Multiquery Optimization for Declarative Compilers

Darshana Balakrishnan
darshbal@amazon.com
Amazon
Toronto, Canada

Oliver Kennedy
okennedy@buffalo.edu
University at Buffalo
Buffalo, NY, USA

Lukasz Ziarek
lziarek@buffalo.edu
University at Buffalo
Buffalo, NY, USA

Johannes Luong
jluong@amazon.de
Amazon
Berlin, Germany

Hinnerk Gildhoff
hinnerk@amazon.de
Amazon
Berlin, Germany

Gaurav Saxena
gssaxena@amazon.com
Amazon
California, USA

1 Introduction

Program analyses grapple with exceedingly large state spaces, a struggle that has historically required breaking down high-level analyses into smaller, manageable chunks (*e.g.*, separate compilation units and limited context-sensitivity). Unfortunately, such a granular analysis methodology can prohibit the idiomatic expression of useful analyses, and the resulting loss of precision limits optimizations. For example Rust's borrow checker can add minutes to compilation times, and at the scale of the Linux Kernel, non-trivial cross-module optimizations are simply not possible. In this talk, we will outline our initial efforts to create a compiler capable of scalable, orders-of-magnitude faster, cross-module program analysis, optimization, and compilation, without increased development complexity. In doing so, our hope is to make feasible many optimizations and analysis techniques thus far unexplored.

Multiple recent efforts have attempted to tackle compiler scalability challenges, for example by leveraging incremental view maintenance to avoid repeated tree-traversals during optimization [1], or by leveraging distributed datalog engines to scale up program analysis [2, 4]. These efforts share a common strategy of re-casting compiler problems (i.e., program analysis and optimization tasks) into a declarative, relational abstraction, and then leveraging already existing relational query processing techniques. By implementing the compiler declaratively, these approaches decouple the compiler's implementation from its performance: **(i)** Compilation performance improvements that may be too complicated or too small to implement safely by hand can be automated; **(ii)** Compilers be implemented through simpler, more idiomatic development patterns, avoiding complexity for the sake of reduced compile times; **(iii)** Static analysis can be used to assert soundness, completeness, and/or stability of the compiler itself; and **(iv)** Data structures that encode the program are decoupled from compiler logic, allowing substantial reorganization with no change to the compiler. In short, a compiler whose rules, analyses, and optimizations are specified declaratively can be analyzed, and accelerated like queries in a database.

Declarative Compiler Engines. Although similar to a database, a declarative compiler engine is unique in several key respects. First, like graph databases, declarative compilers involve high-width joins (e.g., subgraph isomorphism over abstract syntax trees), and recursive queries (e.g., existential tests over subtrees). However, unlike graph databases, joins often have low fan-outs [3, 4], or even simply foreign-key joins. Finally, declarative compilers feature large numbers of queries (e.g., Spark 3.2 has over 300 distinct optimization rules, each being effectively a query), all known upfront, and with many work-sharing opportunities.

2 Multiquery Optimization

The search for optimization opportunities, in particular accounts for up to 50% of the runtime of Spark's Optimizer and up to 20% of Orca's optimizer [1], and is a huge source of work sharing opportunities. For example, Spark's fixed point optimizer performs a full tree traversal for every one of its 300 rules, with each traversal applying predicates to the nodes of the tree, and replacing nodes when the predicate succeeds (i.e., when an optimization is found). If a node is replaced, the process must restart, and all rules must be applied again. As a result, the optimizer's runtime is $O(|\text{Rules}| \cdot |\text{Abstract Syntax Tree}|)$ *per fixed point iteration.*

Fortunately for us, many of these passes are replicating work. For example, Constant Folding and Predicate Pushdown, both standard rules, both unwrap and manipulate Selection (Filter) operators. This work can be shared if both rules (along with all other Filter-oriented rules) are in-lined into a single pass: (i) Unwrapping and matching structured data (e.g., S-expressions or Scala Case Classes) only needs to happen once; (ii) Expensive predicates only need to be evaluated once; (iii) The resulting access pattern is more cache friendly for especially large queries.

Our talk will focus on our preliminary efforts on multiquery optimization, leveraging opportunities for work sharing in query optimization. We will focus, in particular on one especially costly predicate: Recursive predicates. For example, consider an optimization rule that removes 'Distinct' operators in a relational algebra query when the subquery is already distinct. Determining whether a subquery is already distinct may require a recursive traversal of the tree, leading to an $O(N^2)$ runtime in tree size, where every candidate node triggers a recursive tree traversal. We will outline our approach to inlined evaluation of sets of optimization rules containing partly overlapping, recursive predicates.

References

[1] Darshana Balakrishnan, Carl Nuessle, Oliver Kennedy, and Lukasz Ziarek. 2021. TreeToaster: Towards an IVM-Optimized Compiler. In *SIGMOD*.

[2] Thomas Gilray, Sidharth Kumar, and Kristopher K. Micinski. 2021. Compiling data-parallel Datalog. In *CC*. ACM, 23–35.

[3] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *POPL*. ACM, 264–276.

[4] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proc. ACM Program. Lang.* 7, PLDI (2023), 468–492.