

1.1 A Motivating Example

Let us illustrate the approach with an example that introduces a dictionary-based index to accelerate a filter query. Our index maps keys to *bags* of values, which is safe only because the original query is invariant to the order of the values. Using bags instead of e.g. lists simplifies the concrete implementation of the index in a lower-level language.

Suppose we have two tables as two mutable variables: **departments** associates department IDs and department names, and **responses** stores survey results each consisting of an employee name, department ID, and feedback string. We write a Conquord program that joins the tables and produces a string containing the employee names, department names, and feedback:

```
let mut all_feedback := "" in
foreach r in
  sort [
    r1 ← !departments,
    r3 ← [
      r2 ← !responses,
      check (r2.department_id = r1.department_id),
      ret r2 ],
```

```
    ret {name: r3.name, department: r1.department_name, feedback: r3.feedback} ]
do
  all_feedback :=
    r.name ++ " from " ++ r.department ++
    " wrote: " ++ r.feedback ++ "\n" ++ !all_feedback
end
```

Now we show how this program is optimized by our system. The subexpression rooted at the sorting operation is parsed to:

$$\begin{array}{l}\mathsf{sort}_{\mathsf{list}} \\ (\mathsf{flatmap}_{\mathsf{list}}\ \textbf{departments}\ (r_1 \rightarrow \\ \quad \mathsf{flatmap}_{\mathsf{list}}\ (\mathsf{flatmap}_{\mathsf{list}}\ \textbf{responses}\ (r_2 \rightarrow \\ \quad\quad \mathsf{if}\ (r_2.\mathsf{department\text{-}id} == r_1.\mathsf{department\text{-}id})\ \mathsf{then}\ [r_2]\ \mathsf{else}\ [])\ (r_3 \rightarrow \\ \quad\quad\quad [\{\ \mathsf{name} : r_3.\mathsf{name}; \mathsf{department} : r_1.\mathsf{department\text{-}name}; \mathsf{feedback} : r_3.\mathsf{feedback}\ \}])))\end{array}$$

After identifying relational-algebra operations (Chapter 4), this expression becomes:

$$\begin{array}{l}\mathsf{sort}_{\mathsf{list}} \\ (\mathsf{flatmap}_{\mathsf{list}}\ \textbf{departments}\ (r_1 \rightarrow \\ \quad \mathsf{proj}_{\mathsf{list}}\ (\mathsf{filter}_{\mathsf{list}}\ \textbf{responses}\ (r_2 \rightarrow r_2.\mathsf{department\text{-}id} == r_1.\mathsf{department\text{-}id}))\ (r_3 \rightarrow \\ \quad\quad \{\ \mathsf{name} : r_3.\mathsf{name}; \mathsf{department} : r_1.\mathsf{department\text{-}name}; \mathsf{feedback} : r_3.\mathsf{feedback}\ \})))\end{array}$$

The sorting operation implies that the semantics of this expression is invariant to the order of the results of the flatmaps. We take advantage of this property by introducing a bag-of cast that further rewriting will push into the expression:

$$\begin{array}{l}\mathsf{sort}_{\mathsf{bag}} \\ (\mathsf{bag\text{-}of}\ (\mathsf{flatmap}_{\mathsf{list}}\ \textbf{departments}\ (r_1 \rightarrow \\ \quad \mathsf{proj}_{\mathsf{list}}\ (\mathsf{filter}_{\mathsf{list}}\ \textbf{responses}\ (r_2 \rightarrow r_2.\mathsf{department\text{-}id} == r_1.\mathsf{department\text{-}id}))\ (r_3 \rightarrow \\ \quad\quad \{\ \mathsf{name} : r_3.\mathsf{name}; \mathsf{department} : r_1.\mathsf{department\text{-}name}; \mathsf{feedback} : r_3.\mathsf{feedback}\ \}))))\end{array}$$

Collection-annotation-pushdown rules (Chapter 5) push the cast down to the **responses** table:

$$\begin{array}{l}\mathsf{sort}_{\mathsf{bag}} \\ (\mathsf{flatmap}_{\mathsf{bag}}\ \textbf{departments}\ (r_1 \rightarrow \\ \quad \mathsf{proj}_{\mathsf{bag}}\ (\mathsf{filter}_{\mathsf{bag}}\ (\mathsf{bag\text{-}of}\ \textbf{responses})\ (r_2 \rightarrow \\ \quad\quad r_2.\mathsf{department\text{-}id} == r_1.\mathsf{department\text{-}id}))\ (r_3 \rightarrow \\ \quad\quad\quad \{\ \mathsf{name} : r_3.\mathsf{name}; \mathsf{department} : r_1.\mathsf{department\text{-}name}; \mathsf{feedback} : r_3.\mathsf{feedback}\ \})))\end{array}$$

The transformations above revealed the opportunity to create an index that is a dictionary mapping each department-id value to the bag of tuples in the **responses** table with the

department-id attribute equal to that value. Index-creation rules (Chapter 6) create this index, store it in **responses**.aux, and rewrite the expression to use the index:

```
sort_bag
(flatmap_bag departments (r1 →
  proj_bag (
    optmatch (dict-lookup (responses.aux) r1.department-id)
    empty_bag
    (b → b)) (r3 →
      { name : r3.name; department : r1.department-name; feedback : r3.feedback })))
```

Rather than iterating through the entire **responses** table for every department-id value, the lookup operation retrieves each tuple from **responses** only once, so the final program is asymptotically faster than the original.

1.2 Contributions

The contributions of this thesis are summarized below.

- We formalized different kinds of database-style optimizations as term rewrites in a language with imperative programming features.
- We proposed the use of collection annotations to denote the possibility of using list, bag and set semantics and transformations that propagate the annotations into the abstract syntax tree. The transformations help the compiler discover subprograms where the interpretation of collections may be relaxed without losing determinism in the semantics of the whole program, thereby exposing new optimization opportunities.
- We expressed the creation and use of indexes as a sequence of related rewrite rules, and the proof of their soundness is conceptually non-trivial.
- We developed a parameterized compiler that takes optimization hints as parameters and used it to optimize example programs with optimization hints generated by a large-language model. The optimized programs achieved the expected speedups. The performance of the optimized programs was comparable to that of similar programs written in Python with access to a SQLite database.
- All the transformations described in this thesis are formally verified for semantic preservation.

The remainder of this thesis is organized as follows. Chapter 2 summarizes the key ideas in this thesis; Chapter 3 presents the syntax, semantics and typing rules of Conquord; Chapters 4, 5 and 6 elaborate on the three categories of rewrites in the compiler; Chapter 7 describes the setup of the experiments used to evaluate the performance of Conquord; Chapter 8 discusses the experimental results; Chapter 9 discusses related work; Chapter 10 concludes this thesis.

Chapter 2

Key Ideas

2.1 From Query Comprehension to Relational Algebra

List-comprehension syntax is found in many programming languages including Python and Haskell, and variations of comprehension syntax have been proposed for database languages as an alternative to the SQL syntax [7,8]. Query comprehensions can be parsed to monadic operations on lists, with flatmap as the monadic bind operator. We use query comprehensions in the concrete syntax of Conquord for their natural integration with the rest of the language, and flatmap is therefore a primitive of the Conquord language. After parsing the input program, the compiler identifies subexpressions that correspond to the filter, projection, and join operations in relational algebra. That normalization facilitates the subsequent database-style optimizations, which often involve relational-algebra operations.

For instance, the following expression is written in the query-comprehension syntax. It fetches each tuple from the **shirts** table and returns it in the output if its color attribute is `"white"`.

```
[ x ← !shirts, check(x.color = "white"), ret x ]
```

The query comprehension is parsed to the following abstract syntax tree.

$$\mathsf{flatmap}_{\mathsf{list}}\ \textbf{shirts}\ (x \rightarrow \mathsf{if}\ (x.\mathsf{color} == \texttt{"white"})\ \mathsf{then}\ [x]\ \mathsf{else}\ [])$$

The compiler can then transform the expression (Chapter 4) to a semantically equivalent relational-algebra operation on **shirts**.

$$\mathsf{filter}_{\mathsf{list}}\ \textbf{shirts}\ (x \rightarrow x.\mathsf{color} == \texttt{"white"})$$

2.2 Interpretation of Collections

In the original relational-database theory, every relation is interpreted as a set of tuples [9]. SQL uses bag semantics by default, so a database table can contain duplicate tuples. However, the bag semantics permit nondeterministic outputs from SQL queries when the order is not specified, and the nondeterminism complicates the definitions of program equivalence and transformation soundness. To mitigate the problem, we use list semantics in the subset of

the Conquord language exposed to the programmer. A potential drawback of this approach is that some optimizations rely on the interpretation of a database table as a bag or a set of tuples, and interpreting database tables as lists everywhere in the program rules out such optimizations. The Conquord compiler recovers opportunities for alternative interpretations of collections by pattern matching for certain local constructs. For example, sorting a list is equivalent to interpreting the list as a bag and sorting the bag. Such interpretations can then be propagated to operations on collections like flatmap, filter, and join.

Continuing from the example above, if the expression is under a sorting operation, then this stage of compilation will recognize that it can be interpreted as a bag. The expression

$$\mathsf{sort}_{\mathsf{list}}\ (\mathsf{filter}_{\mathsf{list}}\ \textbf{shirts}\ (x \to x.\mathsf{color} == \texttt{"white"}))$$

gets transformed to

$$\mathsf{sort}_{\mathsf{bag}}\ (\mathsf{bag\text{-}of}\ (\mathsf{filter}_{\mathsf{list}}\ \textbf{shirts}\ (x \to x.\mathsf{color} == \texttt{"white"})))$$

and the bag annotation gets pushed (Chapter 5) into the subexpression to give

$$\mathsf{sort}_{\mathsf{bag}}\ (\mathsf{filter}_{\mathsf{bag}}\ (\mathsf{bag\text{-}of}\ \textbf{shirts})\ (x \to x.\mathsf{color} == \texttt{"white"}))$$

2.3 Creating and Using Indexes

Creating the right indexes for a database table and using them for queries is an important way to improve the runtime efficiency of the queries. Many prior works have studied how to make better choices based on the queries and workload [3,10–19]. Rather than considering the heuristics used to choose the optimal combination of indexes, this work assumes the knowledge of a set of indexes that the compiler should create for a table. Such information can be derived through static analysis of the application code, and the choice of indexes will not compromise the soundness of the compilation pipeline as long as the pipeline is proven to be sound for any of the indexes it may create for a table. The Conquord compiler has a sequence of program transformations (Chapter 6) that create the indexes and rewrite the application code to use the created indexes, and the transformations together preserve the semantics of programs.

One way to optimize the example program above is to use a dictionary-based index that maps each color value to the tuples in the **shirts** table where the color attribute takes that value. The filter operation can then be replaced by the corresponding lookup operation on the index. Since the lookup only needs to return a bag of tuples, the implementation of that index does not need to preserve the order of the tuples in **shirts**. As such, more optimization opportunities can arise due to the relaxed interpretation of collections.

2.4 Limitations

The design of the Conquord compiler assumes all the application code that will run on the data is available at compile time. This assumption is shared with much of the prior work on layout optimization and still captures an interesting space of data-processing queries,

allowing the compiler to apply database-style optimizations based on the code, automatically and effectively. Moreover, we assume that database tables are loaded fully into memory during program execution, so the compiler does not handle a storage hierarchy in the layout-optimization problem. This assumption is supported by an increasing number of real database systems focusing entirely on in-memory workloads [20]. Finally, this thesis focuses on the construction of transformation pipelines with soundness proofs rather than the heuristics for choosing the optimal transformations. The transformations described in this thesis are source-to-source transformations on the high-level Conquord language, and compilation to executable code is left for future work.

Chapter 3

Conquord

In this chapter, we describe the syntax and semantics of Conquord.

3.1 Types and Abstract Syntax

Conquord is a language with static type checking. Figure 3.1 shows the types in the Conquord language. A record type contains a list of pairs of tags (attribute names) and types sorted by a total order over the tags. They are well-formed only when tags are distinct. The semantics of a record expression are similar. Hence, the order of entries in a record expression does not matter, and the two well-typed expressions $\{\ \mathsf{tag\text{-}1} : e_1;\ \mathsf{tag\text{-}2} : e_2\ \}$ and $\{\ \mathsf{tag\text{-}2} : e_2;\ \mathsf{tag\text{-}1} : e_1\ \}$ have the same record type and value.

$$
\begin{array}{lrcl}
\text{Record type entry} & \mathcal{H} & ::= & \mathsf{tag} : \mathcal{T} \\
\text{Type} & \mathcal{T} & ::= & \mathsf{int} \mid \mathsf{bool} \mid \mathsf{string} \mid () \\
 & & & \mid \mathsf{option}\ \mathcal{T} \mid \mathsf{list}\ \mathcal{T} \mid \mathsf{bag}\ \mathcal{T} \mid \mathsf{set}\ \mathcal{T} \\
 & & & \mid \mathsf{dict}\ \mathcal{T}\ \mathcal{T} \mid \{\ \mathcal{H}; \ldots\ \}
\end{array}
$$

Figure 3.1: Conquord types

Figure 3.2 shows a fragment of the abstract syntax of Conquord. For brevity, we omit most unary and binary operations that are Conquord expressions. The different kinds of collections are built into the Conquord language design through a totally ordered set of three collection categories, which are used to annotate other primitives. The atoms in the Conquord language include integers n, Booleans b, and string literals s, as well as empty collections and some other common primitives. We write $\mathsf{empty}_{\mathsf{list}}$ as [] and $\mathsf{insert}_{\mathsf{list}}\ e_2\ e_1$ as $e_1 :: e_2$. The abstract syntax of the empty collections and the none expression permits an optional type annotation, which can be helpful for type inference in the bidirectional typechecker [21]. Conquord expressions include both immutable variables and mutable variables. Mutable variables are useful for expressing updates to database tables, and we use bold symbols to denote them. The Conquord language is first-order, with a new variable binding x in the expression e expressed as $(x \to e)$. There is structural recursion through the fold operation on lists but no general recursion. Record construction, sorting, conversion from lists to bags and sets, and basic relational-algebra operations are also available as Conquord expressions,

but the concrete syntax exposed to the programmer does not involve bags or sets. Conquord commands include common imperative constructs, but there is no while loop, and the foreach constructs are guaranteed to terminate.

$$
\begin{array}{lrcl}
\text{Collection} & \chi & ::= & \mathsf{set} \mid \mathsf{bag} \mid \mathsf{list} \\
\text{Record entry} & \mathcal{R} & ::= & \mathsf{tag} : \mathcal{E} \\
\text{Expr} & \mathcal{E} & ::= & x \mid \mathbf{x} \mid n \mid b \mid s \mid \mathsf{none} \mid \mathsf{none}^t \mid \mathsf{empty\text{-}dict} \\
 & & & \mid \mathsf{empty}_\chi \mid \mathsf{empty}^t_\chi \mid () \\
 & & & \mid \mathcal{E} < \mathcal{E} \mid \mathcal{E} + \mathcal{E} \\
 & & & \mid \mathsf{some}\ \mathcal{E} \mid \mathsf{length}\ \mathcal{E} \mid \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ \mathcal{E}\ \mathsf{else}\ \mathcal{E} \\
 & & & \mid \mathsf{let}\ x \leftarrow \mathcal{E}\ \mathsf{in}\ \mathcal{E} \mid \mathsf{dict\text{-}insert}\ \mathcal{E}\ (\mathcal{E} \mapsto \mathcal{E}) \\
 & & & \mid \mathsf{dict\text{-}lookup}\ \mathcal{E}\ \mathcal{E} \mid \mathsf{optmatch}\ \mathcal{E}\ \mathcal{E}\ (x \to \mathcal{E}) \\
 & & & \mid \mathsf{sort}_\chi\ \mathcal{E} \mid \mathsf{sum}\ \mathcal{E} \mid \mathsf{count}\ \mathcal{E} \mid \mathsf{min}\ \mathcal{E} \\
 & & & \mid \mathsf{flatmap}_\chi\ \mathcal{E}\ (x \to \mathcal{E}) \mid \mathsf{fold}\ \mathcal{E}\ \mathcal{E}\ (x, y \to \mathcal{E}) \\
 & & & \mid \{\ \mathcal{R}; \ldots\ \} \mid \mathcal{E}.\mathsf{tag} \mid \mathsf{bag\text{-}of}\ \mathcal{E} \mid \mathsf{set\text{-}of}\ \mathcal{E} \\
 & & & \mid \mathsf{insert}_\chi\ \mathcal{E}\ \mathcal{E} \\
 & & & \mid \mathsf{filter}_\chi\ \mathcal{E}\ (x \to \mathcal{E}) \\
 & & & \mid \mathsf{join}_\chi\ \mathcal{E}\ \mathcal{E}\ (x, y \to \mathcal{E}\ \mathsf{if}\ \mathcal{E}) \\
 & & & \mid \mathsf{proj}_\chi\ \mathcal{E}\ (x \to \mathcal{E}) \\
\text{Command} & \mathcal{C} & ::= & \mathsf{skip} \mid \mathcal{C}; \mathcal{C} \mid \mathsf{let\ mut}\ \mathbf{x} \coloneqq \mathcal{E}\ \mathsf{in}\ \mathcal{C} \mid \mathbf{x} \coloneqq \mathcal{E} \\
 & & & \mid \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ \mathcal{C}\ \mathsf{else}\ \mathcal{C}\ \mathsf{end} \mid \mathsf{foreach}\ x\ \mathsf{in}\ \mathcal{E}\ \mathsf{do}\ \mathcal{C}\ \mathsf{end}
\end{array}
$$

Figure 3.2: Abstract syntax of Conquord

The construct $\mathsf{flatmap}_c$ for $c \in \{\mathsf{list}, \mathsf{bag}, \mathsf{set}\}$ represents flat-map operations on the respective collections, likewise for other constructs with collection subscripts. We point out once again that the concrete syntax exposed to the programmer assumes list semantics, while bags and sets are part of the internal abstract syntax tree known to the compiler. In the rest of this thesis, we will omit the subscript when referring to operations on lists.

3.2 Type System

Figure 3.3 shows the typing rules on Conquord atoms. The type well-formedness judgment $\vdash t$ enforces that the entries of each record type in t are ordered and there is no duplication among the tags in a record type.

The typing judgment on Conquord expressions is defined by inductive rules, some of which are shown in Figure 3.4. The typing environment for mutable variables is denoted by Δ, and that for immutable variables is denoted by Γ.

The well-typedness judgment on Conquord commands can be defined using the typing rules on Conquord expressions, as shown in Figure 3.5. We have developed a bidirectional typechecker and proven its soundness with respect to the typing rules.

$$\frac{}{\Delta; \Gamma \vdash n : \mathsf{int}} \qquad \frac{}{\Delta; \Gamma \vdash b : \mathsf{bool}} \qquad \frac{}{\Delta; \Gamma \vdash s : \mathsf{string}} \qquad \frac{}{\Delta; \Gamma \vdash () : ()}$$

$$\frac{\vdash t}{\Delta; \Gamma \vdash \mathsf{none} : \mathsf{option}\ t} \qquad \frac{\vdash t}{\Delta; \Gamma \vdash \mathsf{none}^t : \mathsf{option}\ t} \qquad \frac{\vdash t_k \qquad \vdash t_v}{\Delta; \Gamma \vdash \mathsf{empty\text{-}dict} : \mathsf{dict}\ t_k\ t_v}$$

$$\frac{\vdash t}{\Delta; \Gamma \vdash [] : \mathsf{list}\ t} \qquad \frac{\vdash t}{\Delta; \Gamma \vdash []^t : \mathsf{list}\ t} \qquad \frac{\vdash t}{\Delta; \Gamma \vdash \mathsf{empty}_{\mathsf{bag}} : \mathsf{bag}\ t} \qquad \frac{\vdash t}{\Delta; \Gamma \vdash \mathsf{empty}^t_{\mathsf{bag}} : \mathsf{bag}\ t}$$

$$\frac{\vdash t}{\Delta; \Gamma \vdash \mathsf{empty}_{\mathsf{set}} : \mathsf{set}\ t} \qquad \frac{\vdash t}{\Delta; \Gamma \vdash \mathsf{empty}^t_{\mathsf{set}} : \mathsf{set}\ t}$$

Figure 3.3: Typing rules on Conquord atoms

3.3 Big-Step Operational Semantics and Type-Soundness Theorems

Figure 3.6 shows the values used to define the semantics of Conquord expressions. Integer literals are denoted by n, Booleans by b, and strings by s. Like record types, a record value also has its entries sorted by the tags. Similarly, our implementation of dictionary, bag, and set values in Rocq ensures that equivalent values have unique and canonical representations. A dictionary is represented by a finite map d, while lists, bags and sets are represented using vectors of values $\vec{l}$ with the appropriate properties.

Some of the typing rules on values are shown in Figure 3.7.

The semantics of Conquord expressions are defined by an interpretation function that maps expressions to values. Let $[\![e]\!]_{\phi,\psi}$ denote the interpretation of the expression e in the value environment ϕ for mutable variables and ψ for immutable variables. We formalized and proved the type-soundness theorem for Conquord as stated in Theorem 1 in the Rocq theorem prover.

Theorem 1 (Type soundness for expressions) *For any typing environments Δ and Γ, Conquord expression e, and Conquord type t, if e has type t under Δ and Γ, then under any value environments ϕ respecting Δ and ψ respecting Γ, e is interpreted to a value of type t. Formally,*

$$\forall \Delta,\ \Gamma,\ e,\ t.\ \Delta; \Gamma \vdash e : t \implies \forall \phi,\ \psi.\ \Delta \vdash \phi \implies \Gamma \vdash \psi \implies \vdash [\![e]\!]_{\phi,\psi} : t$$

Similarly, $[\![c]\!]_{\phi,\psi}$ denotes the interpretation of the command c in the value environments, and it evaluates to a value environment for mutable variables after executing c from a state with ϕ and ψ. Theorem 2 is also formalized and proven in Rocq.

Theorem 2 (Type soundness for commands) *For any typing environments Δ and Γ and Conquord command c, if c is well-typed under Δ and Γ, then under any value environments*

$$\frac{\Delta[\mathbf{x}] = t}{\Delta; \Gamma \vdash \mathbf{x} : t} \qquad \frac{\Gamma[x] = t}{\Delta; \Gamma \vdash x : t} \qquad \frac{\Delta; \Gamma \vdash e_1 : \mathsf{bool} \qquad \Delta; \Gamma \vdash e_2 : t \qquad \Delta; \Gamma \vdash e_3 : t}{\Delta; \Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : t}$$

$$\frac{\Delta; \Gamma \vdash e_1 : t_1 \qquad \Delta; \Gamma, x : t_1 \vdash e_2 : t_2}{\Delta; \Gamma \vdash \mathsf{let}\ x \leftarrow e_1\ \mathsf{in}\ e_2 : t_2}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \mathsf{list}\ t_1 \qquad \Delta; \Gamma \vdash e_2 : t_2 \qquad \Delta; \Gamma, x : t_1, y : t_2 \vdash e_3 : t_2}{\mathsf{fold}\ e_1\ e_2\ (x, y \to e_3) : t_2}$$

$$\frac{\texttt{no-dup}\ [\mathsf{tag\text{-}1}, \dots] \qquad \Delta; \Gamma \vdash e_1 : t_1 \qquad \dots}{\Delta; \Gamma \vdash \{\ \mathsf{tag\text{-}1} : e_1, \dots\ \} : \{\ \mathsf{tag\text{-}1} : t_1, \dots\ \}}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \mathsf{list}\ t_1 \qquad \Delta; \Gamma, x : t_1 \vdash e_2 : \mathsf{list}\ t_2}{\Delta; \Gamma \vdash \mathsf{flatmap}_{\mathsf{list}}\ e_1\ (x \to e_2) : \mathsf{list}\ t_2} \qquad \frac{\Delta; \Gamma \vdash e : \mathsf{list}\ t}{\Delta; \Gamma \vdash \mathsf{bag\text{-}of}\ e : \mathsf{bag}\ t}$$

$$\frac{\Delta; \Gamma \vdash e : \mathsf{bag}\ t}{\Delta; \Gamma \vdash \mathsf{sort}_{\mathsf{bag}}\ e : \mathsf{list}\ t} \qquad \frac{\Delta; \Gamma \vdash e_d : \mathsf{dict}\ t_k\ t_v \qquad \Delta; \Gamma \vdash e_k : t_k \qquad \Delta; \Gamma \vdash e_v : t_v}{\Delta; \Gamma \vdash \mathsf{dict\text{-}insert}\ e_d\ (e_k \mapsto e_v) : \mathsf{dict}\ t_k\ t_v}$$

$$\frac{\Delta; \Gamma \vdash e_d : \mathsf{dict}\ t_k\ t_v \qquad \Delta; \Gamma \vdash e_k : t_k}{\Delta; \Gamma \vdash \mathsf{dict\text{-}lookup}\ e_d\ e_k : \mathsf{option}\ t_v}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \mathsf{option}\ t_1 \qquad \Delta; \Gamma \vdash e_2 : t_2 \qquad \Delta; \Gamma, x : t_1 \vdash e_3 : t_2}{\Delta; \Gamma \vdash \mathsf{optmatch}\ e_1\ e_2\ (x \to e_3) : t_2}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \mathsf{list}\ t \qquad \Delta; \Gamma \vdash e_2 : t}{\Delta; \Gamma \vdash \mathsf{insert}_{\mathsf{list}}\ e_1\ e_2 : \mathsf{list}\ t} \qquad \frac{\Delta; \Gamma \vdash e : \mathsf{set\ int}}{\Delta; \Gamma \vdash \mathsf{min}\ e : \mathsf{option\ int}}$$

$$\frac{\Delta; \Gamma \vdash e : \mathsf{list}\ t \qquad \Delta; \Gamma, x : t \vdash p : \mathsf{bool}}{\Delta; \Gamma \vdash \mathsf{filter}_{\mathsf{list}}\ e\ (x \to p) : \mathsf{list}\ t} \qquad \frac{\Delta; \Gamma \vdash e_1 : \mathsf{list}\ t_1 \qquad \Delta; \Gamma, x : t_1 \vdash e_2 : t_2}{\Delta; \Gamma \vdash \mathsf{proj}_{\mathsf{list}}\ e_1\ (x \to e_2) : \mathsf{list}\ t_2}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e_1 : \mathsf{list}\ t_1 \\ \Delta; \Gamma \vdash e_2 : \mathsf{list}\ t_2 \qquad \Delta; \Gamma, x_1 : t_1, x_2 : t_2 \vdash p : \mathsf{bool} \qquad \Delta; \Gamma, x_1 : t_1, x_2 : t_2 \vdash e_3 : t_3\end{array}}{\Delta; \Gamma \vdash \mathsf{join}_{\mathsf{list}}\ e_1\ e_2\ (x_1, x_2 \to e_3\ \mathsf{if}\ p) : \mathsf{list}\ t_3}$$

Figure 3.4: Typing rules on Conquord expressions, omitting some rules analogous to the ones presented here

ϕ respecting Δ and ψ respecting Γ, the value environment resulting from the execution of c also respects Δ. Formally,

$$\Delta; \Gamma \vdash c \implies \forall \phi,\ \psi.\ \Delta \vdash \phi \implies \Gamma \vdash \psi \implies \Delta \vdash [\![c]\!]_{\phi,\psi}$$

$$\frac{}{\Delta;\Gamma \vdash \mathsf{skip}} \qquad \frac{\Delta;\Gamma \vdash c_1 \qquad \Delta;\Gamma \vdash c_2}{\Delta;\Gamma \vdash c_1; c_2} \qquad \frac{\Delta;\Gamma \vdash e : t \qquad \Delta, \mathbf{x} : t;\Gamma \vdash c}{\Delta;\Gamma \vdash \mathsf{let\ mut}\ \mathbf{x} \coloneqq e\ \mathsf{in}\ c}$$

$$\frac{\Delta[\mathbf{x}] = t \qquad \Delta;\Gamma \vdash e : t}{\Delta;\Gamma \vdash \mathbf{x} \coloneqq e} \qquad \frac{\Delta;\Gamma \vdash e : \mathsf{bool} \qquad \Delta;\Gamma \vdash c_1 \qquad \Delta;\Gamma \vdash c_2}{\Delta;\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2}$$

$$\frac{\Delta;\Gamma \vdash e : \mathsf{list}\ t \qquad \Delta;\Gamma, x : t \vdash c}{\Delta;\Gamma \vdash \mathsf{foreach}\ x\ \mathsf{in}\ e\ \mathsf{do}\ c\ \mathsf{end}}$$

Figure 3.5: Typing rules on Conquord commands

List	L	::=	$\mathsf{list}\ \vec{l}$
Bag	B	::=	$\mathsf{bag}\ \vec{l}$
Set	S	::=	$\mathsf{set}\ \vec{l}$
Dictionary	D	::=	$\mathsf{dict}\ d$
Value	v	::=	$n \mid b \mid s \mid \mathsf{none} \mid \mathsf{some}\ v \mid \{\mathsf{tag} : v; \dots\} \mid () \mid D \mid L \mid B \mid S$

Figure 3.6: Values for defining the semantics of Conquord

$$\frac{}{\vdash n : \mathsf{int}} \qquad \frac{}{\vdash b : \mathsf{bool}} \qquad \frac{}{\vdash s : \mathsf{string}} \qquad \frac{\vdash v : t}{\vdash \mathsf{some}\ v : \mathsf{option}\ t} \qquad \frac{}{\vdash \mathsf{none} : \mathsf{option}\ t}$$

$$\frac{\forall v.\ v \in \vec{l} \implies \vdash v : t}{\vdash \mathsf{list}\ \vec{l} : \mathsf{list}\ t} \qquad \frac{\texttt{sorted}\ \vec{l} \qquad \forall v.\ v \in \vec{l} \implies \vdash v : t}{\vdash \mathsf{bag}\ \vec{l} : \mathsf{bag}\ t}$$

$$\frac{\texttt{no-dup}\ \vec{l} \qquad \texttt{sorted}\ \vec{l} \qquad \forall v.\ v \in \vec{l} \implies \vdash v : t}{\vdash \mathsf{set}\ \vec{l} : \mathsf{set}\ t}$$

$$\frac{\forall k,\ v.\ D[k] = v \implies \vdash k : t_k\ \wedge\ \vdash v : t_v}{\vdash D : \mathsf{dict}\ t_k\ t_v} \qquad \frac{\texttt{no-dup}\ [\mathsf{tag\text{-}1}, \dots] \qquad \vdash v_1 : t_1, \dots}{\vdash \{\mathsf{tag\text{-}1} : v_1; \dots\} : \{\mathsf{tag\text{-}1} : t_1, \dots\}}$$

Figure 3.7: Typing rules on Conquord values

3.4 Concrete Syntax

This section briefly introduces the concrete syntax of Conquord, i.e., the syntax that a programmer can use to write programs in this language. While the concrete syntax is not essential to the development of the ideas in this thesis, it offers conciseness and familiarity that make the discussion of example programs more pleasant. We developed a parser that parses the concrete syntax to a fragment of the abstract syntax. The concrete syntax of

imperative constructs is similar to that found in mainstream programming languages, and the concrete syntax of commands corresponds closely with the presentation of the abstract syntax in Figure 3.2. The reading of a mutable variable **x** is written as the dereferencing of the variable, `!x`. The pattern matching of an expression e that takes an option value is written as `optmatch e as x with a:b end`. The notation corresponds to the (optmatch e a $(x \rightarrow b)$) construct in the abstract syntax. If e is evaluated to none, then the value of the entire expression is the value of a. Otherwise, e is evaluated to (some v) for some value v, and the value of the entire expression can be obtained by evaluating b with v assigned to x. The concrete syntax of most other expressions is straightforward, so we focus on the parsing of query-comprehension notations for the rest of this section. The correspondence between the notations and the abstract syntax will shed light on the intuition behind some rewrite rules presented in Chapter 4.

Query comprehensions are notations for the list monad, with flatmap as the bind operator of the monad, so the notations are readily converted to flatmap operations. There are three main constructs in query-comprehension notations, as shown in Figure 3.8. They occur inside square brackets and are separated by commas. The constructs associate to the right.

`x ← e, ...`	$\Rightarrow$ flatmap e $(x \rightarrow ...)$
`check(e), ...`	$\Rightarrow$ if e then ... else []
`ret e`	$\Rightarrow [e]$

Figure 3.8: Rules for parsing query-comprehension notations in the concrete syntax of Conquord

Applying the rules to parse the following expression,

```
[x ← !tbl, check(cond), ret e]
```

we obtain

flatmap **tbl** $(x \rightarrow$ if *cond* then $[e]$ else $[])$

For brevity, we will refer to expressions that result from the parsing of query-comprehension notations as query-comprehension expressions in the rest of this thesis.

Chapter 4

Rewrites Related To Relational-Algebra Operations

A Conquord program expresses a relational database table as a list of records. Relational operations on a table are written as query comprehensions. Hence, database-style optimizations that are derived from equivalences in relational algebra correspond to rewrites of query-comprehension expressions. While some of the optimizations are easily expressed as transformations between query-comprehension expressions, it is often cleaner to reason about semantic preservation when the expressions are in the form of relational-algebra constructs. Therefore, our compiler is not only capable of transformations between query-comprehension expressions, but it can also rewrite expressions into their relational-algebra equivalents to facilitate the subsequent optimizations in the pipeline. This rewriting is incomplete—not all expressions equivalent to relational-algebra operations will be recognized as such, but expressions that are not transformed remain valid expressions of Conquord and can be evaluated as usual. Using filter pushdown as a core idea, we illustrate how rewrites before and after the conversion to relational-algebra constructs can improve program performance and enable further optimizations.

4.1 Identifying Relational-Algebra Operations

We start by describing rewrite rules that identify expressions equivalent to relational-algebra constructs, as these rules motivate the other rewrites presented in this chapter. The rewrites between query-comprehension expressions, which usually precede the application of these rewrite rules, should aim to create expressions that match the left-hand sides of these rules. The rewrites after the application of these rules follow naturally from equivalences between relational-algebra expressions. Figure 4.1 shows some rewrite rules transforming expressions into relational-algebra constructs.

While most rewrite rules in this thesis are written as equations, the compiler applies them as transformations from left to right.

We define the soundness of a transformation by Definitions 1 and 2.

Definition 1 (Soundness of expression transformations) *A transformation* $f : \mathsf{expr} \to \mathsf{expr}$ *is sound if and only if* $\forall \Delta,\ \Gamma,\ e,\ t.\ \Delta; \Gamma \vdash e : t \implies \Delta; \Gamma \vdash f\ e : t \ \wedge\ \forall \phi,\ \psi.\ \Delta \vdash$

$$\frac{\Delta; \Gamma \vdash e : \mathsf{list}\ t \qquad \Delta; \Gamma, x : t \vdash p : \mathsf{bool}}{\mathsf{flatmap}_{\mathsf{list}}\ e\ (x \to \mathsf{if}\ p\ \mathsf{then}\ [x]\ \mathsf{else}\ []) = \mathsf{filter}_{\mathsf{list}}\ e\ (x \to p)}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \mathsf{list}\ t_1 \qquad \Delta; \Gamma, x : t_1 \vdash e_2 : t_2}{\mathsf{flatmap}_{\mathsf{list}}\ e_1\ (x \to [e_2]) = \mathsf{proj}_{\mathsf{list}}\ e_1\ (x \to e_2)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e_1 : \mathsf{list}\ t_1 \qquad \Delta; \Gamma \vdash e_2 : \mathsf{list}\ t_2 \qquad \Delta; \Gamma, x_1 : t_1, x_2 : t_2 \vdash p : \mathsf{bool} \\ \Delta; \Gamma, x_1 : t_1, x_2 : t_2 \vdash e_3 : t_3 \qquad x_1 \neq x_2 \qquad x_1 \notin \mathsf{free\text{-}immuts}(e_2)\end{array}}{\mathsf{flatmap}_{\mathsf{list}}\ e_1\ (x_1 \to \mathsf{flatmap}_{\mathsf{list}}\ e_2\ (x_2 \to \mathsf{if}\ p\ \mathsf{then}\ [e_3]\ \mathsf{else}\ [])) = \mathsf{join}_{\mathsf{list}}\ e_1\ e_2\ (x_1, x_2 \to e_3\ \mathsf{if}\ p)}$$

Figure 4.1: Rules for identifying relational-algebra constructs in Conquord programs

$\phi \implies \Gamma \vdash \psi \implies [\![f\ e]\!]_{\phi,\psi} = [\![e]\!]_{\phi,\psi}$.

Definition 2 (Soundness of command transformations) *A transformation f : command $\to$ command is sound if and only if* $\forall \Delta,\ \Gamma,\ c.\ \Delta; \Gamma \vdash c \implies \Delta; \Gamma \vdash f\ c\ \wedge\ \forall \phi,\ \psi.\ \Delta \vdash \phi \implies \Gamma \vdash \psi \implies [\![f\ c]\!]_{\phi,\psi} = [\![c]\!]_{\phi,\psi}$.

All the transformations presented as equations in this thesis have soundness proofs checked in Rocq.

Theorem 3 *All transformations in Figure 4.1 are sound.*

4.2 Rewrites Between Query-Comprehension Expressions

Using the parsing rules in Figure 3.8, the expressions on the lefts of the rewrite rules in Figure 4.1 can be seen as the following query comprehensions.

```
x ← e, check(p), ret x
x ← e1, ret e2
x1 ← e1, x2 ← e2, check(p), ret e3
```

On the other hand, when writing a query, it is natural to bind variables to rows of multiple tables, perform multiple `check`s, and yield an arbitrary expression as the result. The compiler can rewrite such queries using the rules in Figure 4.2 to reveal the patterns we identified above. The equations in the rules correspond to the following equations in query-comprehension notation (omitting the conditions here).

```
x ← e1, check(p), ret e2 = check(p), x ← e1, ret e2
x ← e1, check(p), ret e2 = x ← [x ← e1, check(p), ret x], ret e2
check(p1), check(p2), ... = check(p2), check(p1), ...
```

When used with suitable heuristics, these rules help push the if constructs closer to the relevant flatmap constructs and extract subexpressions that can be identified as filter operations.

$$\frac{\Delta;\Gamma \vdash e_1 : \mathsf{list}\ t_1 \qquad \Delta;\Gamma \vdash p : \mathsf{bool} \qquad \Delta;\Gamma, x : t_1 \vdash e_2 : t_2}{\mathsf{flatmap}_{\mathsf{list}}\ e_1\ (x \to \mathsf{if}\ p\ \mathsf{then}\ [e_2]\ \mathsf{else}\ []) = \mathsf{if}\ p\ \mathsf{then}\ \mathsf{flatmap}_{\mathsf{list}}\ e_1\ (x \to [e_2])\ \mathsf{else}\ []}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \mathsf{list}\ t_1 \qquad \Delta;\Gamma, x : t_1 \vdash p : \mathsf{bool} \qquad \Delta;\Gamma, x : t_1 \vdash e_2 : t_2}{\begin{array}{c}\mathsf{flatmap}_{\mathsf{list}}\ e_1\ (x \to \mathsf{if}\ p\ \mathsf{then}\ [e_2]\ \mathsf{else}\ []) = \\ \mathsf{flatmap}_{\mathsf{list}}\ (\mathsf{flatmap}_{\mathsf{list}}\ e_1\ (x \to \mathsf{if}\ p\ \mathsf{then}\ [x]\ \mathsf{else}\ []))\ (x \to [e_2])\end{array}}$$

$$\frac{\Delta;\Gamma \vdash p_1 : \mathsf{bool} \qquad \Delta;\Gamma \vdash p_2 : \mathsf{bool} \qquad \Delta;\Gamma \vdash e : \mathsf{list}\ t}{\mathsf{if}\ p_1\ \mathsf{then}\ (\mathsf{if}\ p_2\ \mathsf{then}\ e\ \mathsf{else}\ []\ )\ \mathsf{else}\ [] = \mathsf{if}\ p_2\ \mathsf{then}\ (\mathsf{if}\ p_1\ \mathsf{then}\ e\ \mathsf{else}\ []\ )\ \mathsf{else}\ []}$$

Figure 4.2: Rules for transforming query-comprehension expressions in Conquord programs

Theorem 4 *All transformations in Figure 4.2 are sound.*

4.3 Rewrites From Relational-Algebra Equivalences

After the identification of relational-algebra operations, typical database-style rewrite rules such as filter pushdown can be applied, and the reasoning about the correctness of index-related transformations becomes more straightforward.

Figure 4.3 shows two rewrite rules that help push down filter operations into a join construct. The first rule gives a typical filter-pushdown transformation, identifying a conjunct in the join operation's predicate that depends only on one of the tables and pushing the filter operation closer to that table. The second rule pushes down a predicate that may depend on both tables of the join construct, much like how a query planner chooses a nested-loop join with a filter operation in the inner loop.

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : \mathsf{list}\ t_1 \qquad \Delta;\Gamma \vdash e_2 : \mathsf{list}\ t_2 \\ \Delta;\Gamma, x_1 : t_1 \vdash p_1 : \mathsf{bool} \qquad \Delta;\Gamma, x_1 : t_1, x_2 : t_2 \vdash p : \mathsf{bool} \qquad \Delta;\Gamma, x_1 : t_1, x_2 : t_2 \vdash e_3 : t_3\end{array}}{\mathsf{join}_{\mathsf{list}}\ e_1\ e_2\ (x_1, x_2 \to e_3\ \mathsf{if}\ p_1 \wedge p) = \mathsf{join}_{\mathsf{list}}\ (\mathsf{filter}_{\mathsf{list}}\ e_1\ (x_1 \to p_1))\ e_2\ (x_1, x_2 \to e_3\ \mathsf{if}\ p)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : \mathsf{list}\ t_1 \\ \Delta;\Gamma \vdash e_2 : \mathsf{list}\ t_2 \qquad \Delta;\Gamma, x_1 : t_1, x_2 : t_2 \vdash p : \mathsf{bool} \qquad \Delta;\Gamma, x_1 : t_1, x_2 : t_2 \vdash e_3 : t_3\end{array}}{\mathsf{join}_{\mathsf{list}}\ e_1\ e_2\ (x_1, x_2 \to e_3\ \mathsf{if}\ p) = \mathsf{flatmap}_{\mathsf{list}}\ e_1\ (x_1 \to \mathsf{flatmap}_{\mathsf{list}}\ (\mathsf{filter}_{\mathsf{list}}\ e_2\ (x_2 \to p))\ (x_2 \to [e_3]))}$$

Figure 4.3: Some rewrite rules for relational-algebra constructs in Conquord programs

Theorem 5 *All transformations in Figure 4.3 are sound.*

Chapter 5

Rewrites for Relaxing Collection Constraints

Conquord supports three collection types: lists, bags, and sets. These collections form a hierarchy: lists preserve order and duplicates, bags preserve duplicates but not order, and sets preserve neither. Some Conquord operations are invariant to order, duplicates, or both, which gives us the flexibility to change collection types without changing program semantics. We capture the requirements on collection variables by eagerly introducing collection conversions and pushing these conversion operations into expressions. Index-introduction rules use these conversion operators to recognize when an index representation of a table can be consumed safely.

5.1 Collection Annotation

After performing the relational-algebra rewrites, the compiler introduces conversions to collections that are less restricted than lists when conversion would not change the semantics of the consuming operation. The rewrite rules are shown in Figure 5.1, and the compiler performs the rewrites from left to right.

5.2 Collection Pushdown

The annotations of bags are pushed down into the subexpressions through the rewrite rules in Figure 5.2. Analogous rules shown in Figure 5.3 hold for annotations of sets, with all occurrences of bag-of replaced with set-of.

Theorem 6 *All transformations in Figure 5.1, Figure 5.2 and Figure 5.3 are sound.*

$$\frac{\Delta;\Gamma \vdash e : \mathsf{list}\ t}{\mathsf{sort}_{\mathsf{list}}\ e = \mathsf{sort}_{\mathsf{bag}}\ (\mathsf{bag\text{-}of}\ e)} \qquad \frac{\Delta;\Gamma \vdash e : \mathsf{list\ int}}{\mathsf{fold}\ e\ 0\ (v, a \to (v + a)) = \mathsf{sum}\ (\mathsf{bag\text{-}of}\ e)}$$

$$\frac{\Delta;\Gamma \vdash e : \mathsf{list}\ t}{\mathsf{length}\ e = \mathsf{count}\ (\mathsf{bag\text{-}of}\ e)}$$

$$\frac{\Delta;\Gamma \vdash e : \mathsf{list\ int}}{\begin{array}{c}\mathsf{fold}\ e\ \mathsf{none}\ (v, a \to \mathsf{optmatch}\ a\ (\mathsf{some}\ v)\ (v' \to \mathsf{if}\ v < v'\ \mathsf{then\ some}\ v\ \mathsf{else}\ a)) \\ = \mathsf{min}\ (\mathsf{set\text{-}of}\ e)\end{array}}$$

$$\frac{\Delta;\Gamma \vdash e : \mathsf{list}\ t}{(e == []) = (\mathsf{bag\text{-}of}\ e == \mathsf{empty}_{\mathsf{bag}})}$$

Figure 5.1: Rules for identifying less-restricted interpretations of collections

$$\frac{\Delta;\Gamma \vdash e_1 : \mathsf{list}\ t_1 \qquad \Delta;\Gamma, x : t_1 \vdash e_2 : \mathsf{list}\ t_2}{\mathsf{bag\text{-}of}\ (\mathsf{flatmap}\ e_1\ (x \to e_2)) = \mathsf{flatmap}_{\mathsf{bag}}\ (\mathsf{bag\text{-}of}\ e_1)\ (x \to (\mathsf{bag\text{-}of}\ e_2))}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : \mathsf{list}\ t_1 \\ \Delta;\Gamma \vdash e_2 : \mathsf{list}\ t_2 \qquad \Delta;\Gamma, x_1 : t_1, x_2 : t_2 \vdash p : \mathsf{bool} \qquad \Delta;\Gamma, x_1 : t_1, x_2 : t_2 \vdash e_3 : t_3\end{array}}{\mathsf{bag\text{-}of}\ (\mathsf{join}\ e_1\ e_2\ (x_1, x_2 \to e_3\ \mathsf{if}\ p)) = \mathsf{join}_{\mathsf{bag}}\ (\mathsf{bag\text{-}of}\ e_1)\ (\mathsf{bag\text{-}of}\ e_2)\ (x_1, x_2 \to e_3\ \mathsf{if}\ p)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \mathsf{list}\ t_1 \qquad \Delta;\Gamma, x : t_1 \vdash e_2 : t_2}{\mathsf{bag\text{-}of}\ (\mathsf{proj}\ e_1\ (x \to e_2)) = \mathsf{proj}_{\mathsf{bag}}\ (\mathsf{bag\text{-}of}\ e_1)\ (x \to e_2)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : t \qquad \Delta;\Gamma \vdash e_2 : \mathsf{list}\ t}{\mathsf{bag\text{-}of}\ (e_1 :: e_2) = \mathsf{insert}_{\mathsf{bag}}\ (\mathsf{bag\text{-}of}\ e_2)\ e_1}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \mathsf{bool} \qquad \Delta;\Gamma \vdash e_2 : \mathsf{list}\ t \qquad \Delta;\Gamma \vdash e_3 : \mathsf{list}\ t}{\mathsf{bag\text{-}of}\ (\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) = \mathsf{if}\ e_1\ \mathsf{then\ bag\text{-}of}\ e_2\ \mathsf{else\ bag\text{-}of}\ e_3}$$

Figure 5.2: Rules for pushdown of collection casts to bags

$$\frac{\Delta;\Gamma \vdash e_1 : \mathsf{list}\ t_1 \qquad \Delta;\Gamma, x : t_1 \vdash e_2 : \mathsf{list}\ t_2}{\mathsf{set\text{-}of}\ (\mathsf{flatmap}\ e_1\ (x \to e_2)) = \mathsf{flatmap}_{\mathsf{set}}\ (\mathsf{set\text{-}of}\ e_1)\ (x \to (\mathsf{set\text{-}of}\ e_2))}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : \mathsf{list}\ t_1 \\ \Delta;\Gamma \vdash e_2 : \mathsf{list}\ t_2 \qquad \Delta;\Gamma, x_1 : t_1, x_2 : t_2 \vdash p : \mathsf{bool} \qquad \Delta;\Gamma, x_1 : t_1, x_2 : t_2 \vdash e_3 : t_3\end{array}}{\mathsf{set\text{-}of}\ (\mathsf{join}\ e_1\ e_2\ (x_1, x_2 \to e_3\ \mathsf{if}\ p)) = \mathsf{join}_{\mathsf{set}}\ (\mathsf{set\text{-}of}\ e_1)\ (\mathsf{set\text{-}of}\ e_2)\ (x_1, x_2 \to e_3\ \mathsf{if}\ p)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \mathsf{list}\ t_1 \qquad \Delta;\Gamma, x : t_1 \vdash e_2 : t_2}{\mathsf{set\text{-}of}\ (\mathsf{proj}\ e_1\ (x \to e_2)) = \mathsf{proj}_{\mathsf{set}}\ (\mathsf{set\text{-}of}\ e_1)\ (x \to e_2)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : t \qquad \Delta;\Gamma \vdash e_2 : \mathsf{list}\ t}{\mathsf{set\text{-}of}\ (e_1 :: e_2) = \mathsf{insert}_{\mathsf{set}}\ (\mathsf{set\text{-}of}\ e_2)\ e_1}$$

Figure 5.3: Rules for pushdown of collection casts to sets

Chapter 6

Rewrites for Index Creation and Use

This chapter uses a mix of Gallina (Rocq's dependently typed functional language) and Conquord syntax, as Gallina tokens are used to abstract over Conquord expressions. Gallina expressions are colored and underlined to avoid confusion.

The indexes in the Conquord compiler implement a common index interface defined as a Gallina type class in Figure 6.1. The to-idx[.] member of the type class is a context that takes a Conquord expression e and returns a Conquord expression that computes the index from e. is-tbl-ty takes a type t and returns a Boolean value. If it returns true for a type t, then for any Conquord expression e with $\Delta; \Gamma \vdash e : t$, to-idx[$e$] correctly computes the index from the table that e evaluates to, and $\Delta; \Gamma \vdash$ to-idx[e] : idx-ty t. idx-wf defines the characterizing properties of the index with respect to the original table, and it helps simplify the soundness proofs of index-specific transformations. idx-ty-wf, to-idx-ty and to-idx-wf are propositions that instances of the type class must satisfy.

$$
\begin{array}{lcl}
\text{to-idx}[.] & : & \text{expr} \to \text{expr} \\
\text{is-tbl-ty} & : & \text{type} \to \text{bool} \\
\text{idx-ty} & : & \text{type} \to \text{type} \\
\text{idx-wf} & : & \text{value} \to \text{value} \to \text{Prop} \\
\text{idx-ty-wf} & : & \forall t.\ \vdash t \implies \text{is-tbl-ty}\ t = \text{true} \implies \vdash \text{idx-ty}\ t \\
\text{to-idx-ty} & : & \forall t.\ \vdash t \implies \text{is-tbl-ty}\ t = \text{true} \implies \epsilon; x : t \vdash \text{to-idx}[x] : \text{idx-ty}\ t \\
\text{to-idx-wf} & : & \forall t.\ \vdash t \implies \text{is-tbl-ty}\ t = \text{true} \implies \vdash v : t \implies \text{idx-wf}\ v\ [\![\text{to-idx}[x]]\!]_{\epsilon;x:v}
\end{array}
$$

Figure 6.1: Gallina type class defining the index interface

6.1 Index Creation

A whole-program transformation is used to create an index for a table while preserving the semantics of the entire program. For index creation, we have the local rewrites that transform the expressions whose values are written into the table. The transformations rely on the

index interface to abstract away the implementation of the index.

$$\mathsf{let\ mut}\ \mathbf{tbl} \coloneqq e\ \mathsf{in}\ c \Rightarrow \mathsf{let\ mut}\ \mathbf{tbl} \coloneqq \{\ \mathsf{id} : e;\ \ \mathsf{aux} : \underline{\text{to-idx}}[e]\ \}\ \mathsf{in}\ c \tag{6.1}$$

$$\mathbf{tbl} \coloneqq e \Rightarrow \mathbf{tbl} \coloneqq \{\ \mathsf{id} : e;\ \ \mathsf{aux} : \underline{\text{to-idx}}[e]\ \} \tag{6.2}$$

To obtain semantic equivalence, reads from the mutable variable **tbl** are transformed in tandem.

$$\mathbf{tbl} \Rightarrow \mathbf{tbl}.\mathsf{id} \tag{6.3}$$

Theorem 7 *A sound transformation is formed by the following steps: take a command of the form* ***let mut tbl*** $\coloneqq e$ ***in*** c *where* e *has a type* t *satisfying* $\underline{\text{is-tbl-ty}}\ t = \underline{\text{true}}$ *in the current typing environments; apply Transformation 6.1 to* ***let mut tbl*** $\coloneqq e$ ***in*** c*; apply Transformations 6.2 and 6.3 in* c *wherever* ***tbl*** *is not shadowed.*

To use the created index, the compiler rewrites expressions that contain **tbl**.id to read from **tbl**.aux instead. What expressions may be rewritten depends on the index instance.

6.2 Index-Specific Transformations

The equation for the most basic transformation common to all indexes has the following form.

$$\underline{\text{to-idx}}[\mathbf{tbl}.\mathsf{id}] = \mathbf{tbl}.\mathsf{aux}$$

Rewriting from left to right of the equation, the transformation identifies an expression equivalent to the creation of the index and replaces it by an expression that retrieves the index, which has been created and stored in a mutable variable.

Another common category of transformations identifies expressions that are equivalent to updates to existing indexes. The expression for index update, however, will depend on the definition of to-idx. The rest of this section presents a few instances of the index interface and transformations using the indexes.

6.2.1 A Dictionary-Based Index

Suppose attr is an attribute of a table. We have implemented a dictionary-based index that maps each value of attr to the bag of tuples in the table with that attribute value. Using bags instead of lists as index lookup results will give more flexibility in the compilation of the index to a lower-level language.

$$\begin{aligned}
&\underline{\text{dict-update}}[d, k, v] \triangleq \\
&\mathsf{dict\text{-}insert}\ d\ (k \mapsto \mathsf{insert}_{\mathsf{bag}}\ (\mathsf{optmatch}\ (\mathsf{dict\text{-}lookup}\ d\ k)\ \mathsf{empty}_{\mathsf{bag}}\ (b \to b))\ v) \\
&\underline{\text{dict-idx.to-idx}}[e] \triangleq \mathsf{fold}\ e\ \mathsf{empty\text{-}dict}\ (\mathit{tup}, \mathit{acc} \to \underline{\text{dict-update}}[\mathit{acc}, \mathit{tup}.\mathsf{attr}, \mathit{tup}])
\end{aligned}$$

Given the definition above, the following rewrite rule can change the computation of $\underline{\text{dict-idx.to-idx}}[e_1 :: e_2]$ to an update to $\underline{\text{dict-idx.to-idx}}[e_2]$ using e_1. In the case where $e_2 \equiv \mathbf{tbl}$, the previous rewrite rule can be applied, and the resulting program uses **tbl**.aux instead of

computing the dictionary-based index from scratch. For brevity, we omit the typing judgment for this rewrite rule.

$$\underline{\text{dict-idx.to-idx}}[e_1 :: e_2] = \text{let } y \leftarrow \underline{\text{dict-idx.to-idx}}[e_2] \text{ in } \underline{\text{dict-update}}[y, e_1.\text{attr}, e_1]$$

where $y \notin \text{free-immuts}(e_1)$.

Moreover, we also have an ad-hoc rewrite rule unique to the dictionary-based index.

$$\begin{aligned}&\text{filter}_{\text{bag}}\ (\text{bag-of } \mathbf{tbl}.\text{id})\ (x \to x.\text{attr} == e) = \\ &\text{optmatch}\ (\text{dict-lookup } \mathbf{tbl}.\text{aux}\ e)\ \text{empty}_{\text{bag}}\ (b \to b)\end{aligned}$$

where $x \notin \text{free-immuts}(e)$.

6.2.2 Aggregations

Figure 5.1 presented some rewrite rules that identify aggregation operations like min and sum. We have implemented indexes to cache the aggregation results and perform incremental updates when a new tuple is inserted into the table. Unlike the dictionary-based index where the transformation that uses the index involves a lookup operation on the index, the use of the two aggregation indexes happens to coincide with the transformations that identify the index creation. In other words, "looking up" a sum-aggregation index is the same as retrieving the index itself, which evaluates to the cached aggregation result.

Like before, we suppose attr is an attribute of the table in question and define aggregations over that attribute. For sum aggregation over the attribute attr of the table,

$$\underline{\text{sum-agg.to-idx}}[e] \triangleq \text{sum}\ (\text{proj}_{\text{bag}}\ (\text{bag-of } e)\ (x \to x.\text{attr}))$$

The following transformation identifies an update to the sum aggregation

$$\underline{\text{sum-agg.to-idx}}[e_1 :: e_2] = (\text{let } x \leftarrow e_1 \text{ in } x.\text{attr}) + \underline{\text{sum-agg.to-idx}}[e_2]$$

The minimum aggregation is slightly more complicated, as it returns an option value to handle the case of aggregating over the empty set.

$$\underline{\text{min-agg.to-idx}}[e] \triangleq \text{min}\ (\text{proj}_{\text{set}}\ (\text{set-of } e)\ (x \to x.\text{attr}))$$

$$\begin{aligned}&\underline{\text{min-agg.to-idx}}[e_1 :: e_2] = \\ &\text{let } y \leftarrow (\text{let } x \leftarrow e_1 \text{ in } x.\text{attr}) \text{ in} \\ &\text{optmatch } \underline{\text{min-agg.to-idx}}[e_2] \\ &(\text{some } y) \\ &(x \to \text{if } (y < x) \text{ then } (\text{some } y) \text{ else } (\text{some } x))\end{aligned}$$

where $y \neq x$ and $y \notin \text{free-immuts}(e_2)$.

6.2.3 Bitmap Index

We have also implemented a simplified variant of bitmap indexes. Fixing an attribute attr of a table and a constant value c that the attribute may take, the bitmap index consists of two parts: a primary-key index with an automatically generated unique ID for each tuple in the table, and a bitmap where each bit indicates whether tup.attr == c for its corresponding tuple tup.

The creation of the primary-key index is defined by

$$\text{pk-idx.to-idx}[e] \triangleq \mathsf{fold}\ e\ (0, \mathsf{empty\text{-}dict})\ (tup, (c, d) \to (c+1, \mathsf{dict\text{-}insert}\ d\ (c \mapsto tup)))$$

It evaluates to a pair, where the first component is the cardinality of the table, and the second is a dictionary mapping the IDs to the tuples.

The update transformation is

$$\text{pk-idx.to-idx}[e_1 :: e_2] = \mathsf{let}\ (c, d) \leftarrow \text{pk-idx.to-idx}[e_2]\ \mathsf{in}\ (c+1, \mathsf{dict\text{-}insert}\ d\ (c \mapsto e_1))$$

The creation of the bitmap is defined by

$$\text{bitmap.to-idx}[e] \triangleq \mathsf{flatmap}\ e\ (tup \to [tup.\mathsf{attr} == \mathsf{c}])$$

The update transformation is

$$\text{bitmap.to-idx}[e_1 :: e_2] = (e_1.\mathsf{attr} == \mathsf{c}) :: \text{bitmap.to-idx}[e_2]$$

The bitmap index can replace filtering operations by the tup.attr == c predicate:

```
use-bitmap m d ≜
fold m (0, []) (b, acc →
(fst acc + 1,
if b
then optmatch (dict-lookup d (fst acc)) (snd acc) (x → x :: snd acc)
else snd acc))

filter tbl.id (x → (x.attr == c)) =
snd (use-bitmap(tbl.aux.bitmap)(snd (tbl.aux.pk)))
```

where the primary-key index is stored in **tbl**.aux.pk and the bitmap in **tbl**.aux.bitmap.

While the transformation above does not speed up programs assuming the entire database is available in the memory, it is a step towards the more interesting use of bitmap indexes, where the predicate may be a conjunction or a disjunction of multiple predicates represented by different bitmaps. The filter operation can then be rewritten to bit-vector operations between the bitmaps, and the resulting bitmap can be used to perform lookups in the primary-key index. The bit-vector operations can parallelize much more efficiently than the predicate computation in the original filter operation, thereby speeding up the program execution.

6.2.4 Composite Indexes

In our implementation, the index created in the whole-program transformation is a composite index that consists of an arbitrary number of indexes. In particular, the to-idx expression of the composite index is a record of the to-idx expressions of the constituent indexes, each associated with a unique tag in the record. For example, suppose the composite index $cidx$ contains two indexes idx_1 and idx_2 associated with the tags idx-1 and idx-2 respectively. Then we have

$$cidx.\text{to-idx} \triangleq \{\ \text{idx-1} : idx_1.\text{to-idx};\ \ \text{idx-2} : idx_2.\text{to-idx}\ \}$$

$$cidx.\text{is-tbl-ty}\ t \triangleq idx_1.\text{is-tbl-ty}\ t \wedge idx_2.\text{is-tbl-ty}\ t$$

$$cidx.\text{idx-ty}\ t \triangleq \{\ \text{idx-1} : idx_1.\text{idx-ty}\ t;\ \ \text{idx-2} : idx_2.\text{idx-ty}\ t\ \}$$

$$cidx.\text{idx-wf}\ v\ d \triangleq idx_1.\text{idx-wf}\ v\ d.\text{idx-1} \wedge idx_2.\text{idx-wf}\ v\ d.\text{idx-2}$$

and idx-ty-wf, to-idx-ty, and to-idx-wf for $cidx$ can be derived from the corresponding propositions for idx_1 and idx_2 respectively.

As a result, **tbl**.aux.idx-1 can be used in place of **tbl**.aux in the examples above to access the instance of idx_1 stored in **tbl**, and likewise for idx_2.

Chapter 7

Experimental Design

We conducted experiments to evaluate the performance of Conquord, testing the effectiveness of the compiler's optimizations and the expressivity of the language. This chapter describes the experimental methodology.

7.1 A Parameterized Compilation Pipeline

The transformations described in this paper can be composed to give a compilation pipeline. The optimal choice of transformations and their orders, however, depend on many factors and are often left to heuristics based on expert knowledge. To facilitate easier construction of compilation pipelines from different permutations of transformations, we annotate the abstract syntax of a Conquord program with optimization hints. The compiler takes such an annotated program and performs the corresponding sequence of transformations to produce the optimized program.

An annotated program consists of three parts: the original program, a list of basic-transformation hints, and a list of index-choice hints. A basic-transformation hint corresponds to one of the transformations described in Chapters 4 and 5, i.e., a transformation in this thesis that is not related to indexes. An index-choice hint is a collection of index hints, and each index hint specifies an index along with its parameters. For example, DictIdx attr specifies a dictionary-based index with the attribute attr as the key. We assume the database tables are declared as mutable variables at the top level of the program through let mut constructs. The n-th index-choice hint in the list represents the indexes recommended for the table declared by the n-th outermost let mut construct. The compiler first applies the basic transformations. Then it creates the indexes indicated by the index hints and applies transformations related to an index if and only if that index is created successfully. Using the soundness lemmas of the transformations in the pipeline, we proved that the compilation pipelines resulting from arbitrary hints are sound.

Theorem 8 *Given any Conquord program as a command c, a list of basic-transformation hints h_b, and a list of index-choice hints h_i, let $(F\ h_b\ h_i\ c)$ denote the program that the compiler obtains by applying the corresponding transformations to c. The transformation $(F\ h_b\ h_i)$ is sound. Formally, $\forall \Delta,\ \Gamma,\ c.\ \Delta; \Gamma \vdash c \implies \forall h_b,\ h_i.\ \Delta; \Gamma \vdash F\ h_b\ h_i\ c\ \wedge\ \forall \phi,\ \psi.\ \Delta \vdash \phi \implies \Gamma \vdash \psi \implies [\![F\ h_b\ h_i\ c]\!]_{\phi,\psi} = [\![c]\!]_{\phi,\psi}$*

7.2 Methodology

We aim to test the efficacy of the language design through the evaluation framework illustrated in Figure 7.1.

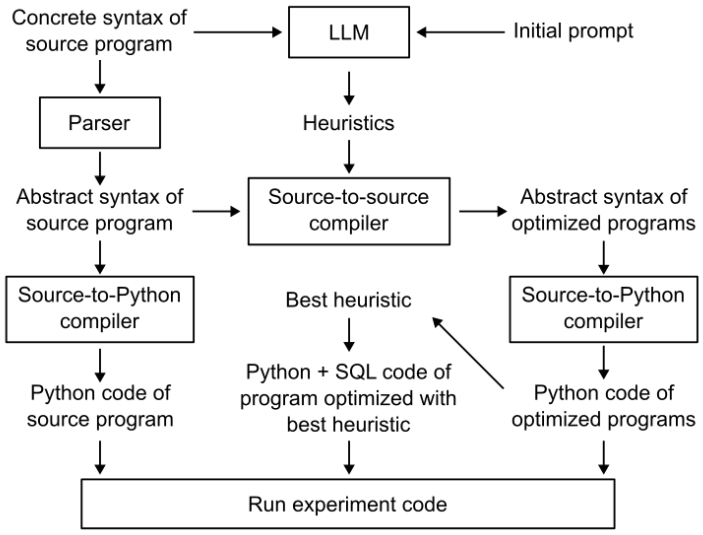

Figure 7.1: The evaluation framework

The workflow of an experiment starts with a source program written in the concrete syntax of Conquord. The program is parsed to obtain its abstract syntax, and the typechecker ensures it is well-typed in the suitable typing environments. Since finding the optimal heuristics is not a goal of this thesis, we delegated the task to a large language model (LLM). The initial prompt given to the LLM introduced the syntax of the language and described the optimization hints accepted by the compiler. Then we provided the source program to the LLM and asked for some candidate heuristics in the form of optimization hints. The abstract syntax was passed to the compiler along with each of the heuristics to produce the optimized programs.

In order to estimate the runtime performance of the programs, we developed a transpiler from Conquord to Python to translate the source program and the optimized programs to Python programs for execution. The translation from Conquord to Python mainly invokes operations and language constructs that already exist in Python, and they have a close correspondence with the Conquord primitives. One exception is the fold operation over a list, which is translated to an imperative loop to avoid stack overflow. The soundness of the transpiler was not formally verified, but the outputs of all the executable programs for the same example were validated to be equal.

The six example programs range from repeated database queries to applications that take parameterized queries and allow the addition of new data to the tables. With randomly generated inputs of various sizes and sufficiently large numbers of queries to dominate the cost of the initial table creation, we tested the runtime of the Python programs and looked for the best-performing heuristic. We varied the table sizes in examples that do not involve the addition of new data to the tables. If the examples have queries that perform data insertion

interleaved with data analysis, then we varied the number of queries as that is directly related to the sizes of the tables during program execution. We recorded the mean of the results from 10 independent repetitions of the experiments.

After determining the best heuristic from the LLM, we asked the LLM to generate Python code that uses a SQLite database for the program with that heuristic. The resulting program resembles what a programmer working with Python and SQL would write, with the programmer manually injecting their judgment about which indexes should be created for each table. For convenience, we call this program the Python-SQL program. We ran the experiment code again on the Python code from the Conquord source program, the Python code from the program optimized with the best-performing heuristic, and the Python-SQL program to compare their performance.

The LLM used for the experiments was GPT-5.4 [22], and the Python programs were executed on a commercial laptop with an Apple M2 Max chip and 32 GB memory.

Chapter 8

Evaluation

This chapter presents the experimental results and evaluates the performance of Conquord. A general observation is that the LLM consistently gave suggestions of index-choice hints that met our expectations, and it provided good heuristics for basic transformations in five out of the six example programs. When multiple heuristics were suggested, the first one always performed the best among them. When working with the example where the LLM did not provide helpful hints for basic transformations, we used a few additional prompts to explain the reason why its chosen hints would not work and provided more details about the transformations. After that, the LLM was able to produce better hints with the correct explanations. For every example, the best-performing heuristic achieved the expected asymptotic speedup, and the runtime of the optimized Conquord program was within a small constant multiple of the corresponding Python-SQL code, with the former sometimes outperforming the latter, provided that the cost of the queries dominated that of the initial table and index creations.

Table 8.1 shows a summary of the runtime results for the smallest and the largest input sizes used in each experiment alongside the best heuristic generated for that experiment. Each heuristic in the table is written as a composition of basic-transformation hints followed by a list of index-choice hints, with the n-th index-choice hint corresponding to the n-th outermost mutable-variable declaration in the program. "Baseline" refers to the Python code derived from the source program, "heu1" corresponds to the Python code derived from the Conquord program after applying the best LLM heuristic to the source program, and "pysql" refers to the Python-SQL program with similar optimizations.

Figure 8.1 visualizes the runtime data with a bar chart. A logarithmic scale is used in this figure due to the large difference in the runtime of baseline programs between different experiments.

Comparison with Python-SQL programs also highlights the expressivity of the Conquord language. On top of working with two entirely different programming styles at once, a programmer writing a Python-SQL program also needs to manage the connection to a SQL database, write SQL queries as strings that cannot be statically checked, and manually specify which indexes should be created for each database table. In contrast, Conquord provides a more consistent programming experience. It has static type checking of the entire program, and the compiler can optimize the program using indexes based on automatically generated hints without violating the strong requirement of semantic preservation.

Example	Best Heuristic	Small Input			Large Input		
		baseline	heu1	pysql	baseline	heu1	pysql
Employee	(PC ∘ AC ∘ TP ∘ JFF ∘ TJ); [(DictIdx "dept-id")]	9.09	3.99	7.81	4810	159	266
Family	(PC ∘ AC ∘ JFF ∘ FP ∘ TJ); [(DictIdx "parent")]	803	28.2	34.3	112000	29.6	38.0
Sum	(PC ∘ AC ∘ TP); [(SumAgg "value")]	107	2.53	2.40	1330	68.4	53.7
BFS	(PC ∘ AC ∘ TP ∘ TF ∘ IFF ∘ TP ∘ TF ∘ IFF); [(DictIdx "src"); (DictIdx "node")]	4.12	0.538	2.89	3770	254	95.8
Triangle	(PC ∘ AC ∘ TP ∘ TF ∘ IFF ∘ SFI ∘ SFI ∘ SPI ∘ SIF ∘ IFF ∘ SFI ∘ SFI); [(DictIdx "src")]	45.0	0.216	0.320	12000	6.89	10.4
Orders	(PC ∘ AC ∘ TP ∘ TF ∘ IFF ∘ TP); [(DictIdx "id"); (SumAgg "price", MinAgg "price")]	162	51.9	5.25	6290	204	125

PC = PushdownCollection; AC = AnnotateCollection
TP = ToProj; TF = ToFilter; TJ = ToJoin
FP = FilterPushdown; JFF = JoinToFlatmapFilter
IFF = IfNilIntoFlatmap; SPI = SplitIf; SIF = SwapIfNil; SFI = SwapFlatmapIf

Table 8.1: Summary of best heuristics and runtime (in milliseconds) for all examples at small and large input sizes.

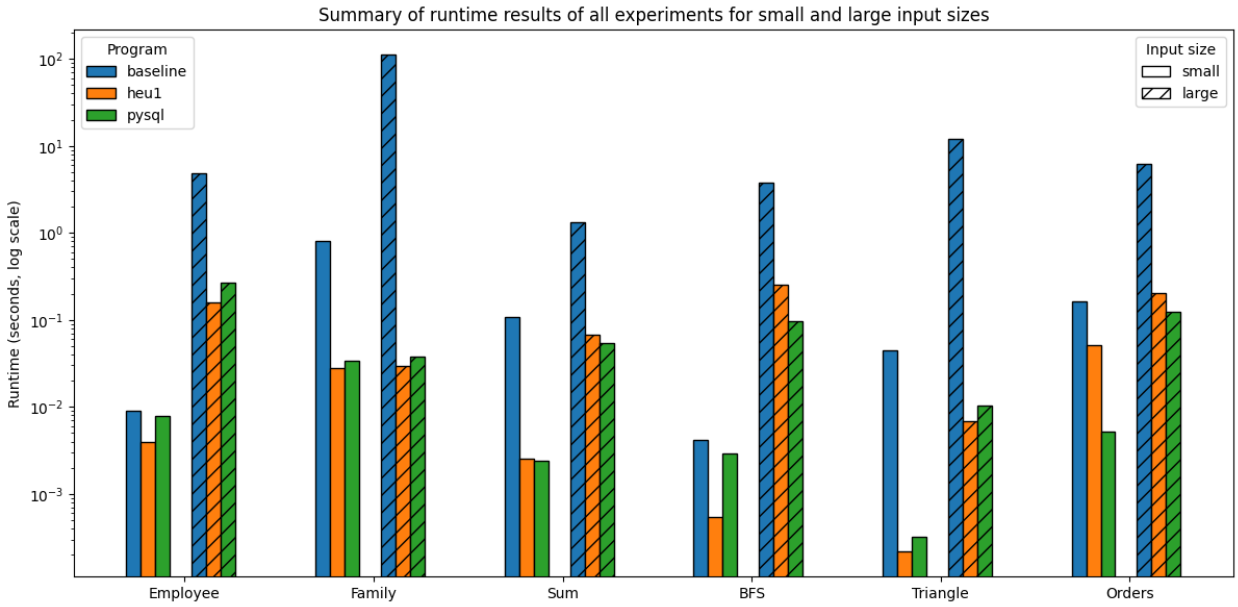

Figure 8.1: Summary of runtime results of all experiments for small and large input sizes.

We will now dive into the details for each example program and discuss the experimental results.

8.1 Example: Employee

The first example program demonstrates a join operation on a foreign key. The program takes a table about departments and a table about employees. It then repeatedly performs the join operation between the two tables on department IDs, producing output strings that list every employee's name with the name of the department they are in.

```
let mut employees := emp_tbl in
let mut departments := dept_tbl in

foreach i in range(0, iters) do
    let mut output := "" in
    let result =
    sort [ d ← !departments,
        e ← !employees,
        check(e.dept_id = d.id),
        ret {name: e.name, dept: d.name} ] in

    foreach r in result do
        output := !output ++ r.name ++ " is in " ++ r.dept ++ "\n"
    end;
    outputs := !output :: !outputs
```

```
end
```

The best heuristic from the LLM is shown in 8.1. The heuristic identified the join construct, pushed down the filter operation, created and propagated collection annotations and created a dictionary-based index on the `dept_id` attribute of the `employees` table.

Figure 8.2 shows the runtime of the Python programs derived from the source program, the program optimized with the best heuristic from the LLM, and the similarly optimized program written in Python with access to a SQLite database. The string concatenations in the Conquord program are parsed such that they associate to the right. Python's compiler recognized that the `output` string could be updated in place and avoided the quadratic cost of building new strings for all concatenations. Therefore, the cost of building the strings did not dominate the performance of the program.

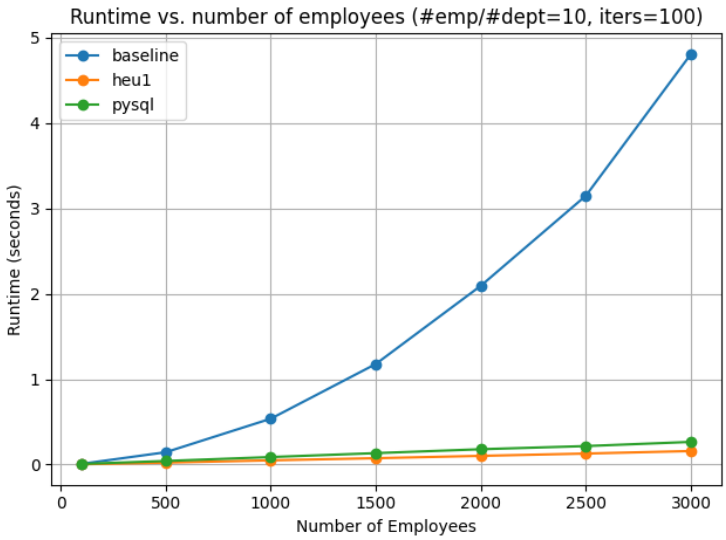

Figure 8.2: Runtime against the number of employees for variants of the "employee" example program, with a constant ratio of 10 between the number of employees and the number of departments.

We maintained a constant ratio between the size of `emp_tbl` and the size of `dept_tbl` as we varied the input table sizes. Let N denote the number of employees in `emp_tbl`. The baseline graph, showing the performance of the source program, exhibits runtime that is quadratic in N. Both the optimized program and the corresponding Python-SQL program have $O(N \log N)$ runtime due to the sorting of the join's output, and their graph appear to have linear growth in N. For this example, the Python code derived from the optimized Conquord program was faster than the Python-SQL program.

8.2 Example: Family

This example is a program that takes a table of parent-child relationships and produces a string that prints all grandchildren of a person. The program assumes a list `queries` that

contains incoming queries, each of which identifies a person, and the program should find all grandchildren of that person.

```
let mut parents := parents_tbl in

foreach person in queries do
    let mut result := "Grandchildren of " ++ person ++ ":\n" in
    let children =
        sort [ p ← !parents,
            q ← !parents,
            check(p.parent = person && q.parent = p.child),
            ret q.child ] in
    foreach child in children do
        result := !result ++ child ++ "\n"
    end;
    outputs := !result :: !outputs
end
```

To generate the parent-child graph for this experiment, we first created a tree of $4N$ edges and then added N edges to the tree to produce a directed acyclic graph (DAG) with $5N$ edges. Each edge corresponds to a row in `parents_tbl`.

The difference between the performance of the optimized Conquord program and that of the Python-SQL program becomes invisible when illustrated in the same axes as the performance of the source program. Hence, we present the graph of the source program and the optimized Conquord program in Figure 8.3, where we can observe an asymptotic speedup from the optimization. Separately, Figure 8.4 shows that the Python code derived from the optimized Conquord program again outperformed the Python-SQL program.

8.3 Example: Sum

This example tests the incremental computation of the sum aggregation. The program starts with a table of orders and a list of queries. Each query can ask for an output string containing the total value of all the orders in the table, or it can give a new order to be added to the table.

```
let mut orders := orders_tbl in

foreach q in queries do
    let mut output := "" in
    if q.type = 0
    then
        let total = sum [ o ← !orders, ret o.value ] in
        output := "Current total value: " ++
            str(total) ++ "\n"
    else
```

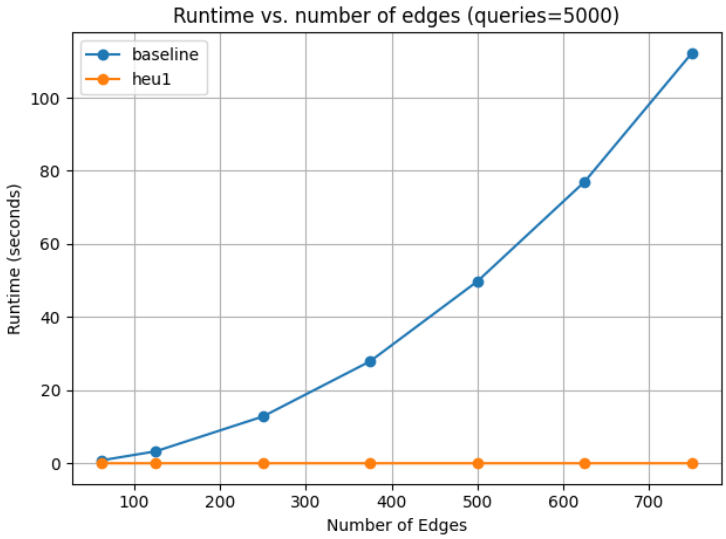

Figure 8.3: Runtime against the number of edges in the parent-child graph for the baseline program and the optimized Conquord program in the "family" example.

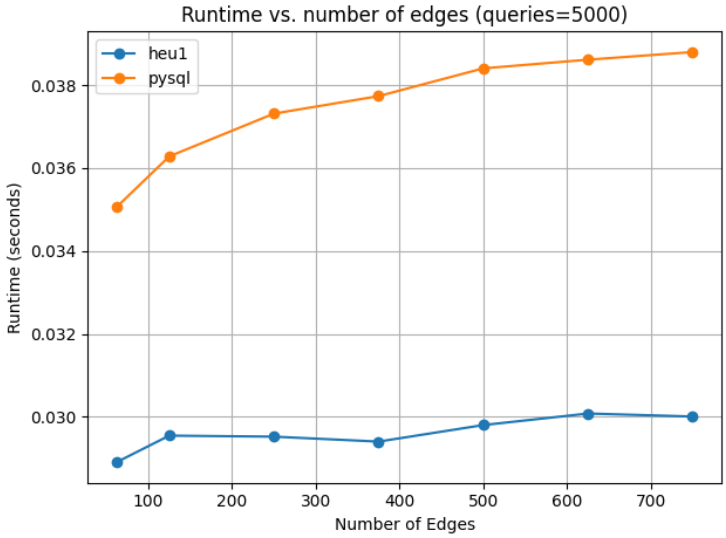

Figure 8.4: Runtime against the number of edges in the parent-child graph for the optimized Conquord program and the Python-SQL program in the "family" example.

```
        orders := q.new_order :: !orders;
        output := "Added a new order\n"
    end;
    outputs := !output :: !outputs
end
```

The source program naively recomputed the sum everytime it was requested, whereas the best-performing heuristic from the LLM correctly created and propagated collection

annotations. It also created a sum-aggregation index to cache the aggregation result and incrementally updated it upon the insertion of new orders.

For the experiment, we started with a randomly generated table of 1000 orders and tested the runtime with increasing numbers of queries. Approximately half of the queries added new orders to the table, while the other half asked for the sum. Figure 8.5 shows the asymptotic speedup of the optimized program. The Python program derived from the optimized Conquord program is only slower than the Python-SQL program by a small constant factor.

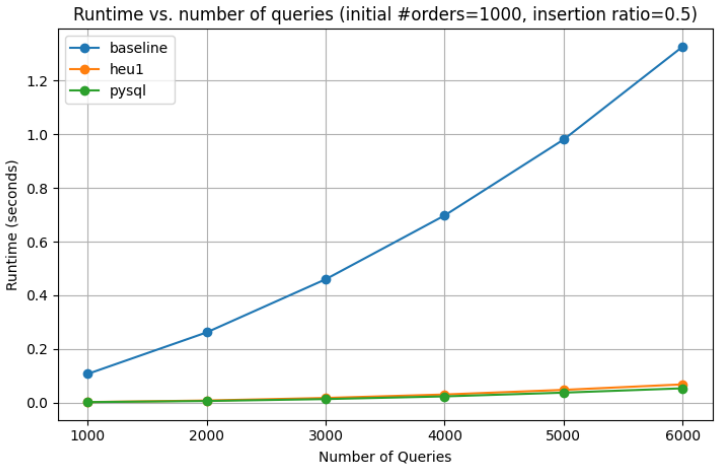

Figure 8.5: Runtime against the number of queries for variants of the "sum" example program.

8.4 Example: Breadth-First Search

This program performs a breadth-first search (BFS) of all nodes reachable from a given start node on a directed graph and generates a string containing the discovered nodes with their depths. The directed graph is represented as a table of edges.

```
let mut edges := edges_tbl in
let mut visited := [ {node: start_node, depth: 0} ] in
let mut frontier := [ start_node ] in
let mut next_frontier := []<int> in
let mut cur_depth := 1 in

foreach i in range(0, len(edges_tbl)) do
  foreach cur_node in !frontier do
    let cur_children =
      sort [ e ← !edges,
        check(e.src = cur_node),
        check([n ← !visited, check(n.node = e.dst), ret ()] = []),
        ret e.dst ] in
```

```
    foreach n in cur_children do
      visited := {node: n, depth: !cur_depth} :: !visited;
      next_frontier := n :: !next_frontier
    end
  end;

  frontier := !next_frontier;
  next_frontier := []<int>;
  cur_depth := !cur_depth + 1
end;

foreach n in sort(!visited) do
  outputs := ("Node " ++ str(n.node) ++ " at depth " ++
    str(n.depth) ++ "\n") :: !outputs
end
```

Every directed graph with N nodes was generated with $5N$ edges, and we measured the runtime of the programs with respect to increasing values of N. The input graphs were generated such that there were no self loops or repeated edges. The LLM successfully suggested optimization hints to expose the filter operations. Based on the index-choice hints from the LLM, the compiler created a dictionary-based index for the `edges` table on the `src` attribute and another one for the `visited` table on the `node` attribute.

From Figure 8.6, we can observe a significant speedup from the source program to the optimized Conquord program, and the performance of the latter is close to its Python-SQL counterpart.

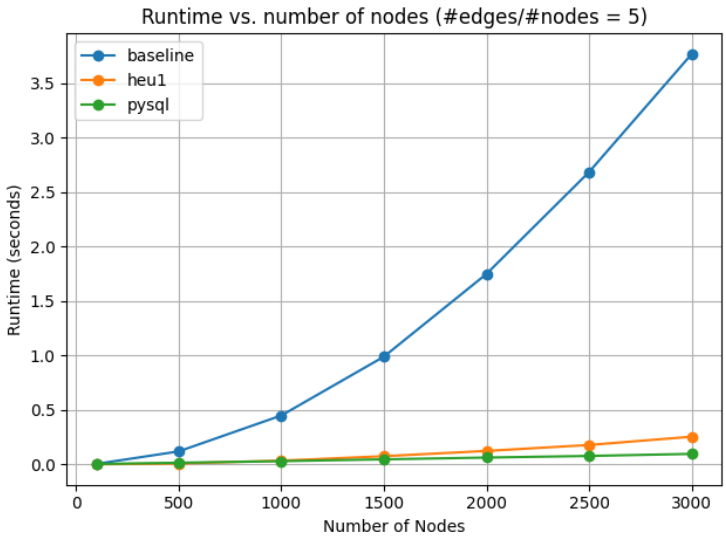

Figure 8.6: Runtime against the number of nodes for variants of the "BFS" example program.

8.5 Example: Triangle

Like the BFS example, this example also involves a directed graph represented as a table of edges. The program accepts a list of queries, each of which can either add a new edge to the graph or ask for three-edged cycles in the graph with strictly increasing node IDs. The lookup uses self-join operations that connect three edges at their vertices. This program has a number of predicates that need to be reordered and pushed down into the flatmap constructs for the most effective use of indexes, and the rewrites between query-comprehension expressions are helpful for the filter pushdown.

```
let mut edges := edges_tbl in

foreach q in queries do
    if q.type = 0
    then
        let triangles =
            sort [ e1 ← !edges,
                e2 ← !edges,
                e3 ← !edges,
                check(e2.src = e1.dst),
                check(e3.src = e2.dst),
                check(e1.dst = e3.dst),
                check(e1.src < e1.dst && e2.src < e2.dst),
                ret { p1: e1.src, p2: e2.src, p3: e3.src } ] in
        let mut result := "" in
        foreach tri in triangles do
            result := !result ++ str(tri.p1) ++ ", " ++
                str(tri.p2) ++ ", " ++ str(tri.p3) ++ "\n"
        end;
        outputs := !result :: !outputs
    else
        edges := q.new_edge :: !edges;
        outputs := "New edge added successfully\n" :: !outputs
    end
end
```

While the LLM correctly suggested creating a dictionary-based index on the `src` attribute of the `edges` table, the list of basic transformation hints it provided did not expose the filter constructs that can be rewritten to index lookups. In a few subsequent prompts, we explained why the hints did not work and how other rewrites could achieve the goal. After that, the LLM managed to give a heuristic that exposed opportunities to use the index. That heuristic is the one that yielded the results below.

We conducted the experiment with increasing numbers of queries, and about half of the queries are additions to the graph. The Conquord program resulting from optimization with the best heuristic was slightly faster than the Python-SQL equivalent. Figure 8.7

shows the comparison between the source program and the optimized Conquord program, and Figure 8.8 displays the difference between the optimized Conquord program and its Python-SQL counterpart.

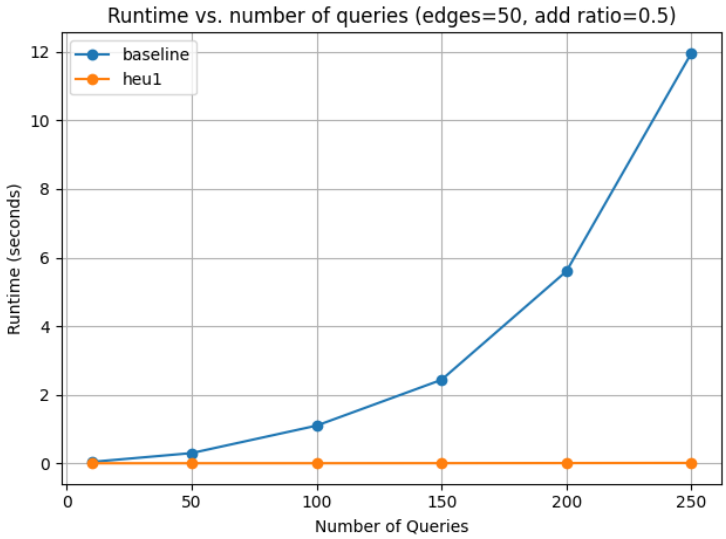

Figure 8.7: Runtime against the number of queries for the baseline program and the optimized Conquord program in the "triangle" example.

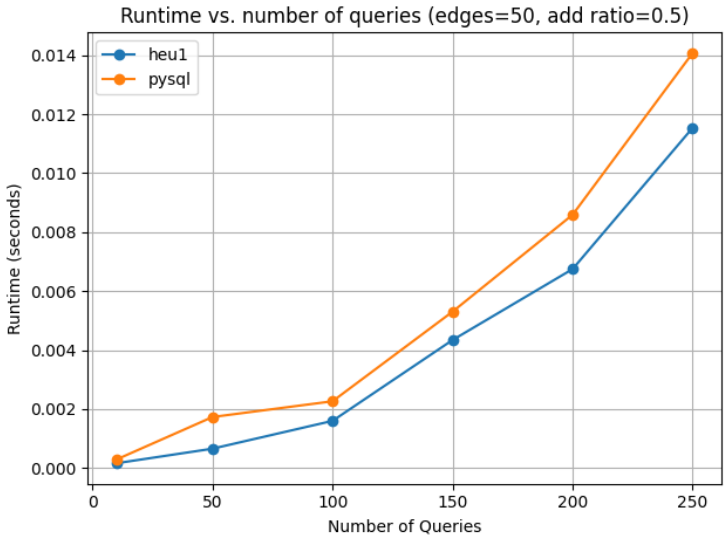

Figure 8.8: Runtime against the number of queries for the optimized Conquord program and the Python-SQL program in the "triangle" example.

8.6 Example: Orders

Consider a sales event where the customer can get the lowest-value item in their orders for free if they spend at least 300 dollars. This example program has access to an inventory table that contains the price information of the items and a table of the current orders. The program receives a list of queries, and for each query the program either adds an order or computes whether the current orders qualify for the discount, and if so, how much can be saved.

```
let mut inventory := inventory_tbl in
let mut orders := orders_tbl in

foreach q in queries do
    let mut new_log := "" in
    if q.type = "add order" then
        let item_price =
            sort [ item ← !inventory,
            check(item.id = q.new_order_id),
            ret item.price ] in
        foreach price in item_price do
            orders := { id: q.new_order_id, price: price } :: !orders
        end;
        new_log := "Added order: " ++ str(q.new_order_id) ++ "\n"
    else if q.type = "discount status" then
            let total = sum([ item ← !orders, ret item.price ]) in
            if total < 300 then
                let diff = 300 - total in
                new_log := "Spend another " ++ str(diff) ++
                    " dollars to get discount\n"
            else
                let min_price_opt =
                    min([ item ← !orders, ret item.price ]) in
                new_log :=
                    optmatch min_price_opt as min_price with
                        "Minimum price undefined: No orders\n"
                        :
                        "Qualified for a discount of " ++
                        str(min_price) ++ " dollars\n"
                    end
            end
        else
            new_log := "Invalid query\n"
        end
    end;
    log := !new_log :: !log
```

```
end
```

The heuristics from the LLM successfully recognized that the program optimization could benefit from a dictionary-based index for the inventory table, as well as a sum-aggregation index and a min-aggregation index for the table of orders.

In the experiment, we fixed the inventory size and increased the number of queries, with about half of the queries adding new orders to the table of orders. Figure 8.9 shows an asymptotic speedup from the source program to the optimized program. While Figure 8.1 shows that the Python-SQL program was about 10 times faster than the optimized Conquord program at the smallest input size, the performance difference diminished to below a factor of two as the number of queries increased to 10,000.

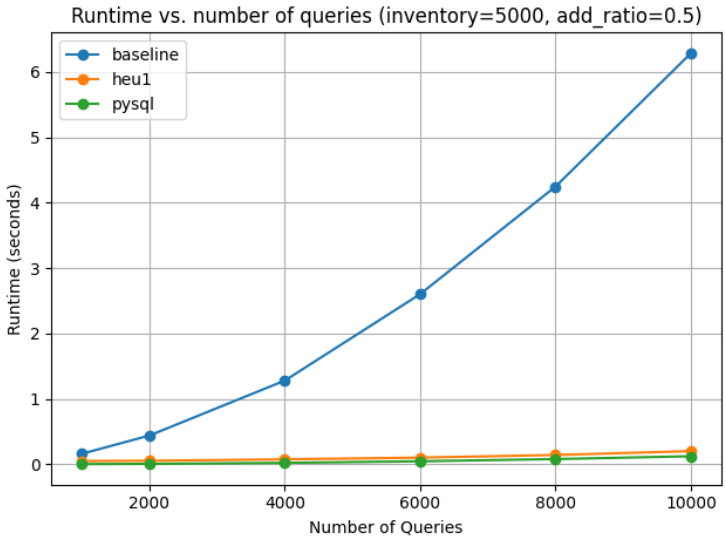

Figure 8.9: Runtime against the number of queries for variants of the "orders" example program.

8.7 Lines of Code

Table 8.2 shows the number of lines of code in each project file. The statistics are derived using the tool from [23]. A significant amount of effort was devoted to the definitions and the proofs about the language, such as the type system and type soundness. Common utilities in Utils.v and TransfUtils.v also take up a significant proportion of the code size. The numbers of lines of code in the different index definitions reflect the complexity of the indexes, from 223 lines in SumAgg.v to 1139 lines in BitmapIndex.v.

File	blank	comment	code
Utils.v	187	1	1744
TypeSound.v	95	2	1573
TypeSystem.v	47	5	1371
Optimize.v	60	20	1342
BitmapIndex.v	85	0	1139
IndexTransf.v	55	0	922
DictIndexImpl.v	54	2	896
RelTransf.v	79	4	780
Substitute.v	38	2	733
TransfUtils.v	49	3	693
CollectionTransf.v	46	2	593
Value.v	72	2	570
OptimizeAnno.v	59	0	535
Interpret.v	28	0	498
ParamPipelineEx.v	54	9	327
TransfSound.v	38	0	298
MinAgg.v	31	0	268
SumAgg.v	31	0	223
ToPython.v	20	5	216
Notations.v	37	13	205
PipelineEx.v	44	5	193
Language.v	17	11	189
DictIndexImplEx.v	27	5	106
Ex_Orders.v	20	4	96
Ex_Triangle.v	21	3	93
CombIdxEx.v	21	3	82
RelTransfEx.v	13	8	79
Ex_Survey.v	17	1	71
Ex_Employee.v	22	6	69
SumAggEx.v	19	2	67
Ex_BFS.v	22	27	58
Ex_Family.v	18	17	54
Ex_Sum.v	19	17	54
IndexInterface.v	3	0	26
SUM	**1448**	**179**	**16163**

Table 8.2: Lines of code

Chapter 9

Related Work

Formalized Query Languages A line of work by Benzaken and Contejean has demonstrated in Coq both a formal semantics of a significant subset of SQL [24] and verified query compilation to imperative code [25]. Other works have used algebraic structures to model subsets of SQL semantics and reason about the equivalence between SQL queries [26,27]. In this work, we avoid the baggage of literal adherence to SQL standards. Our language is more expressive than traditional query languages in two ways: it includes explicit control over data-structure selection, and it includes imperative control flow to allow writing more complex data-processing applications. This additional flexibility allows us to tackle the orthogonal challenge of integrating SQL-style compilation within a larger compiler for a general-purpose language.

Verified Compilers Past work on end-to-end-verified computing stacks has often involved high-level languages on top, starting with Lisp in the Computational Logic, Inc. stack [28]. A more recent example is CakeML [29] and its connection to a verified processor [30]. CakeML supports a high-level functional-programming experience, complete with garbage collection. Those functional programs can be written as either normal ML syntax or embedded inside the HOL4 proof assistant, verified in either case, with those results extended into full-stack correctness theorems. We continue this line of work and push the level of abstraction even higher by starting with a language of query comprehensions and making data-structure choices available to the optimizer.

Relational Query Optimization The rules we formalize in Chapter 4 are similar to those used in traditional rule-based SQL optimizers [31,32]. As in many databases, queries are written in a comprehension syntax similar to the relational calculus [33] and translated to the relational algebra [9] for rewriting. Unlike normal database optimizers, data-structure selection is part of the optimization process, with rules that examine the way that relations are accessed and introduce index structures accordingly.

While data-layout optimization is not usually treated as part of query optimization, the databases community has considered the problem. Work on automatic view and index selection [15–19] treats index introduction as an optimization problem where extra storage space is traded for increased query performance. Their work considers the problem in the

context of SQL databases, rather than the more general data-processing-program setting we use, and does not come with guarantees of correctness.

The idea of optimizing a data-processing program by introducing indexes in response to query workloads is not novel, nor are the particular index structures that we introduce. Our contribution lies in the formalization and verification of this process.

Data-Representation Optimization Prior work in programming languages and compilers has considered the problem of choosing an optimal data representation for a query or collection of queries [3,12–14]. The literature differs in approach (inductive [12–14] or deductive [3] program synthesis), expressivity of the query and data-structure language, and the level of automation, but in all cases the result of optimization is either unverified or checked using unsound bounded verification. We show how these sorts of ideas can be fully integrated into end-to-end system verification. By requiring mechanized proofs, we allow for safer component architectures, where tools can be taught new optimizations without endangering soundness.

The idea of equipping a high-level programming language with alternative types of collections has been an interesting topic on its own. SETL [34] is a language that has sets and tuples as compound data types, with imperative constructs like loops and conditional control flow. It has a sublanguage that permits optional annotations to guide the machine-level data representation. However, the runtime behavior of the annotated program is allowed to differ from that of the original program if errors or inconsistencies are present. Moreover, iterations over sets are allowed to have nondeterministic outputs, as the operation in the loop body may not be associative and commutative. Later works [35,36] have demonstrated the feasibility of automatically generating efficient implementations of a restricted subclass of programs written in such a language through top-down refinement. As in those works, we consider incremental computation and data-structure selection in our compiler, but instead of refinement steps, we prove semantic preservation using rewrite rules that ensure the programs before and after the transformation produce exactly the same result. Conquord avoids the nondeterminism by enforcing list semantics in the source program written by the user. The design choice seems to forgo the benefits from the greater variety of data structures that may be used to implement a bag or a set, but we recover the optimization opportunities through an analysis that rediscovers when a table may be interpreted using bag or set semantics without altering the result of the program.

Fiat [6] considers the problem of data-structure optimization in a verified context, but it implements its compiler transformations as Rocq proof scripts, which makes it particularly difficult to use and reduces performance. We instead deeply embed our query language in Gallina and implement our transformations as Gallina programs with proofs that they are equivalence-preserving. The resulting compiler is efficient and simple to use and extend.

Chapter 10

Conclusion and Future Work

The style of transformation we presented here represents the beginning of a potential verified pipeline from very high-level programs to native code with good performance. Most critically, the inference of bag and set semantics and choice of index structures for source-program tables anticipate later compiler phases where we expect our tables to be compiled to corresponding low-level data structures. Our starting point in high-level code with clear semantics of data access may also prove valuable for compiling to concurrent code, ideally implementing classic atomic database transactions. For instance, it may be possible to infer custom locking schemes through static analysis of applications.

So far, we implemented rewriting by directly pattern matching on ASTs like most compilers, but we plan to investigate the use of e-graphs [37–39] to do more robust rewriting modulo provable equalities. Since e-graphs remember terms as they are rewritten and select optimal outputs from among them, we hope that they will allow us to use more speculative rewrites for further optimization.

Building verified compilers can be quite labor-intensive, and one established way to divide the work into manageable units is structuring in terms of verified rewrite rules and untrusted heuristics for applying them. We have shown here that such an approach applies more broadly in database-related code than had previously been appreciated.

References

[1] Surabhi Gupta and Karthik Ramachandra. "Procedural extensions of SQL: understanding their usage in the wild." *Proc. VLDB Endow.*, **14**(8), Apr. 2021, pp. 1378–1391. ISSN: 2150-8097. DOI: 10.14778/3457390.3457402. URL: https://doi.org/10.14778/3457390.3457402.

[2] William Cook and Ali Ibrahim. "Integrating Programming Languages & Databases: What's the Problem?" Jan. 2005.

[3] John Feser, Sam Madden, Nan Tang, and Armando Solar-Lezama. "Deductive optimization of relational data storage." *Proceedings of the ACM on Programming Languages*, **4**(OOPSLA), 2020, pp. 1–30.

[4] Wilmer Ricciotti and James Cheney. "A Formalization of SQL with Nulls." *Journal of Automated Reasoning*, **66**(4), 2022, pp. 989–1030.

[5] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. "Links: Web programming without tiers." In: *International Symposium on Formal Methods for Components and Objects*. Springer. 2006, pp. 266–296.

[6] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. "Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant." In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 689–700. DOI: 10.1145/2676726.2677006. URL: https://doi.org/10.1145/2676726.2677006.

[7] Phil Trinder and Philip Wadler. "Improving list comprehension database queries." In: *Fourth IEEE Region 10 International Conference TENCON*. 1989, pp. 186–192. DOI: 10.1109/TENCON.1989.176921.

[8] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. "Comprehension Syntax." English. *SIGMOD Record*, **23**(1), Mar. 1994, pp. 87–96. DOI: 10.1145/181550.181564.

[9] Edgar Frank Codd. "A relational model of data for large shared data banks." *Commun. ACM*, **13**(6), June 1970, pp. 377–387. ISSN: 0001-0782. DOI: 10.1145/362384.362685. URL: https://doi.org/10.1145/362384.362685.

[10] Sai Wu, Ying Li, Haoqi Zhu, Junbo Zhao, and Gang Chen. "Dynamic index construction with deep reinforcement learning." *Data Science and Engineering*, **7**(2), 2022, pp. 87–101.

[11] Stéphane Azefack, Kamel Aouiche, and Jérôme Darmont. "Dynamic index selection in data warehouses." In: *2007 Innovations in Information Technologies (IIT)*. IEEE. 2007, pp. 1–5.

[12] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. "Data representation synthesis." In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. Ed. by Mary W. Hall and David A. Padua. ACM, 2011, pp. 38–49. DOI: 10.1145/1993498.1993504. URL: https://doi.org/10.1145/1993498.1993504.

[13] Calvin Loncaric, Michael D. Ernst, and Emina Torlak. "Generalized data structure synthesis." In: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Ed. by Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman. ACM, 2018, pp. 958–968. DOI: 10.1145/3180155.3180211. URL: https://doi.org/10.1145/3180155.3180211.

[14] Cong Yan and Alvin Cheung. "Generating Application-Specific Data Layouts for In-Memory Databases." *Proc. VLDB Endow.*, **12**(11), 2019, pp. 1513–1525. DOI: 10.14778/3342263.3342630. URL: http://www.vldb.org/pvldb/vol12/p1513-yan.pdf.

[15] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. "Index Selection for OLAP." In: *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997, Birmingham, UK*. Ed. by W. A. Gray and Per-Åke Larson. IEEE Computer Society, 1997, pp. 208–219. DOI: 10.1109/ICDE.1997.581755. URL: https://doi.org/10.1109/ICDE.1997.581755.

[16] Zohreh Asgharzadeh Talebi, Rada Chirkova, Yahya Fathi, and Matthias F. Stallmann. "Exact and inexact methods for selecting views and indexes for OLAP performance improvement." In: *EDBT 2008, 11th International Conference on Extending Database Technology, Nantes, France, March 25-29, 2008, Proceedings*. Ed. by Alfons Kemper, Patrick Valduriez, Noureddine Mouaddib, Jens Teubner, Mokrane Bouzeghoub, Volker Markl, Laurent Amsaleg, and Ioana Manolescu. Vol. 261. ACM International Conference Proceeding Series. ACM, 2008, pp. 311–322. DOI: 10.1145/1353343.1353383. URL: https://doi.org/10.1145/1353343.1353383.

[17] Michael Stonebraker. "The choice of partial inversions and combined indices." *International Journal of Parallel Programming*, **3**(2), 1974, pp. 167–188. DOI: 10.1007/BF00976642. URL: https://doi.org/10.1007/BF00976642.

[18] Nicolas Bruno and Surajit Chaudhuri. "Automatic Physical Database Tuning: A Relaxation-based Approach." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*. Ed. by Fatma Özcan. ACM, 2005, pp. 227–238. DOI: 10.1145/1066157.1066184. URL: https://doi.org/10.1145/1066157.1066184.

[19] Rafi Ahmed, Randall Bello, Andrew Witkowski, and Praveen Kumar. "Automated generation of materialized views in Oracle." *Proceedings of the VLDB Endowment*, **13**(12), 2020, pp. 3046–3058.

[20] Mohit Kumar Gupta, Vishal Verma, and Megha Singh Verma. "In-memory database systems—A paradigm shift." *arXiv preprint arXiv:1402.1258*, 2014.

[21] Benjamin C. Pierce and David N. Turner. "Local type inference." *ACM Transactions on Programming Languages and Systems*, **22**(1), 2000, pp. 1–44.

[22] OpenAI. *GPT-5.4 [Large Language Model]*. https://openai.com. Accessed: 2026-04-06. 2026.

[23] Albert Danial. *cloc: v2.06*. Version v2.06. June 2025. DOI: 10.5281/zenodo.15734241. URL: https://doi.org/10.5281/zenodo.15734241.

[24] Véronique Benzaken and Évelyne Contejean. "A Coq Mechanised Formal Semantics for Realistic SQL Queries: Formally Reconciling SQL and Bag Relational Algebra." In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2019. Cascais, Portugal: Association for Computing Machinery, 2019, pp. 249–261. ISBN: 9781450362221. DOI: 10.1145/3293880.3294107. URL: https://doi.org/10.1145/3293880.3294107.

[25] Véronique Benzaken, Évelyne Contejean, Mohammed Houssem Hachmaoui, Chantal Keller, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. "Translating Canonical SQL to Imperative Code in Coq." *Proc. ACM Program. Lang.*, **6**(OOPSLA1), Apr. 2022. DOI: 10.1145/3527327. URL: https://doi.org/10.1145/3527327.

[26] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. "HoTTSQL: proving query rewrites with univalent SQL semantics." In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 510–524. ISBN: 9781450349888. DOI: 10.1145/3062341.3062348. URL: https://doi.org/10.1145/3062341.3062348.

[27] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. "Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries." *arXiv preprint arXiv:1802.02229*, 2018.

[28] Strother J Moore. "A Grand Challenge Proposal for Formal Methods: A Verified Stack." *10th Anniversary Colloquium of UNU/IIST*, 2002, pp. 161–172. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.7.8607&rep=rep1&type=pdf.

[29] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. "CakeML: A Verified Implementation of ML." In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. 10.1145/2535838.2535841. San Diego, California, USA: Association for Computing Machinery, 2014, pp. 179–191. ISBN: 9781450325448. URL: https://ts.data61.csiro.au/publications/nicta_full_text/7494.pdf.

[30] Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony C. J. Fox. "Verified compilation on a verified processor." In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by Kathryn S. McKinley and Kathleen Fisher. ACM, 2019, pp. 1041–1053. DOI: 10.1145/3314221.3314622. URL: https://doi.org/10.1145/3314221.3314622.

[31] Surajit Chaudhuri. "An Overview of Query Optimization in Relational Systems." In: *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*. Ed. by Alberto O. Mendelzon and Jan Paredaens. ACM Press, 1998, pp. 34–43. DOI: 10.1145/275487.275492. URL: https://doi.org/10.1145/275487.275492.

[32] Matthias Jarke and Jürgen Koch. "Query Optimization in Database Systems." *ACM Comput. Surv.*, **16**(2), 1984, pp. 111–152. DOI: 10.1145/356924.356928. URL: https://doi.org/10.1145/356924.356928.

[33] Edgar Frank Codd. "A Database Sublanguage Founded on the Relational Calculus." In: *Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, California, USA, November 11-12, 1971*. Ed. by E. F. Codd and A. L. Dean. ACM, 1971, pp. 35–68.

[34] Jacob T Schwartz, Robert B.K. Dewar, Edward Dubinsky, and Edith Schonberg. *Programming with sets: An introduction to SETL*. Springer Science & Business Media, 2012.

[35] Jacob T Schwartz. "Automatic data structure choice in a language of very high level." In: *Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '75. Palo Alto, California: Association for Computing Machinery, 1975, pp. 36–40. ISBN: 9781450373517. DOI: 10.1145/512976.512981. URL: https://doi.org/10.1145/512976.512981.

[36] Robert Paige and Fritz Henglein. "Mechanical translation of set theoretic problem specifications into efficient RAM code—A case study." *Journal of Symbolic Computation*, **4**(2), 1987, pp. 207–232. ISSN: 0747-7171. DOI: https://doi.org/10.1016/S0747-7171(87)80066-4. URL: https://www.sciencedirect.com/science/article/pii/S0747717187800664.

[37] Greg Nelson and Derek C. Oppen. "Fast Decision Procedures Based on Congruence Closure." *J. ACM*, **27**(2), Apr. 1980, pp. 356–364. ISSN: 0004-5411. DOI: 10.1145/322186.322198. URL: https://doi.org/10.1145/322186.322198.

[38] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. "Egg: Fast and Extensible Equality Saturation." *Proc. ACM Program. Lang.*, **5**(POPL), Jan. 2021. DOI: 10.1145/3434304. URL: https://doi.org/10.1145/3434304.

[39] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. "Better together: Unifying Datalog and equality saturation." *Proceedings of the ACM on Programming Languages*, **7**(PLDI), 2023, pp. 468–492.