PLDI: U: Towards a Relational E-graph

Yihong Zhang
University of Washington
yz489@cs.washington.edu

1 Problem & Motivation

The congruence closure data structure, also known as the e-graph, compactly represents a set of terms and an equivalence relation over the terms. It is a central component of equality saturation-based optimizers [13, 15]. It has been rejuvenated with the recent egg framework and proven successful in many areas, including floating-point arithmetic [12], ML compilers [16], and digital signal processors [14]. Such applications typically populate a large number of equivalence classes (or e-classes). Therefore, the performance of e-graph frameworks is key to the success of modern e-graph based applications. E-matching, the procedure of performing pattern matching over an e-graph, is a major bottleneck in these modern e-graph workloads. In a typical application [15], e-matching is responsible for 60–90% of the overall run time.

The wide variety of e-graph applications is also placing new demands on the capability of e-graph frameworks. For example, e-graph–based tensor graph optimizers [16] use a standard extension to e-matching called multi-patterns. Efficient support for multi-patterns requires complicated modification of the basic backtracking algorithm provided. Most existing frameworks either do not support multi-patterns or support them inefficiently. As another example, practical applications may interleave equational reasoning with non-equational ones. However, non-equational reasoning like logical implication is fairly non-trivial and potentially inefficient in existing e-graph frameworks like egg.

To improve the performance and expressiveness of e-graphs, we argue that a systematic approach to e-graphs should be based on relational databases. As the first step, we propose to solve e-matching on an e-graph by reducing it to answering conjunctive queries on a relational database, named **relational e-matching**. There are several benefits. First, by reducing e-matching to conjunctive queries, we simplify e-matching by taking advantage of decades of study by the database community. Second, the relational representation unifies different kinds of constraints in e-matching patterns and allows query optimizers to generate asymptotically faster query plans in many cases. Finally, by leveraging the generic join algorithm, a recent theoretical advance in the database community, our technique achieves the first worst-case optimal bound for e-matching. This work is published at POPL 2022 [17].

The relational e-matching approach hints at the fundamental connection between e-graphs and relational databases. However, some pitfalls exist. To have both efficient e-graph update (update) and efficient relational e-matching (query), one has to switch back and forth between the e-graph to its relational representation. Such cost can sometimes take a significant proportion of the run time, potentially offsetting the benefit of our approach [17]. A single self-contained relational representation is therefore desired to fully enjoy the benefits of relational e-matching. We call this design **relational e-graphs**. There are several challenges to it. First, we need to express operations over e-graphs in a relational manner. As performance is critical, they further need to be executed efficiently. From our experience, although e-graphs can be encoded in relational systems like Soufflé [10], they are far less efficient. Moreover, e-class analysis, an important extension of e-graphs that allows the user to express domain-specific logic, is widely supported in state-of-the-art e-graph frameworks. Such extensions are critical to the success of modern e-graph applications, and supporting them in the relational representation is a necessity.

Our design is based on the following observations. First, Datalog, a fixpoint-based relational language, is able to express non-equational relations (e.g., reachability) and can compute them using efficient algorithms, such as semi-naïve evaluations [6]. Modern Datalog engines [10] are also being extended to support efficient equational reasoning. Moreover, data dependencies in relational databases generalize congruence and rewrite rules in e-graphs. Data dependencies are extensively studied in the database community and can be efficiently reasoned with the chase procedure [2]. Finally, the lattice semantics of relational databases [8] precisely capture the monotonic nature of e-class analyses.

We argue that the relational e-graphs design should be based on Datalog with the lattice semantics and data dependencies. Besides the superior performance of relational e-matching, our ongoing research effort on relational e-graphs demonstrates that it has the following advantages:

- Committing to a single relational representation avoids the cost of converting between different representations.
- Efficient algorithms for Datalog like semi-naïve evaluations could benefit rule rewriting in e-graphs by incrementalizing e-matching.
- Rules in Datalog are naturally multi-patterns. This will allow first-class support for multi-patterns, whose performance will also benefit from relational e-matching.
- The lattice semantics of relations subsume e-class analyses. Datalog's non-equational reasoning further expands the expressive power of e-graphs.

The rest of the paper is organized as follows: Section 2 reviews background on e-graphs and relational databases. Section 3 presents relational e-matching, evaluates its performance, and discusses our ongoing work on relational e-graphs. Section 4 concludes.

2 Background & Related Work

2.1 E-Graphs

Terms. Let Σ be a set of function symbols and V be a set of variables. $T(\Sigma, V)$ is the set of terms inductively constructed using symbols from Σ and variables from V. A *ground term* is a term that contains no variables, a non-ground term is a *pattern*, and an f*-term* is one whose top-level function symbol is f.

Collaborators: Remy Wang, Philip Zucker, Max Willsey, and Zachary Tatlock.

E-graphs. A *congruence relation* $\cong$ is an equivalence relation over ground terms where $f(t_1,\dots,t_n) \cong f(t'_1,\dots,t'_n)$ whenever $\wedge_i\ t_i \cong t'_i$. An e-graph E efficiently represents sets of terms in a congruence relation. It consists of a set of e-classes. Each e-class consists of a set of e-nodes, and each e-node consists of an n-ary function symbol f and n children e-classes. An e-node associated with function symbol f is called an f-*application* e-node.

An e-node $f(c_1,\dots,c_n)$ is said to *represent* a term $f(t_1,\dots,t_n)$ if e-class c_i represents term t_i. An e-class c represents a term t if an e-node a in c represents term t. Terms represented by the same e-class are congruent. Figure 1a shows an e-graph representing the set of terms (where $[N]=\{1,2,\dots,N\}$):

$$[N] \cup \{g(i) \mid i \in [N]\} \cup \{f(i, g(j)) \mid i, j \in [N]\}.$$

In addition, all g-terms are equivalent, and all f-terms are equivalent. Note that the e-graph has size $O(N)$, yet it represents $\Omega(N^2)$ many terms. In general, an e-graph is capable of representing exponentially many terms in polynomial space.

One of the key challenges of relational e-graphs is efficient invariant maintenance [15]. Algorithms based on e-graphs like equality saturations [13] perform intensive rewrites to grow the e-graphs. These workloads require efficient maintenance of the congruent invariant under updates. In our approach, we translate e-graph invariants into data dependencies and use a specialized chase procedure to efficiently maintain e-graphs under updates.

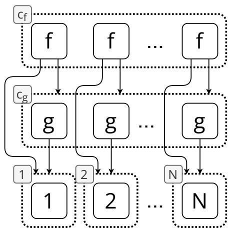
E-matching. E-matching performs pattern matching in an e-graph and yields a set of substitutions. A *substitution* σ is a function that maps variables to e-classes. Given a pattern p, we use $\sigma(p)$ to denote the set of terms obtained by replacing every occurrence of variable v_i in p with terms in $\sigma(v_i)$. Given an e-graph and a pattern p, e-matching finds the set of all possible pairs (σ, r) such that every term in $\sigma(p)$ is represented in the root e-class r. For example, matching the pattern $f(\alpha, g(\alpha))$ against the e-graph in Figure 1a produces the N substitutions, each with the same root c_f: $\{(\{\alpha \mapsto j\}, c_f) \mid j \in [N]\}$.

E-matching is a bottleneck in many e-graph based applications. Several algorithms have been proposed for e-matching [3, 4, 9]. However, most algorithms implement some form of backtracking search, which is inefficient in many cases. Figure 2 shows an abstract backtracking-based e-matching algorithm. It performs a top-down search following the shape of the pattern and prunes the result set of substitutions when necessary. Note that matching pattern $f(\alpha, g(\alpha))$ against the example e-graph will produce only N matches, yet the backtracking search runs in time $O(N^2)$. In contrast, relational e-matching runs in time $O(N)$.

E-class Analyses. Many domain-specific applications use domain knowledge when reasoning in e-graphs. Such domain knowledge is expressed as e-class analyses. An e-class analysis annotates each e-class with a (semi)lattice value. As the e-graph grows, knowledge is introduced, propagated, and joined (by taking the least-upper bound). To support e-class analyses in relational e-graphs is important since many domain-specific applications rely on it.

2.2 Relational Databases

Conjunctive Queries. A relational schema S_D over domain D is a set of relation symbols with associated arities. A relation R under a schema S_D is a set of tuples $\boldsymbol{t} \in R$. A database instance (or simply database) I of S_D is a set of relations under S_D.



(a)

arg_1	arg_2	id
1	c_g	c_f
2	c_g	c_f
⋮	⋮	⋮
N	c_g	c_f

(b)

arg_1	id
1	c_g
2	c_g
⋮	⋮
N	c_g

(c)

Figure 1: (a) An e-graph over $T(\Sigma, \emptyset)$, where $\Sigma = \{f, g, 1, \dots, N\}$. Each solid box denotes an e-node and each dashed box denotes an e-class. (b) Relation of f. (c). Relation of g.

$$\begin{aligned}\mathsf{match}(x, c, S) =& \{\sigma \cup \{x \mapsto c\} \mid \sigma \in S, x \notin \mathsf{dom}(\sigma)\} \cup \\ & \{\sigma \mid \sigma \in S, \sigma(x) = c\} \\ \mathsf{match}(f(p_1,\dots,p_k), c, S) =& \bigcup_{f(c_1,\dots,c_k)\in c} \mathsf{match}(p_k, c_k, \dots, \mathsf{match}(p_1, c_1, S))\end{aligned}$$

Figure 2: A declarative backtracking-based e-matching algorithm (reproduced from [3]). The set of substitutions for pattern p on e-graph G with e-classes C can be obtained by computing $\bigcup_{c\in C} \mathsf{match}(p, c, \{\emptyset\})$.

$$\begin{aligned} tc(x, y) &\leftarrow edge(x, y) \\ tc(x, y) &\leftarrow edge(x, z), tc(z, y)\end{aligned}$$

Figure 3: A Datalog program that computes the transitive closure of *edge*.

A conjunctive query Q over the schema S_D has the form:

$$Q(\boldsymbol{u}) \leftarrow R_1(\boldsymbol{x_1}), \dots, R_n(\boldsymbol{x_n}),$$

where $\boldsymbol{u}$ and $\boldsymbol{x_i}$ are lists of variables, and R_i are relation symbols in S_D. We call $Q(\boldsymbol{u})$ the *head* of the query, the remainder is the *body*. Each $R_i(\boldsymbol{x_i})$ is called an *atom*. Variables that appear in the head are called *free variables*, and they must appear in the body. Variables that appear in the body but not the head are called *bound variables*.

Evaluating a conjunctive query yields substitutions that map free variables in Q to elements in the domain such that there exists a mapping of the bound variables that causes every substituted atom to be present in the database.

Extensive research has been done on conjunctive queries. The AGM bound [1] and worst-case optimal joins [11] are recent advances in database theory. The AGM bound characterizes the worst-case output size of a conjunctive query, and worst-case optimal join is an algorithm that achieves this bound. As we will show, they allow us to derive new bounds for e-matching and guarantee worst-case optimality of relational e-matching.

We observe that conjunctive query and e-matching are structurally similar: both search for substitutions whose instantiations are present in a database. Therefore, it is tempting to reduce e-matching to a conjunctive query over the relational database, thereby benefiting from well-studied techniques from the database community, including join algorithms and query optimization.

Datalog and Semi-Naïve Evaluations. A Datalog program is a set of conjunctive queries whose head refers to relations in S_D. Figure 3 shows an example Datalog program that computes the transitive closure of the relation *edge*. It is inductively defined as follows: (1) if there is an edge from x to y, then y is in x's transitive closure, and (2) if there is an edge from x to z, and y is in z's transitive closure, then y is also in x's transitive closure.

The semantics of Datalog is well studied, and many semantic extensions are proposed [7, 8]. The database community has also extensively studied efficient evaluation and optimization algorithms for Datalog programs. Particularly, semi-naïve evaluation [6] uses monotonicity to incrementalize the iterative evaluation of Datalog programs so that no work is duplicated across each iteration.

Data Dependencies and the Chase. Data dependencies describe dependencies between columns in a relational database. *Tuple-generating dependencies* (TGDs) generalize Datalog rules with multi-heads and existential quantifiers. For example,

$$\exists b.supervise(a, b), employee(b) \leftarrow employee(a)$$

asserts that every employee a will be supervised by b, who is also an employee. *Functional dependency* (FD) is another kind of dependencies that identifies a relation's "determinant columns". For example,

$$b_1 = b_2 \leftarrow supervise(a, b_1), supervise(a, b_2)$$

indicates that if a is supervised by b_1 and b_2, then b_1 and b_2 are the same person. In other words, *a functionally determines* their (unique) supervisor.

In relational e-graphs, data dependencies generalize congruence and rewrite rules. Data dependencies can be efficiently reasoned about using *the chase*. Variants of the chase exist, leading to slightly different semantics. Our evaluation algorithm for relational e-graphs can be viewed as a chase procedure specialized to the semantics of e-graphs.

Lattice Semantics of Datalog. Flix extends Datalog by allowing the last column of a relation to be lattices, which can grow over time with rule derivations [8]. Lattices in Flix capture the monotonicity of many program analysis algorithms that are either inefficient or difficult to express in classical Datalog.

3 Approach and Uniqueness

In this section, we first introduce relational e-matching [17] and evaluate its performance. Next, we present our early designs on relational e-graphs.

3.1 Relational E-matching

E-matching via top-down backtracking search is inefficient because it visits obviously unsatisfying terms. For example, on pattern $f(\alpha, g(\alpha))$, whenever there are many terms that have the shape $f(\alpha, g(\beta))$ where α is not necessarily equivalent to β, backtracking will waste time visiting terms that do not yield a match.

We propose to view e-matching as conjunctive query answering: we represent every n-ary function symbol f as a relation R_f with $n+1$ columns. Every f-application e-node is now a tuple in R_f. The first n columns denote the e-class labels of children of n_f. The last column denotes the e-class label of n_f. Figures 1b and 1c show the relational representation of the e-graph in Figure 1a.

Under this relational view, an e-matching problem comes out as a conjunctive query naturally. For example, the pattern $f(\alpha, g(\alpha))$ corresponds to conjunctive query: $Q(root, \alpha) \leftarrow R_f(\alpha, x, root), R_g(\alpha, x)$.

$$\begin{aligned}
\textsc{Compile}(p) &= Q(v_1, \ldots, v_k, root) \leftarrow atoms \\
&\qquad \text{where } v_1 \ldots v_k \text{ are variables in } p \\
&\qquad \text{and } \textsc{Aux}(p) = root \sim atoms \\
\textsc{Aux}(f(p_1, \ldots, p_k)) &= v \sim R_f(v_1, \ldots, v_k, v), A_1, \ldots, A_k \\
&\qquad \text{where } v \text{ is fresh and } \textsc{Aux}(p_i) = v_i \sim A_i \\
\textsc{Aux}(x) &= x \sim \emptyset \qquad \text{where } x \text{ is a pattern variable}
\end{aligned}$$

Figure 4: Compiling a pattern to a conjunctive query.

Evaluating Q with a simple hash join strategy exemplifies the benefits of the relational approach. The hash join will only consider terms that fully match the pattern and so is asymptotically better: it builds indices on (α, x) for R_f and performs a linear scan through R_g for matches. Figure 4 shows the compilation algorithm that unnests an e-matching pattern to a conjunctive query.

This observation leads us to a very simple algorithm for relational e-matching. Relational e-matching takes an e-graph E and a set of patterns ps. It first transforms the e-graph to a relational database I. Then, it reduces every pattern p to a conjunctive query q. Finally, it evaluates the conjunctive queries over I.

Answering CQs with Generic Join. We propose to use generic join, an efficient worst-case optimal join algorithm, to solve the generated conjunctive queries. Traditional query plans, which are well-adopted and based on two-way joins such as hash joins and merge-sort joins, may suffer on certain queries compiled from patterns. For example, the pattern $f(g(\alpha), h(\alpha))$ compiles to a cyclic conjunctive query: $Q(\alpha, root) \leftarrow R_f(x, y, root), R_g(\alpha, x), R_h(\alpha, y)$. For any such query, [11] shows there exist databases on which *any* two-way join plan is suboptimal. In contrast, generic join is guaranteed to run in time linear to the worst-case output size. Moreover, generic join can have comparable performance on acyclic queries with two-way join plans. These properties make generic join our ideal solver for conjunctive queries generated from e-matching patterns.

Complexity of Relational E-matching. Relational e-matching preserves the worst-case optimality generic join guarantees:

Theorem 1. *Relational e-matching is worst-case optimal; that is, fix a pattern p, let $M(p, E)$ be the set of substitutions yielded by e-matching on an e-graph E with N e-nodes, relational e-matching runs in time $O(\max_E(|M(p, E)|))$.*

The structure of e-matching patterns lets us derive an additional bound dependent on the *actual* output size rather than the worst-case output size.

Theorem 2. *Fix an e-graph E with N e-nodes that compiles to a database I, and fix a pattern p that compiles to conjunctive query $Q(\overline{X}) \leftarrow R_1(\overline{X_1}), \ldots, R_m(\overline{X_m})$. Relational e-matching p on E runs in time $O\left(\sqrt{|Q(I)| \times \Pi_i |R_i|}\right) \leq O\left(\sqrt{|Q(I)| \times N^m}\right)$.*

Supporting Multi-patterns. Multi-patterns are a widely used extension to e-matching [3, 16]. A multi-pattern is a list of patterns of the form $(p_1, \ldots, p_k)$ that are to be simultaneously matched. For example, e-matching the multi-pattern $(f(\alpha, \beta), f(\alpha, \gamma))$ searches for pairs of two f-applications whose first arguments are equivalent. Efficient support for multi-patterns on top of backtracking search requires complicated additions to state-of-the-art e-matching algorithms [3], and they are suboptimal in many cases. Relational e-matching supports multi-patterns "for free": a multi-pattern is compiled to a single conjunctive query just like a single pattern. For example, multi-pattern $(f(\alpha, \beta), f(\alpha, \gamma))$ naturally compiles to $Q(\alpha, \beta, \gamma, root_1, root_2) \leftarrow R_f(\alpha, \beta, root_1), R_g(\alpha, \gamma, root_2)$. This shows the wide applicability of the relational e-matching approach.

3.2 Evaluation

We implemented relational e-matching inside the egg equality saturation toolkit [15]. Our implementation consists about 80 lines of Rust inside egg itself to convert patterns into conjunctive queries and a separate, e-graph-agnostic Rust library to implement generic join in fewer than 500 lines. egg's existing e-matching infrastructure is also about 500 lines of Rust, and it is interconnected to various other parts of egg. Qualitatively, we claim that the relational approach is simpler to implement, especially since the conjunctive query solver is completely modular.

In this section, we refer to egg's existing e-matching implementation as "EM" and our relational approach as "GJ."

Setup. We use patterns from `math` and `lambda`, egg's two largest benchmark suites. To construct the e-graphs in the benchmarks we ran equality saturation on a set of terms selected from egg's test suite, stopping before the e-graph reached 1e5, 1e6, 2e6, and 3e6 e-nodes. The result is four increasingly large e-graphs for each benchmark suite filled with terms generated by the suite's rewrite rules. For each benchmark suite and each of the four e-graph sizes, we then ran e-matching on the e-graph using both EM and GJ. We ran each approach 10 times and took the minimum run time.

For GJ, we ran each trial twice. The first time builds the indices necessary for generic join just-in-time, and the run time includes that. On the second trial, GJ uses the pre-built index tries from the first run, so the time to build them is excluded.

Results. Figure 5 shows the results of our benchmarking experiments. Each group of bars shows the results of e-matching a single pattern on 4 increasingly large e-graphs (top to bottom), comparing egg's built-in e-matching (EM) with relational e-matching using generic join (GJ). The orange bar shows the multiplicative speedup of our approach: EM/GJ. The blue bar shows the same, but *excluding* the time spent building the indices needed for generic join: EM/(GJ − idx). The text above each group of bars shows the pattern itself and the number of substitutions found on the largest e-graph; the patterns are sorted by this quantity.

Relational e-matching can be up to 6 orders of magnitude faster than traditional e-matching on complex patterns. Speedup tends to be greater when the output size is smaller, and when the pattern is larger and non-linear. A large output indicates the e-graph is densely populated with terms matching the given pattern, therefore backtracking search wastes little time on unmatched terms, and using relational e-matching contributes little or no speedup. Large and complex patterns require careful query planning to be processed efficiently. For example, the pattern experiencing the largest speedup in Figure 5 is 4 e-nodes deep with 4 occurrences of the variable f. Relational e-matching using generic join can find a query plan that puts smaller relations with fewer children on the outer loop, thereby pruning down a large search space early. In contrast, backtracking search must traverse the e-graph top-down.

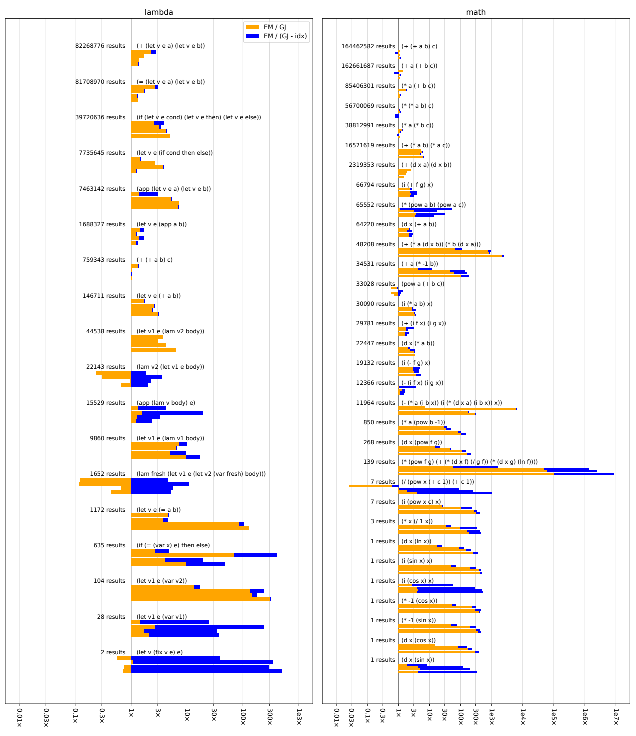


Figure 5

In some cases index building time takes a significant proportion of the run time in relational e-matching, sometimes offsetting the gains. Overall, relational e-matching remains competitive with the index building overhead.

In summary, relational e-matching is almost always faster than backtracking search, and is especially effective in speeding up complex patterns.

3.3 Relational E-graphs

In this subsection, we will discuss some of our ongoing effort on relational e-graphs. We built a prototype named egg💡 (pronounced as egg lite) and are designing a full-fledged language for relational e-graphs named egg$^\sharp$ (pronounced as egg sharp).

egg💡 is an e-graph prototype on top of SQLite, an embedded relational database system. egg💡 has full-featured support for functionalities like match-apply iterations and multi-patterns, allowing users to write real-world applications like equality saturation. To our knowledge, this is the first full-fledged e-graph framework fully on top of a relational database.

egg💡 can be viewed both as a multi-pattern language for e-graphs and as a Datalog language with an internalized congruence. egg💡 has a Datalog-like surface language, and it translates e-graph operations into SQL statements, which are executed in SQLite. As is in relational e-matching, we use relation $\texttt{Add}(x_1, x_2, c)$ to represent a term $\texttt{Add}(x_1, x_2)$ with e-class id c. The associativity rule for Add can be specified as

$$(\texttt{Add}\ \alpha_1\ (\texttt{Add}\ \alpha_2\ \alpha_3))@\alpha_4 \Rightarrow (\texttt{Add}\ (\texttt{Add}\ \alpha_1\ \alpha_2)\ \alpha_3)@\alpha_4.$$

The semantics of rules is as follows: for each pattern on the left-hand side, substitute and populate patterns on the right-hand side, and merge e-classes annotated with the same id (e.g., α_4).

egg	**egg$^\sharp$**
Equational rewrites	Tuple-generating dependencies
Congruence rules	Functional dependencies (FD)
E-classes	User-defined sorts
E-class merges	FD repair through unification
E-class analyses	User-defined lattices
E-class analysis maintenance	FD repair through lattice joins

Figure 6: The correspondence between egg and in egg$^\sharp$.

egg$^{💡}$ is performant. Matching the left-hand side is essentially relational e-matching. Moreover, it uses a novel batched rewriting algorithm and an efficient algorithm for maintaining congruence. Both algorithms can be viewed as chase procedures specialized for reasoning about data dependencies in e-graphs. and lay a solid foundation for efficient computations in relational e-graphs. egg$^{💡}$ also demonstrates how far we can push the limits of an in-memory database system as an e-graph engine. The preliminary benchmark shows that, even with the overheads for interpretation, parsing, and communication, egg$^{💡}$ is within one order of magnitude slower than egg, which is highly customized for the e-graph workload.

egg$^{💡}$ also exhibits some interesting designs. For example, unlike a traditional e-graph implementation, egg$^{💡}$ does not have a global union-find data structure. Instead, a local union-find is created transiently during each rebuilding. This challenges the traditional view that an e-graph is a set of terms and an equivalence relation over the terms [5]. Moreover, although our current egg$^{💡}$ implementation only supports congruence rules, we realized that congruence is just a special kind of FD over the database. This again challenges the long-held belief that congruence rules are indispensable to e-graphs. We have discovered several interesting non-congruent FDs so far, which are useful for different scenarios.

Based on the experience with egg$^{💡}$, we started a design for relational e-graphs called egg$^\sharp$. In egg$^\sharp$, rewrite rules are expressed as TGDs, and relations are annotated with FDs. Values in egg$^\sharp$ are divided into sorts and lattices. Sort values can be "unified" so that they denote the same element in the underlying model. Relations in egg$^\sharp$ generalize both e-nodes and e-class analyses. For example, let E be a sort for expressions and $L_{\max}$ be the max lattice. $R_+(E, E) \to E$ is a ternary relation where the first two columns determine the last column. It represents the binary + constructor of an e-graph. $lo(E) \to L_{\max}$ is a binary relation where the expression determines its lattice value. This could denote an e-class analysis that computers the lower bound of an expression e-class.

When two sort values are unified (e.g., during e-class merges), they are no longer distinguishable. Such unification could potentially break the integrity of FDs, i.e., multiple distinct tuples with the same determinant columns. egg$^\sharp$ remedies these violations. For each sort column in dependent columns, egg$^\sharp$ unifies the sort values in that column, making them indistinguishable and therefore unique again. For each lattice column in the dependent columns, the new, unique value for each column is computed as the lattice join of values in that column. As a result, FD repair in egg$^\sharp$ unifies the semantics of e-classes and e-class analyses. We have also developed a model semantics for egg$^\sharp$ based on data dependencies.

egg$^\sharp$ greatly expands the expressivity of egg: It has natural support of multi-patterns. As an extension to Datalog, it supports reasoning expressible in Datalog, including non-equational ones. Finally, lattice values in egg$^\sharp$ generalize e-class analyses. As an example, egg$^\sharp$ allows interdependent analyses, while traditional e-class analyses are not composable in general. With the new expressive power, egg$^\sharp$ is able to express the classical type inference algorithm for Hindley-Milner type systems.

egg$^\sharp$ also benefits from the efficient semi-naïve evaluations in Datalog. Semi-naïve evaluation incrementalizes the matching in egg$^\sharp$. The literature has also considered algorithms for incremental e-matching [3]. However, these optimizations are often complex to implement and are inefficient when the e-graph grows rapidly, a typical scenario in modern e-graph applications. In contrast, the semi-naïve evaluation algorithm is straightforward to implement and works well with batched updates.

4 Conclusion

This paper reviews our work on relational e-matching and discusses our recent progress on relational e-graphs. The research presented summarizes our work in PLDI 2021 SRC, a POPL 2022 paper, a forthcoming EGRAPH workshop talk, and a research blog post. I am the first or sole author of the above work.

References

[1] Atserias, A., Grohe, M., and Marx, D. Size bounds and query plans for relational joins. In *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science* (USA, 2008), FOCS '08, IEEE Computer Society, p. 739–748.

[2] Benedikt, M., Konstantinidis, G., Mecca, G., Motik, B., Papotti, P., Santoro, D., and Tsamoura, E. Benchmarking the chase. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (New York, NY, USA, 2017), PODS '17, Association for Computing Machinery, p. 37–52.

[3] de Moura, L., and Bjørner, N. Efficient e-matching for smt solvers. In *Automated Deduction – CADE-21* (Berlin, Heidelberg, 2007), F. Pfenning, Ed., Springer Berlin Heidelberg, pp. 183–198.

[4] Detlefs, D., Nelson, G., and Saxe, J. B. Simplify: A theorem prover for program checking. *J. ACM 52*, 3 (May 2005), 365–473.

[5] Downey, P. J., Sethi, R., and Tarjan, R. E. Variations on the common subexpression problem. *J. ACM 27*, 4 (oct 1980), 758–771.

[6] Green, T. J., Huang, S. S., Loo, B. T., and Zhou, W. Datalog and recursive query processing. *Found. Trends Databases 5*, 2 (Nov. 2013), 105–195.

[7] Köstler, G., Kiessling, W., Thöne, H., and Güntzer, U. Fixpoint iteration with subsumption in deductive databases. *J. Intell. Inf. Syst. 4*, 2 (mar 1995), 123–148.

[8] Madsen, M., Yee, M.-H., and Lhoták, O. From datalog to flix: A declarative language for fixed points on lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2016), PLDI '16, Association for Computing Machinery, p. 194–208.

[9] Moskal, M., Łopuszański, J., and Kiniry, J. R. E-matching for fun and profit. *Electron. Notes Theor. Comput. Sci. 198*, 2 (May 2008), 19–35.

[10] Nappa, P., Zhao, D., Subotić, P., and Scholz, B. Fast parallel equivalence relations in a datalog compiler. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2019), IEEE, pp. 82–96.

[11] Ngo, H. Q., Porat, E., Ré, C., and Rudra, A. Worst-case optimal join algorithms. *J. ACM 65*, 3 (Mar. 2018).

[12] Panchekha, P., Sanchez-Stern, A., Wilcox, J. R., and Tatlock, Z. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2015), PLDI '15, Association for Computing Machinery, p. 1–11.

[13] Tate, R., Stepp, M., Tatlock, Z., and Lerner, S. Equality saturation: A new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2009), POPL '09, Association for Computing Machinery, p. 264–276.

[14] VanHattum, A., Nigam, R., Lee, V. T., Bornholt, J., and Sampson, A. *Vectorization for Digital Signal Processors via Equality Saturation.* Association for Computing Machinery, New York, NY, USA, 2021, p. 874–886.

[15] Willsey, M., Nandi, C., Wang, Y. R., Flatt, O., Tatlock, Z., and Panchekha, P. Egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang. 5*, POPL (Jan. 2021).

[16] Yang, Y., Phothilimthana, P., Wang, Y., Willsey, M., Roy, S., and Pienaar, J. Equality saturation for tensor graph superoptimization. In *Proceedings of Machine Learning and Systems* (2021), A. Smola, A. Dimakis, and I. Stoica, Eds., vol. 3, pp. 255–268.

[17] Zhang, Y., Wang, Y. R., Willsey, M., and Tatlock, Z. Relational e-matching. *Proc. ACM Program. Lang. 6*, POPL (jan 2022).