Using Relational Algebra to Facilitate Safe Refactoring for Performance in Rust

UNIVERSITY OF ICELAND
SCHOOL OF ENGINEERING AND NATURAL SCIENCES
FACULTY OF INDUSTRIAL ENGINEERING, MECHANICAL ENGINEERING AND COMPUTER SCIENCE

Eric Michael Sumner

Supervisors: Snorri Agnarsson and Hjálmtýr Hafsteinsson, Háskóli Íslands

May 2021

Purpose

A common goal for refactoring projects is to improve the performance of a piece of software without affecting its overall functionality. Achieving this goal often requires the programmer to alter the program's internal data layout to enable the use of algorithms with asymptotically better performance. Unfortunately, a program's domain logic is often tightly coupled with the chosen data layout, such that changing the layout requires extensive changes in the program's logic routines. In addition to being costly, this creates a risk of unintentionally introducing a change in behavior. The goal of this project is to develop a mechanism that mitigates these problems by decoupling domain logic code from the internal layout of data.

Background

An early attempt to address this problem, *information hiding*, was presented by David Parnas. He proposed that programs should be divided into modular components, and that the interface between these components should be defined in terms of functionality rather than any particular data structure. This allows the algorithms and data structures used within a module to be changed without affecting the other modules.

Around the same time, E. F. Codd recognized a similar problem in the design of database systems. Contemporary databases were tree-structured and provided only limited access paths to the data they contained; reorganizing the data in response to changing demands regularly broke these access paths, requiring extensive changes to the programs and processes that depended on the data. To address this problem, he developed *relational algebra*, a mathematical model of data that is independent of how it is represented internally.

In the intervening decades, both of these developments have proven to be pivotal innovations in their respective domains. Both Parnas and Codd were awarded ACM Fellowships for their work, and Codd received a Turing award for the invention of relational algebra.

The project: Memquery

Memquery is a framework for managing a program's internal data, based on the principles of relational algebra. It is designed to reduce the coupling between three distinct programming roles:

Application programmers who are primarily concerned with the correctness of a single feature or use case of a program,

Architects who are primarily concerned with the overall performance and maintainability of a program, and

Library authors who are primarily concerned with inventing new data organization schemes that can be used in multiple programs.

Memquery presents an abstract, relational, view of a program's data to application programmers which is independent of the data layout chosen by the architect. The run-time cost of this abstraction is minimized by leveraging Rust's algebraic type system and other compile-time programming features, which allows query planning to occur during program compilation.

Listing 1 shows typical usage. (a) is the schema definition, which describes how the data is laid out in memory. In this case, it consists of three relations: part descriptions, project descriptions, and a commitment relation describing the number of each part assigned to various projects. (b) is a piece of code that, given a project name, will print the name and quantity of all the parts committed to that project. Changing the indexing strategy or adding additional fields to the schema does not require any change to this query code.

***Listing 1** Sample Memquery usage*

```
// (a) Schema definition
//     NB: Column definitons omitted
pub struct Inventory {
    parts: BTreeIndex<PartId, Option<(PartId, PartName)>>,
    projects: BTreeIndex<ProjectId, Option<(ProjectId, ProjectName)>>,
    commits: RedundantIndex<ProjectId, PartId,
                BTreeIndex<PartId, Vec<(ProjectId, PartId, Quantity)>>>,
}

// (b) Sample query
for (qty, name) in self.projects.as_ref()
                        .join(self.commits.as_ref())
                        .join(self.parts.as_ref())
                        .where_eq(&ProjectName(String::from(proj_name)))
                        .iter_as::<(Quantity, PartName)>() {
    println!("Part #{}: {} units committed", name.0, qty.0);
}
```

Case Study

Codd describes five different data structures for an inventory management problem; two indexing strategies were chosen for each structure. Each of these ten solutions (see Table 1) was implemented with both Memquery and traditional techniques. The performance of each Memquery implementation is then compared with its traditional counterpart, and the query code is analyzed for maintainability.

Description	Structure	Part Index	Project Index	Commitment Index
Projects subordinate to parts	1a	Id	Id	
	1b	Id	Name	
Parts subordinate to projects	2a	Id	Id	
	2b	Id	Name	
Parts and projects as peers, commitment relationship subordinate to projects	3a	Id	Id	Part Id
	3b	Id	Name	Part Id
Parts and projects as peers, commitment relationship subordinate to parts	4a	Id	Id	Project Id
	4b	Id	Name	Project Name
Parts, projects, and commitment relationship as Peers	5a	Id	Id	(Part Id, Project Id)
	5b	Id	Id	(Project Name, Part Id)

***Table 1** Studied data structures*

Results

Figure 1 shows the performance results of a sample query for each of the implemented data structures; (a) shows the Memquery results and (b) shows the results for a traditional approach. In each case, Memquery's performance is comparable to the corresponding traditional implementation.

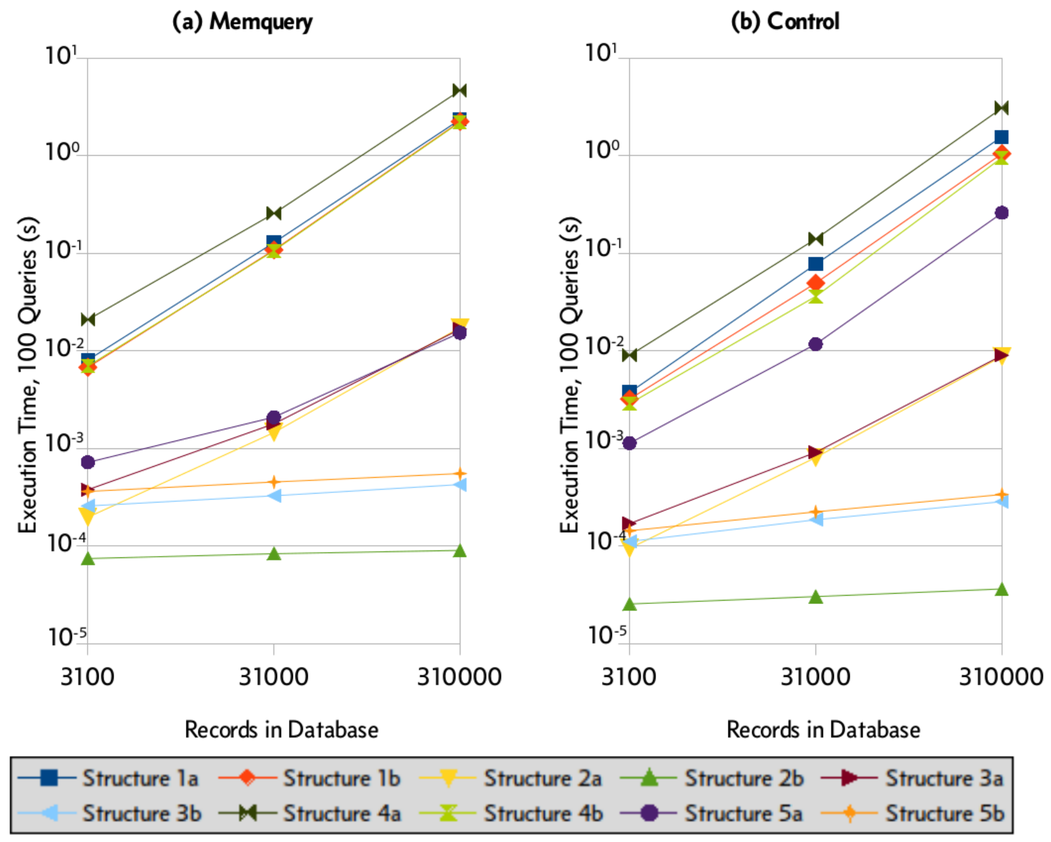

***Figure 1** Performance of Sample Queries*

In the control implementations, the query code is tightly coupled to the chosen data structure: Changing to a different structure requires re-engineering the query from first principles. All of the Memquery implementations, on the other hand, use nearly the same query code: Adopting a new structure for the data requires only minimal changes, which can be applied mechanically.

Discussion and further work

The Memquery prototype system demonstrates that relational algebra can be used in general-purpose programming to decouple program logic from the organization of data in memory, while retaining the performance benefits that come from reorganizing that data.

Further investigation is required to determine the best way to integrate this capability into modern software development practices. Additionally, the prototype does not currently support some operations commonly provided by databases, such as grouped and aggregate queries.

References

D. L. Parnas. 1972. *On the criteria to be used in decomposing systems into modules*. Commun. ACM 15, 12 (Dec. 1972), 1053–1058. DOI:https://doi.org/10.1145/361598.361623

E. F. Codd. 1970. *A relational model of data for large shared data banks*. Commun. ACM 13, 6 (June 1970), 377–387. DOI:https://doi.org/10.1145/362384.362685