Incremental Equality Saturation

Stop Throwing Away Your E-Graphs!

RUPANSHU SOI, Stanford University, USA
BENJAMIN DRISCOLL, Stanford University, USA
KE WANG, Visa Research, USA
ALEX AIKEN, Stanford University, USA

We introduce a new equality saturation algorithm that efficiently optimizes terms available at different points in time using a single (incremental) e-graph, thereby extending the structure sharing benefits of standard equality saturation temporally.

1 INTRODUCTION

A standard equality saturation (EqSat) pipeline (Figure 1) has the following steps: (1) apply domain-specific pre-processing to the input term; (2) insert the term into an empty e-graph; (3) run EqSat; (4) extract an optimized term; and, (5) apply domain-specific post-processing. This pipeline is run afresh for every term, and no state is maintained. As a side effect, EqSat produces a database of equalities and, while the equalities it finds are a function of its input, many of them can reused for other terms. Why, then, are we always throwing the equalities away?

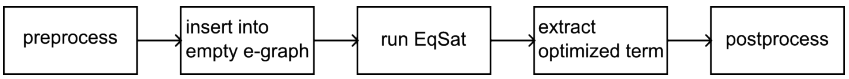

Fig. 1. Standard EqSat pipeline.

We are not the first to ask this question. Willsey et al. [4] made essentially the same observation in their paper introducing the egg library, showing that EqSat time could be significantly reduced by using a single e-graph to exploit shared structure between different terms. Their technique, *batched EqSat*, works by jointly processing all terms of interest. Steps 1 and 2 of the pipeline are run for each term, then all terms are inserted into the same, empty e-graph, and, finally, steps 4 and 5 and run for each term.

However, many applications cannot leverage batched EqSat because it requires that all terms be available simultaneously. This limitation precludes leveraging batched EqSat for interactive applications as, in any such application, there are delays as the user provides successive terms. Projects that fall into this category include Herbie [2] and Cranelift [1].

Moreover, there is no straightforward way to salvage batched EqSat for these applications. The naive approach would be to use a standard pipeline but persist the e-graph instead of beginning with an empty one each time. One might expect this approach to provide some of the benefits of batched EqSat, but that is not what happens. Since the e-graph has no notion of a current term, every time EqSat is run it applies rewrites to the entire e-graph. But, most of the e-graph might be unreachable from the current term, in which case applying rewrites to it is useless. This problem is only exacerbated as more terms are inserted into the e-graph, making this approach infeasibly slow.

Our algorithm, *incremental EqSat*, addresses this problem. *Incremental e-graphs* have a notion of a current term, implemented by assigning a *version* (a natural number) to each e-class. Versions are

Authors' addresses: Rupanshu Soi, rsoi@cs.stanford.edu, Stanford University, USA; Benjamin Driscoll, bdrisc@cs.stanford.edu, Stanford University, USA; Ke Wang, kewang@visa.com, Visa Research, USA; Alex Aiken, aaiken@stanford.edu, Stanford University, USA.

leveraged by incremental EqSat to apply rewrites to only the part of the e-graph that is relevant to the current term.

2 PRELIMINARIES

Before describing incremental EqSat in detail, we discuss our setting and terminology. We assume that the reader is familiar with standard e-graphs.

We are given a sequence of input terms t_i. Each term must be optimized and returned to the user before accepting the next. The *ruleset* R used to rewrite the terms is constant; R is a list of *rules* (rewrites) of the form `l -> r`, where `l` and `r` are S-expressions consisting of *symbols* and *pattern variables*.[1] E.g., the rule for commutativity of addition is `(+ ?a ?b) -> (+ ?b ?a)`, where "+" is a symbol and "`?a`" and "`?b`" are pattern variables. The symbol or pattern variable at the head of the S-expression of the left-hand side (LHS) of the rule is called the *root*. A match for a rule on an e-graph consists of a mapping from symbols to e-nodes and pattern variables to e-classes.[2] The mapping of the root gives the e-class into which the e-node corresponding to the right-hand side (RHS) will be inserted.

EqSat with R might be terminating or non-terminating. Virtually all rulesets used in practice are non-terminating because it is quite easy to introduce non-termination via the interaction of seemingly innocuous rules (and undecidable, in general, whether you have done so [3]), which necessitates forcing termination in some fashion, generally by using a *budget*. A budget can be a maximum number of e-nodes to add, an iteration count, or a time limit.

A cost function C is used for extracting the final term. If C can be computed using only on an e-node's function symbol and the costs of its children, then C is *local*. Local cost functions have the nice property that they can be implemented as an analysis (in the `egg` [4] sense), obviating the need for a separate extraction step. We will assume that C is local and discuss the difficulties particular to incremental EqSat with a non-local cost functions later.

3 RUNNING EXAMPLE

Consider a small algebraic simplifier. Let R consist of common algebraic identities you might find in a high-school textbook. For this example, the important rules are: `?x -> (+ ?x 0)`, `?x -> (* ?x 1)`, `(pow ?x 0) -> 1` and `(d ?x ?x) -> 1`, where `(d x y)` denotes $\frac{\mathrm{d}}{\mathrm{d}x}\, y$. Say the cost function is just the size of the term (smaller terms preferred). We will examine the behavior of standard and incremental EqSat on the same two input terms: `(pow x 0)` and `(d x (* x 1))`.

Figure 2 depicts the two e-graphs obtained after optimizing these terms via standard EqSat. Note the duplicated e-class, which represents redundant work performed by standard EqSat.

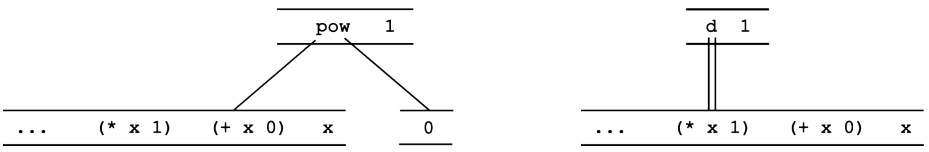

Fig. 2. Standard e-graphs obtained after optimizing `(pow x 0)` and `(d x (* x 1))`. Some e-nodes are duplicated and/or flattened to simplify the diagrams.

[1] In egg, rules can additionally do some computation on the RHS. That is also possible in incremental EqSat but unimportant for our discussion so we omit it here.

[2] In egg, the mapping for symbols is not provided as part of the match, but it nevertheless exists.

4 INCREMENTAL EQSAT

Incremental e-graphs are standard e-graphs augmented with:

(1) A *version* for each e-class and the current version, V, global to the e-graph.
(2) A special *top-k* analysis that integrates with rule matching.

V is initialized to 0 and incremented before each t_i is inserted into the e-graph. On insertion of a term, new e-classes are assigned version V. Old e-classes do not change version with one exception. If an e-node in the current term, at the time of its insertion, already exists in an e-class with just that e-node, then we update that e-class's version to V. We explain this design choice in Section 5.2.1. We also note that new e-classes can have their version downgraded to an older one—see Section 5.

The top-k analysis maintains, for every e-class, the set of e-nodes contained within it having the lowest cost, second-lowest cost, and so on up to the k^{th}-lowest cost. Note that the cost of an e-node is well-defined for a local cost function, but non-local cost functions provide a cost only for entire terms or e-graphs. In practice, we find $k = 2$ is a reasonable value. If an e-node is in the top-k set for its e-class, we say that e-node *is top-k*.

Rule matching works differently in incremental EqSat. Every match must satisfy two additional conditions:

(1) The root must map to either an e-node in an e-class with version V or an e-class with version V (dependent on whether the root is a symbol or pattern variable). Note that non-root positions are allowed to map to old e-nodes or e-classes. Focusing on saturating new e-classes while leveraging what is already known about old e-classes effectively prunes large parts of the e-graph when searching for a match.
(2) If a symbol is mapped to an old e-node e, then e must be top-k.

4.1 Running Example, Incrementally

Let us see how incremental EqSat optimizes the two terms from the previous section. We begin by adding `(pow x 0)` to a fresh incremental e-graph. V is initialized to 0, so all e-classes get version 0. Since the e-graph was previously empty, when we run incremental EqSat we will find the same equalities as standard Eqsat.

Now, we increment V to 1 and insert `(d x (* x 1))` into the same incremental e-graph. The e-class of `d` is assigned version 1 because it is new, the rest will remain at version 0. Note that, without doing any rule application, we have obtained the equality `(d x (* x 1)) = (d x x)` by our previously derived equalities and the structure sharing of e-graphs. Once we begin rule application, there is only the derivative rule to apply, resulting in the equality `(d x (* x 1)) = 1`. Adding this equality decrements the version of `d`'s e-class since `1` is an old e-node (see Section 5).

Figure 3 shows the final state of the incremental e-graph. There are no new e-classes left so incremental EqSat has saturated. Overall, it is clear how incremental EqSat benefits from reusing previously found equalities, and avoids spending the budget on e-classes which are not relevant to the current term.

5 DESIGNING FOR INCREMENTALITY

The rest of the design of incremental EqSat is motivated by the following two questions:

(1) *Soundness.* Is incremental EqSat sound?
(2) *Performance.* How can incremental EqSat be made performant?

5.1 Soundness

At each step, the rule matches valid under incremental EqSat are a subset of the matches valid under standard EqSat on the same e-graph, and matches are applied identically. Taken together

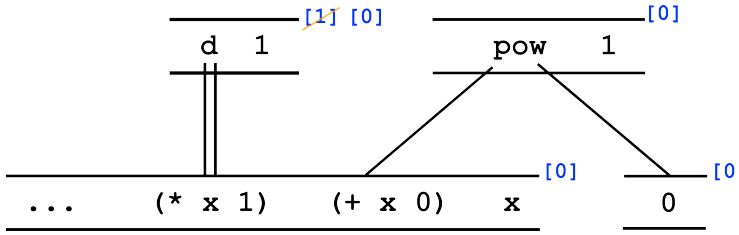

Fig. 3. Incremental e-graph obtained after optimizing the same two terms. Versions are shown in blue. The orange line depicts a version downgrade.

with the fact that standard EqSat can be initialized with any e-graph of correct equalities (though typically run with the empty e-graph), we conclude that incremental EqSat is sound.

It is, however, interesting to consider the relationship between the set of equalities found by incremental and standard EqSat. In particular, given the same term and ruleset, if an equality is reachable via standard EqSat, then is it also reachable via incremental EqSat starting from any incremental e-graph? In general, we believe the answer is no. Incrementality demands some policy that, given a term t, chooses sub-terms of t to exclude from exploration. In our algorithm, this choice is made by assigning a version to the e-class containing each sub-term of t and exploring only e-classes that are assigned the current global version. Assuming that, for an arbitrary ruleset, it is not possible to compute whether an e-class contains all e-nodes found by standard EqSat within some budget more efficiently than by running standard EqSat, a reasonable policy cannot know whether its decision to not explore a sub-term renders unreachable an equality that is reachable by standard EqSat.

We believe a useful way to think about standard and incremental EqSat is that they search the same underlying space, but in different ways. They have different starting points (e-graphs) and different ways of taking steps (applying rules). Consequently, there exist cases where standard EqSat finds a better solution than incremental EqSat. However, the important question is whether incremental EqSat is Pareto optimal in practice. We give evidence in the affirmative in Section 6.

5.2 Performance

5.2.1 Policy. Perhaps the most important consideration for incremental EqSat's performance is the choice of P, the policy by which sub-terms are enabled or disabled for exploration.

Our P is based on a *well-exploredness heuristic*. It says that, since the ruleset and budget are fixed, each old e-class is *well-explored* because it was at some point new, i.e., its version was equal to that round's V and we explored it until that round's budget was exhausted. By this logic, we need not explore any sub-term whose e-class is old. But, e-classes created just before a round's budget is exhausted are likely to have had little or no rule application performed on them. To detect under explored e-classes, we parameterize our heuristic by m and say that old e-classes with fewer than m e-nodes are not well-explored. In practice, we find $m = 2$ works well. This choice of m implies that old singleton e-classes are not considered well-explored and underlies the exception in the version update rule in Section 4.

5.2.2 Version merges. Another important design point is the rule for updating versions when two e-classes are merged. We give the resultant e-class the lower version, and do not modify the versions of their parents. Consequently, if a new e-class gets merged with an old one, the resultant e-class is considered old and subsequently will not be allowed to match as root. Intuitively, such a

merge "fast-forwards" the new e-class to well-exploredness because we now know its e-nodes to be equal to well-explored e-nodes.

5.2.3 Top-k and Runaway e-classes. The top-k restriction on rule matching turns out to be crucial for our approach to incremental EqSat.

The issue is best explained by example. Consider the algebraic simplifier, now augmented with associativity and commutativity (AC) rules. Consider a term `(f X)` where `X` is a large sum of sub-terms such that `X = Y` is derivable under our ruleset and `Y` is much better cost-wise than `X`. Say `X` is old and `f` is new.

Without top-k, the entire budget may be exhausted by matching on `(f X')` where `X'` is a permutation of `X` induced by the AC rules. But it is much more likely that the beneficial thing to do is to match on `Y` or something close to it. That is exactly what top-k enforces.

Basically, as the incremental e-graph gets larger, for many practical rulesets e-classes tend to runaway in the sense that the fraction of e-nodes that can lead to a cost minima decreases because of the presence of rules that lead to an exponential explosion of sub-terms of equivalent cost. If we do not provide any guidance regarding rule application, incremental EqSat starts squandering its entire budget matching on useless e-nodes.

The main limitation of incremental EqSat, then, is its reliance on the locality of the cost function, since that is a precondition for defining top-k. We leave solving the runaway e-class problem for non-local cost functions as an open problem.

5.2.4 Incremental Rebuilding. Incremental EqSat uses a more efficient rebuilding algorithm than standard EqSat (as implemented in egg). The standard algorithm canonicalizes the entire e-graph at every rebuild, but doing so is very slow and breaks incrementality because most of the incremental e-graph might be irrelevant to the current term. Instead, we continuously track the set of e-classes that require canonicalization as we are applying rules, and only canonicalize those e-classes during rebuilding. The details are omitted due to lack of space.

6 EVALUATION

We implement incremental EqSat as a fork of the egg [4] library. We evaluate it on our running example: an algebraic simplifier. The code for the simplifier is a fork of the code for egg's `math` example. It supports polynomials, exponentials, logarithms, trigonometric functions, derivatives and integrals. The cost function tries to get rid of derivatives and integrals while minimizing the size of the term. It also implements constant folding via a simple analysis. Since egg does not support multiple analyses, the combination of this analysis with top-k had to be manually constructed.

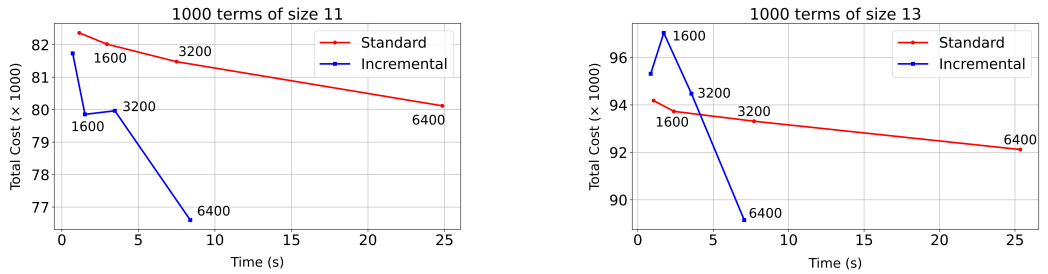

Fig. 4. Results for optimizing 1000 terms of size 11 and 13 via standard and incremental EqSat. Cost on the Y-axis is the sum of costs of extracted terms for all 1000 input terms. Lower and to the left is better. Some data points are labeled with their e-node limit.

Test inputs are randomly generated by a Python script. Each input is a function of x of a given size. Test inputs do not contain numeric constants. Each test case corresponds to 1000 test inputs of a fixed size (11 and 13 in the plots). We optimize inputs sequentially via both incremental and standard EqSat, and plot the total time against the the sum of costs of extracted terms.

Each data point corresponds to a run with a different e-node limit. In standard EqSat an e-node limit of N means that rule application will continue until the e-graph has at least N e-nodes. In incremental EqSat we adapt this to a requirement that the e-graph has at least N *new* e-nodes. Other adaptations are possible but we believe this is fair since, in most cases, it forces incremental EqSat to do at least as much work as standard EqSat.

There is no separate extraction step in incremental EqSat as top-k allows us to directly extract the best e-node from the root e-class. For a fair comparison we modified standard EqSat with a similar analysis that maintained the best e-node in each e-class. Of course, in standard EqSat this analysis does not constrain rule matching.

We see that incremental EqSat is Pareto optimal over standard EqSat for many different e-node limits (Figure 4). As we increase the size of the input terms, incremental EqSat requires a larger budget to achieve Pareto optimality as more exploration is required on larger terms to make the top-k sets useful.

The plot for terms of size 13 (Figure 4) contains a data point for incremental EqSat with unexpectedly high cost (at e-node limit 1600). In general, due to the top-k restriction on rule matching incremental EqSat can have variable performance at intermediate e-node limits, but we are currently unsure of the precise cause of the behavior seen at this data point.

1000×11	E-Node Limit	Time (s)	Total Cost	Incremental Better	Standard Better
Incremental	6400	8.3	76600	56/1000 terms	40/1000 terms
Standard	6400	24.8	80118		

1000×13	E-Node Limit	Time (s)	Total Cost	Incremental Better	Standard Better
Incremental	6400	7.7	89146	46/1000 terms	37/1000 terms
Standard	6400	26.3	92119		

Fig. 5. Results for optimizing 1000 terms of sizes 11 and 13. Incremental EqSat is Pareto optimal in both cases.

Figure 5 shows detailed numbers for both test cases in a configuration where incremental EqSat is Pareto optimal in terms of time and total cost. Note that Pareto optimality does not mean that incremental EqSat found a better solution for every term—the total cost is the sum of costs of 1000 optimized terms. As shown in the figure, standard EqSat finds a better solution than incremental EqSat in $\approx 4\%$ of cases and vice-versa. For the vast majority of terms, incremental and standard EqSat found solutions of equal cost.

REFERENCES

[1] Chris Fallin. [n. d.]. ægraphs: Acyclic E-graphs for Efficient Optimization in a Production Compiler. https://pldi23.sigplan.org/details/egraphs-2023-papers/2/-graphs-Acyclic-E-graphs-for-Efficient-Optimization-in-a-Production-Compiler. Accessed: 2025-04-15.

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

[3] Dan Suciu, Yisu Remy Wang, and Yihong Zhang. 2025. Semantic Foundations of Equality Saturation. In *28th International Conference on Database Theory (ICDT 2025) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 328)*, Sudeepa

Roy and Ahmet Kara (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 11:1–11:18. https://doi.org/10.4230/LIPIcs.ICDT.2025.11

[4] 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