My mission is to shape a future where anyone can write programs to gain insights from data. I believe this future will be built on an emerging paradigm of programming which I call **relational programming**. My research develops languages, theories and tools that make relational programming practical and scalable.

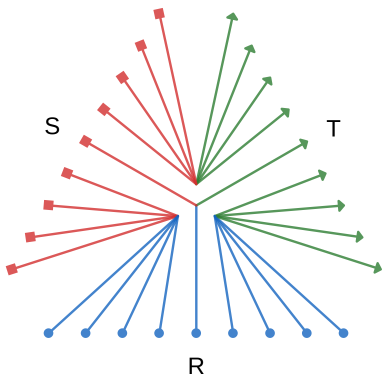
Relational programming stands on the same foundation supporting relational databases, namely the relational model of data. The relational model has established itself as the de-facto abstraction for tabular data, empowering analysts to wrangle information with ease, while at the same time enabling engineers to build effective systems. Relational programming emerges from this relational foundation, but evolves far beyond the traditional paradigm of running simple queries over tables. On one hand, data may come in a variety of forms that do not resemble tables, be it graphs, matrices, or tensors. They may conform to quantitative constraints such as sparsity (in matrices), or qualitative constraints such as acyclicity (in graphs). On the other hand, a relational program can be far more complex than a SQL query, involving intricate mathematical operations running iteratively. This increased complexity in both data and program poses many challenges to building effective platforms for relational programming. My research tackles these challenges in three thrusts:

- A new relational programming language (Section 1)
- A new algorithm for the relational join (Section 2)
- Optimization techniques for relational programs (Section 3)

1. A Relational Programming Language

Today's data analysts increasingly leverage machine learning (ML) algorithms in their workflow. These ML algorithms are usually specified in the language of linear algebra and tensor algebra. The algorithms are usually iterative, repeating computation until certain convergence criteria are met. Such ML-powered workflow calls for new programming abstractions that serve two purposes: first, they should allow the analyst to specify complex algorithms over matrices and tensors; second, they should assist the data system to optimize and execute the algorithms efficiently. My research proposes a new relational programming language, $\mathsf{Datalog}^\circ$ [1], that achieves precisely these goals. $\mathsf{Datalog}^\circ$ was developed in collaboration with researchers from RelationalAI who have integrated the language into their product. Our work on $\mathsf{Datalog}^\circ$ received a best paper award at PODS 2022.

As the name suggests, $\mathsf{Datalog}^\circ$ descends from Datalog, a relational language that was designed for iterative algorithms. The key idea of $\mathsf{Datalog}^\circ$ is a simple yet powerful extension to the relational model of data: instead of viewing each relation simply as a set of tuples, we augment the relation with an algebraic structure called a semiring. Each semiring comes with a set of values and two operations: sum and product. This extension is known in the literature as K-relations. The extension allows $\mathsf{Datalog}^\circ$ to model a tensor by attaching a value to each tuple of coordinates, and model tensor operations by composing semiring operations with the relational join. For example, matrix multiplication can be expressed with the simple $\mathsf{Datalog}^\circ$ program $C(i,j) = \sum_k A(i,k) \times B(k,j)$ which is essentially the same as the Einstein notation $C_{ij} = \sum_k A_{ik} \times B_{kj}$. The true power of $\mathsf{Datalog}^\circ$ lies in its flexibility to allow user-specified semirings. That is, a $\mathsf{Datalog}^\circ$ programmer has the freedom



```
for (x, a) in R:
  s = S[x]?
  for b in s:
    t = T[x]?
    for c in t:
      output(x,a,b,c)
```

(A) Binary join

```
for (x, a) in R:
  s = S[x]?
  t = T[x]?
  for b in s:
    for c in t:
      output(x,a,b,c)
```

(B) Free Join

FIGURE 1. Computing the query $Q(x, a, b, c) = R(x, a), S(x, b), T(x, c)$. Every edge represents a tuple; x values are at the center, and a, b, and c values are on the peripheral. The three edges at the center make up the only output. The `?` operator indicates a *short-circuiting* lookup that continues to the enclosing loop upon failure. We optimize binary join to Free Join by *factoring out* the underlined lookup.

to customize the value domain as well as the sum and product operations to meet their needs. For example, one may choose min to be the semiring sum, and + to be the semiring product, in which case the program above becomes $C(i, j) = \min_k A(i, k) + B(k, j)$ which finds the length of the shortest 2-path between any pair of vertices. While being flexible, Datalog$^\circ$ retains many of the elegant properties of Datalog. In particular, it has first-class support for recursion with a rigorous semantics. The combination of customizable semirings and built-in recursion makes Datalog$^\circ$ very expressive: we have used it to define the classic gradient descent algorithm, Dijkstra's algorithm for finding shortest paths, and Brandes' algorithm for betweenness centrality.

Nevertheless, having an expressive language is not enough, as any general purpose language can express all aforementioned algorithms. We have designed Datalog$^\circ$ to balance expressiveness with performance. In the following sections I describe techniques to efficiently evaluate and optimize relational programs, all of which are applicable to Datalog$^\circ$.

2. A New Join Algorithm

The central operation over relational data is the relational join, which combines information from different sources and composes computation from different queries. Join algorithms are the workhorse of relational data systems, and modern data analytics demand even greater efficiency from them. My research develops a new join algorithm called Free Join [6] that generalizes and improves upon state-of-the-art algorithms.

Unlike traditional database workloads where foreign key relationships are prevalent, analysts today frequently face data with heavy skew. For example, in Figure 1 the query Q returns only a single output on the given input, yet the binary join of any two relations produces $O(N^2)$ results. Another challenge comes from the structure of the query. Traditional relational queries are often acyclic, and most database systems compute them with binary joins. In contrast, modern relational programs are often cyclic. For example, many analyses over social networks depend on finding triangles in graphs. The combination of cyclic queries and skewed data degrades the performance of traditional databases, and classic join algorithms are known to be asymptotically suboptimal on such workloads.

Free Join tackles these challenges by unifying binary joins with a new breed of theoretically optimal algorithms called Worst-case Optimal Joins (WCOJs). Similar to WCOJs, Free Join computes cyclic queries efficiently in the presence of skew. At the same time, key techniques developed for binary joins, like column-oriented layout and vectorized execution can also be naturally adapted for Free Join. Figure 1b shows the execution of Free Join to compute Q. Note that the program runs in time linear to the size of the relations, in contrast to the quadratice run time of binary join in Figure 1a.

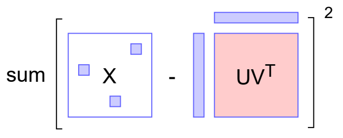
A core contribution of our work is an algorithm that compiles any binary join plan into a more efficient Free Join plan. As an example, we compile the binary join in Figure 1a to Free Join in Figure 1b by simply *factoring out* the underlined lookup. This compilation algorithm is of great pratical importance, because it allows system builders to integrate Free Join more easily. Thanks to this algorithm, we were able to *plug in* Free Join to reuse the query optimizer of DuckDB, a state-of-the-art database system. Starting from the same query plan, Free Join consistently outperforms both DuckDB (up to 19x) and a WCOJ baseline (up to 30x), on both cyclic and acyclic queries.

3. Optimizing Relational Programs

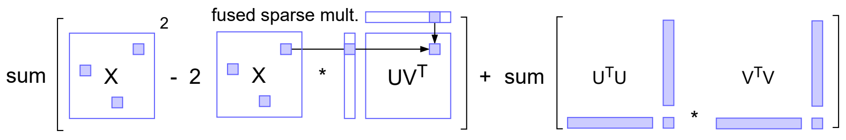
The query optimizer lies at the heart of any relational data system. It allows the analyst to specify computation declaratively at a high level, yet be assured the results will be computed efficiently. This section describes my research on optimizing relational programs, including a general framework for building optimizers (Section 3.1, [2, 7, 9, 10]), an optimizer for large scale linear algebra (Section 3.2, [3]), an optimizer for recursive queries (Section 3.3, [4]), as well as applications in other domains (Section 3.4, [7, 8, 9]).

3.1. **A Framework for Building Optimizers.** My collaborators at UW have been developing a new framework for building optimizers called *equality saturation* (EqSat), and I have contributed to all major milestones of the project [2, 7, 9, 10]. The core principle of EqSat is to represent a large collection of equivalent programs using a compact data structure, and expand this collection by applying a set of simple equivalence axioms. EqSat is able to represent exponentially many programs with little space, thanks to which it is able to cover a much larger search space than traditional optimizers. While earlier versions of our EqSat system used specialized data structures and algorithms, I recognized that *the core* EqSat *algorithm can be formulated as a relational program.* This insight has steered us to improve EqSat with relational database techniques, improving its correctness and performance [9, 10]. Our work on EqSat has received two distinguished paper awards at POPL and OOPSLA.

3.2. **An Optimizer for Large Scale Linear Algebra.** The first optimizer I built with EqSat, called SPORES, aims to speed up large-scale linear algebra computation [3]. Traditionally, linear algbebra systems implement ad hoc optimization rules, and apply them with heuristics. These rules are *incomplete*, in the sense that they often fail to cover important optimizations. For example, Figure 2a shows the computation of the expression $\text{sum}[(X-UV^\top)^2]$, which involves materializing a dense intermediate matrix. Figure 2b shows an equivalent expression that avoids the materialization. This new expression is cheaper to



(A) Original program.



(B) Optimized program.

FIGURE 2. The linear algebra program sum$[(X - UV^\top)^2]$ before and after optimization. Note X is a sparse matrix; $*$ and $\cdot^2$ are point-wise. The original program materializes a dense matrix $UV^\top$, while the optimized program uses a fused sparse multiplication $X * UV^\top$ to avoid the materialization.

evaluate, but the optimizer of SystemML, a state-of-the-art linear algebra system, fails to discover it.

Instead of directly rewriting the linear algebra code, we first translate the program into relational algebra (RA) over semirings (essentially non-recursive Datalog°). We then optimize the RA code using a set of rewrite rules which we prove to be *complete*. In other words, two RA programs are equivalent if and only if we can rewrite one to the other using these rules. We then supply these rewrite rules to our EqSat system which applies the rules in all possible orders to obtain a large collection of equivalent programs. Finally, to extract the most efficient program from the compact data structure used by EqSat, we encode the search problem as an integer linear program (ILP). SPORES automatically discovers the optimization in Figure 2 by chaining together multiple rewrite rules.

3.3. **An Optimizer for Recursive Queries.** While SPORES can optimize non-recursive linear algebra programs, many pratical relational programs are recursive. My research proposes a new optimization rule, called the FGH rule, to optimize recursive programs [4]. The FGH rule captures several classic optimizations for Datalog like semi-naïve evaluation, magic sets, and pushing extrema into recursion. Nevertheless, the FGH rule does not prescribe an algorithm to directly transform programs into their optimized forms. Instead, we make use of a powerful tool from programming languages called counterexample-guided inductive synthesis (CEGIS). CEGIS in turn leverages SMT solvers to reason about the complex arithmetic operations in relational programs. This has enabled us to perform *semantic optimizations* that exploit global data properties like acyclicity, by encoding such properties as SMT formulas. All told, our optimizer achieves orders-of-magnitude speedups across different programs. It can even automatically change the underlying algorithm of the program, lowering its asymptotic complexity.

3.4. **Other Applications.** My collaborators and I have applied the techniques described above to many other domains. This includes improving the accuracy of floating point computation [7, 9], speeding up deep learning inference [8], and improving points-to analysis for programming languages [9]. Our optimizers improve performance beyond the state of the art while taking less time to run.

4. A Vision for Relational Programming

Short-term plans. My research has focused on improving the performance of relational programs, and I plan to continue this work in two directions, going both down to the hardware level and up to higher levels of theoretical understanding. At the hardware level, I am experimenting with building a relational data system on top of tensor processing engines. In particular, the TACO compiler for tensor algebra kernels has extensive support for hardware acceleration, which can make it easier to leverage specialized hardwares to speed up relational programming. My extension of TACO to support WCOJ has already been merged into the system. At the theoretical level, I have started to uncover connections among equality saturation, the classic Chase algorithm, and finite state tree automata [5]. These connections can push forward research in all three subjects, producing new theorems and tools.

Long-term plans. For relational programming to succeed, it needs much more than just performance. Popular programming languages provide many important tools to help the programmer write correct and efficient code. These include debuggers, static analyzers, and profilers. Since the optimizer for relational programs can completely change the underlying algorithm, it can be challenging to understand why a program produces incorrect results, or why it runs slow. As a first step in this direction, I have designed and implemented a static analysis that helps the programmer avoid non-terminating code. The analysis now runs on every program written in the relational programming language developed by RelationalAI.

The future of relational programming is in the hands of a new generation of system builders. Research aiming to help these system builders will therefore have great impact. The LLVM compiler infrastructure has revolutionized the field of programming languages, empowering people with limited resources to develop their own languages. My long-term research goal is to develop similar infrastructure for building relational programming systems. I believe such infrastructure will shape the landscape of research and practice.

References

[1] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and **Yisu Remy Wang**. Convergence of datalog over (pre-) semirings. In Leonid Libkin and Pablo Barceló, editors, *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022,* **Best Paper Award**, pages 105–117. ACM, 2022.

[2] Chandrakana Nandi, Max Willsey, Amy Zhu, **Yisu Remy Wang**, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. Rewrite rule inference using equality saturation. *Proc. ACM Program. Lang.*, 5(OOPSLA, **Distinguished Paper Award**):1–28, 2021.

[3] **Yisu Remy Wang**, Shana Hutchison, Dan Suciu, Bill Howe, and Jonathan Leang. SPORES: sum-product optimization via relational equality saturation for large scale linear algebra. *Proc. VLDB Endow.*, 13(11):1919–1932, 2020.

[4] **Yisu Remy Wang**, Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, and Dan Suciu. Optimizing recursive queries with progam synthesis. In Zachary Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 79–93. ACM, 2022.

[5] **Yisu Remy Wang**, James Koppel, Altan Haan, and Josh Pollock. E-graphs, VSAs, and Tree Automata: a rosetta stone. *EGRAPHS*, 2022.

[6] **Yisu Remy Wang**, Max Willsey, and Dan Suciu. Free join: Unifying worst-cast optimal and traditional joins, 2022 (*in submission*).

[7] 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, **Distinguished Paper Award**):1–29, 2021.

[8] Yichen Yang, Phitchaya Mangpo Phothilimthana, **Yisu Remy Wang**, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. In Alex Smola, Alex Dimakis, and Ion Stoica, editors, *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021.* mlsys.org, 2021.

[9] Yihong Zhang, **Yisu Remy Wang**, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Max Willsey, and Zachary Tatlock. Better together: Unifying datalog and equality saturation, 2022 (*in submission*).

[10] Yihong Zhang, **Yisu Remy Wang**, Max Willsey, and Zachary Tatlock. Relational e-matching. *Proc. ACM Program. Lang.*, 6(POPL):1–22, 2022.