Relational E-matching

YIHONG ZHANG, University of Washington, USA
YISU REMY WANG, University of Washington, USA
MAX WILLSEY, University of Washington, USA
ZACHARY TATLOCK, University of Washington, USA

We present a new approach to e-matching based on relational join; in particular, we apply recent database query execution techniques to guarantee worst-case optimal run time. Compared to the conventional backtracking approach that always searches the e-graph "top down", our new *relational e-matching* approach can better exploit pattern structure by searching the e-graph according to an optimized query plan. We also establish the first data complexity result for e-matching, bounding run time as a function of the e-graph size and output size. We prototyped and evaluated our technique in the state-of-the-art egg e-graph framework. Compared to a conventional baseline, relational e-matching is simpler to implement and orders of magnitude faster in practice.

CCS Concepts: • **Theory of computation → Equational logic and rewriting**.

Additional Key Words and Phrases: E-matching, Relational Join Algorithms

ACM Reference Format:
Yihong Zhang, Yisu Remy Wang, Max Willsey, and Zachary Tatlock. 2022. Relational E-matching. *Proc. ACM Program. Lang.* 6, POPL, Article 35 (January 2022), 22 pages. https://doi.org/10.1145/3498696

1 INTRODUCTION

The congruence closure data structure, also known as the e-graph, is a central component of SMT-solvers [Barrett et al. 2011; de Moura and Bjørner 2008; Detlefs et al. 2005; Moskal et al. 2008] and equality saturation-based optimizers [Tate et al. 2009; Willsey et al. 2021]. An e-graph compactly represents a set of terms and an equivalence relation over the terms. An important operation on e-graphs is e-matching, which finds the set of terms in an e-graph matching a given pattern. In SMT-solvers, e-matching is used to instantiate quantified formulas over ground terms. In equality saturation, e-matching is used to match rewrite rules on an e-graph to discover new equivalent programs. The efficiency of e-matching greatly affects the overall performance of the SMT-solver [Barrett et al. 2011; de Moura and Bjørner 2008], and slow e-matching is a major bottleneck in equality saturation [Nandi et al. 2020; Willsey et al. 2021; Yang et al. 2021]. In a typical application of equality saturation, e-matching is responsible for 60–90% of the overall run time [Willsey et al. 2021].

Several algorithms have been proposed for e-matching [de Moura and Bjørner 2007; Detlefs et al. 2005; Moskal et al. 2008]. However, due to the NP-completeness of e-matching [Kozen 1977], most algorithms implement some form of backtracking search, which are inefficient in many cases. In particular, backtracking search only exploits *structural constraints*, which are constraints about the

Authors' addresses: Yihong Zhang, University of Washington, USA, yz489@cs.washington.edu; Yisu Remy Wang, University of Washington, USA, remywang@cs.washington.edu; Max Willsey, University of Washington, USA, mwillsey@cs.washington.edu; Zachary Tatlock, University of Washington, USA, ztatlock@cs.washington.edu.

2475-1421/2022/1-ART35
https://doi.org/10.1145/3498696

shape of a pattern, but defers checking *equality constraints*, which are constraints that variables should be consistently mapped. This leads to suboptimal run time when the equality constraints dominate the structural constraints.

To improve the performance of backtracking-based e-matching, existing systems implement various optimizations. Some of these optimizations only deal with patterns of certain simple shapes and are therefore *ad hoc* in nature [Moskal et al. 2008]. Others attempt to incrementalize e-matching upon changes to the e-graph, or match multiple similar patterns together to eliminate duplicated work [de Moura and Bjørner 2007]. However, these optimizations are complex to implement and fail to generalize to workloads where the e-graph changes rapidly or when the patterns are complex.

To tackle the inefficiency and complexity involved in e-matching, we propose a systematic approach to e-matching called **relational e-matching**. Our approach is based on the observation that e-matching is an instance of a well-studied problem in the databases community, namely answering *conjunctive queries*. We therefore propose to solve e-matching on an e-graph by reducing it to answering conjunctive queries on a relational database. This approach has several benefits. First, by reducing e-matching to conjunctive queries, we simplify e-matching by taking advantage of decades of study by the databases community. Second, the relational representation provides a unified way to express both the structural constraints and equality constraints in patterns, allowing query optimizers to leverage both kinds of constraints to generate asymptotically faster query plans. Finally, by leveraging the generic join algorithm, a novel algorithm developed in the databases community, our technique achieves the first worst-case optimal bound for e-matching.

Relational e-matching is provably optimal despite the NP-hardness of e-matching. The databases community makes a clear distinction between *query complexity*, the complexity dependent on the size of the query, and *data complexity*, the complexity dependent on the size of the database. The NP-hardness result [Kozen 1977] is stated over the size of the pattern, yet in practice only small patterns are matched on a large e-graph. When we hold the size of each pattern constant, relational e-matching runs in time polynomial over the size of the e-graph.

Our approach is widely applicable. For example, *multi-patterns* are typically framed as an extension to e-matching that allows the user to find matches satisfying multiple patterns simultaneously. Efficient support for multi-patterns requires modifying the basic backtracking algorithm [de Moura and Bjørner 2007]. In contrast, relational e-matching inherently supports multi-patterns for free. The relational model also opens the door to entirely new kinds of optimizations, such as persistent or incremental e-graphs.

To evaluate our approach, we implemented relational e-matching for `egg`, a state-of-the-art implementation of e-graphs. Relational e-matching is simpler, more modular, and orders of magnitude faster than `egg`'s e-matching implementation.

In summary, we make the following contributions in this paper:

- We propose relational e-matching, a systematic approach to e-matching that is simple, fast, and optimal.
- We adapt generic join to implement relational e-matching, and provide the first data complexity results for e-matching.
- We prototyped relational e-matching[1] in `egg`, a state-of-the-art e-graph implementation, and we show that relational e-matching can be orders of magnitude faster.

The rest of the paper is organized as follows: Section 2 reviews relevant background on the e-graph data structure, the e-matching problem, conjunctive queries and join algorithms. Section 3 presents our relational view of e-graphs, our e-matching algorithm, and the complexity results. Section 4 discusses optimizations on our core algorithm and addresses various practical concerns.

[1]We will open source our implementation.

function symbols	$\langle fun \rangle$	::=	$f \mid g \mid \ldots$
variables	$\langle var \rangle$	::=	$x \mid y \mid z \mid \ldots \mid \alpha \mid \beta \mid \ldots$
e-class ids	$\langle id \rangle$	::=	$i \mid j \mid \ldots$
ground terms	$\langle t \rangle$	::=	$\langle fun \rangle \mid \langle fun \rangle\ (\langle t \rangle, \ldots, \langle t \rangle)$
patterns	$\langle p \rangle$	::=	$\langle fun \rangle \mid \langle fun \rangle (\langle p \rangle, \ldots, \langle p \rangle) \mid \langle var \rangle$
e-nodes	$\langle n \rangle$	::=	$\langle fun \rangle \mid \langle fun \rangle (\langle id \rangle, \ldots, \langle id \rangle)$
e-classes	$\langle c \rangle$	::=	$\{\langle n \rangle, \ldots, \langle n \rangle\}$

Fig. 1. Syntax and metavariables used in this paper.

Section 5 evaluates our algorithm and implementation with a set of experiments in the context of equality saturation. Section 6 discusses how the relational model opens up many avenues for future work in e-graphs and e-matching, and Section 7 concludes.

2 BACKGROUND

Throughout the paper we follow the notation in Figure 1. We define the e-graph data structure and the e-matching problem, and review background on relational queries and join algorithms that form the foundation of our e-matching algorithm.

2.1 E-Graphs and E-Matching

Let Σ be a set of function symbols with associated arities. A function symbol is called a *constant* if it has zero arity. Let V be a set of variables. We define $T(\Sigma, V)$ to be the set of terms constructed using function symbols from Σ and variables from V:

Definition 1 (Terms and patterns). The set of terms $T(\Sigma, V)$ over function symbols Σ and variables V is the smallest set such 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. All terms in $T(\Sigma, \emptyset)$ are ground terms. 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.

We define a *congruence relation* over the terms as an equivalence relation that is *congruent*:

Definition 2 (Equivalence Relation). An *equivalence relation* $\equiv_\Sigma$ is a binary relation over $T(\Sigma, \emptyset)$ that is reflexive, symmetric, and transitive.

Definition 3 (Congruence Relation). A *congruence relation* $\cong_\Sigma$ is an equivalence relation satisfying:

$$\forall k\text{-ary function symbols } f.\ \left(\forall i \in \{1, \ldots, k\}.\ t_i \cong t_i'\right) \implies f(t_1, \ldots, t_k) \cong f(t_1', \ldots, t_k')$$

The *congruence closure* of a binary relation R on $T(\Sigma, \emptyset)$ is the smallest congruence relation that contains R.

We write $\equiv$ and $\cong$ when Σ is clear from the context.

An e-graph is a data structure that represents a congruence relation. An e-graph is built up from *e-classes* and *e-nodes*, defined as follows:

Definition 4 (E-classes and E-nodes). An e-class is a set of e-nodes. Every e-class is identified by one or more ids. An e-node is a tuple (f, args) where f is a function symbol and args is a (possibly empty) list of e-class ids. Similar to terms, we call an e-node of the form $(f, i_1, \ldots i_k)$ an f-application e-node. We will write $f(i_1, \ldots, i_k)$ for the e-node $(f, i_1, \ldots i_k)$

Definition 5. A e-graph is a tuple $(C, E, I, U, M, \text{lookup})$ where:

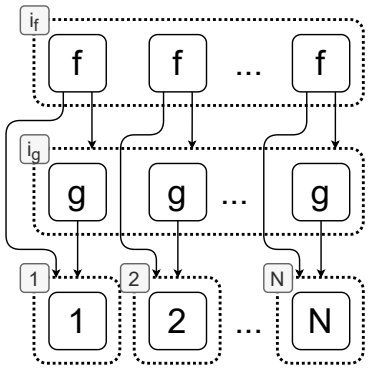


Fig. 2. An example e-graph.

- C is a set of e-classes over E which is a set of e-nodes; I is a set of e-class ids.
- A union-find [Tarjan 1975] data structure U stores an equivalence relation (denoted with $\equiv_{\text{id}}$) over e-class ids. The union-find provides a function find that *canonicalizes* e-class ids such that $\text{find}(i_1) = \text{find}(i_2) \iff i_1 \equiv_{\text{id}} i_2$. An e-class id i is canonical if $i = \text{find}(i)$.
- The *e-class map* M is a surjective function that maps e-class ids to e-classes. All equivalent e-class ids map to the same e-class, i.e., $a \equiv_{\text{id}} b$ iff $M[a]$ is the same set as $M[b]$. An e-class id a is said to *refer to* the e-class $M[\text{find}(a)]$.
- A function lookup that maps e-node n to an id of e-class that contains it: $n \in M[\text{lookup}(n)]$.

Note that by definition, no two e-nodes in the same e-graph can have the same symbol and children, i.e., an e-node's symbol and children together uniquely identify the e-class that contains it. This property is necessary for lookup to be a function. Section 4.3 explains how this property also translates to a *functional dependency* in the e-graph's relational representation, which could be leveraged to further optimize relational e-matching.

Figure 2 shows an example e-graph, where each dotted box is an e-class, and each solid box together with argument pointers make up an e-node. In this e-graph there is one unique id for each e-class, shown in the shaded labels on the top-left corner of each e-class. For example, the e-class in the middle contains the set of e-nodes $g(1), \ldots, g(N)$ and has id i_g.

An e-graph E efficiently represents sets of ground terms in a congruence relation. An e-graph, e-class, or e-node is said to *represent* a term t if t can be "found" within it.

Definition 6 (Representation). Representation is defined recursively:

- An e-graph represents a term if any of its e-classes does.
- An e-class c represents a term if any e-node $n \in c$ does.
- An e-node $f(j_1, \ldots, j_k)$ represents a term $f(t_1, \ldots, t_k)$ if they have the same function symbol f and e-class $M[j_i]$ represents term t_i for $i \in \{1, \ldots, k\}$.

The e-graph in Figure 2 represents the set of terms (where $[N] = \{1, 2, \ldots, N\}$):

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

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. If the e-graph has cycles, it can even represent an infinite set of terms. For example, the e-graph with a single e-class $c = \{f(c), a\}$ represents the infinite set of terms $\{a, f(a), f(f(a)), \ldots\}$.

E-matching. E-matching finds the set of terms in an e-graph matching a given pattern. Specifically, e-matching finds the set of e-matching substitutions and a root class that represents the terms.

Definition 7 (E-matching substitution). An *e-matching substitution* σ is a function that maps every variable in a pattern to an e-class.

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 represented by $\sigma(v_i)$.

Definition 8 (The E-matching problem). 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, and r is said to be the root of matched terms.

For example, matching the pattern $f(\alpha, g(\alpha))$ against the e-graph G in Figure 2 produces the following N substitutions, each with the same root i_f:

$$\left\{(\{\alpha \mapsto j\}, i_f) \mid j \in [N]\right\}.$$

Existing e-matching algorithms perform backtracking search directly on the e-graph [de Moura and Bjørner 2007; Detlefs et al. 2005; Willsey et al. 2021]. Figure 3 shows an abstract backtracking-based e-matching algorithm. Most e-matching algorithms using backtracking search can be viewed as optimizations based on this abstract algorithm. Specifically, it will perform a top-down search following the shape of the pattern and prune the result set of substitutions when necessary. To match pattern $f(\alpha, g(\alpha))$ against G, backtracking search visits terms in the following order (each $\hookrightarrow$ marks a backtrack step):

$$\begin{aligned} f(1, g(1)) &\to \cdots \to f(1, g(N)) \\ \hookrightarrow f(2, g(1)) &\to \cdots \to f(2, g(N)) \\ \hookrightarrow f(N, g(1)) &\to \cdots \to f(N, g(N)) \end{aligned}$$

For each term $f(i, g(j))$ visited, whenever $i = j$ the algorithm yields a match $\alpha \mapsto i$. Despite there being only N matches, backtracking search runs in time $O(N^2)$.

This inefficiency is due to the fact that naïve backtracking does not use the equality constraints to prune the search space *globally*. Specifically, the above e-matching pattern corresponds to three constraints for a potential matching term t:

(1) t should have function symbol f.
(2) t's second child should have function symbol g.
(3) t's first child should be equivalent to the child of t's second child.

We can categorize these constraints into two kinds:

- *Structural constraints* are derived from the structure of the pattern. The structure of pattern $f(\alpha, g(\alpha))$ constrains the root symbol and the second symbol to be f and g respectively (i.e., constraints 1 and 2).
- *Equality constraints* are implied by multiple occurrences of the same variable. Here, the occurrences of α implies that the terms at these positions should be equivalent with each other for all matches (i.e., constraint 3), which we call *equality constraints*. Following Moskal et al. [2008], we define patterns without equality constraints to be *linear patterns*.

Backtracking search exploits the structural constraints first and defers checking the equality constraints to the end. In our example pattern $f(\alpha, g(\alpha))$, backtracking search enumerates all $f(i, g(j))$, regardless of whether i and j are equivalent, only to discard nonequivalent matches later. Complex query patterns may involve many variables that occur at several places, which will makes naïve backtracking search enumerate a very large number of candidates, even though the result size is small.

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

Fig. 3. A declarative backtracking-based e-matching algorithm (reproduced from de Moura and Bjørner [2007]). The set of substitutions for pattern p on e-graph G with e-classes C can be obtained by computing $\bigcup_{c \in C} \text{match}(p, c, \emptyset)$.

2.2 Conjunctive Queries

Conjunctive queries are a subset of queries in relational algebra that use only select, project, and join operators (as opposed to union, difference, or aggregation). Conjunctive queries have many desirable theoretical properties (like computable equivalence checking), and they enjoy efficient execution thanks to decades of research from the databases community.

Relational Databases. 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; for each tuple $(t_1, \ldots, t_k) \in R$, k is the arity of R in S_D and t_i is an element in D. A database instance (or simply database) I of S_D is a set of relations under S_D.

We use the notation $R(x, y).x$ to denote projection, i.e., $R(x, y).x = \{x \mid (x, y) \in R\}$.

Conjunctive Queries. A conjunctive query Q over the schema S_D is a formula of the form:

$$Q(x_1, \ldots x_k) \leftarrow R_1(x_{1,1}, \ldots, x_{1,k_1}), \ldots, R_n(x_{n,1}, \ldots, x_{n,k_n}),$$

where $R_1 \ldots R_n$ are relation symbols in S_D with arities $k_1, \ldots k_n$ and the x are variables.[2] We call the $Q(\ldots)$ part the *head* of the query, the remainder is the *body*. Each $R_i(\ldots)$ 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*, since they are implicitly existentially quantified.

Semantics of Conjunctive Queries. Similar to e-matching, evaluating a conjunctive query Q yields substitutions. Specifically, evaluation 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. Bound variables are *projected out* and not present in resulting the substitutions.

More formally, let I be a database of schema S_D and let Q be a conjunctive query over the same schema with k variables in its head. Let the n atoms in the body of Q be $R_1, \ldots, R_n$ where R_j has arity k_j. Evaluating Q over I yields a substitution $\sigma = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$ iff there exists a $\sigma' \supset \sigma$ mapping all variables in Q such that:

$$\bigwedge_{j \in [n]} (\sigma'(x_{j,1}), \ldots, \sigma'(x_{j,k_j})) \in R_j$$

In practice, conjunctive queries are often evaluated according to a *query plan* which dictates each step of execution. For example, many industrial database systems will construct tree-like query plans, where each node describes an operation like scanning a relation or joining two

[2]Some definitions of conjunctive queries allow both variables and constants. We only allow variables without loss of generality: any constant c can be specified with a distinguished relation $R_c = \{c\}$.

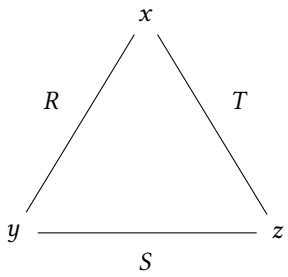


Fig. 4. Query hypergraph of $Q(x, y, z) \leftarrow R(x, y), S(y, z), T(z, x)$.

intermediate relations. Industrial database systems typically construct query plans based on binary join algorithms such as hash joins and merge-sort join, which process two relations at a time. The quality of a query plan critically determines the performance of evaluating a conjunctive query.

We observe that conjunctive query and e-matching are structurally similar: both are defined as finding 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 databases community, including join algorithms and query optimization. We achieve exactly this in Section 3.

2.3 Worst-Case Optimal Join Algorithms

The run time of any algorithm for answering conjunctive queries is lower-bounded by the output size, assuming the output must be materialized. How large can the output of a conjunctive query be on a particular database? A naïve bound is simply the product of the size of each relation, which is the size of their cartesian product. Such a naïve bound fails to consider the query's structure. The AGM bound [Atserias et al. 2008] gives us a bound for the worst-case output size. In fact, the AGM bound is tight; there always exists a database where the query output size is the AGM bound.

The AGM bound and worst-case optimal joins are recent developments in databases research. We do not attempt to provide a comprehensive background on these topics here; familiarity with the existence of the AGM bound and the generic join algorithm is sufficient for this paper.

Consider $Q(x, y, z) \leftarrow R(x, y), S(y, z), T(x, z)$, also known as the "triangle query", since output tuples are triangles between the edge relations R, S, T. We calculate a trivial bound $|Q| \leq |R| \times |S| \times |T|$. If $|R| = |S| = |T| = N$, then $|Q| \leq N^3$. We can derive a tighter bound from $|Q| \leq |R| \times |S| = N^2$. That is because Q contains fewer tuples than the query $Q'(x, y, z) \leftarrow R(x, y), S(y, z)$ as Q further requires $(x, z) \in T$. The AGM bound for Q is even smaller: $N^{3/2}$. It is computed from the *fractional edge cover* of the *query hypergraph*.

Query Hypergraph. The hypergraph of a query is simply the hypergraph with a vertex for each variable and a (hyper)edge for each atom. The edge for an atom $R(x_i, \ldots, x_k)$ connects the vertices corresponding to the variables $x_i, \ldots, x_k$. Figure 4 illustrates Q's hypergraph.

Cyclic and Acyclic Queries. Certain queries can be represented by a tree, called the *join tree*, where each node corresponds to an atom in the query. Furthermore, for each variable x the nodes corresponding to the atom containing x must form a connected subtree. Queries that admit such a join tree are said to be *acyclic*; otherwise, the query is *cyclic*.[3] The triangle query is cyclic because it

[3] A cycle in the hypergraph does not necessarily entail a cyclic query, since the hypergraph may still admit a join tree.

cannot be represented by a join tree. Acyclic queries can be answered more efficiently than cyclic ones.

Fractional Edge Cover. A set of edges *cover* a graph if they touch all vertices. For Q's hypergraph, any two edges form a cover. A *fractional edge cover* assigns a weight in the interval $[0, 1]$ to each edge such that, for each vertex v, the weights of the edges containing v sum to at least 1. Every edge cover is a fractional cover, where every edge is assigned a weight of 1 if it is in the cover, and 0 otherwise. For Q's hypergraph, $\{R \mapsto 1/2, S \mapsto 1/2, T \mapsto 1/2\}$ is the fractional edge cover with lowest total weight.

The AGM Bound. The AGM bound [Atserias et al. 2008] for a query with body atoms $R_i(\ldots)$ for $i \in [k]$ is defined as $\min_{w_1,\ldots,w_k} \Pi_{i\in[k]}|R_i|^{w_i}$, where $\{R_i \mapsto w_i \mid i \in [k]\}$ forms a fractional edge cover. For example, the AGM bound for Q is $|R|^{1/2}|S|^{1/2}|T|^{1/2} = N^{3/2}$ when $|R| = |S| = |T| = N$. This is the upper bound of Q's output size; i.e. in the worst case Q outputs this many tuples.

Generic Join. A desirable algorithm for answering conjunctive queries should run in time linear to the worst case output size. Recent developments in the databases community have led to such an algorithm [Ngo et al. 2018], one of which is *generic join* [Ngo et al. 2014]. Generic join has one parameter: an ordering of the variables in the query. Any ordering guarantees a run time linear to the worst-case output size, but different orderings can lead to dramatically different run time in practice [Aberger et al. 2017].

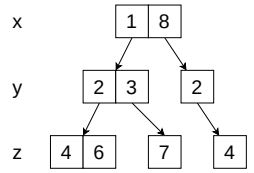
Algorithm 1: Generic join for the general query $Q(x_1, \ldots, x_k) \leftarrow R_1(\overline{X}_1), \ldots, R_n(\overline{X}_n)$

Result: $\mathrm{GJ}(Q, \emptyset)$ computes the output of query Q
Input: query Q, partial substitution σ
/* k indicates how many variables remain in query Q */
if $k = 0$ **then** /* there are no more variables, so σ is complete */
  output σ
else
  choose a variable x;
  /* Compute D_x, which all possible values of x, by intersecting the attributes of the relations where x occurs. Intersection must be computed in $O(\min(|R_j.x|))$ time. */
  $J = \{j \mid x \in \overline{X}_j\}$;
  $D_x = \cap_{j\in J} R_j.x$;
  for $v \in D_x$ **do**
    /* compute residual query by replacing variable x with constant v */
    $Q' = Q[v/x]$;
    $\mathrm{GJ}(Q', \sigma \cup \{x \mapsto v\})$
  end
end

Algorithm 1 shows the generic join algorithm. Generic join is recursive, proceeding in two steps. First, it chooses a variable from the query and collects all possible values for that variable in the query. Then, for each of those values, it builds a *residual query* by replacing occurrences of the variable in the query with a possible value for that variable. These residual queries are solved recursively, and when there are no more variables in the residual query, the algorithm yields the substitution it has accumulated so far.

x	y	z
1	2	4
1	2	6
1	3	7
8	2	4

(a) Table for relation $R(x, y, z)$.



(b) Trie for relation $R(x, y, z)$ using ordering $[x, y, z]$.

Fig. 5. A trie is a tree where every node is a map (typically a hashmap or sorted map) from a value to a trie. Every path from the root of a trie to a leaf represents a tuple in the relation. Tries allow efficient computation of residual relations. For example, $R(1, y, z)$ can be computed quickly by following the $x = 1$ edge from the root.

Generic join requires two important performance bounds to be met in order for its own run time to meet the AGM bound. First, the intersection on line 6 must run in $O(\min(|R_j.x|))$ time. Second, the residual relations should be computed in constant time, i.e., computing from the relation $R(x, y)$ the relation $R(v_x, y)$ for some $v_x \in R(x, y).x$ must take constant time. Both of these can be solved by using tries (sometimes called prefix or suffix trees) as an indexing data structure. Figure 5 shows an example trie. Tries allow fast intersection because each node is a map which can be intersected in time linear to the size of the smaller map. Tries also allow constant-time access to residual relations according to a compatible variable ordering.

A useful way to understand generic join is to observe the loops and intersections it performs on a specific query. Algorithm 2 shows generic join computing the triangle query. Given a variable ordering ($[x, y, z]$ in this case), generic join assumes the input relations are stored in tries according to the ordering, so $R(x, y)$ is stored in a trie with xs on the first level, and ys on the second. This makes accessing the residual relations ($R(v_x, y)$) fast since the replacement of variables with values is done according to the given variable ordering. Note how the algorithm is essentially just nested `for` loops. There is no explicit filtering step; the intersection of residual queries guarantees that once a complete tuple of values is selected, it can be immediately output without additional checking.

Algorithm 2: Generic join for the triangle query, with ordering $[x, y, z]$.

```
   Result: compute Q(x, y, z) ← R(x, y), S(y, z), T(z, x)
1  X = R(x, y).x ∩ T(z, x).x;
2  for v_x ∈ X do                          /* compute Q(v_x, y, z) = R(v_x, y), S(y, z), T(z, v_x) */
3  |  Y = R(v_x, y).y ∩ S(y, z).y;
4  |  for v_y ∈ Y do                       /* compute Q(v_x, v_y, z) = R(v_x, v_y), S(v_y, z), T(z, v_x) */
5  |  |  Z = S(v_y, z).z ∩ T(z, v_x).z;
6  |  |  for v_z ∈ Z do                    /* yield join results Q(v_x, v_y, v_z) */
7  |  |  |  output(v_x, v_y, v_z)
8  |  |  end
9  |  end
10 end
```

3 RELATIONAL E-MATCHING

E-matching via backtracking search is inefficient because it handles equality constraints suboptimally. In fact, backtracking search follows edges in the e-graph and only visits concrete terms that

$f(1, i_g)$	$g(1)$	✓
	⋮	
	$g(N)$	×
$f(2, i_g)$	$g(1)$	×
	$g(2)$	✓
	⋮	
	$g(N)$	×
$f(N, i_g)$	$g(1)$	×
	⋮	
	$g(N)$	✓

(a) Backtracking takes time $O(N^2)$

build hash ↓	probe ↓	
$R_g(i_g, 1)$	$R_f(i_f, 1, i_g)$	✓
$R_g(i_g, 2)$	$R_f(i_f, 2, i_g)$	✓
$R_g(i_g, 3)$	$R_f(i_f, 3, i_g)$	✓
⋮	⋮	
$R_g(i_g, N)$	$R_f(i_f, N, i_g)$	✓

(b) Hash join takes time $O(N)$.

Fig. 6. E-matching $f(\alpha, g(\alpha))$ with backtracking search and a simple hash join on the e-graph/database in Figure 7.

satisfy structural constraints. However, equality constraints are checked *a posteriori* only after the search visits a (partial) term.[4] Whenever there are many terms that satisfy the structural constraints but not the equality constraints, as is in our example pattern $f(\alpha, g(\alpha))$, backtracking will waste time visiting terms that do not yield a match.

By reducing e-matching to evaluating conjunctive queries, we can use join algorithms that take advantage of both structural *and* equality constraints. Figure 6 conveys this intuition using the pattern $f(\alpha, g(\alpha))$ and the example e-graph and database from Figure 7. The backtracking approach considers every possible assignment to the variables, even those where the two occurrences of α do not agree.

We can instead formulate a conjunctive query that is equivalent to the following pattern:

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

Later subsections will detail how this conversion is done, but note how the auxiliary variable x captures the structural constraint from the pattern. Evaluating Q with a simple hash join strategy exemplifies the benefits of the relational approach: it considers structural *and* equality constraints (in this case by doing a hash join keyed on (x, α)); indeed, the relational perspective sees no difference between the two kinds of constraints.

This observation leads us to a very simple algorithm for relational e-matching, shown in Algorithm 3. 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. These intermediate steps will be detailed in the following subsections.

3.1 From the E-Graph to a Relational Database

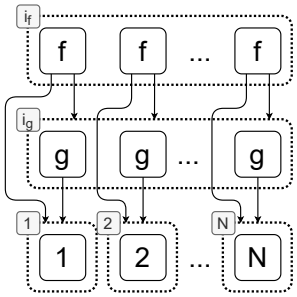
The first step of relational e-matching is to transform the e-graph E into a relational database I. The domain of the database is e-class ids, and its schema is determined by the function symbols in Σ. Every e-node with symbol f in the e-graph corresponds to a tuple in the relation R_f in the database. If f has arity k, then R_f will have arity $k + 1$; its first attribute is the e-class id that contains the

[4]Backtracking e-matching can perform the check as soon as it has traversed enough of the pattern to encounter a variable more than once.

Algorithm 3: RelationalEMatching

Input: An e-graph E and a list of e-matching patterns ps
Output: The result of running ps on E

```
1 I ← EGraphToDatabase(E);
2 qs ← {PatternToCQ(p) | p ∈ ps};
3 return {EvalCQ(q, I) | q ∈ qs}
```



(a) An example e-graph, reproduced from Figure 2.

id	$\arg_1$	$\arg_2$
i_f	1	i_g
i_f	2	i_g
⋮	⋮	⋮
i_f	N	i_g

(b) Relation of f.

id	$\arg_1$
i_g	1
i_g	2
⋮	⋮
i_g	N

(c) Relation of g.

Fig. 7. An e-graph and its relational representation. Each e-class (dotted box) is labeled with its id.

corresponding e-node, and the remaining attributes are the k children of the f e-node. Figure 7 shows an example e-graph and part of its corresponding database. In particular, only the relations of function symbols f and g are presented in this figure. There are N other relations; each relation R_j represents a constant j and has exactly one tuple (i.e., singleton (j)).

We construct the database I by simply looping over every e-node in every e-class in E and making a tuple in the corresponding relation:

$$I = \left\{R_f \leftarrow (\mathsf{find}(i), \mathsf{find}(j_1), \ldots, \mathsf{find}(j_k)) \mid M[i] = f(j_1, \ldots, j_k)\right\}$$

Note that the tuples in the database contain only canonical e-class ids returned from the find function.

Our presentation in this paper specifically targets e-matching use cases like equality saturation, where the building of the database I can be amortized. In this setting, e-matching is done in large batches, and expensive work like congruence closure can be amortized between these batches using a technique called "rebuilding" [Willsey et al. 2021]. The time complexity of building this database is always linear, which is subsumed by the time complexity of most non-trivial e-matching patterns. In Section 6.4, we discuss how this technique could be generalized to the non-amortized setting of frequently updated e-graph as future work.

3.2 From Patterns to Conjunctive Queries

Once we have a database that corresponds to the e-graph, we must convert each pattern we wish to e-match to a conjunctive query. We use the algorithm in Figure 8 to "unnest" a pattern to a conjunctive query by connecting nested patterns with *auxiliary variables*.

The Aux function returns a variable and a conjunctive query atom list. Particularly, for non-variable pattern $f(p_1, \ldots p_k)$, Aux produces a fresh variable v and a concatenation of $R_f(v, v_1, \ldots, v_k)$ 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

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

Fig. 8. Compiling a pattern to a conjunctive query.

simply returns x and an empty list. Note that the auxiliary variables introduced by $\textsc{Aux}(f(\ldots))$ are not included in the head of the query, and thus are not part of the output.

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 a one-to-one correspondence between the output of them. Specifically, each e-matching output (i_{root}, σ) corresponds to a query output of $\{root \mapsto i_{root}\} \cup \sigma$. The only difference is that returning the root e-class id is a special consideration for e-matching, but it is just another variable in the conjunctive query.

The Compile function (specifically the Aux subroutine) relies on the fact that the database contains only canonical e-class ids. Without this fact, nested patterns would require an additional join on the equivalence relation $\equiv_{\text{id}}$. But since $i \equiv_{\text{id}} j \iff i = j$ if i and j are canonical e-class ids, we can omit introducing the additional join, instead joining nested patterns directly on the auxiliary variable.

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

$$Q(root, \alpha) \leftarrow R_f(root, \alpha, x), R_g(x, \alpha). \tag{1}$$

Compared to the original e-matching pattern, this flattened representation enables relational e-matching to utilize both the structural and the equality constraints. For example, a reasonable query plan that database optimizers will synthesize is a hash join on both join variables (i.e., x and α), which takes $O(N)$ time. In contrast, backtracking-based e-matching takes $O(N^2)$ time.

Figure 6 shows the traces for running a direct backtracking search on the e-graph and running hash join on the relational representation. Every term enumerated by hash join will simultaneously satisfy all the constraints. Conceptually, backtracking-based e-matching can be seen as a hash join that only builds and look-ups a single variable (i.e., x), and filters the outputs using the equality predicate on α. In other words, existing e-matching algorithms will consider all $f(\alpha, g(\beta))$ terms regardless of 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 tuples where $\alpha \cong \beta$.

3.3 Answering CQs 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 on certain queries compiled from patterns. For example, consider the pattern $f(g(\alpha), h(\alpha))$. The compiled conjunctive

query is:

$$Q(root, \alpha) \leftarrow R_f(root, x, y), R_g(x, \alpha), R_h(y, \alpha). \tag{2}$$

Like the classic triangle query, this is a cyclic conjunctive query (Section 2.3). We call e-matching patterns that generate cyclic conjunctive queries *cyclic patterns*. For such cyclic queries, Ngo et al. [2018] show 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.

Using the generic join algorithm, suppose we fix the variable ordering to be $[\alpha, x, root]$ on the generated conjunctive query 1. The algorithm below shows generic join instantiated on this particular CQ:

Algorithm 4: Relational e-matching using GJ for $f(\alpha, g(\alpha))$, with ordering $[\alpha, x, root]$.

Result: compute $Q(root, \alpha) \leftarrow R_f(root, \alpha, x), R_g(x, \alpha)$

```
// compute all possible values of α
```
$A = R_f(root, \alpha, x).\alpha \cap R_g(x, \alpha).\alpha$;
for $i_\alpha \in A$ **do**
  `// compute all possible values of x given` $\alpha = i_\alpha$
  $X = R_f(root, i_\alpha, x).x \cap R_g(x, i_\alpha).x$;
  for $i_x \in X$ **do**
    `// compute all possible values of` *root* `given` $\alpha = i_\alpha$ `and` $x = i_x$
    $Roots = R_f(root, i_\alpha, i_x).root \cap R_g(i_x, i_\alpha).root$;
    for $i_{root} \in Roots$ **do**
      output(i_{root}, i_α)
    end
  end
end

3.4 Complexity of Relational E-matching

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

Theorem 9. *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 $O(\max_E(|M(p, E)|))$.*

Proof. Notice that there is an one-to-one correspondence between output tuples of the generated conjunctive query and the e-matching pattern. 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. □

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

Theorem 10. *Fix an e-graph E with N e-nodes that compiles to a database I, and a fix 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 $O\left(\sqrt{|Q(I)| \times \Pi_i |R_i|}\right) \leq O\left(\sqrt{|Q(I)| \times N^m}\right)$.*

Proof. Let $\overline{X^\circ}$ be the set of isolated variables, those that occur in only one atom. Note that $\overline{X^\circ} \subseteq \overline{X}$, since $\overline{X}$ is precisely the pattern variables and the root, and auxiliary variables must occur in at least two atoms. Using these, define two new queries:

$$C(\overline{X^\circ}) \leftarrow R_1(\overline{X_1}), \ldots, R_m(\overline{X_m})$$
$$Q'(\overline{X}) \leftarrow R_1(\overline{X_1}), \ldots, R_m(\overline{X_m}), C(\overline{X^\circ})$$

Since $\overline{X^\circ} \subseteq \overline{X}$, C is the same query as Q but with zero or more variables projected out. Therefore, every tuple in $C(I)$ corresponds to one in the output $Q(I)$, so $C(I) \subseteq Q(I)$ and $|C(I)| \leq |Q(I)|$.

Now we can compute the AGM bound for Q'. Our new atom $C(\overline{X^\circ})$ includes all those variables that only appear in one atom of Q. Therefore, every variable in Q' occurs in at least two atoms, so assigning $1/2$ to each edge is a fractional edge cover. Thus:

$$\begin{aligned}
\text{AGM}(Q') &= \sqrt{|C(I)| \times \Pi_i |R_i|} \\
&\leq \sqrt{|Q(I)| \times \Pi_i |R_i|} && \text{since } |C(I)| \leq |Q(I)| \\
&\leq \sqrt{|Q(I)| \times N^m} && \text{since } |R_i| < N
\end{aligned}$$

Let $\text{GJ}(Q', I)$ denote the running time of generic join with query Q' on database I. We know that $\text{GJ}(Q', I) \leq \text{AGM}(Q')$. Because $C(I) \subseteq Q(I)$, we also know $Q'(I) = Q(I)$, and we can use $GJ(Q', I)$ to bound $GJ(Q, I)$. Now we show that $\text{GJ}(Q, I) \leq \text{GJ}(Q', I)$.

The query Q' is just Q with an additional atom $C(\overline{X^\circ})$ that covers the variables that only appeared in one atom from Q. Fix a variable ordering for generic join that puts those variables in $\overline{X^\circ}$ at the end. So loops of both GJ instantiations are the same, except that, in Q', each loop corresponding to a variable in $\overline{X^\circ}$ performs an intersection with C, but not in Q. But these intersections are in the innermost loops, at which point all intersections with atoms from Q have already been done. So the intersections with C do nothing, since C is precisely Q projected down to the variables in $\overline{X^\circ}$! Since those intersections are not helpful and Q simply does not do them, $\text{GJ}(Q, I) \leq \text{GJ}(Q', I)$.

Putting the inequalities together, we get:

$$\text{GJ}(Q, I) \leq \text{GJ}(Q', I) \leq AGM(Q') \leq O\left(\sqrt{|Q(I)| \times \Pi_i |R_i|}\right) \leq O\left(\sqrt{|Q(I)| \times N^m}\right)$$

. □

Example 11 (Complexity of relational e-matching). Consider the pattern $f(g(\alpha))$, which compiles to the query $Q(r, \alpha) \leftarrow R_f(r, x), R_g(x, \alpha)$. Following the proof we define $C(r, \alpha) \leftarrow R_f(r, x), R_g(x, \alpha)$ and $Q'(r, \alpha) \leftarrow R_f(r, x), R_g(x, \alpha), C(r, \alpha)$. The AGM bound for Q' is $N^{1/2} N^{1/2} |C|^{1/2} = N\sqrt{|C|} = N\sqrt{|Q|}$. This also bounds the run time of generic join on Q.

The above bound is tight for linear patterns, in which case each variable occurs exactly twice in Q'. In the case of nonlinear patterns, we may find tighter covers than assigning $1/2$ to each atom, thereby improving the bound.

3.5 Supporting Multi-patterns

Multi-patterns are an extension to e-matching used in both SMT solvers [de Moura and Bjørner 2007] and program optimizations [Yang et al. 2021]. A multi-pattern is a list of patterns of the form

$(p_1, \ldots, p_k)$ that are to be simultaneously matched (i.e., the instantiation of each contained pattern should use the same substitution σ). 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 [de Moura and Bjørner 2007]. 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, the conjunctive query for the multi-pattern $(f(\alpha, \beta), f(\alpha, \gamma))$ is

$$Q(root_1, root_2, \alpha, \beta, \gamma) \leftarrow R_f(root_1, \alpha, \beta), R_g(root_2, \alpha, \gamma). \tag{3}$$

This is one example that shows the wide applicability of the relational model adopted in relational e-matching.

4 OPTIMIZATIONS

Our implementation of relational e-matching using generic join is simple (under 500 lines), but that does not preclude having several optimizations important for practical performance.

4.1 Degenerate Patterns

Not all patterns correspond to conjunctive queries that involve relational joins. Non-nested patterns (whether linear or non-linear) will produce relational queries without any joins:

$$f(\alpha, \beta) \leftrightarrow R_f(root, \alpha, \beta)$$
$$f(\alpha, \alpha) \leftrightarrow R_f(root, \alpha, \alpha)$$

The corresponding query plan is simply a scan of a relation with possible filtering. For these queries, generic join (or any other join plan) offers no benefit, and building the indices for generic join incurs unnecessary overhead. A relational e-matching implementation (or any other kind) should have a "fast path" for these relatively common types of queries that simply scans the e-graph/database for matching e-nodes/tuples. For this reason, we exclude these kinds of patterns from our evaluation in Section 5.

4.2 Variable Ordering

Different variable orderings can dramatically affect performance for generic join [Aberger et al. 2017; Amler 2017], so choosing a variable ordering is important. Compared to join plans for binary joins, query plans for generic join is much less studied. In relational e-matching we choose a variable ordering using two simple heuristics: First, we prioritize variables that occur in many relations, because the intersected set of many relations is likely to be smaller. Second, we prioritize variables that occur in small relations, because intersections involving a small relations are also likely to be smaller. Performing smaller intersections first can prune down the search space early.

Using these two heuristics, the optimizer is able to find more efficient query plans than the top-down search of backtracking-based e-matching. This is even true for linear patterns, where our relational e-matching has no more information than e-matching, but it does have more flexibility. Consider the linear pattern $f(g(h(\alpha)))$ compiled to the query $R_f(root, x), R_g(x, y), R_h(y, \alpha)$. When there are very few h-application e-nodes in the e-graph, R_h will be small. The variable ordering $[y, x, root, \alpha]$ takes advantage of this by intersecting $R_g.y \cap R_h.y$ first, resulting in an intersection no larger than R_h. This "bottom-up" traversal is not possible in conventional e-matching.

4.3 Functional Dependencies

Functional dependencies describe the dependencies between columns. For example, a functional dependency on relation $R(y, x_1, x_2, x_3)$ of the form $x_1, x_2, x_3 \rightarrow y$ indicates that for each tuple of R,

the values of x_1, x_2, and x_3 uniquely determines the value y. Functional dependencies are ubiquitous in relational e-matching. In fact, every transformed schema of the form $R_f(\textit{e-class}, \textit{arg}_1, \ldots, \textit{arg}_k)$ has a functional dependency from $\textit{arg}_1, \ldots, \textit{arg}_k$ to *e-class*. When the variable graph formed by functional dependencies is acyclic [5], we can speed up generic join by ordering the variables to follow the topological order of the functional dependency graph. Every conjunctive query compiled from an e-matching pattern has acyclic functional dependencies, because each dependency goes from the e-node's children to the e-node's parent e-class. Relational e-matching can therefore always choose a variable ordering that is consistent with the functional dependency. Our implementation tries to respect functional dependencies, but prioritizes larger intersections more.

As an example, consider conjunctive query 2 again. It is synthesized from pattern $f(g(\alpha), h(\alpha))$ and, assuming each relation has size N, an AGM bound of $O(N^{3/2})$. Suppose however that we pick the variable ordering π to be $[\alpha, x, y, \textit{root}]$. For every possible value of α chosen, there will be at most one possible value for x, y, and *root* by functional dependency, which can be immediately determined. This reduces the run time from $O(N^{3/2})$ to $O(N)$.

4.4 Batching

Generic join always processes one variable at a time, even if multiple consecutive variables are from the same atom. We find this strategy to be inefficient in practice, as it results in deeper recursion that does little useful work.

Consider the query $Q(x, y, z, w) \leftarrow R(x, y, \alpha), S(\alpha, z, w)$. The right variable ordering places α at the front, since it is the only intersection. We observe that variables that only appear in one atom can be "batched" with others that only appear in the same atom. Batched variables are treated as a single variable in the trie and intersections. So instead of variable ordering $[\alpha, x, y, z, w]$, we can use $[\alpha, (x, y), (z, w)]$. This lowers the recursion depth of generic join (from 5 to 3) and improves data locality by reducing pointer-chasing.

5 EVALUATION

To empirically evaluate relational e-matching, we implemented it inside the egg equality saturation toolkit [Willsey et al. 2021]. Our implementation consists about 80 lines of Rust inside egg itself to convert patterns into conjunctive queries, paired with a 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 CQ solver is completely modular. We could plug in a different generic join implementation [6], or even a more conventional binary join implementation.

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

5.1 Benchmarking Setup

We use egg's two largest benchmark suites as the basis for our two benchmark suites. The `math` suite implements a simple computer algebra system, including limited support for symbolic differentiation and integration. The `lambda` suite implements a partial evaluator for the lambda calculus. Each egg benchmark suite provides a set of rewrite rules, each with a left and right side pattern, and a set of starting terms.

[5] Cyclicity of functional dependencies is unrelated to cyclicity of the query.
[6] There is no reusable generic join implementation at the time of writing.

Table 1. Summary statistics across patterns for each benchmark suite. The "Idx" column shows whether the time to build indices in GJ is included (+) or not (–); the row color corresponds to the colors in Figure 9. The "Suite" columns shows the benchmark suite, and the "EG Size" shows the number of e-nodes in the e-graph used to benchmark. The "GJ" and "EM" columns show how many patterns that algorithm was fastest on in this configuration. "Total" shows the cumulative speedup across all patterns in that configuration. The remaining columns show statistics about the EM/GJ ratios for each pattern: harmonic mean, geometric mean, max, median, and min.

Idx	Suite	EG Size	GJ	EM	Total	HMean	GMean	Best	Medn	Worst
+	lambda	4,142	15	3	1.69	.84	1.71	13.62	1.60	.12
–	lambda	4,142	18	0	2.58	2.99	4.23	39.17	3.68	1.10
+	lambda	57,454	16	2	2.60	.95	2.66	136.54	2.65	.12
–	lambda	57,454	18	0	2.87	3.33	9.11	406.70	4.05	1.03
+	lambda	109,493	15	3	1.66	1.75	3.11	148.96	2.03	.65
–	lambda	109,493	18	0	1.70	3.32	7.46	291.18	4.10	1.05
+	lambda	213,345	15	3	2.20	1.55	3.40	304.33	1.72	.43
–	lambda	213,345	18	0	2.21	2.96	8.23	501.12	5.04	1.04

Idx	Suite	EG Size	GJ	EM	Total	HMean	GMean	Best	Medn	Worst
+	math	8,205	30	2	5.49	0.64	4.61	66.54	2.79	0.03
–	math	8,205	30	2	5.21	2.93	8.62	1,630.00	5.48	0.62
+	math	53,286	29	3	311.23	2.61	13.50	50,030.29	3.62	0.74
–	math	53,286	30	2	318.95	3.39	29.60	1,325,802.56	30.72	0.74
+	math	132,080	29	3	96.55	2.66	15.18	61,488.73	4.02	0.60
–	math	132,080	30	2	97.84	3.46	34.16	2,447,939.38	68.71	0.75
+	math	217,396	30	2	119.82	2.83	18.34	101,023.37	3.91	0.72
–	math	217,396	31	1	119.73	3.45	41.35	8,575,830.58	80.84	0.76

To construct the e-graphs used in our 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 that are 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 our GJ approach, we ran each trial twice. The first time builds the index tries 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. In both Figure 9 and Table 1, orange bars/rows show the first runs (including indexing), and blue bars/rows show the second runs (excluding indexing).

All benchmarks are single-threaded, and they were executed on a 4.6GHz CPU with 32GB of memory.

5.2 Results

Figure 9 show the results of our benchmarking experiments. GJ can be over 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

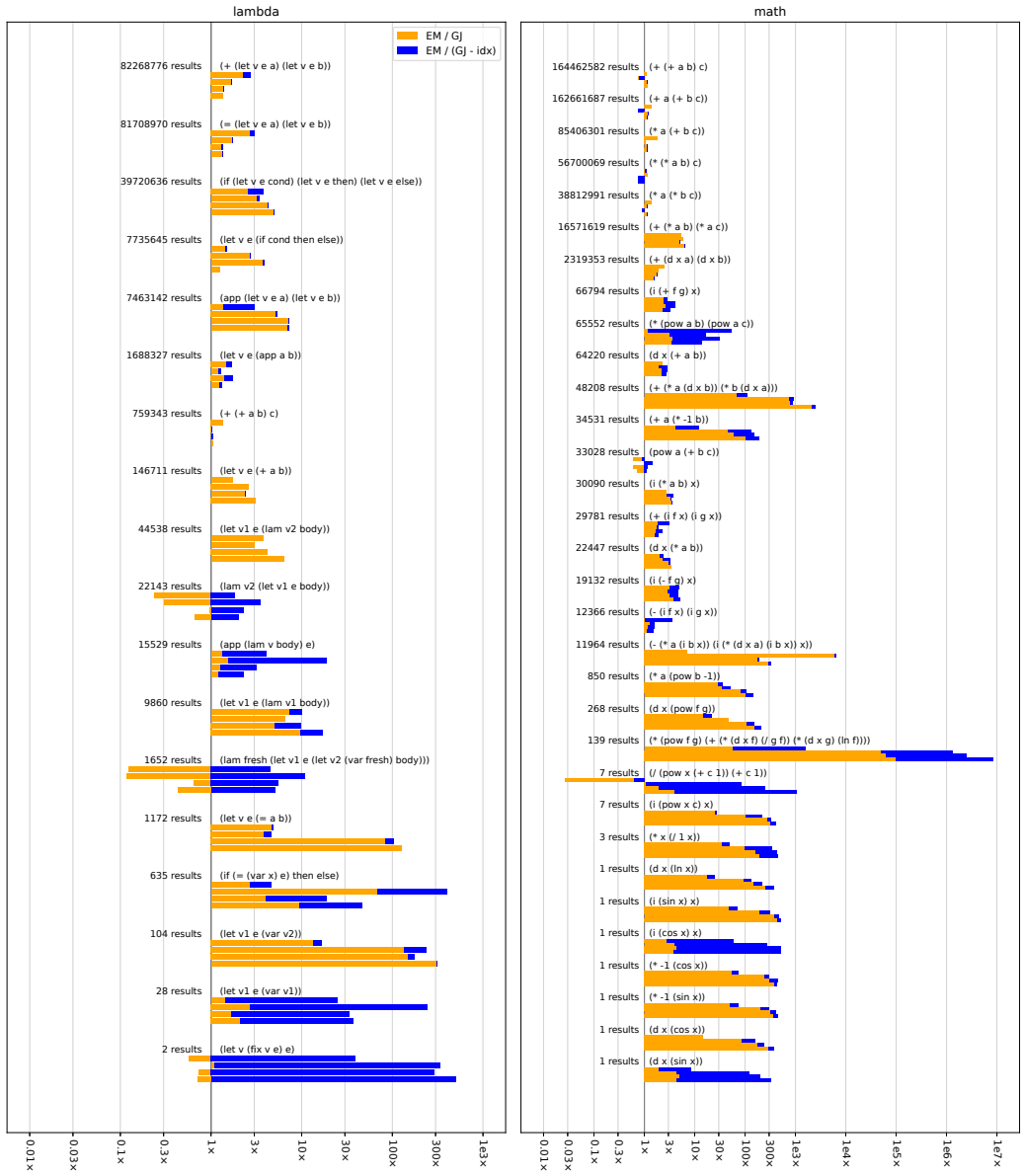


Fig. 9. Relational e-matching can be up to 6 orders of magnitude faster than traditional e-matching on complex patterns. Speedup tends to be greater when the output size is smaller. Bars to the right of the "1×" line indicate that relational e-matching is faster. The plots show two benchmark suites, `lambda` and `math`, taken from the egg test suite. Each group of bars shows the benchmarking 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 our relational e-matching approach 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.

wastes little time on unmatched terms, and using relational e-matching contributes little or no speedup. Large and complex (non-linear) patterns require careful query planning to be processed efficiently. For example, the pattern experiencing the largest speedup in Figure 9 is 4 e-nodes deep with 4 occurrences of the variable f. Relational e-matching using generic join can devise a variable ordering to put 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.

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 Section 6.4 we discuss potential remedies to alleviate this overhead.

Table 1 shows summary statistics across all patterns for each benchmark configuration. Notably, GJ is faster across patterns in every benchmarking configuration (the "Total" column). Much of the total benchmarking run time is dominated by simple, linear patterns (e.g. `(+ (+ a b) c)`) that return many results. Terms matching such patterns come to dominate the e-graph over time, due to the explosive expansion of associativity and commutativity. As a result, the total speedup does not necessarily increase as the e-graph grows, whereas the best speedup as well as different average statistics steadily increase.

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

6 DISCUSSION

The relational model for e-matching is not only simple and fast, but it opens the door to a wide range of future work to further improve e-graphs and e-matching.

6.1 Relational Representation of Code

The representation of e-nodes as relations and patterns as relational queries is simple and natural. It may even feel obvious to someone familiar with Prolog-style programming, where a function with arity k is often written as a relation with arity $k+1$, with 1 extra argument for the output. This relational representation can also be found in the literature on congruence closure. For example, Rümmer [2012] uses the same encoding to simulate congruence closure procedures with a hyper-resolution calculus. Unlike our focus on the performance of e-matching, their goal is to improve the completeness of quantified reasoning in SMT solvers. Research on large-scale code search and analysis [Gle [n.d.]; Antoniadis et al. 2017; Avgustinov et al. 2016; Urma and Mycroft 2013] has also explored storing program repositories in a database. Programmers may issue queries against the database to find code that match certain patterns, which may be examples of API usage or "anti-patterns" that pose security risks. Certain complex queries can even express sophisticated analyses like the points-to analysis [Antoniadis et al. 2017; Avgustinov et al. 2016]. These code search and analysis engines need to balance the need for expressiveness, efficiency, and ease of use.

While the idea to represent code as relations is not new, we are the first, to the best of our knowledge, to leverage relational join algorithms to speed up e-matching. We designed optimizations specialized for e-matching, and also derived the first non-trivial complexity bound for e-matching from a careful analysis of the generic join algorithm.

6.2 Pushdown Optimization

An e-matching pattern may have additional filtering condition associated with it. For example, a rewrite rule with left hand side `(* (/ x y) y)` may additionally require that $y \neq 0$. When the variables involved in conditions all occur in a single relation (e.g., R_* and $R_/$), this relation can be

effectively filtered even before being joined (e.g., using predicate $\text{find}(\sigma(y)) \neq \text{find}(\text{lookup}(0))$), which could immediately prune a large search space.

We call this *pushdown optimization*, which can be considered as e-graph's version of the relational query optimization that always pushes the filter operations down to the bottom of the join tree. Note that the ability to do pushdown optimization stems from relational e-matching's ability to consider the constraints in any order; backtracking e-matching could not support this technique. We currently do not implement this, because it requires breaking changes to egg's interface.

Conditions that involve multiple variables can be "pushed down" as well. In generic join, the filter can occur immediately after the variables appear in the variable ordering. Thus, an implementation that supports these conditional filters should take this into account when determining variable ordering.

6.3 Join Algorithms

Research in databases has proposed a myriad of different join algorithms. For example, state-of-the-art database systems implement two-way joins like hash join and merge-sort join. They have a longer history than generic join, and benefit from various constant factor optimizations. Extensive research has focused on generating highly efficient query plans using two-way joins. On the other hand, Yannakakis' algorithm [Papadimitriou and Yannakakis 1999] is proven to be optimal on a class of queries called *full ayclic queries*, running in time linear to the total size of the input and the output. All linear patterns correspond to acyclic queries, but some nonlinear patterns correspond to cyclic ones. Recent research [Freitag et al. 2020; Mhedhbi and Salihoglu 2019] has also experimented with combining traditional join algorithms with generic join, achieving good performance. In this paper we choose generic join for its simplicity, and future work may consider other join algorithms for relational e-matching.

6.4 Incremental Processing

We have focused on improving the core e-matching algorithm in this paper, yet prior work has successfully sped up e-matching by making it incremental [de Moura and Bjørner 2007]. When the changes to the e-graph are small and the results of e-matching patterns are frequently queried, maintaining the already-discovered matches becomes crucial for efficiency. From our relational perspective, incremental e-matching is captured precisely by the classic problem of incremental view maintenance (IVM) [Ceri and Widom 1991; Salem et al. 2000; Zhuge et al. 1995] in databases. IVM aims to efficiently update an already-computed query result upon changes to the database, without recomputing the query from scratch. Future research can follow our approach but implement e-graphs directly on top of a relational database engine, inheriting the incrementality from the underlying system. For example, a datalog engine provides semi-naive evaluation.

There is opportunity to improve relational e-matching even without a full-fledged IVM solution. For example, we have shown in Figure 9 that index building can take up a significant portion of the run time. Our index implementation is based on a hash trie, which is simple but difficult to update efficiently. We are experimenting with an alternative index design based on sort tries, in the hope that it can make updates as simple as inserting into a sorted array.

6.5 Building on Existing Database Systems

Given our reduction from e-matching to conjunctive query answering, one may wonder if other e-graph operations could be reduced to relational operations so that a fully functioning e-graph engine can be implemented purely on top of an off-the-shelf database system. There are many benefits to it. For example, we could enjoy an industrial-strength query optimization and execution engine for free (although most industrial databases do not use worst-case optimal join algorithms),

and eliminates the cost of transforming an e-graph to a relational database. Moreover, this approach would enjoy any properties of the host database system, including persistence, incremental maintenance, concurrency, and fault-tolerance.

As a proof of concept, we implemented a prototype e-graph implementation on top of SQLite, an embedded relational database system, with 160 lines of Racket code. E-graph operations like insertion and merging are translated into high-level SQL queries and executed using SQLite. This naïve prototype is not competitive with highly optimized implementations like egg, especially given our relational e-matching approach. However, with appropriate indices and query plan, it could have similar asymptotic performance. Specialized data structures to represent equivalence relations [Nappa et al. 2019] could also help performance. Therefore, not only e-matching but also other e-graph operations can be expressed as relational queries, which hints at the possibility of developing real-world e-graph engines on top of existing relational database systems.

7 CONCLUSION

In this paper, we present relational e-matching, a novel e-matching algorithm that is conceptually simpler, asymptotically faster, and worst-case optimal. We reduce e-matching to conjunctive queries answering, a well-studied problem in databases research. This relational presentation provides a unified way to exploit in query planning not only structural constraints, but also equality constraints, which are constraints that traditional e-matching algorithms cannot effectively leverage. We implement relational e-matching with the worst-case optimal generic join algorithm, using which we derive the first data complexity for e-matching. We integrate our implementation in the state-of-the-art equality saturation engine egg, and show relational e-matching to be flexible (readily supports multi-patterns) and efficient (achieves orders of magnitude speedup).

ACKNOWLEDGMENTS

This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA. This material is based upon work supported by the National Science Foundation under Grant No. 1749570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[n.d.]. Glean System for collecting, deriving and querying facts about source code. https://glean.software. Accessed: 2021-10-12.

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.

Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. Porting Doop to Soufflé: A Tale of Inter-Engine Portability for Datalog-Based Analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis* (Barcelona, Spain) *(SOAP 2017)*. Association for Computing Machinery, New York, NY, USA, 25–30. https://doi.org/10.1145/3088515.3088522

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

Pavel Avgustinov, Oege De Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

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

Stefano Ceri and Jennifer Widom. 1991. Deriving Production Rules for Incremental View Maintenance. In *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, Guy M. Lohman, Amílcar Sernadas, and Rafael Camps (Eds.). Morgan Kaufmann, 577–589. http://www.vldb.org/conf/1991/P577.PDF

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

Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *Proc. VLDB Endow.* 13, 12 (July 2020), 1891–1904. https://doi.org/10.14778/3407790.3407797

Dexter Kozen. 1977. Complexity of Finitely Presented Algebras. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing* (Boulder, Colorado, USA) *(STOC '77)*. Association for Computing Machinery, New York, NY, USA, 164–177. https://doi.org/10.1145/800105.803406

Amine Mhedhbi and Semih Salihoglu. 2019. *Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins*. arXiv:1903.02076v2 [cs.DB]

Michał Moskal, Jakub Łopuszański, and Joseph R. Kiniry. 2008. E-Matching for Fun and Profit. *Electron. Notes Theor. Comput. Sci.* 198, 2 (May 2008), 19–35. https://doi.org/10.1016/j.entcs.2008.04.078

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

Patrick Nappa, David Zhao, Pavle Subotić, and Bernhard Scholz. 2019. Fast Parallel Equivalence Relations in a Datalog Compiler. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 82–96.

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

Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Rec.* 42, 4 (Feb. 2014), 5–16. https://doi.org/10.1145/2590989.2590991

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

Philipp Rümmer. 2012. E-matching with free variables. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 359–374.

Kenneth Salem, Kevin S. Beyer, Roberta Cochrane, and Bruce G. Lindsay. 2000. How To Roll a Join: Asynchronous Incremental View Maintenance. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein (Eds.). ACM, 129–140. https://doi.org/10.1145/342009.335393

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

Raoul-Gabriel Urma and Alan Mycroft. 2013. Expressive and Scalable Source Code Queries with Graph Databases. (2013).

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]

Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. 1995. View Maintenance in a Warehousing Environment. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, Michael J. Carey and Donovan A. Schneider (Eds.). ACM Press, 316–327. https://doi.org/10.1145/223784.223848