Faster and Worst-Case Optimal E-Matching

Yihong Zhang

Supervised by Professor Zachary Tatlock

A senior thesis submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Science
with Departmental Honors

Computer Science & Engineering,
University of Washington

June, 2021

Presentation of work given on ______________________________

Thesis and presentation approved by ______________________________

Date ______________________________

Abstract

An e-graph is a data structure that efficiently represents sets of congruent terms (Nelson, 1980). It has a long history of applications in automated theorem proving (Nelson, 1980; de Moura and Bjørner, 2007; Selsam and Moura, 2016; Nieuwenhuis and Oliveras, 2005; Detlefs et al., 2005) and is re-purposed in the last decade for program optimization, known as equality saturation (Tate et al., 2009; Joshi et al., 2006; Wang et al., 2020; Premtoon et al., 2020; Panchekha et al., 2015; Willsey et al., 2021). A fundamental query operation on e-graph is e-matching (de Moura and Bjørner, 2007; Detlefs et al., 2005), which finds the set of terms in the e-graph matching a given pattern. E-matching is the bottleneck of many equality saturation–based program optimizers (Wang et al., 2020; Panchekha et al., 2015; Willsey et al., 2021) and is a central procedure in state-of-the-art SMT solvers, including `Z3` (de Moura and Bjørner, 2008) and `CVC4` (Barrett et al., 2011). Therefore, the performance of e-matching is critical. Several optimizations are proposed for e-matching (de Moura and Bjørner, 2007; Detlefs et al., 2005). However, to our knowledge, existing e-matching algorithms are based on naïve backtracking, which is suboptimal in many cases. In particular, they do not exploit the equality constraints implied by multi-occurrences of a variable, which is nonetheless common in practice.

To tackle this inefficiency, we propose to take a relational view of e-graphs. Under this view, e-matching corresponds naturally to a restricted class of relational queries, known as conjunctive query. By treating e-matching as an instance of conjunctive query, we benefit from decades of researches from the database community. In particular, by using the recently discovered generic join algorithm, our e-matching algorithm guarantees worst-case optimality and achieves more than 400× speed-up in our preliminary experiments.

This thesis is based on an extended abstract to appear at PLDI 2021 Student Research Competition and a forthcoming paper.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

Along my journey of academics and life, I have owed thanks to many people. They teach me, help me, encourage me, and care about me. I thank them for everything they did for me.

I would like to first thank my thesis advisor Professor Zachary Tatlock, who is always encouraging and gives me great help during my thesis work. I also would like to thank Professor Dan Suciu, who was kind enough for recognizing my worth and introduced me to my first research on genetic algorithms for counterfactual explanations. I thank Max Schleich for his kind guidance on the counterfactual explanation project and my academic journey. I also would like to thank James Wilcox. I worked both as a student of and a teaching assistant for Zach and James, and I deeply appreciate their dedication to students and enthusiasm to creating a wonderful learning environment for all.

I would like to thank my collaborators and mentors Remy Wang and Max Willsey on the work described in this thesis. It is always my pleasure to discussing ideas with them and learning from their insights, which I deeply appreciate. It is them who make this project possible.

I would like to thank my roommate Mike He, Jiuru Li, and Bruce Li for their care and tolerance throughout my collegiate life. We share interests on many things. In particular, Mike and I have the same enthusiasm in programming languages and I learned so many things from him.

I want to thank my parents, who brought me to the world and supports me, emotionally and financially, to study abroad at the States. I recognize it is such a privilege for me to study here at UW, and this is impossible without my parents.

I want to thank Jiawei Huang, my love, who is always here when I am down and celebrates for my achievements. It's my biggest fortune to meet her and fall in love with her.

Finally, I want to thank my maternal grandfather Peihua Zhao and grandmother Jinhua Yu. Words cannot express how deeply owed I am to them. They taught me to be a righteous human being and were always proud of what I did. My heart broke when they passed away in 2016 and in 2020, but they keep appearing in my dreams. This thesis is dedicated to them.

DEDICATION

To my maternal grandmother and grandfather,

I miss you.

外公外婆，我想念你们。

Chapter 1
INTRODUCTION

The e-graph data structure has been the focus of programming language and verification research for a long time. First invented in the 1980s, it is originally designed for automated theorem proving (ATP) and is extensively used in modern theorem provers (Nelson, 1980; de Moura and Bjørner, 2007; Nieuwenhuis and Oliveras, 2005; Detlefs et al., 2005; de Moura and Bjørner, 2008; Barrett et al., 2011). During the last decade, e-graphs are also re-purposed for program optimization, known as equality saturation (Tate et al., 2009; Joshi et al., 2006; Wang et al., 2020; Premtoon et al., 2020; Panchekha et al., 2015; Willsey et al., 2021). At its core, e-graphs generalize the union-find data structure (Tarjan, 1975), which efficiently represents equivalence relations, to congruence relations. A congruence relation $\cong$ is an equivalence relation where two terms are also congruent (i.e., $f(t_1, \ldots, t_k) \cong f(t'_1, \ldots, t'_k)$) if all children of the two terms are pairwise congruent (i.e., $t_i \cong t'_i$). E-graphs maintains the congruence invariant by maintaining a set of equivalent classes (known as *e-classes*) consisting of a set of nodes (known as *e-nodes*), while every e-node consists of a head function symbol and a list of children e-classes.

E-matching is a fundamental operation on e-graphs (de Moura and Bjørner, 2007; Detlefs et al., 2005). It finds the set of terms in an e-graph matching a given pattern. For example, for pattern $f(\alpha, g(\alpha))$[1], e-matching finds every term whose head symbol is f and whose second argument has a head symbol g and whose first argument to f and first argument to g are the same. E-matching is a central procedure in state-of-the-art SMT solvers, including Z3 (de Moura and Bjørner, 2008) and CVC4 (Barrett et al., 2011). SMT solvers

[1] α is a variable in this pattern. We use a, b, c for constants and α, β, γ for variables as a convention.

use e-matching to instantiate quantified formulas over ground terms. Moreover, e-matching is also the bottleneck of many equality saturation–based program optimizers (Wang et al., 2020; Panchekha et al., 2015; Willsey et al., 2021). In equality saturation, the optimizer uses e-matching to match the left-hand side of the rewrite rules and fires new terms based on the matches, and e-matching usually takes 60–90% of the overall run time. Therefore, in both applications, the performance of e-matching is critical.

Several algorithms are proposed for e-matching (de Moura and Bjørner, 2007; Detlefs et al., 2005). However, to our knowledge, existing e-matching algorithms are based on naïve backtracking, which is suboptimal in many cases. In particular, their query planning does not exploit the constraints implied by multi-occurrences of the same variable, which is nonetheless a common pattern in practice. To see this, consider the pattern $f(\alpha, g(\alpha))$ again. This pattern generates three constraints for a potential matching e-node n:

$$n.\mathit{symbol} = f \tag{1.1}$$

$$n.\mathit{child}_2.\mathit{symbol} = g \tag{1.2}$$

$$n.\mathit{child}_1 = n.\mathit{child}_2.\mathit{child}_1 \tag{1.3}$$

There are two kinds of constraints here. The first kind of constraint is derived from the structure of the pattern. In this example, the structure of $f(\alpha, g(\alpha))$ constrains the head symbol of all (sub)terms to be f and g respectively (i.e., constraint 1.1 and 1.2), which we call *structural constraints*. The second kind of constraints is implied by the multi-occurrence of the same variable. Here, the occurrence of α implies that the terms at these positions should be equivalent with each other for all matches (i.e., constraint 1.1), which we call *equality constraints*.

To our knowledge, existing backtracking-based algorithms only exploit the structural constraint during query planning and fail to consider equality constraints in general. Therefore, running backtracking-based algorithms on the above pattern, terms violating constraint 1.3 are only pruned away *a posteriori* when the partial match containing violating variables are instantiated (See Chapter 2 for more details). Therefore, excessive terms like $f(a, g(b))$ for

$a \not\cong b$ will be enumerated during search and only pruned away before yielding.

We solve this issue by taking a relational view of e-graphs and e-matching, which we call *relational e-matching*. Under this view, e-graphs correspond to a relational database and e-matching patterns correspond to a restricted class of relational queries, known as *conjunctive queries*. One key advantage of this relational view is that the conjunctive query representation provides a unified way to explicitly encode *both* structural constraints and equality constraints, of which the conjunctive query solver could take advantage during query planning.

One particular kind of conjunctive queries practical e-matching patterns may correspond to is cyclic queries, where query plans based on traditional two-way joins, including hash joins and merge-sort joins, are known to be suboptimal (Atserias et al., 2008). On cyclic queries, two-way joins cannot leverage all the constraints to prune the search space at once, so candidates that does not satisfy all the constraints will be enumerated. Therefore, we take inspirations from the theoretical advances in database theory community and utilize generic join algorithm as the conjunctive query solver. Generic join algorithm is a worst-case optimal algorithm on conjunctive queries, and it avoids to enumerate candidates that do not satisfy all the constraints. Moreover, it is particularly efficient with complex queries (e.g., cyclic queries). Using generic joins, our e-matching algorithm preserves the guarantees of worst-case optimality. Finally, we instantiate our algorithm in EGG (Willsey et al., 2021), a program optimization framework based on equality saturation. We evaluate our implementation using some microbenchmarks, which shows asymptotic speedup and order of magnitude performance gain.

In summary, we made the following contributions:

- We propose relational e-matching, a worst-case optimal and efficient approach to e-matching by running generic joins on e-matching-translated conjunctive queries over the relational representations of e-graphs.

- We instantiate relational e-matching in an efficient implementation integrates to EGG and evaluate it on several benchmarks.

Chapter 2
BACKGROUNDS

In this chapter, we briefly review related backgrounds.

2.1 E-graphs and e-matching

Terms. Let Σ be a set of function symbols with associated arities. A function symbol is called a *constant* if it has a zero arity. Let V be the set of variables. We define $T(\Sigma, V)$ to be the set of terms constructed using function symbols from Σ and variables from V. More formally, $T(\Sigma, V)$ is the smallest set that (1) all variables and constants are in $T(\Sigma, V)$ and (2) $t_1, \ldots, t_k \in T(\Sigma, V)$ implies $f(t_1, \ldots, t_k) \in T(\Sigma, V)$, where $f \in \Sigma$ has arity k. A *ground term* is a term in $T(\Sigma, V)$ that contains no variables. A non-ground term is also called a *pattern*. We call a term of the form $f(t_1, \ldots, t_k)$ an f-application term.

Congruence relation. An *equivalence relation* $\equiv_\Sigma$ is a binary relation over $T(\Sigma, \emptyset)$ that is reflexive, symmetric, and transitive. A *congruence relation* $\cong_\Sigma$ is an equivalence relation satisfying that if $t_i \cong t'_i$ for all $i = 1, \ldots, n$ hold, $f(t_1, \ldots, t_n) \cong f(t'_1, \ldots, t'_n)$ holds as well, where f is a n-ary function symbol. We wrote $\cong$ when Σ is clear from the context.

E-graph. Intuitively, an e-graph E is a set of e-classes $\{c_1, \ldots, c_k\}$, where each e-class c is a set of e-nodes $\{n_1, \ldots, n_k\}$. Each e-node consists of a function symbol f and a list of children e-classes. Similar to patterns, we call an e-node of the form $f(c_1, \ldots c_n)$ an f-application e-node. More formally, we define an e-graph E to be an tuple $(\Sigma, N, C, \mathit{symbol}, \mathit{lookup}, \mathit{child})$ where

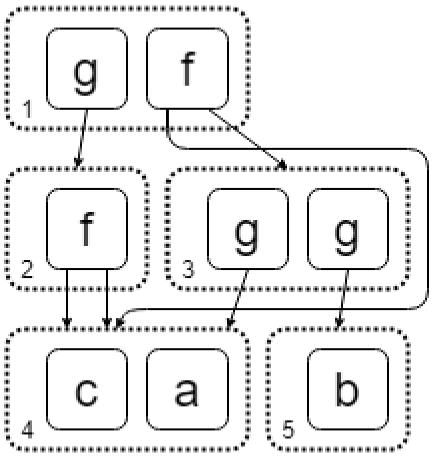
- Σ is a set of function symbols with associated arities,

- N is the set of e-nodes,
- C is the set of e-classes,
- *symbol* is a function that maps each e-node in N to a function symbol in Σ,
- *lookup* is a function that maps every e-node in N to the e-class in C that contains it, and
- *child* is a function that maps an e-node to a list of e-class that are the children of this e-node.

For convenience, we use $n.symbol$, $n.id$, and $n.child_i$ to denote $symbol(n)$, $lookup(n)$, and the ith element of $child(n)$.

An e-graph E efficiently represents sets of ground terms in a congruence relation. An e-graph is said to *represent* a ground term t if its e-classes represent it. An e-class c represents a ground term if c contains an e-node a that represents it. An e-node $f(c_1, \ldots, c_k)$ represents a ground term $f(t_1, \ldots, t_k)$ if they have the same function symbol f and each e-class c_i represents term t_i. Terms represented by an e-class are equivalent to each other. Consequently, an e-graph forms a congruence relation. Consider Figure 2.1a for example. $g(f(c,c))$, $f(c,g(b))$, and $f(a,g(a))$ are three ground terms represented by e-class c_1 in this e-graph. Therefore, they are congruent with each other by definition.

E-matching. E-matching is the task of finding e-matching substitutions that instantiate patterns to set of terms represented in the e-graph. An *e-matching substitution* σ is a function that maps every variable in a pattern to e-classes. For convenience, we use $\sigma(p)$ to denote the set of terms obtained by replacing every occurrence of variable v_i in p with terms in $\sigma(v_i)$. Formally, given an e-graph E 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 e-class r. Terms in $\sigma(p)$ are said to be matched by pattern p. r is said to be the root of matched terms. For instance, pattern



(a)

eclass-id	child$_1$	child$_2$
1	4	3
2	4	4

(b)

eclass-id	child$_1$
1	2
3	4
3	5

(c)

Figure 2.1: (a) an e-graph over $T(\Sigma, \emptyset)$ and $\cong_\Sigma$ where $\Sigma = \{f, g, a, b, c\}$ and a, b, c are nullary functions. Each solid box denotes an e-node and each dashed box denotes an e-class. Every term represented by an e-class is mutually equivalent. For example, $a \cong_\Sigma c$, $g(a) \cong_\Sigma g(b)$, and $f(a, g(a)) \cong_\Sigma g(f(a, a))$. The labels of e-classes are at bottom left. (b) relation representing f. (c) relation representing g.

$f(\alpha, g(\alpha))$ matches four terms in e-class c_1: $f(a, g(a))$, $f(a, g(c))$, $f(c, g(c))$, and $f(c, g(a))$; all of which are witnessed by the substitution $\{\alpha \mapsto c_4\}$.

Existing approaches to e-matching rely on backtracking (de Moura and Bjørner, 2007; Detlefs et al., 2005; Willsey et al., 2021). For example, de Moura and Bjørner (2007) proposed a backtracking-based e-matching algorithm that is used by Z3 (de Moura and Bjørner, 2008) and EGG (Willsey et al., 2021), two state-of-the-art e-graph implementations. To match the pattern $f(\alpha, g(\alpha))$ on the e-graph in Figure 2.1a, their algorithm does a depth-first search over the e-graph: it searches for all f-application e-nodes n_f, adds $\alpha \mapsto n_f.\mathit{child}_1$ to substitution σ, iterates through all g-application e-nodes n_g in e-class $n_f.\mathit{child}_2$, and only yield σ if $n_g.\mathit{child}_1 = \sigma(\alpha)$. In general, this procedure runs in time that is quadratic of the e-graph size. In a large e-graph, there may be thousands of pairs of n_f and n_g where n_g is in e-class $n_f.child_2$, but only a few satisfy the constraint $n_f.child_1 = n_g.child_1$. Therefore, backtracking-based e-matching enumerates an unnecessarily large pool of candidates. Even worse, complex query patterns may involve many variables that occur at several places, which makes naïve backtracking enumerates an excessive number of terms and therefore extremely slow. This inefficiency is due to the fact that naïve backtracking does not use the equality constraints to prune the search space *globally*. This is in contrast to our approach, which exploits the equality constraints during query planning for greater performance and guarantees worst-case optimality with respect to the output size.

2.2 Conjunctive queries

Relational schema. A relational schema S_D over domain D is a set of relation symbols with associated arities. An atom under a schema S_D is an expression $R(t_1, \ldots, t_k)$, where k is the arity of R in S_D and t_i is an element in D. An instance of S_D is a set of atoms over S_D.

Conjunctive queries. A conjunctive query over schema S_D and a set of variables V is a formula of the form

$$ans(x_1, \ldots x_k) \text{ :- } R_1(v_{1,1}, \ldots, v_{1,k_1}), \ldots, R_n(v_{n,1}, \ldots, v_{n,k_n}),$$

where $n \geq 0$, $R_1 \ldots R_n$ are relation names in S_D with arities $k_1, \ldots k_n$, ans is the name of the resulting relation not in S_D, $v_{i,j}$ are variables in V[1], and $x_1, \ldots x_k$ are variables occurring in $v_{i,j}$. We call such $R_i(v_{i,1}, \ldots, v_{i,k_1})$ an atom of the conjunctive query.

Semantics of conjunctive queries. Similar to e-matching substitutions, we define a conjunctive query substitution as a function that maps every variable occurring in $v_{i,j}$ to elements in D. The semantics of conjunctive queries is defined as follows: Let q be a conjunctive

[1]Some definitions of conjunctive queries allow both variables and constants, but we only allow variables in conjunctive queries for simplicity.

query, I be an instance of S_D, conjunctive queries yield the set of all $ans(\sigma(x_1), \ldots, \sigma(x_k))$ where σ is a substitution satisfying that $R_i(\sigma(t_1), \ldots \sigma(t_k))$ are atoms in I for $i = 1, \ldots, n$. We denote the result set as $q(I)$.

We make the observation that the definition of conjunctive query and e-matching are definitionally similar to each other: both are defined as finding substitutions whose instantiations are present in a (relational or graph) database. In fact, the e-matching problem can be viewed as a "nested" conjunctive queries on a database where the atoms are nested. Therefore, it is tempting to reduce the e-matching problem to a conjunctive query over the relational database, thereby benefiting from well-studied techniques from the database community, including join algorithms, query optimization, and query evaluations.

2.3 The AGM bound and Generic Join [2]

The AGM Bound Given a database and a conjunctive query, the AGM bound Atserias et al. (2008) is a bound on how large the query output could be. A simple bound just multiplies the cardinally of each relation, which is the size of the Cartesian product of all relations. For example, for triangle query $Q(x, y, z)$:- $R(x, y), S(y, z), T(x, z)$, this bound computes to $|Q| \leq |R| \times |S| \times |T|$. If $|R| = |S| = |T| = N$, then $|Q| \leq N^3$. However, we can do better: we observe that Q contains fewer atoms than the query $Q'(x, y, z)$:- $R(x, y), S(y, z)$, as Q is further filtered by joining with T. Therefore, we know $|Q| \leq |R| \times |S| = N^2$. In fact, the theoretical bound of conjunctive query output size is given as the AGM bound (Atserias et al., 2008), which is $N^{3/2}$ in this case.

The AGM bound proposes a question: is there an algorithm that always achieves the performance described by the AGM bound? In particular, *cyclic queries* is a kind of conjunctive queries whose query hypergraph contains cycles, such as the triangle query Q above, and they are known to make traditional join plans (e.g., join plans using two-way joins like hash joins or merge joins) suboptimal. Most of real-world database systems therefore have

[2]This section is inspired by Remy Wang's introduction to generic join algorithms.

```
procedure SolveTriangleQuery($(R, S, T)$)
    $A := R(x, y).x \cap T(z, x).x$
    ▷ Compute $Q(\alpha, y, z)$ :- $R(\alpha, y), S(y, z), T(z, \alpha)$
    for $\alpha \in A$ do
        $B := R(\alpha, y).y \cap S(y, z).y$
        ▷ Compute $Q(\alpha, \beta, z)$ :- $R(\alpha, \beta), S(\beta, z), T(z, \alpha)$
        for $\beta \in B$ do
            $C := S(\beta, z).z \cap T(z, \alpha).z$
            ▷ Yield join results $Q(\alpha, \beta, \gamma)$
            for $\gamma \in C$ do
                output $Q(\alpha, \beta, \gamma)$
            end for
        end for
    end for
end procedure
```

Figure 2.2: Generic join algorithm for the triangle query $Q(x, y, z)$:- $R(x, y), S(y, z), T(z, x)$, with ordering $[x, y, z]$.

bad performance on cyclic queries.

Generic join. Fortunately, theoretical progress has been made in the database theory community to achieve optimal performance on conjunctive queries and on cyclic queries in particualr. In particular, the generic join algorithm for conjunctive query is developed that runs in time linear to the worst-case output size with a log factor (Ngo et al., 2018). The generic join algorithm has great performance in practice when the query is cyclic, and is competitive to traditional join algorithms in other cases. Given a conjunctive query and a relational database, generic join is parameterized over a ordering of the set of variables in the query, say $[x, y, z]$ for the query above. The ordering does not matter for the optimality guarantee, but may impact practical performance greatly. Given an ordering, we assume the input relations are stored in tries sorted by the ordering. That is, given the ordering $[x, y, z]$, $R(x, y)$ is sorted by x and then y, and the first-level trie nodes are the x's. We can very efficiently intersect (join) relations on their first (according to the variable ordering) variable given such tries. Algorithm 2.2 shows the generic join algorithm for computing Q. Note that selection, e.g. $R(a, y)$ is very fast with the built tries. $A \cap B$ can also be done in $\tilde{O}(\min(|A|, |B|))$ time ($\tilde{O}$ means O with a log factor). For general queries we may have to intersect more than two relations, in which case the intersection should be performed in $\tilde{O}(\min_i |A_i|)$ time (using the merge in merge-sort).

Chapter 3
RELATIONAL E-MATCHING

In this chapter, we introduce our relational e-matching algorithm, shown in Figure 3.1. The algorithm takes an e-graph E and a set of patterns ps. It first transforms the e-graph to its relational representation, denoted as I. Next, for each pattern, it compiles the pattern to a conjunctive query q, and executes this conjunctive query over I. This is particularly suited for scenarios like equality saturation. In such scenarios, the program optimizer switches between e-matching and batched rewrite applications in each iteration to amortize the run-time required for maintaining the congruence invariant, a technique known as "rebuilding" (Willsey et al., 2021). In such scenarios, the run time for materializing the e-graph can be neglected compared to the e-matching workload. In other scenarios, where the e-graph is frequently updated and e-matching is performed interactively, it is possible to use a totally relational representation to avoid the cost of frequent materialization of the relational database and use standard update operations from a SQL-like language[1]. However, we suspect this would incur a larger constant factor to the run time compared to the graph representation

[1]We actually implemented a prototype e-graph purely on top of an in-memory relational database that does not use graph data structure nor union-finds.

```
1: procedure RelationalEMatching($E, ps$)
2:     $I \leftarrow$ EgraphToDatabase($E$)
3:     return $\{\text{EvalCQ}(q, I) \mid$ for $p \in ps, q \leftarrow \text{PatternToCQs}(p)\}$
4: end procedure
```

Figure 3.1: The relational e-matching algorithm

$$\begin{aligned}\textsc{EgraphToDatabase}(E) &= \{\textsc{EnodeToAtom}(n) \mid \text{for e-node } n \in E\} \\ \textsc{EnodeToAtom}(f(c_1, \ldots, c_n)) &= R_f(\mathit{lookup}(f(c_1, \ldots, c_n)), c_1, \ldots, c_n)\end{aligned}$$

Figure 3.2: Transforming e-graphs to relational database

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

Figure 3.3: Compiling a pattern to a conjunctive query.

of e-graph. In general, we left the problem of interactive e-matching, where e-matching is performed on a frequently updated e-graph, as future work (See Chapter 8).

3.1 Reducing e-graphs to relations

The first step of relational e-matching is to transform the e-graph to a relational database. An e-graph E representing a congruence relation over $T(\Sigma, \emptyset)$ will be mapped to a relational database over Id, the set of e-class ids. For every n-ary symbol f in Σ, We map it to an $n+1$-ary symbol R_f in schema S_{Id}. We refer to fields of R_f as *eclass-id*, $child_1, \ldots, child_n$ respectively, and they denote the e-class this e-node belongs to and the children e-classes of this e-node respectively. For every e-node in the e-graph, we map it to an atom in the relational database. The algorithm is presented in Figure 3.2.

Take the e-graph in Figure 2.1a for example. There are 8 e-classes in total. The g-application e-node in e-class c_1 translates to atom $R_g(1, 2)$, where 1 is the id of the e-class that contains it and 2 is the id of its child e-class. Similarly, the f-application e-node in e-class c_2 translates to the atom $R_f(2, 4, 4)$, where 2 is the id of the e-class that contains it and $4, 4$ are the ids of its child e-classes. Figure 3.2 only shows the relation representing f and g. The relations representing constants a, b, c, which are all relations with ≤ 1 atom, are omitted for brevity.

3.2 Reducing e-matching patterns to conjunctive queries

We reduce every e-matching to a conjunctive query so that it can be run on the relational representation of e-graphs. We use the algorithm in Figure 3.3 to "unnest" a pattern to a conjunctive query. The Aux function returns a variable and a conjunctive query atom list. Particularly, for non-variable pattern $f(p_1, \ldots p_n)$, Aux produces a fresh variable v and a concatenation of $R_f(v, v_1, \ldots, v_n)$ and atoms from A_i, where $v_i \sim A_i$ is the result of calling $\textsc{Aux}(p_i)$. For variable pattern x, Aux simply returns x and an empty list. Given a pattern p, the Compile function returns a conjunctive query with body atoms from $\textsc{Aux}(p)$ and the head atom consisting of the root variable and variables in p. The compiled conjunctive query and the original e-matching query are equivalent because there is an one-to-one correspondence between the output of them. Specifically, each output atom of the form $Q(c_{root}, c_{x_1}, \dots c_{x_k})$ denotes an e-matching output $(c_{root}, \{x_1 \mapsto c_{x_1}, \dots, x_k \mapsto c_{x_k}\})$.

With this algorithm, the example pattern $f(\alpha, g(\alpha))$ is compiled to the following conjunctive query

$$Q(root, \alpha) \text{ :- } R_f(root, \alpha, x), R_g(x, \alpha).$$

Compared to the original e-matching pattern, this flattened representation enables the pattern search to utilize both the structural and the equality constraints. For example, most of the database optimizers will synthesize query plans that build and lookup indices on both join variables (i.e., x and α), which runs in linear time. In contrast, backtracking-based e-matching takes quadratic time. Backtracking-based e-matching can be seen as a hash join that only builds and look-ups a single variable (i.e., x), and filter the outputs using the equality predicates on α. In other words, existing e-matching algorithms will consider all $f(\alpha, g(\beta))$ regardless whether α is congruent to β, while the generated conjunctive query gives the query optimizer the freedom to synthesize query plans that will consider only atoms where $\alpha \cong \beta$.

3.3 Answering conjunctive queries with generic join

Finally, we consider the problem of efficiently solving the compiled conjunctive queries. We propose to use the generic join algorithm to solve the generated conjunctive queries. Although traditional query plans, which are based on two way joins such as hash joins and merge-sort joins, are extensively used in industrial relational database engine, they may suffer from certain conjunctive queries e-matching generates. For example, consider the pattern $f(g(\alpha), g(\alpha))$. The synthesized conjunctive query is

$$Q(\alpha) \leftarrow R_f(root, x, y), R_g(x, \alpha), R_g(y, \alpha),$$

which is a cyclic conjunctive query. Two-way join plans suffer on these cyclic queries because they are unable to leverage all the constraints in the conjunctive query to prune the space during enumeration. Consequently, two-way join plans will enumerate terms that do not satisfy the constraint and prune them away *a posteriori*. In contrast, the generic join algorithm will only consider a concrete term when the term satisfies all the constraints. Moreover, generic joins has 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.

Chapter 4
DISCUSSION

4.1 Complexity and optimality

Generic join guarantees worst-case optimality with respect to the output size. Our relational e-matching preserves this optimality. In particular, we have the following theorem:

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 nodes, relational e-matching runs in time $\tilde{O}(\max_E(|M(p, E)|))$.*

Proof. Notice that there is an one-to-one correspondence between an output atoms of the generated conjunctive query and the e-matching. Therefore, the worst-case bound is the same across an e-matching pattern and the conjunctive query it generated. Because generic join is worst-case optimal, relational e-matching also runs in worst-case optimal time with respect to the output size. $\square$

4.2 Other join algorithms

Although we choose generic join algorithm in relational e-matching, there are also other choices in the design space. For example, traditional two-way join plans are efficient on acyclic queries, and extensive research has been done on synthesizing highly efficient query plans. Moreover, Yannakakis' algorithm (Papadimitriou and Yannakakis, 1999) is an optimal algorithm on ayclic queries, running in time equal to the size of the output (with possible log factors). However, both of the queries suffer on cyclic conjunctive queries. Two-way join plans spend time enumerating unsatisfying terms, while Yannakakis' algorithm are not applicable to cyclic conjunctive queries. It is possible to choose different join algorithms based on the cyclicity of the query to take advantages each algorithms. However, we currently does not implement this.

4.3 Comparison to graph pattern matching

The idea of representing graph data structure as relational databases are not new. For example, many Datalog programs represent nodes and edges as their own relations to use relational joins to support efficient queries on graph database, known as graph pattern matching. Compared to the common encoding of graphs, our relational representation of e-graphs in a relational database is slightly different and specialized for e-graphs. It is a future work to absorb researches on graph pattern matching for relational e-matching.

Chapter 5
IMPLEMENTATION

We implemented our relational e-matching algorithms for the EGG equality saturation framework. In this section, we describe several implementation details.

5.1 Overview

Given an e-matching pattern, we first transform it to a conjunctive query using the algorithm described in Section 3.2. Next, we lower the conjunctive query to an execution plan, represented as a sequence of variables that the generic join will go through. The execution plan is reconstructed for every new e-graph, because the query compiler may depend on the statistics of the e-graph to generate the execution plan. We then build the indices for each atom $R(v_1, \ldots, v_k)$ in the conjunctive query according to the variable ordering. Finally, we use an efficient interpreter to run the generic join algorithm.

5.2 Variable Ordering

Different variable ordering may result in asymptotically different performance (Amler, 2017; Aberger et al., 2017). Therefore, choosing an variable ordering is important. Compared to join plans for binary joins, there is few literature on query plans for generic joins. Therefore, we developed two heuristics based on empirical experiment. First, we prioritize variables that are associated to more relations, because the intersected set of more relations are more likely to have a smaller cardinality. Moreover, we prioritize variables that are attached to small relations. By intersecting on these variables first, generic join could use the smaller relations to early prune a large portion of atoms in other relations that is not satisfying.

Using these two constraints, the optimizer is able to find superior query plans than the top-down search of backtracking-based e-matching even for linear patterns, where our relational e-matching does not have more information than e-matching. For example, for queries like $x + \alpha$, our optimizer is able to exploit the fact that there is only one atom in the R_x (i.e., the relation representing constant x), so enumerating this relation and use attributes of R_x to prune R_+ will immediately produces all valid substitutions. Meanwhile, in the top-down fashion of backtracking-based e-matching, the matching procedure will need to go through all atoms of R_+ and check if constant x resides in their second child. Such query plan examines much more unnecessary atoms in R_+, because only a small number of $+$-application e-nodes are connected to x.

5.3 Indexing

The generic join assumes that indices are built for the given variable ordering. However, they are crucial to the actual performance of generic join (Amler, 2017). We currently use a recursive trie structure for indexing. The trie is a hash map from the e-class ids to itself. An atom is represented as a path from the root in the trie, according to the order of visiting. The indices are built on demand and shared among different e-matching pattern search on the same e-graph. For most non-trivial e-matching queries, the index building time is neglectable. We believe there are still space for improvement surrounding indices, such as optimizing the set intersection operations and using a more cache-friendly representation.

5.4 Other optimizations

Finally, we write specialized implementations for various numbers of relations being intersected at each nested layer of generic join. This makes possible unrolling most of the loops when manipulating multiple relations together. Moreover, during index building, if the residual relation is known not to join with any other relations, we switch from trie to a flattened vector that directly stores the relation, which is more cache friendly and avoid backtracking during enumeration of the residual relation. In our experiment on handwritten queries, this optimization enables up to $4\times$ performance gain even on simple queries.

Chapter 6
EVALUATION

In this section, we first categorize e-matching patterns into three categories. Next, we evaluate our prototype of relational e-matching with two preliminary experiments, choosing patterns from each of the categories, and discussed our progress on evaluating full-system benchmarks. More experiment will be presented in the forthcoming paper.

To investigate the kinds of conjunctive queries an e-matching pattern would generate and its performance implication on the join algorithms, we categorize the e-matching patterns into three categories according to the kinds of conjunctive queries they generate:

1. Linear patterns: this is the simplest form of patterns, where no variables occur more than once in the pattern. Examples of this category include $f(\alpha, \beta)$ and $f(g(\alpha), g(\beta)))$.

2. Non-linear acyclic patterns: we define non-linear acyclic patterns to be patterns that are not linear and satisfy that every multi-occurrence of a variable must occur between e-nodes that has "distance" ≤ 1 with each other. Examples of this category include $f(\alpha, g(\alpha))$ and $f(f(\alpha, \beta), \alpha)$.

3. Cyclic patterns: we define all other patterns to be cyclic patterns. Examples include $f(g(\alpha), g(\alpha))$ and $f(f(\alpha, \beta), g(\beta)))$. Note that e-matching patterns of this category are always reduced to cyclic queries.

All three categories of patterns exist in real-world applications like equality saturation. For example, the search patterns for commutative law (e.g., $a + b$) and associative law (e.g., $(a + b) + c$) is linear, and the search patterns for the distributive law (e.g., $a \times b + a \times c$) is cyclic, while that for the rule for reciprocal (e.g., $x \times (1/x)$) is non-linear acyclic.

We choose two benchmarks from the EGG's test suites, namely the math test suite, which saturates mathematical expressions, and the lambda test suite, which saturates lambda calculus terms. Both of them are representative of a standard application of equality saturation. For example, the math test suite has many overlapping rules with Herbie (Panchekha et al., 2015), an application of equality saturation in the domain of floating-point arithmetic.

We did two preliminary experiments. The first experiment tries to understand the asymptotic speedup of relational e-matching. In this experiment, we manually write the implementation for relational e-matching for three patterns, which are representative of linear patterns, non-linear acyclic patterns, and cyclic patterns. We benchmark against the e-matching algorithm in EGG, which implements the backtracking-based e-matching algorithm described in de Moura and Bjørner (2007)[1], on different e-graph sizes.

The result is shown in Figure 6.1. On both cyclic and non-linear acyclic case, our relational e-matching achieves asymptotically better performance, up to 426×, over backtracking-based e-matching by taking advantages of the equality constraints. In the linear case, because no variable occurs more than once, relational e-matching achieves similar performance as the backtracking-based e-matching up to a constant factor.[2]

Next, we compare the performance of relational e-matching as we implemented in EGG against EGG's original e-matching implementation on several benchmarks. The result is presented with two figures in Figure 6.2. Figure 6.2a only compares the enumeration procedure of backtracking-based e-matching to that of relational e-matching, excluding time that may need to build the index before performing generic join, which are shared among different patterns, and Figure 6.2a makes the same comparison but includes the index building time.

According to Figure 6.2a, relational e-matching achieves substantial speedups for com-

[1]In fact, EGG does not implement all of the instructions of the backtracking virtual machine described in de Moura and Bjørner (2007), since some of the instructions and their compilation are very complicated.

[2]Note that the comparison here is between the handwritten relational e-matching, which is compiled, and the e-matching engine in EGG, which is interpreted, because this experiment is performed for the PLDI SRC, when we have not developed a fully working version of relational e-matching inside EGG. Therefore, the comparison is not perfectly apple-to-apple. However, we can still see the asymptotic trends.

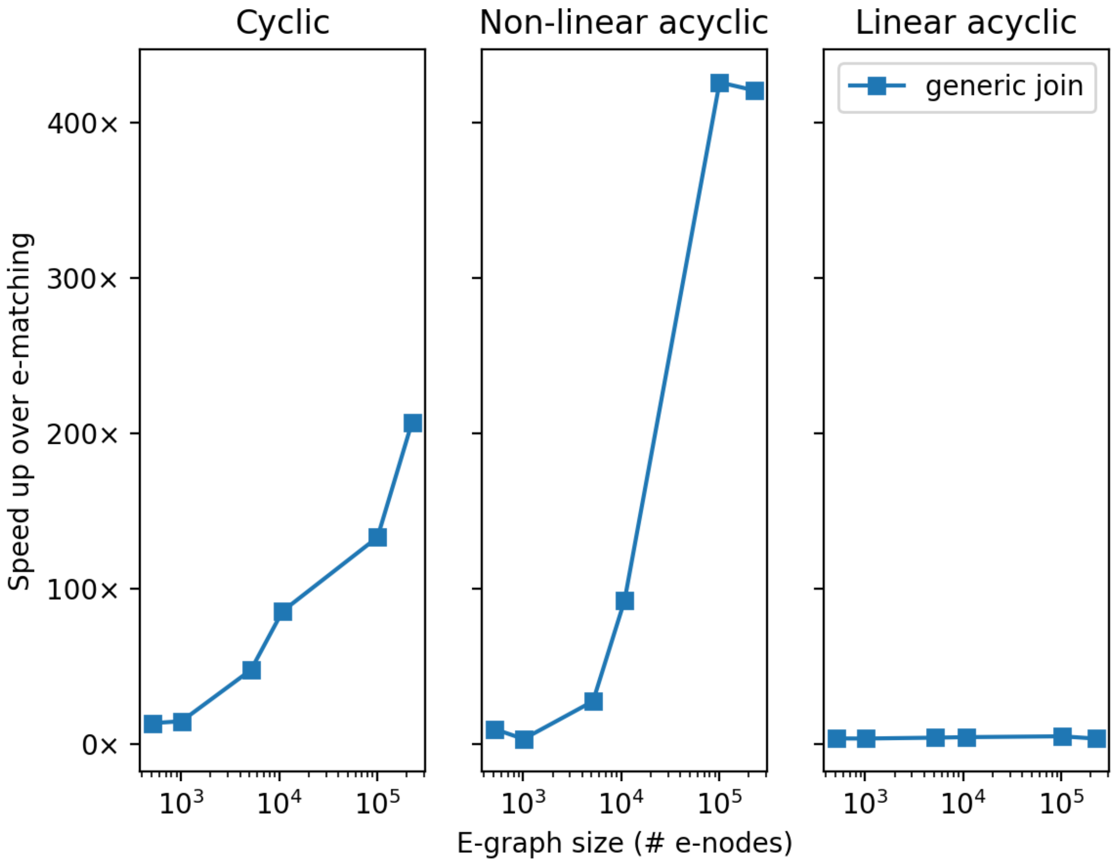
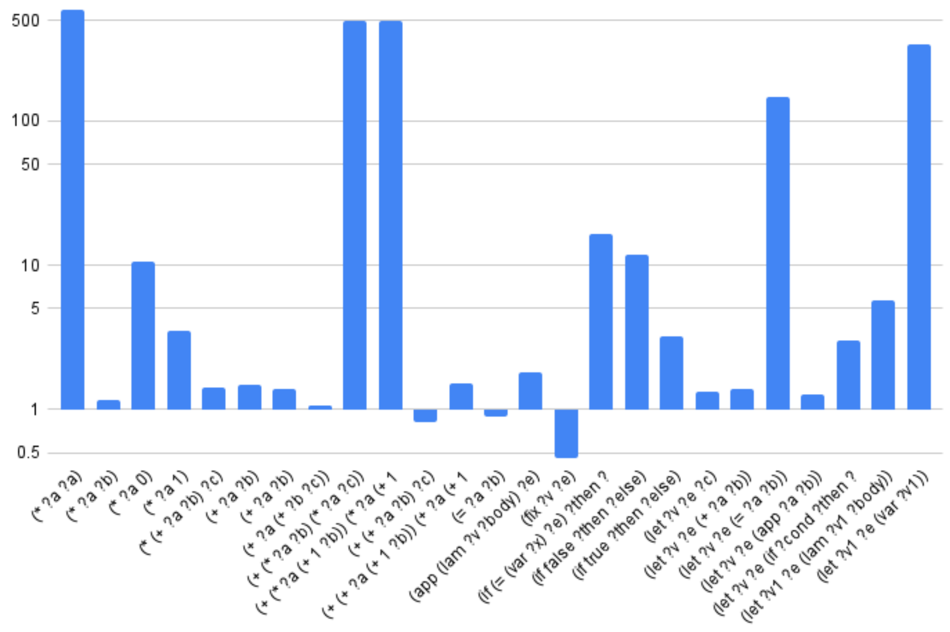
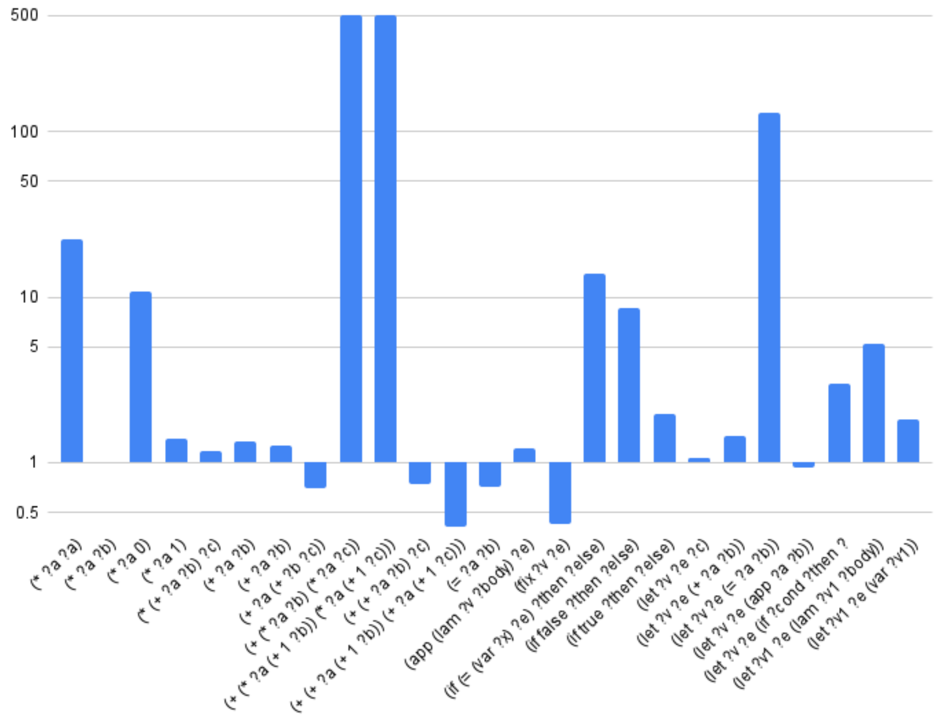


Figure 6.1: Speedup of relational e-matching over backtracking-based e-matching algorithms



(a)



(b)

Figure 6.2: (a) Speedup of relationl e-matching over backtracking-based e-matching on microbenchmarks without index building time. (b) Speedup of relationl e-matching over backtracking-based e-matching on microbenchmarks with index building time.

plex patterns. However, on some simple patterns, relational e-matching does not achieve a significant speedup compared to backtracking-based e-matching algorithms, and sometimes relational e-matching is slower. Most of such patterns are linear patterns like `(+ (+ ?a ?b) ?c)` and `(fix ?v ?e)`. In particular, patterns like `(fix ?v ?e)` are equivalent to a simple enumeration of all f-application terms for some function symbol f, which we call singleton patterns. The relational e-matching has the most slowdown on several singleton patterns because running singleton patterns is very fast, taking only a few microseconds, so the overhead related to relational e-matching such as query compilation dominates the run time. Still, because they only take a few microseconds, the run time for matching such patterns are not an issue for real applications. Moreover, note the relational e-matching performs order of magnitude better on pattern `(* ?a ?a)`, which is a singleton pattern, as our query optimizer exploits the fact that both children of the *-application node are the same and thus only does a filter over the corresponding relation. In contrast, backtracking-based e-matching without ad hoc handling can only express this as a backtracking process, which has a larger overhead.

Comparing Figure 6.2a against Figure 6.2b, we observe the time taken for building indices to be significant. Many plans where relational e-matching is faster now becomes slower. This is consistent with the observation made by Amler (2017), where index building is sometimes the bottleneck of the generic join algorithm. Since our current implementation is only a prototype, we are currently working on improving the index building time so that the performance of relational e-matching is able to dominate that of the current implemented e-matching in EGG.

Finally, our next step of experiment is to evaluate relational e-matching on full-system applications that use EGG. We attempted to run Herbie with relational e-matching as the e-matching procedure. However, we are unable to observe any significant speedup by using relational e-matching. We speculate the cause is that the workload of Herbie is running only simple, linear patterns on small e-graphs, so there are no additional equality constraints that relational e-matching could exploit. We are working on evaluating relational e-matching on other applications, such as Tensat (Yang et al., 2021) and Szalinski (Nandi et al., 2020).

Chapter 7
CONCLUSION

Existing e-matching algorithms are based on backtracking and suffer from problems like implementation difficulty, run inefficiency, and a loose complexity bound. To tackle these problems, we propose relational e-matching, an algorithm that efficiently solves the e-matching problem by reducing it to conjunctive queries. Relational e-matching is based on the observation that e-matching is definitionally similar to conjunctive queries. Leveraging this observation, relational e-matching maps an e-graph to a set of relations and compiles every e-matching pattern to a conjunctive query. As conjunctive query has been extensively studied in the database community for decades, relational e-matching is *conceptually simpler*. Moreover, such relational representation provides an unified way to express not only structural constraints, but also equality constraints, which are constraints that backtracking-based e-matching fails to take advantage of during query planning. By exploiting equality constraints, relational e-matching performs *asymptotically faster* than existinge-matching algorithms on many queries. Finally, to avoid enumerate an unnecessarily large space of candidates, relational e-matching uses generic join to solve the generated conjunctive queries, which makes relational e-matching *worst-case optimal*.

We implemented relational e-matching in EGG and evaluate our relational e-matching on some microbenchmarks. Preliminary experiment indicates that relational e-matching shows asymptotic speedup over backtracking-based e-matching algorithms on both cyclic and nonlinear acyclic patterns. However, for linear patterns, relational e-matching shows similar performance as backtracking-based e-matching algorithms, up to a constant factor. In the future, we plan to evaluate relational e-matching more extensively and on more realistic e-graph applications.

Chapter 8
FUTURE WORKS

We believe there is a deep connections between e-graphs and relational dabase and e-graph can really benefit from research fruits of the database community. This thesis is only a very first attempt to bridging e-graphs and relational databases. Many things are left to be done. This sections sketches a few future works that are worth investigating. Some of them are technical, while others require insights into both areas.

Experiments. The current experiments are still very preliminary. To fully explore the design space of various join algorithms (e.g., hash join, Yannarakis' join algorithm) and query optimization strategies (e.g., different variable ordering for generic join, as well as standard optimization techniques for relational queries) for the scenarios of e-matching.

Multi-patterns. A common extension to e-matching is multi-patterns (de Moura and Bjørner, 2007). Multi-patterns generalize single patterns to multiple patterns that share the the same of variables. For example, $\alpha \times \beta, \alpha \times \gamma$ is a multi-pattern that searches for substitutions σ that satisfy $\sigma(\alpha \times \beta)$ and $\sigma(\alpha \times \gamma)$ are both present simultaneously. They are extensively used in SMT solvers and are also useful in various equality saturation scenarios. Specialized algorithms are developed for multi-patterns. However, from a relational perspective, the conjunctive query generated by a multi-pattern is just a sequence of conjunctive query atoms generated from each pattern placed together. Therefore, the relational e-matching naturally generalizes to the scenarios of multi-patterns.

Optimizing query plans for generic joins. There are not as many research done on optimizing the variable ordering as on optimizing traditional join plans. EmptyHeaded is the first graph database engine that uses generalized hypertree decompositions (GHD) to optimize generic join and achieves performance similar to low-level graph engines. However, it only focuses achieves a stronger run-time worst-case optimality guarantee, but does not take variable ordering into considerations. Yet it has been shown different variable ordering may leads to asymptotically different results (Amler, 2017). Therefore, it is one of our future directions to design algorithms to search for an optimal variable orderings during query optimization.

E-graph engine purely on relational databases. This paper only exploits the possibility of doing e-matching on a relational representation of e-graphs. It is our assumption that normal e-graph operations like merge and lookup are still done on the original e-graph representation and the relational representation will be re-computed during each (batch of) e-matching. However, it is possible to build an e-graph engine purely on top of relational databases by canonicalizing every e-node on the fly during merge. Although we expect it incurs performance penalty to use a purely relational database to describe e-graph and operations on e-graph, it is ideal for situations where interactive e-matching is required. In such cases, it is unrealistic to rebuild the relational representation for every e-matching invocation, while such cost can be avoided if all e-graph operations are directly applied to its relational representation. Moreover, by describing every e-matching operations in terms of the language of relational databases, it is possible to benefit from the literature on incremental maintenance of relational databases and design an algorithm for *incremental* e-matching.

Exploring functional dependency for relational e-matching. Functional dependencies are constraints such as two atoms having the same value on attributes $\mathit{attr}_1, \ldots \mathit{attr}_{n-1}$ implies they also share the same value on attribute attr_n. Functional dependency are ubiquitous in relational databases, and they are also present in the relational representation of e-graphs. In particular, For a relation $R_f(\mathit{eclass\text{-}id}, \mathit{child}_1, \ldots, \mathit{child}_k)$ representing f, the value of *eclass-id* are determined by the values of $\mathit{child}_1, \ldots \mathit{child}_k$.

Our current relational e-matching does not take functional dependencies into considerations, but they may have an impact on the bound of worst-case output. Fortunately, there are worst-case optimal join algorithms that guarantee optimality with the presence of functional dependencies. Therefore, it is worth incorporating functional dependencies into relational e-matching and derives a tighter worst-case bound for relational e-matching.

Datalog engine with internalized congruence. One of our initial motivation for this work is to develop a more flexible language for EGG. Tasks in EGG are described as rewrite rules that add the rewritten terms into the same e-class as the matched terms. Therefore, EGG only supports equivalence reasoning, and it is very hard to describe relations that are not symmetric, such as a transitive closure. However, such directed relations are prevalent in real-world applications. One idea is to make the e-graph engine an extension of the Datalog language (Green et al., 2013). Datalog is a logic programming language that can be efficiently evaluated, and many relational properties such as transitivity, reflexivity, and symmetry can be naturally expressed in Datalog. However, many relations, such as the equivalence relation, may be asymptotically slower if directly expressed in Datalog compared to a dedicated data structure. Soufflé is a Datalog tool that support efficient equivalence relations as a specialized relation and uses an efficient union-find data structure to maintain equivalence relations. Therefore, it is desired to extend Datalog language to efficient maintain congruence closures. On the one hand, this extends the equivalence reasoning in e-graphs to other kinds of reasoning as well. On the other hand, it makes certain existing Datalog programs efficient by using a more dedicated data structure.

BIBLIOGRAPHY

Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4, Article 20 (Oct. 2017), 44 pages. https://doi.org/10.1145/3129246

Andreas Amler. 2017. *Evaluation of Worst-Case Optimal Join Algorithm.* Master's thesis.

Albert Atserias, Martin Grohe, and Dániel Marx. 2008. Size Bounds and Query Plans for Relational Joins. In *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS '08)*. IEEE Computer Society, USA, 739–748. https://doi.org/10.1109/FOCS.2008.43

Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

Leonardo de Moura and Nikolaj Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Automated Deduction – CADE-21*, Frank Pfenning (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 183–198.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *J. ACM* 52, 3 (May 2005), 365–473. https://doi.org/10.1145/1066100.1066102

Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. 2013. Datalog and Recursive Query Processing. *Found. Trends Databases* 5, 2 (Nov. 2013), 105–195. https://doi.org/10.1561/1900000017

Rajeev Joshi, Greg Nelson, and Yunhong Zhou. 2006. Denali: A Practical Algorithm for Generating Optimal Code. *ACM Trans. Program. Lang. Syst.* 28, 6 (Nov. 2006), 967–989. https://doi.org/10.1145/1186632.1186633

Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 31–44. https://doi.org/10.1145/3385412.3386012

Charles Gregory Nelson. 1980. *Techniques for Program Verification.* Ph.D. Dissertation. Stanford, CA, USA. AAI8011683.

Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-Case Optimal Join Algorithms. *J. ACM* 65, 3, Article 16 (March 2018), 40 pages. https://doi.org/10.1145/3180143

Robert Nieuwenhuis and Albert Oliveras. 2005. Proof-Producing Congruence Closure. In *Proceedings of the 16th International Conference on Term Rewriting and Applications* (Nara, Japan) *(RTA'05)*. Springer-Verlag, Berlin, Heidelberg, 453–468. https://doi.org/10.1007/978-3-540-32033-3_33

Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/2737924.2737959

Christos H. Papadimitriou and Mihalis Yannakakis. 1999. On the Complexity of Database Queries. *J. Comput. Syst. Sci.* 58, 3 (June 1999), 407–427. https://doi.org/10.1006/jcss.1999.1626

Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic Code Search via Equational Reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1066–1082. https://doi.org/10.1145/3385412.3386001

Daniel Selsam and Leonardo Moura. 2016. Congruence Closure in Intensional Type Theory. In *Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706*. Springer-Verlag, Berlin, Heidelberg, 99–115. https://doi.org/10.1007/978-3-319-40229-1_8

Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (April 1975), 215–225. `https://doi.org/10.1145/321879.321884`

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 264–276. `https://doi.org/10.1145/1480881.1480915`

Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proc. VLDB Endow.* 13, 12 (July 2020), 1919–1932. `https://doi.org/10.14778/3407790.3407799`

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. `https://doi.org/10.1145/3434304`

Yichen Yang, Phitchaya Mangpo Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. *arXiv e-prints*, Article arXiv:2101.01332 (Jan. 2021), arXiv:2101.01332 pages. arXiv:2101.01332 [cs.AI]