Database Theory in Action: Search-Based Program Optimization

Yihong Zhang
University of Washington, Seattle, WA, USA

Dan Suciu
University of Washington, Seattle, WA, USA

Yisu Remy Wang
University of California, Los Angeles, CA, USA

Max Willsey
University of California, Berkeley, CA, USA

Abstract

Recent work in programming languages developed an approach to term rewritings based on *equality saturation (EqSat)*, which, instead of applying destructively the rewrite rules, maintains all equivalent expressions in a structure called an E-graph. This paper describes two surprising connections between EqSat and databases, going both ways. On one hand equality saturation can be viewed as a query evaluation problem, with great benefits. On the other hand, most sophisticated SQL query optimizers are based on the Volcano/Cascades framework which, we explain, is a variant of EqSat.

2012 ACM Subject Classification Theory of computation → Equational logic and rewriting; Theory of computation → Database query processing and optimization (theory)

Keywords and phrases Query optimization, program optimization, Cascades framework, equality saturation, Datalog

Digital Object Identifier 10.4230/LIPIcs.ICDT.2025.34

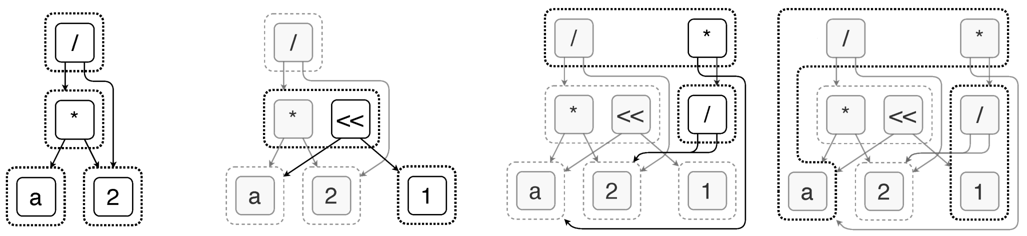
Funding This work is supported by NSF IIS 2314527 and NSF SHF 2312195, and is conducted while some of the authors participated in the Simons Program on Logic and Algorithms in Databases and AI.

1 Program Optimization with E-graphs & Equality Saturation

Challenges with Rewriting. Program optimization in the compilers literature is typically viewed from the perspective of term rewriting. Given an input term t and a set of rewrite rules $\mathcal{R}$ of the form $expr_1 \to expr_2$, a system repeatedly chooses a rule from $\mathcal{R}$ to apply to the term t, hopefully moving it closer and closer to a "better" program. When $\mathcal{R}$ is convergent (i.e. terminating and confluent) this works great! No matter the order of rewrite application, this process finishes (termination) and arrives at the *same term* (confluence).

What happens when a term rewriting system does not enjoy properties like confluence? Consider trying to optimize the program $(a \times 2)/2$ to a. One promising transformation in general is $a \times 2 \to a \ll 1$: replacing a multiplication with a cheaper bitshift. However, in this particular case, applying that locally-good transformation would obscure the globally optimal solution. In the compilers literature, this is called the phase-ordering problem: different orderings of analyses and transformations produce different results.

E-graphs and Equality Saturation. *Equality saturation* (EqSat) is technique that promises to mitigate these above challenges with conventional term rewriting. The key idea is that rewriting no longer destroys the old term, instead it monotonically adds an equality between the new and old term, so applying the "incorrect" rewrite at any given time is no longer



(a) E-graph that represents $(a \times 2)/2$. **(b)** Rewrite $x \times 2 \rightarrow x \ll 1$. **(c)** Rewrite $(x \times y)/z \rightarrow x \times (y/z)$. **(d)** Rewrite $x/x \rightarrow 1$ and $1 \times x \rightarrow x$.

Figure 1 Applying rewrites over an example e-graph (figures from [25]). A solid box denotes an e-node, and a dotted box denotes an e-class. E-nodes consist of a function symbol and children e-classes, and e-classes contain a set of equivalent e-nodes. In traditional rewriting, the step in (b) would remove the multiplication node, preventing the application of the rewrite in (c). Since rewriting in the e-graph is monotonic, one will not prevent the other from being applied.

as punishing. To represent this large space of programs, EqSat borrows the e-graph data structure [14] from the automated theorem proving community, where it sits in the core of SMT solvers like Z3 [4] and CVC [1] to reason about the equality of uninterpreted functions. The original EqSat work [20] demonstrated the e-graph's promise as the basis for a search-based program optimization technique. Recently, EqSat has seen a flurry of new work advancing the technique [25, 28, 9, 29], developing new toolchains like `egg`, and applying it to new areas [17, 13, 27, 23, 22, 12].

Figure 1 shows an example e-graph and how EqSat applies rewrites to grow the space of programs that it represents. Using the definition from [25], an e-graph is a set of e-classes (equivalence classes) that contain equivalent e-nodes. An e-node is a function application that points to e-classes (not e-nodes) as children; this captures a notion of *congruence*: if $a = b$ then $f(a) = f(b)$. Rewrites over the e-graph are somewhat of a misnomer, they only add new e-nodes and e-classes to the e-graph. Figure 1b shows a potentially "problematic" rewrite that in the traditional setting would hinder the optimization of $(a \times 2)/2$ to a.

The typical equality saturation workflow is as follows:

1. (Figure 1a) Insert the input program into the e-graph; say the input is stored in e-class c.
2. (Figure 1b-d) Apply rewrites to the e-graph until a fixed point or a timeout is reached; note that a fixed point may not exist.
3. Extract from c the cheapest program according to some cost function.

E-matching. Applying rewrites is typically the most expensive part of the EqSat workflow. This complexity comes from the fact that the e-graph represents many, many programs (in fact, potentially infinite programs in e-graphs with cycles) that could match the left-hand side of a rewrite. The *e-matching* (pattern matching modulo equality) problem is the task of, given a pattern p and an e-class c, find all substitutions σ such that $p\sigma \in c$ where σ maps variables in p to e-classes in the e-graph.

E-class Analyses. Notice the rule $x/x \rightarrow 1$ in Figure 1d is not always sound. If x is instantiated with a term that may evaluate to 0, the left-hand side x/x is undefined and should not be rewritten to 1. E-class analyses provides a mechanism for incorporating semantic information during EqSat: An e-class analysis annotates each e-class with an element from a join semi-lattice. As e-classes are merged through rewrites, semantic information about e-classes is propagated and aggregated by taking the join (least upper bound) of individual

```
(sort Node)
(function mk (i64) Node)
(relation edge (Node Node))
(relation path (Node Node))

(rule ((edge x y))
      ((path x y)))
(rule ((path x y) (edge y z))
      ((path x z)))

(edge (mk 1) (mk 2))
(edge (mk 2) (mk 3))
(edge (mk 5) (mk 6))

(union (mk 3) (mk 5))
(run)
(check (edge (mk 3) (mk 6)))
(check (path (mk 1) (mk 6)))
```

(a) A path reachability program using unification to combine nodes.

```
(datatype Math
  (Num i64)
  (Var String)
  (Add Math Math)
  (Mul Math Math))

;; expr1 = 2 * (x + 3)
(define expr1 (Mul (Num 2) (Add (Var "x") (Num 3))))
;; expr2 = 6 + 2 * x
(define expr2 (Add (Num 6) (Mul (Num 2) (Var "x"))))

(rewrite (Add a b)         (Add b a))
(rewrite (Mul a (Add b c)) (Add (Mul a b) (Mul a c)))
(rewrite (Add (Num a) (Num b)) (Num (+ a b)))
(rewrite (Mul (Num a) (Num b)) (Num (* a b)))

(run)
(check (= expr1 expr2))
```

(b) Datalog + functions + unification allows `egglog` to implement EqSat-style verification and optimization.

Figure 2 Datalog + functions allows `egglog` to implement traditional Datalog programs and Datalog over lattices. The unification feature adds new capabilities to Datalog as in (a), and also enables EqSat-style verification and optimization (b).

e-classes' annotated values. For example, an e-classes analysis can track the possible ranges with interval arithmetic. If two e-classes with possible ranges $[-2, 5]$ and $[1, 7]$ are merged (discovered to be equivalent), it can be concluded that the merged e-class can only have possible range $[1, 5]$. This semantic information can then be used to safeguard rules like $x/x \to 0$ with appropriate side conditions (e.g., range of x should not contain 0). There are, however, some limitations with e-class analyses: it only allows one e-class analysis per E-graph, and the analysis data are only propagated bottom up, but not top down, making certain analyses like type analysis infeasible.

2 Database Techniques in EqSat

Relational E-matching. Both e-matching and database queries aim to find homomorphisms: the former over the e-graph data structure and the latter over a database instance. As the e-matching problem grows more complex – patterns with repeated variable use, searching multiple correlated patterns simultaneously, the need for incremental search, etc. – the techniques employed become more complex as well [3]. Instead of building on existing e-matching techniques, recent work [29] encoded e-matching as conjunctive queries over a relational database instance, then used a Worst Case Optimal Join (WCOJ) [15] to perform e-matching. The work followed the standard flattening of terms into tables. For example, the term $y = f(f(a))$ can be encoded in a binary relation, call it $R_f(\mathit{child}, \mathit{id})$, containing the tuples (a, x), (x, y), where x, y are the identifiers of the nodes $f(a)$ and $f(f(a))$ respectively. To represent $z = g(f(f(a)), b)$, add a second relation $R_g(\mathit{child}_1, \mathit{child}_2, \mathit{id})$, containing the tuple (y, b, z). When two e-nodes are in the same e-class, they simply share the same *id*. This encoding naturally supports complex patterns, since a query optimizer can exploit the query structure for asymptotic speedup over the traditional e-matching algorithm [29].

egglog: Connecting EqSat to Datalog. egglog is a recent system that unifies EqSat and Datalog. There is a simple intuition here: both EqSat and Datalog perform non-destructive, fixpoint reasoning over some (term or relational) databases. The resulting system is beneficial to both EqSat and Datalog. For EqSat, this brings more expressive program analyses over e-graphs, powerful query optimization, and incrementalization via semi-naïve evaluation. Compared to Datalog, the union-find-based equational reasoning from EqSat makes tasks like unification-based pointer analysis [18] require asympototically less space. Figure 2 shows some egglog code in both a Datalog and EqSat style.

At the core of egglog is the concept of functional dependency (FD). egglog is based on a "functional" database design – functions are just relations with functional dependencies. Let f be a binary function table; under the hood it is really just a ternary relation $R_f(c_1, c_2, c)$ with FD $(c_1, c_2) \to c$. What if the FDs are violated? For example, suppose $f(a, b) = c$ and later we learned also that $f(a, b) = c'$. In egglog there are only two cases. If c and c' are e-classes, then we can unify c and c' – from now on, c and c' should be viewed as the same e-class and will not be distinguishable. If c and c' are lattice values, then we can take the join of c and c' and claim $f(a, b) = c \vee c'$, since the knowledge $c \vee c'$ subsumes both c and c'. This functional dependency repair is implemented via a congruence-closure-style algorithm [5].

3 EqSat Techniques for Databases

The database perspective has already provided immense benefits to the performance and flexibility of EqSat; here we pose some connections in the opposite direction.

Volcano/Cascades Query Optimization. The dominant query optimization architecture in production database systems is derived from the Volcano [7] and Cascades [8] systems. Yongwen Xu's Master thesis [26] describes an enhanced version of Cascades, and the optd project[1] started at Carnegie Mellon University provides a modern implementation. These query optimizers can be characterized as rewrite-based, memoized search for all queries equivalent to the input, selecting the cheapest in a top-down manner. The rewrite-based aspect makes the system extensible; if a new logical or physical operator is introduced, one can just add new rewrites to enable optimization of queries with the new construct. The memoization of the search is critical for performance, as many transformations of (sub)queries will lead to programs that have already been explored. These characteristics are all present in EqSat: the memo table in Cascades corresponds to the e-graph data structure, which employs the classic union-find data structure [19] to efficiently store and maintain the equivalence classes. Unlike Cascades, EqSat computes the congruence closure [5] of the equivalence relation to further compress the e-graph. These similarities have already spawned interest in using EqSat for portions of query optimization [24] or implementing the entire process (e.g., in CubeSQL and RisingLightDB). The Volcano/Cascades system also critically employs top-down search to enable a branch-and-bound technique, while EqSat/Datalog are bottom-up. Yet both support top-down search, as we explain next.

Magic Set and Demand. Magic set optimization [11], demand transformation [21], and more recent works around ADTs and functional programming in Datalog [6, 16] all aim to provide some degree of top-down control to Datalog. Previous works employing Datalog for query optimization [10, 2] have also found it necessary to implement top-down search

[1] The source code of optd is available at `https://github.com/cmu-db/optd?tab=readme-ov-file`.

in a manual manner. EqSat and `egglog` offer an additional perspective on this problem. EqSat is bottom-up, but it provides an additional tool that can be used to "place demand" on certain tuples/terms. In EqSat, the term $f(a)$ can be considered on its own (in its own e-class) without knowledge of its output. This is similar to labeled nulls; the e-class can always be unified with the result at a later time.

References

1 Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag. `doi:10.1007/978-3-642-22110-1_14`.

2 Martin E. Bidlingmaier. An evaluation algorithm for datalog with equality, 2023. `doi:10.48550/arXiv.2302.05792`.

3 Leonardo de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In Frank Pfenning, editor, *Automated Deduction – CADE-21*, pages 183–198, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

4 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1792734.1792766`.

5 Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, October 1980. `doi:10.1145/322217.322228`.

6 Thomas Gilray, Arash Sahebolamri, Sidharth Kumar, and Kristopher Micinski. Higher-order, data-parallel structured deduction, 2022. `doi:10.48550/arXiv.2211.11573`.

7 G. Graefe and W.J. McKenna. The volcano optimizer generator: extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*, pages 209–218, 1993. `doi:10.1109/ICDE.1993.344061`.

8 Goetz Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995. URL: `http://sites.computer.org/debull/95SEP-CD.pdf`.

9 Thomas Kundefinedhler, Andrés Goens, Siddharth Bhat, Tobias Grosser, Phil Trinder, and Michel Steuwer. Guided equality saturation. *Proc. ACM Program. Lang.*, 8(POPL), January 2024. `doi:10.1145/3632900`.

10 Mengmeng Liu, Zachary G Ives, and Boon Thau Loo. Enabling incremental query re-optimization. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1705–1720, 2016. `doi:10.1145/2882903.2915212`.

11 I. S. Mumick, S. J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is relevant. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 247–258, New York, NY, USA, 1990. Association for Computing Machinery. `doi:10.1145/93597.98734`.

12 Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 31–44, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3385412.3386012`.

13 Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. Rewrite rule inference using equality saturation. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. `doi:10.1145/3485496`.

14 Charles Gregory Nelson. *Techniques for Program Verification.* PhD thesis, Stanford University, Stanford, CA, USA, 1980. AAI8011683.

15 Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, 2013. doi:10.1145/2590989.2590991.

16 André Pacak and Sebastian Erdweg. Functional programming with datalog. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

17 Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. *SIGPLAN Not.*, 50(6):1–11, June 2015. doi:10.1145/2813885.2737959.

18 Bjarne Steensgaard. Points-to analysis in almost linear time. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 32–41. ACM Press, 1996. doi:10.1145/237721.237727.

19 Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975. doi:10.1145/321879.321884.

20 Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 264–276, New York, NY, USA, 2009. ACM. doi:10.1145/1480881.1480915.

21 K. Tuncay Tekle and Yanhong A. Liu. Precise complexity analysis for efficient datalog queries. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pages 35–44, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1836089.1836094.

22 Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. *Vectorization for Digital Signal Processors via Equality Saturation*, pages 874–886. Association for Computing Machinery, New York, NY, USA, 2021. doi:10.1145/3445814.3446707.

23 Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. SPORES: Sum-product optimization via relational equality saturation for large scale linear algebra. *Proceedings of the VLDB Endowment*, 2020.

24 Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q Ngo, Reinhard Pichler, and Dan Suciu. Optimizing recursive queries with program synthesis. *arXiv preprint arXiv:2202.10390*, 2022. arXiv:2202.10390.

25 Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi:10.1145/3434304.

26 Yongwen Xu. Efficiency in the columbia database query optimizer. Master's thesis, Citeseer, 1998.

27 Yichen Yang, Phitchaya Mangpo Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. In *Proceedings of Machine Learning and Systems*, 2021. arXiv:2101.01332.

28 Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. Better together: Unifying datalog and equality saturation. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi:10.1145/3591239.

29 Yihong Zhang, Yisu Remy Wang, Max Willsey, and Zachary Tatlock. Relational e-matching. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. doi:10.1145/3498696.