Incrementalizing Production CodeQL Analyses

Tamás Szabó
GitHub
Germany

ABSTRACT

Instead of repeatedly re-analyzing from scratch, an incremental static analysis only analyzes a codebase once completely, and then it updates the previous results based on the code changes. While this sounds promising to achieve speed-ups, the reality is that sophisticated static analyses typically employ features that can ruin incremental performance, such as inter-procedurality or context-sensitivity. In this study, we set out to explore whether incrementalization can help to achieve speed-ups for production CodeQL analyses that provide automated feedback on pull requests on GitHub. We first empirically validate the idea by measuring the potential for reuse on real-world codebases, and then we create a prototype incremental solver for CodeQL that exploits incrementality. We report on experimental results showing that we can indeed achieve update times proportional to the size of the code change, and we also discuss the limitations of our prototype.

CCS CONCEPTS

• **Software and its engineering** → **Automated static analysis**.

KEYWORDS

Static Analysis, Incremental Computing, Datalog, CodeQL

ACM Reference Format:
Tamás Szabó. 2023. Incrementalizing Production CodeQL Analyses. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3611643.3613860

1 INTRODUCTION

Static analyses play a key role in modern software development because they help catch potential runtime errors already at development time. CodeQL [3] is a static analysis framework that has seen widespread adoption in industry. CodeQL analyses are written in a declarative language called QL, which compiles to Datalog under the hood. The subject program (i.e. codebase to be analyzed) is extracted into a relational database format, and the CodeQL solver executes analyses as queries over the database. GitHub Code Scanning comes with a suite of CodeQL analyses for various languages that provide automated feedback about security vulnerabilities and potential runtime issues at pull request (PR) review time.

Performance and precision are key requirements when it comes to static analyses. On the one hand, the execution time is a key factor for the adoption of static analyses. Prior research shows that static analyses have at most a few minutes to compute their results when they are used as part of a code review process in continuous-integration (CI) environments [9, 17, 27]. Failing to meet this timing constraint interrupts the development flow, which quickly leads to disuse of the analyses. On the other hand, developers expect that static analyses faithfully capture the program behavior, so that they do not report errors that cannot actually occur at runtime. Unfortunately, these two requirements are at odds with each other, and static analysis designers must carefully strike a balance between them. CodeQL approaches this trade-off from the angle of precision because CodeQL analyses are highly sophisticated by design. CodeQL supports many popular programming languages, and, for all these languages, the CodeQL analyses use features and building blocks that are responsible for good precision; ranging from inter-procedural reasoning through context-sensitivity to field-sensitivity. This also means that the CodeQL solver has a difficult task, which it addresses by using a range of optimizations developed in the Datalog and database community [3, 14]. Still, for large projects, even this is not enough, and the execution time can grow well beyond the desired few minutes ballpark.

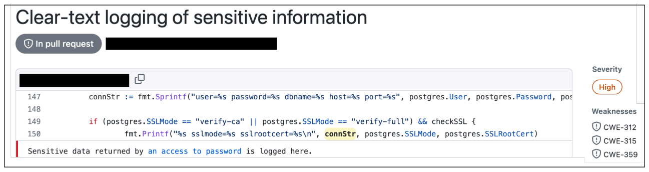
Researchers have long been investigating various techniques that can help speed up static analyses. Examples include parallel execution [16, 22], compositional analysis formulation [8, 11], partial evaluation [18, 23], demand-driven execution [28, 30], or incrementalization [34]. When it comes to delivering updated analysis results on a PR, incrementalization seems like an obvious choice. This is because, instead of repeated re-analysis from scratch, an incremental analysis reuses the previously computed analysis results and updates that based on the changed code parts. This aligns with the life cycle of a PR because that includes multiple rounds of reviews with small code changes in each iteration. If the computational effort required to update the previous results is proportional to the size of the code change (and not the size of the entire subject program), then significant speed-ups *can* be achieved. Prior work shows that this kind of speed-up is a reality for real-world static analyses [13, 23, 28, 32], but CodeQL does not employ incrementalization yet. It is important to emphasize that it is not necessarily the case that a small input change *always* leads to a small change in the analysis result. All those features that make static analyses precise (e.g. inter-procedurality, context-sensitivity) directly go against the potential benefits of incrementalization because even a small program change can have far-reaching transitive effects requiring re-analysis of a significant part of the subject program. There is prior work on investigating the incrementalizability of analyses implemented in Datalog [32, 36], but the results are inconclusive, and they do not obviously generalize to CodeQL.

In this paper, we conduct a *study* to investigate if and how we could use incrementalization to speed up production CodeQL analyses. We start off by clearing our doubts about whether incrementalization can be effective at all for CodeQL analyses. Following the idea of Szabó et al. [32], we conduct an *impact benchmark* using real-world Ruby codebases with a sophisticated data flow analysis. Our goal is to empirically reason about how much of the previously computed analysis results can be reused on average throughout a series of real-world commits, thereby measuring how much potential benefit we could get from an incremental analysis. We conduct this experiment without an actual incremental solver, just by using the existing CodeQL analysis pipeline. We find that there is a lot of potential for reuse and, with that, for speed-ups in updating previously computed results. This observation proves for us that it indeed makes sense to experiment with prototyping an incremental solver for CodeQL.

Given the complexity of the existing CodeQL solver, it is a daunting task to rewrite that to be incremental. Instead, for the purpose of our study, we prototype an incremental solver termed iQL based on an off-the-shelf incremental Datalog solver called Viatra Queries (VQ) [5]. We chose VQ because it has been reported to deliver good incremental performance and because it is already used as the back end of an incremental static analysis framework called IncA [33]. The challenge in our work is that while VQ has good expressive power in terms of the kinds of operations it can incrementalize, it is still not sufficient when compared to the kinds of analyses QL can express. We show how we transform the QL representation of an analysis to the representation that VQ can understand while bridging the abstraction gap between the two systems. Given that CodeQL scales significantly better than VQ in terms of raw performance, we also show how to execute parts of an analysis non-incrementally with CodeQL in an otherwise incremental evaluation governed by VQ. Ultimately, we end up with a prototype incremental solver that can execute production CodeQL analyses fully incrementally or in a hybrid setting where the non-incremental and incremental modes are mixed.

We measure the performance of iQL on real-world Ruby projects by running the above-mentioned CodeQL analysis incrementally on their commit histories. We find that the fully-incremental analysis takes maximum ~15 seconds to update analysis results for commits affecting up to 1000 lines of code. However, it takes more than an hour to perform the first from-scratch analysis, and the memory use (due to caching) of the analysis can go as high as ~70 GB, which is prohibitive. This is where the hybrid approach shines because it presents an interesting trade-off opportunity. We find that by executing all non-recursive parts of the analysis implementation non-incrementally, we manage to reduce the initialization time to ~15 minutes and the memory use to ~20 GB. In turn, the incremental update time gets higher compared to the fully-incremental approach, but it is still below a minute. In both cases, we find that the incremental update time is actually proportional to the size of the commit. These results are promising for CodeQL because they show that incrementalization can deliver fast feedback, but we also acknowledge that the high memory use requires further work before incrementalization can make its way to production CodeQL.

This paper makes the following contributions:


Clear-text logging of sensitive information

In pull request [redacted]

[redacted]

```
        connStr := fmt.Sprintf("user=%s password=%s dbname=%s host=%s port=%s", postgres.User, postgres.Password, pos

        if (postgres.SSLMode == "verify-ca" || postgres.SSLMode == "verify-full") && checkSSL {
                fmt.Printf("%s sslmode=%s sslrootcert=%s\n", connStr, postgres.SSLMode, postgres.SSLRootCert)
Sensitive data returned by an access to password is logged here.
```

Severity

High

Weaknesses

CWE-312

CWE-315

CWE-359

Figure 1: A concrete security vulnerability identified by CodeQL in a Go codebase.

- We identify the challenges that come with the incrementalization of production CodeQL analyses (Section 2).
- We present our impact benchmark which investigates whether incrementalization can be beneficial at all for production CodeQL (Section 4).
- We develop iQL based on an existing incremental Datalog solver called VQ. We discuss how we transform the QL analysis representation to VQ, and we also discuss the idea of the hybrid solver (Section 5).
- We benchmark the performance of iQL and show that incrementalization can deliver fast enough feedback for PRs (Section 6).

2 CHALLENGES AND HIGH-LEVEL SOLUTION APPROACH

We present a motivating example that shows CodeQL in action. Then we set the stage for our study to figure out whether incrementalization can help at all to speed up CodeQL analyses.

2.1 Motivating Example

CodeQL analyses are widely used in production to identify security vulnerabilities in GitHub repositories. Figure 1 shows an example where CodeQL reported a vulnerability in a Go codebase as part of a PR review. This vulnerability is a clear-text logging of sensitive information, and it is also categorized in the Common Weakness Enumeration (CWE) database.[1] Based on this feedback, developers can fix the vulnerability before the code gets merged into the main branch and then shipped to production. Even though this vulnerability is about a specific analysis and language, CodeQL supports many programming languages and frameworks,[2] and it comes with a rich set of analyses covering many important vulnerabilities.[3]

The standard analysis suite for Go contains, among others, the above-mentioned clear-text logging analysis and takes around 6 minutes to run on this particular codebase comprising 10 KLoC. While the feedback comes as part of a PR review in Figure 1, the size of the code change in the PR does not actually matter for the analysis: It executes from scratch on the whole codebase every time the PR is updated. For larger projects, the run time can get significantly higher, and it needs to be paid repeatedly. Research shows that developers are willing to wait at most a few minutes for automated feedback on a PR, afterwards they switch context, and the usefulness of the feedback quickly degrades [9, 17]. Our goal is to make the analysis time proportional to the size of the code

[1]https://cwe.mitre.org/
[2]https://codeql.github.com/docs/codeql-overview/supported-languages-and-frameworks/
[3]https://codeql.github.com/codeql-query-help/full-cwe/

change. To this end we conduct a study where we experiment with incrementality to deliver fast updates for PRs.

2.2 Prior Work on the Incrementalizability of Static Analyses

Static analyses often use inter-procedurality or context-sensitivity to deliver good precision. However, these are exactly the features that make an analysis computationally expensive. They are also the reasons why it is not straightforward to answer whether incrementality can help to achieve speed-up for such analyses. For example, if an analysis reasons about the subject program inter-procedurally, then it is not necessarily true that a small program change always leads to a small change in the analysis result because the change may have far-reaching effects across transitively reachable functions in the call graph. Inter-procedurality is essential to almost all production CodeQL analyses because vulnerabilities are rarely local to a single function but instead manifest across a series of function calls. Thus, it is a question if incrementality can help at all to speed up CodeQL analyses.

Prior research has already investigated the incrementalizability of inter-procedural analyses implemented in Datalog. For example, Szabó et al. used inter-procedural points-to, interval, and constant propagation analyses on Java codebases from the Qualitas corpus [32, 35]. They measured performance on synthesized IDE-style program changes to simulate the kind of changes that developers typically make in an IDE, modifying only individual expressions or statements. This is in contrast to larger program changes, as one might see in a commit. The kinds of program changes they introduced are tailored to trigger the worst-case behavior of the analyses in the sense that, e.g., changing an allocation site will affect the result of a points-to analysis with high probability, so that the analysis will definitely have work to do to update the analysis result. They reason about the *impact* of program changes. They introduce the notion of impact based on how the analysis result is represented in Datalog, which is about storing both the subject program and the analysis result as sets of relations in databases. They measure impact as the size of the diff between two database snapshots (representing two analyses results) in terms of the overall number of tuples that get inserted or deleted in any relation. They find that the majority of the program changes have small impact, which is a necessary condition for incrementalizability.

Zhao et al. also investigate this topic with an inter-procedural points-to analysis adapted from Doop running on a Java codebase [7, 36]. They also synthesize program changes: They randomly select a subset of the tuples (up to 1000) from the database that represents just the subject program (without the analysis result), and they delete and then re-insert those tuples while measuring the impact as explained before. They find large variability in impacts and argue that incrementalization does not necessarily always help. They create an incremental solver that does not actually always perform an incremental update, but, based on heuristics, restarts the analysis from scratch if the impact of a change is deemed too high. It is not clear if these two pieces of related work contradict each other because they use different analyses and code changes, but they are indicators that more exploration is needed to figure out if CodeQL analyses can benefit from incrementalization with commit-style changes. This leads us to our first challenge:

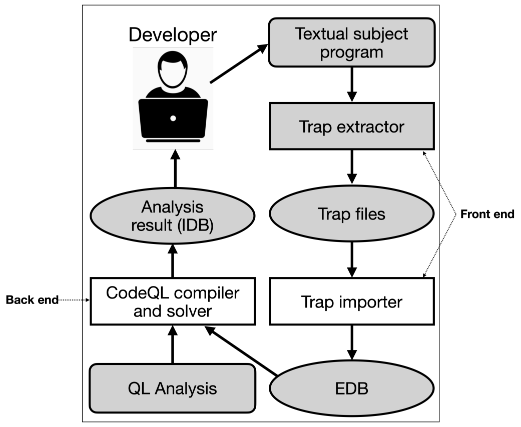


Figure 2: High-level architecture of the CodeQL analysis pipeline. Rectangles are individual analysis components. Rounded rectangles are the inputs to CodeQL: the subject program and the analysis. Ovals are output artifacts. Language-specific components are shaded.

Incrementalizability (C1): Does incrementalization have potential for speeding up production CodeQL analyses? Can we answer this without actually implementing an incremental solver?

Solution approach We also conduct an impact measurement. We use real-world Ruby codebases with their commit histories and a sophisticated CodeQL data flow analysis and show that incrementalization indeed has potential to speed up CodeQL analyses (Section 4).

2.3 Challenges of Incrementalizing CodeQL

Assuming that the impact measurement yields promising results, we now want to investigate what it would take to actually incrementalize the CodeQL analysis pipeline and the challenges that come with that. We must first briefly review the main components of CodeQL.

Figure 2 shows the high-level architecture of CodeQL. The subject program is given to CodeQL as text, i.e., the code that is stored in a GitHub repository. The *trap extractor* is a language-specific component that creates so called *trap* files. Trap files encode various information about the subject program that CodeQL analyses later could require, such as an abstract syntax tree (AST) representation of the program, represented in a relational format. Each supported language has its own database schema defining the available relations. For dynamically-typed languages, such as Ruby or Python, the trap extractor only extracts information from the AST. For other languages, the trap extractor may require compilation and build tasks, so that it can also extract derived information about the program, i.e. type hierarchy or macro expansions. The *trap importer* is language-independent, and it builds the extensional database (EDB) from a set of trap files. The EDB represents the subject program in relational form.

The *CodeQL compiler and solver* takes the EDB and the QL analysis definition as input to first compile the analysis and then compute the *analysis result* which is also called the intensional database

(IDB). The QL analysis is defined with the declarative QL language, and the CodeQL solver completely hides the execution details. It is important to emphasize that CodeQL does not usually store the entire IDB explicitly but only populates that on demand. Using the analysis result, we can then provide feedback to developers on the GitHub UI.

Incrementalization of the CodeQL analysis pipeline requires that every component of the pipeline is either truly incremental or fast enough that it does not become a bottleneck for performance. This leads us to two more challenges:

Front end (C2): Can we make the CodeQL front end (trap extractor and importer) incremental?
Back end (C3): Can we create an incremental solver that supports production CodeQL analyses?

Finally, we also want to get empirical evidence about performance, which leads us to the following challenges:

Update time (C4.1): Can we achieve incremental update times that are (i) below a minute on average per commit and (ii) proportional to the size of the analyzed commit? The minute threshold is reasonable for PR-style feedback based on prior studies in this area [9, 17].
Init time (C4.2): Does our incrementalization approach come with an acceptable initialization time?
Memory use (C4.3): Is the extra memory use induced by incrementalization acceptable? CodeQL analyses are typically executed on standard GitHub Actions runners with 7 GB of memory,[4] so we ideally want to stay below that threshold.

Solution approach When it comes to the front end of CodeQL, it turns out that non-incremental parsers are already able to parse source files in milliseconds which is fast enough to enable their use even in an incremental analysis pipeline [34, Chapter 7]. We acknowledge though that for compiled languages CodeQL requires more than just parsing, so this is not a full solution in general, but it is still sufficient for our study given that we target Ruby. To this end, we focus on the construction and incremental maintenance of the EDB. We show that the current CodeQL front end is not compatible with incrementalization because it is unable to reuse parts of the EDB that belong to unchanged code parts which degrades incremental performance later in the analysis phase. The root cause of this issue lies in the way how ids are generated for program elements in the EDB. We fix this problem by using an alternative id generation approach which ensures that unchanged program elements get the same ids assigned across commits (Section 4). Next, we create an incremental solver called iQL for QL analyses based on an off-the-shelf incremental Datalog solver called VQ (Section 5). Finally, we benchmark the performance of iQL on real-world Ruby projects (Section 6).

3 BACKGROUND ON CODEQL AND VQ

Before we delve into the details of our approach, we first present some background material on CodeQL and VQ.

[4]https://gh.io/AAjpnuo

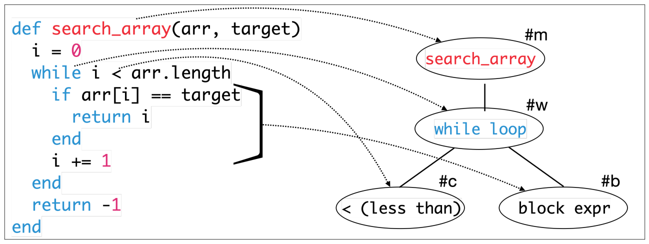

```
def search_array(arr, target)
  i = 0
  while i < arr.length
    if arr[i] == target
      return i
    end
    i += 1
  end
  return -1
end
```

Figure 3: An example Ruby code snippet and a fragment of its AST. Each AST node has an associated label.

3.1 CodeQL

QL language features While the surface syntax of QL looks quite unlike Datalog, it actually gets compiled down to a fairly standard Datalog dialect called DIL. Recursion is central to QL, and it is implemented using least fixpoint semantics. Since QL also supports negation and non-monotonic aggregates, restrictions have to be put into place to ensure a well-defined semantics. Other Datalog dialects generally insist on strict stratification whereby recursion through negation is not allowed [14]. QL relaxes this requirement and allows parity-stratified negation which means that recursive calls under an even number of negations are allowed, though recursion through other non-monotonic operators is still forbidden.[5]

QL also defines a number of *built-ins* that are awkward or impossible to express in standard Datalog to provide operations that are frequently used in static analyses, such as computation of dominator trees or shortest paths. Built-ins have implementations in Java, which CodeQL invokes during the evaluation process. QL also supports user-defined algebraic data types (ADTs), which are used heavily in the standard analysis suite. This is a big jump in expressive power considering that standard Datalog does not allow the construction of any kind of new value. Virtually every QL analysis uses ADTs. At runtime, ADTs also require the generation of fresh ids in order to distinguish between the various instances [21]. Finally, QL supports complex recursive aggregations (including non-monotonic ones) which are challenging even in a non-incremental setting [10].

Front end We already discussed that both the subject program and the analysis result are represented in relations. The trap extractor emits instructions in trap files that will determine what kind of tuples get inserted into the EDB during trap import. As an example, consider the code snippet on the left in Figure 3 and a fragment of its AST on the right. A fragment of the trap file associated with this code snippet looks as follows.

```
#m = @"example.rb#search_array"
#w = *, #b = *, #c = *
ruby_method_def(#m, 'search_array(arr, target)')
ruby_while_def(#w, #b, #c)
```

The trap extractor uses *keys* to refer to program elements. Keys will get turned into integer ids later by the trap importer, but the actual values of integer ids do not matter at extraction time.

[5]It is important to note here that QL (and DIL) are not Horn-clause based, and rule bodies can contain arbitrarily nested first-order logic formulas. In Horn clauses, of course, calls always appear under at most one negation, so parity stratification does not make sense.

Global keys are represented with a string value, and multiple occurrences of the same global key will get the same id assigned through string interning. For example, we assign the global key `@"example.rb#search_array"` to *label* `#m`. Here, we prepend the container file's path before the name of the method because this gives us a globally unique reference to the method. For the other AST nodes, we use local keys (`*`), which simply prescribes for the trap importer the generation of a fresh integer id without sharing of the value. The trap importer uses a globally incremented counter, and it rolls out ids sequentially. We access the local keys through the other labels (`#w`, `#b`, `#c`) and then emit one tuple each in relations `ruby_method_def` and `ruby_while_def`. The trap importer will instantiate the real tuples later and insert them into the EDB.

Back end QL code first gets compiled to DIL, which, in turn, gets compiled to a low-level relational algebra (RA) representation. The CodeQL compiler performs numerous optimizations both on the DIL and on the RA level, including constant folding, dead-code elimination, and join ordering. Join ordering is particularly important because real-world codebases typically produce relations with millions of tuples, so a single inefficient join can make the evaluation intractable. The evaluation of recursive analyses is challenging in general in Datalog. CodeQL employs a well-known technique called semi-naïve evaluation for this purpose. This means that a recursive program gets evaluated with an iterative fixpoint computation where each iteration only considers newly computed results from the previous iteration to improve performance [14]. At least a conceptual level, semi-naïve evaluation operates with different auxiliary predicates that compute e.g. the set of all tuples computed in the previous iteration, the set of new tuples (i.e. not inferred before) computed in the previous iteration, or the tuples computed thus far in the current iteration. Typically, Datalog solvers perform semi-naïve computation implicitly without requiring the auxiliary predicates to be present in the input Datalog program. However, CodeQL generates the helper predicates explicitly during compilation, and the presence of the predicates is heavily woven into the downstream compilation stages.

3.2 Viatra Queries

VQ is a model transformation framework primarily used for Eclipse Modeling Framework (EMF) models. On the surface, it comes with the Viatra Query Language (VQL) to formulate queries used to retrieve information from modeling artifacts. A VQ query is essentially the same as a Datalog predicate. VQL is similar to Datalog both in expressive power and in terms of its language constructs. VQ uses an incremental Datalog solver that can execute queries incrementally in response to changes in the input model. The model representation and change notification mechanism of VQ is abstracted away from EMF, so we can plug in any data representation that adheres to the VQ APIs.

Under the hood, VQ uses a computation network for incrementalization. The input to the network is a set of relations. A VQ query is broken down into relational algebra operations (e.g. join, filter, union), each having an incremental implementation. Changes to the input must be communicated to VQ in the form of tuple insertions and deletions. Each node in the network caches its results. In the face of incoming changes, the nodes incrementally update their results and then propagate the *delta* (that is, the difference between the old and the new result) on the outgoing edges. The change propagation goes on until the result of each node stabilizes. The fixpoint computation follows semi-naïve just like in CodeQL, but VQ does not need any auxiliary predicate to be present.

In terms of absolute performance of from-scratch execution, CodeQL scales better and to larger programs, but VQ trades off initialization time and memory use for incremental update time. For example, IncA [32] which is an incremental static analysis framework based on VQ could achieve millisecond update times for inter-procedural analyses. Compared to CodeQL, VQ does not employ so many optimizations at compile time, but VQ is a highly extensible framework, so optimizations or even new language features are fairly easy to add. VQ only uses dynamic join ordering based on simple heuristics.

4 INCREMENTALIZABILITY OF CODEQL ANALYSES

This section investigates the potential for reuse when it comes to production CodeQL analyses and real-world codebases, thereby giving an estimate on how much we could gain with incrementality.

Measurement setup To measure the incrementalizability of CodeQL analyses, we also perform impact measurements inspired by prior work [32, 36]. We use real-world subject programs. The repositories we selected are all trending projects on GitHub with several thousand stars. Our benchmark analysis is based on a production CodeQL analysis used to find SQL injection vulnerabilities.[6] The original analysis is based on taint tracking across data flow paths, and it employs many of the features that are responsible for good precision, e.g. inter-procedurality, context-sensitivity, or field-sensitivity. The analysis also has a lazy implementation, which means that it stops with the propagation of data flow facts as early as possible if it finds that there are infeasible data flow paths. Given that our benchmark codebases do not necessarily have SQL injection vulnerabilities, we modified the original analysis by changing what kind of data flow nodes should be considered as sources and sinks by the analysis. We connect all formal parameters of the functions with all actual arguments of function calls to eliminate the laziness of the analysis, thereby making it computationally expensive even in the absence of SQL injection vulnerabilities. We selected altogether 6 Ruby projects for benchmarking. The complete list is available in the extended version of this paper [31]. We chose Ruby because the Ruby fact extractor has the simplest implementation out of all the language front ends in CodeQL. It only requires parsing the source code, without any complex integration with a build system as for compiled languages. Choosing Ruby only really simplifies the fact extraction, but otherwise the analysis is representative in terms of its complexity across the production CodeQL analyses.

For each project, we use the newest 1000 commits from the history. For each commit, we take the EDB, and we force CodeQL to create an explicit representation of the entire IDB (which it would not do otherwise due to optimizations). We measure the size of each commit by summing up the number of lines of code that get deleted

[6] https://cwe.mitre.org/data/definitions/89.html

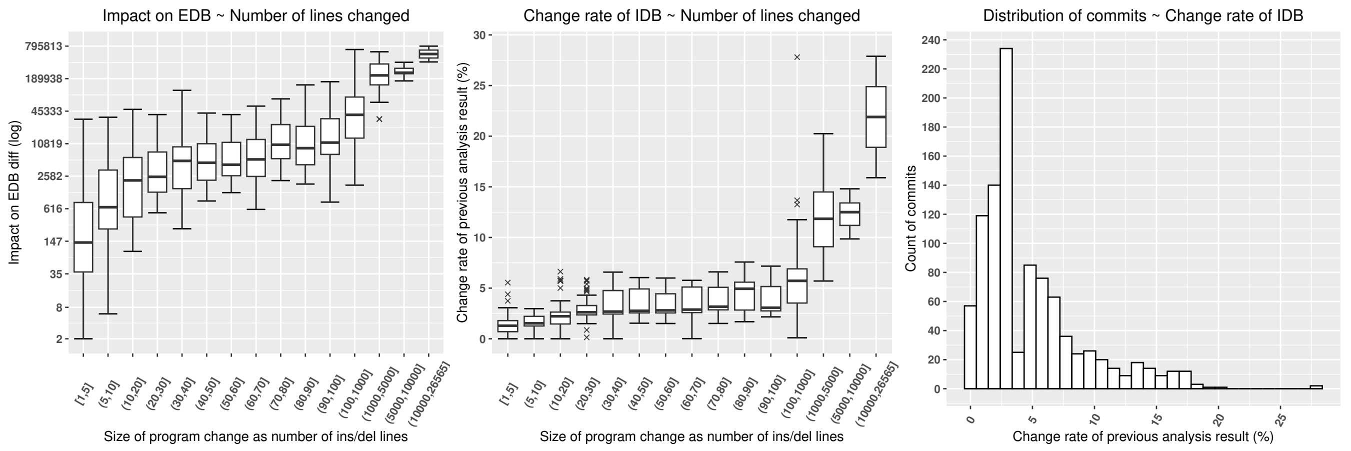


Figure 4: Impact measurement results with optimized id generation on the spree project.

or inserted. A line modification equals to a deletion-insertion pair. We diff consecutive EDB and IDB snapshots to compute impacts. We emphasize that there is no incrementality involved in these measurements. We simply would like to understand first how impact relates to the commit size without an actual incremental solver.

Using the production CodeQL pipeline First, we used production CodeQL for the measurements. We report on the *spree* project, which is an e-commerce platform comprising ~ 73 KLoC Ruby code. We find that throughout the commit history the average number of tuples in the EDB is 2.76 million and the average number of tuples in the IDB is 217.41 million. We find that *irrespective of the commit size*, the average impact on the EDB throughout the commit history is 4 million tuples. This number is actually higher than the size of an individual EDB. This means that, even for small commits, we essentially reconstruct a new EDB because we delete most of the previous contents and then insert a large number of new tuples. This is obviously not compatible with incrementalization. We would expect that, at least for small commits, only a small fraction of the EDB changes. Unfortunately, we have not even computed the analysis result: This is just for the representation of the subject program in the EDB! The average impact on the IDB is 209 million, which is again bad for incrementality. For space reasons, we only talk about one specific project here, but, for all other projects we considered, we found similar high impacts.

Using stable ids We investigated in detail the root cause of the high impacts, and we found that to be due to the lack of stable ids for AST nodes and ADT values. Recall that the trap importer rolls out ids sequentially based on a counter, so a small change to one file may end up shifting ids for all the AST nodes later in the file *and* all subsequently imported files. Similarly, the CodeQL solver assigns ids to ADT instances sequentially, again offering no guarantee that the same ADT instance will be given the same id across different runs. This leads to a huge number of spurious changes. We mitigate this problem in two steps.

First, we modified both the trap extractor and importer. In the extractor, we implemented a new approach for constructing global keys for every AST node based on paths in the AST. When we traverse the AST to create the trap files, we recursively assemble a node path that describes how we can get to a node across containment edges. For example, a path like *r*_1_3 gets associated with a node which can be reached from the root if we take the first child under the root, then the third child after that. If we prepend the full file path of a source file before such node paths, we get globally unique keys. We do not use an id counter anymore in the trap importer. Instead, we calculate an integer hash from the global key and use that in the actual tuples. This approach of course does not help in all situations to ensure stability because e.g. it does not survive a node shifting across commits, but it still significantly helps in id reuse. This kind of id generation is a must if we want to incrementalize the front end of CodeQL (cf. **Front end (C2)**).

Second, we employ a workaround to reuse ids for ADT instances. CodeQL can dump the id assignment of ADT instances to disk. As we proceed along commit pairs in the measurement, we dump the id pool when we process the old commit, and we augment the analysis run for the new commit to reuse the same id assignment whenever possible based on the pool.

Using these two improvements, Figure 4 shows our new measurements on the spree project. The extended version of this paper presents the results for several other projects [31]. The first sub-figure shows the correlation between the impact on EDB and the size of commit. The commits are arranged into buckets in terms of how many lines of code they affect. The impact on the EDB is proportional to the size of the commit, and the impact maxes out at around 800,000 tuples which is much smaller than the overall size of the EDB. We show change rates for the IDB. The change rate tells us the ratio of overall number of tuples deleted or inserted (to get to the new IDB) to the size of the old IDB. The lower the value the better. The middle sub-figure shows that the change rate is also proportional to the commit size. Up to 1000 changed lines in a commit, the change rate is at or below 5 %, which is exactly the kind of number we look for because it demonstrates that we can gain a lot with incrementality. The right sub-figure shows the distribution of the commits. The x-axis shows the change rate, and the y-axis shows how many commits (out of the 1000) produce a certain change rate. The majority of the commits produce a low change rate. The other codebases we considered also produced similar results.

Regarding **Incrementalizability (C1)**: Given that our benchmark analysis is representative in terms of its complexity when it comes to production CodeQL analyses and based on the results of the impact measurements, we conclude that CodeQL analyses are amenable to incrementalization.
Regarding **Front end (C2)**: We found that the production CodeQL front end is not compatible with incrementalization. For an incremental analysis pipeline, it is crucial to employ an id generation strategy that ensures stability as we process code changes. Our approach based on node paths satisfies this requirement.

The impacts measured here are solver-independent, and they give a lower bound on the amount of work that any solver would need to do to update the analysis result. Even though we used the CodeQL solver in our setup, ultimately the analysis specification and the schema of the database determine the impact values.

We also conducted some initial experiments with the Java front end of CodeQL. The Java front end is much more involved in the sense that it extracts information beyond the AST. It actually invokes the build process of the given project (e.g. Gradle or maven build) and dumps information about e.g. type hierarchy or generics, as well. Due to these differences, it turned out to be technically much more difficult to employ our node path-based id generation strategy because that would have required changes to the Java compiler. Instead, we came up with a different strategy that ensures that the same ids get rolled out for program elements originating from unchanged source files considering a commit pair without any kind of id alignment guarantee for program elements originating from changed source files. Due to space reasons, we discuss this strategy and present results only in the extended version of this paper [31]. In sum, we found that the impact is proportional to the size of the code change in case of Java codebases, as well.

5 IQL: INTEGRATING CODEQL WITH VQ

As mentioned above, our prototype system iQL works by translating CodeQL queries to VQ queries, which are then run by the VQ solver. To translate from QL to VQ, we first run the CodeQL compilation and optimization pipeline to convert QL (through DIL) to CodeQL RA and, from there, compile to VQ RA. This has two main benefits: (i) RA is a much smaller language than QL (or DIL), with a lot of syntactic sugar removed, and (ii) we can take advantage of the optimizations that the CodeQL compiler performs. Table 1 shows the main challenges of the translation and how we address them.

Challenges due to differences arising at runtime The first two challenges have to do with the way the systems deal with recursion and join ordering. As mentioned in Section 2.3, CodeQL implements semi-naïve evaluation by explicit code generation, while VQ implements it natively in the solver. This discrepancy can be resolved by turning off the pass of the CodeQL compiler that generates the auxiliary predicates.[7] Join ordering in CodeQL is done at compile time, with the RA statically encoding the join order. VQ dynamically reorders joins at runtime. We deal with this by extending VQ to allow passing join-ordering hints taken from the CodeQL compiler, which the VQ solver uses at runtime.

[7]Note that this means the resulting program is no longer runnable by CodeQL, but this is not a problem for our study.

Challenges due to different expressive power The remaining challenges have to do with missing or divergent features in VQ. Two commonly-used key features of QL without a direct equivalent were reimplemented from scratch. One is the so-called *tuple numbering* operator, which assigns a unique integer to each tuple in a relation and serves as the basis of CodeQL's implementation of ADTs. VQ does not have a direct equivalent and the CodeQL implementation of this operation is too deeply tied to the internals of CodeQL to port, so we instead rewrote it from scratch, using a simple global cache keeping a mapping from all previously seen tuples to their unique identifiers. This is not a plausible solution for real-world usage since we can never know when we do not need a tuple anymore and hence the cache grows without bound, but it suffices for our study. The other key feature we reimplemented are QL's aggregation operations on strings that have no direct VQ equivalent, but were fairly straightforward to add.

CodeQL's built-in operations are a bit more complicated. While VQ offers similar operations in some cases (e.g. transitive closure), we decided against adapting them or reimplementing missing built-ins from scratch to avoid the risk of introducing subtle incompatibilities. Instead, we directly lifted the Java implementation of these built-ins from the CodeQL engine, wrapped in a translation layer to convert between the respective storage formats for CodeQL and VQ relations. This ensures not only that the operations are semantically equivalent, but also that they have the same performance. However, the CodeQL built-ins are not incremental, and fitting them into the otherwise incremental evaluation in VQ is somewhat involved. There is a VQ computation node behind every built-in appearing in the analysis. Whenever such a node gets notified about changes to its input relations, it computes the updated input relations in full, and feeds them to the wrapped CodeQL built-in (which can only work on complete relations, not on deltas). The result of the built-in is then diffed against the previous result, and the delta is propagated to the dependent nodes. Given that VQ normally propagates individual tuples without batching them, repeatedly re-evaluating a built-in from scratch can easily become a performance bottleneck. To avoid this situation, we modified the implementation of the computation network in VQ to (i) batch the inputs of built-ins and (ii) schedule the invocation of nodes evaluating built-ins only after all their incoming updates have arrived.

The two most complicated features of QL that we had to deal with are parity-stratified recursion and recursive aggregates. However, it turns out that in practice neither of these two features is very widely used, and the code that does use them can be rewritten to avoid them, at some cost in readability. Since our aim is to study the potential for incrementality in CodeQL, not to provide a complete incremental CodeQL implementation, we decided to take this approach.

Hybrid evaluation While the translation outlined before works and produces correct VQ programs whose results can be updated very efficiently, these programs are much bigger than what VQ has been designed to handle. As we will see in Section 6, even on small subject programs, the memory use of iQL can be prohibitive.

To address this problem, we study a *hybrid* evaluation strategy where parts of the program are evaluated non-incrementally by

Table 1: Summary of how the translation from QL to VQ handles missing and divergent features

Feature	QL	VQ	Translation
semi-naïve evaluation	explicit	implicit	turn off auxiliary predicate generation
join ordering	compile-time	runtime	modify VQ to inherit the join order computed by CodeQL
tuple numbering	supported	not supported	add support to VQ
aggregation operators	many	few	add support for missing operators to VQ
built-ins	many	few	add support to VQ by wrapping CodeQL implementation
parity stratification	supported	not supported	eliminate by rewriting
recursive aggregates	different semantics		eliminate by rewriting

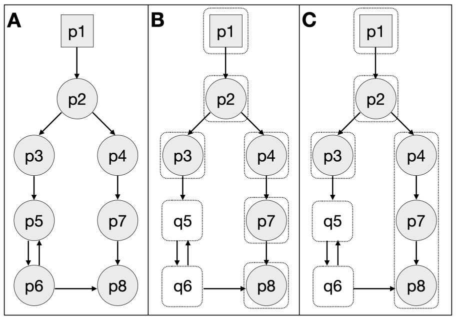


Figure 5: Transformation strategies from CodeQL to VQ: (A) Example dependencies between QL predicates, (B) transformation strategy which creates one VQ query per QL predicate, (C) chaining transformation strategy.

CodeQL, and the results are integrated into the overall VQ program similar to how built-ins are evaluated. The reason why this is interesting is that the parts of the analysis that are evaluated non-incrementally do not require intermediate caching but only at the point where the final (partial) results appear. By choosing which parts of the program to evaluate non-incrementally, we can trade off CodeQL's superior scalability for VQ's incrementality.

As mentioned above, we have to turn off CodeQL's semi-naïve transformation to enable translation to VQ. This means that the recursive predicates have to be evaluated by VQ. We still have freedom in deciding how we evaluate the non-recursive predicates. Consider the QL program shown schematically in Figure 5 A: It consists of one EDB predicate `p1`, and seven IDB predicates `p2` through `p8`. The arrows indicate dependencies, so, for example, predicate `p2` depends on `p1`, while `p5` and `p6` recursively depend on each other. For the non-recursive predicates, a very simple approach would be to wrap each of them individually in a VQ query that calls out to CodeQL for evaluation, translating back and forth between the respective storage formats similar to how built-ins are handled. Figure 5 B shows this approach: each of `p1`, `p2`, `p3`, `p4`, `p7`, and `p8` are wrapped in a VQ query, while `p5` and `p6` are translated into VQ queries `q5` and `q6`. This approach, however, makes things worse: Not only are the wrapped predicates evaluated non-incrementally, but the transformation happening at the VQ-CodeQL boundary introduces significant overhead both in memory use and run time.

To address this, we use a refined translation strategy shown in Figure 5 C. As before, EDB predicates are individually wrapped in VQ queries. IDB predicates are grouped into *chains*, where a chain is a maximal sequence of (non-recursive) IDB predicates that each flow into at most one other predicate. In the example, we identify one non-trivial chain `p4, p7, p8`, while the remaining predicates form a single-element chain each. Note that while it may sound like a good idea to group all the non-recursive predicates together, we can't actually do that because they may interact with recursive predicates. Also, the requirement for at most one outgoing flow is intentional because this way, if a predicate is a fork point (like `p2`), then we will benefit from the caches (at `p2`) shared by the fork endpoints (`p3` and `p4`). In the following, when we use the term *hybrid* solver, we refer to the approach based on chaining.

6 PERFORMANCE BENCHMARKING OF IQL

This section investigates all research challenges formulated in Section 2 that are related to performance.

Measurement setup We benchmark the performance of iQL with real-world Ruby projects and their commit histories using the same benchmark analysis that we used for the impact measurements in Section 4. Technically, the Ruby front end is implemented in Rust, and iQL and VQ are implemented in Java. iQL is a closed-source project because it integrates tightly with CodeQL, and CodeQL is a closed-source project. Due to scalability issues in memory use, we only use two smaller Ruby projects: pagy comprising 6 KLoC code and errbit comprising 9 KLoC code. For each project, we use 200 commits from the history. Our Ruby front end makes use of the id generation strategy described in Section 4, but the front end itself is not incremental. Instead, we precompute the EDB differences: Given every commit pair (c_{old}, c_{new}), we compute EDB_{old} and EDB_{new} and then diff them. We then perform a from-scratch analysis using EDB_{old} to compute IDB_{old}. We make use of both kinds of incremental solvers in iQL: the *fully-incremental* one which only executes the built-ins non-incrementally and the *hybrid* one which makes use of production CodeQL for the execution of non-recursive predicates, as well. We measure the wall-clock initialization time. Then, we take the EDB delta between EDB_{old} and EDB_{new} and let iQL perform an incremental update on IDB_{old} to compute IDB_{new}. We measure the incremental update time and also the stationary memory use (after explicit GC invocation) of iQL before and after the incremental update. We automated all of these steps and executed the benchmarks on GitHub Actions.

Correctness To ensure that iQL performs correct incremental updates, we have set up automated verification machinery. Before and after an incremental update, we check that iQL computes the exact same results as what production CodeQL would compute. We diff all relations in the IDBs, and we make sure that those are identical. The only difference we allow is the actual values of fresh ids, but that is expected as the two systems use different strategies.

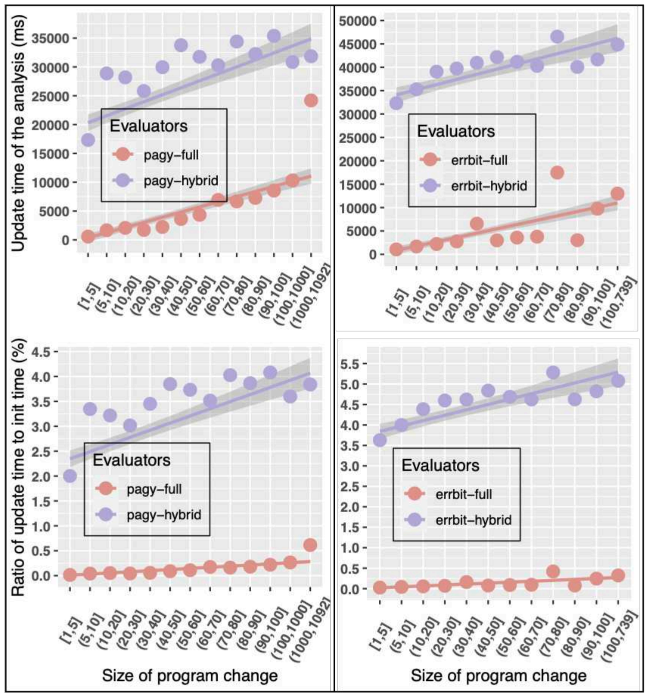


Figure 6: Results of the performance benchmarking: pagy on the left and errbit on the right.

Results Figure 6 shows our benchmark results. The plots on the top show the incremental update times relative to the size of the code changes. The commits are grouped into change size buckets. The dots represent the mean value in a given bucket. The gray areas show the standard deviation of the values with 95 % confidence. The lines are shown to provide visual guidance, and they are computed as a linear regression fit onto the data points. We show the results for both codebases using the two evaluator approaches. The plots at the bottom show the ratio of the incremental update times relative to the initialization time. We summarize the initialization time and memory use below:

	full		*hybrid*	
	init (min)	mem (GB)	init (min)	mem (GB)
pagy	66.10	70.68	14.40	21.52
errbit	67.10	72.61	14.72	22.51

In terms of the incremental performance, we see very promising results. There is a linear correlation between the update time and the commit size in all cases. The fully incremental approach is better in terms of update times than the hybrid one, but this is expected, as the latter one executes a large part of the analysis non-incrementally. For all size buckets, the fully incremental approach delivers update times in around 15 seconds or less, while the hybrid one is also well within the one-minute ballpark. The ratio of update time to initialization time is less than 1 % in case of the fully incremental approach, which shows that iQL can indeed exploit the potential for speed-ups as forecast in Section 4. While it is not a fair comparison, it is worth pointing out that the production CodeQL solver requires around a minute to execute the same analysis from scratch on these codebases when all caches are turned off.[8] This means that the fully incremental approach is actually competitive in this regard against a production system, even though it is only a prototype.

The initialization time is now a different story when it comes to the fully incremental approach. We see run times of about an hour, which is significantly slower than production CodeQL. However, the hybrid approach presents an interesting trade-off opportunity, as its initialization time is only around 15 minutes. Considering that the incremental update times of the hybrid one were also less than a minute, this option may present a good choice in general. Interestingly, the initialization times do not change much between the two codebases. It turns out that our benchmark analysis produces a very complex and large computation network in VQ, and building that actually becomes a dominating factor in the run time. These differences also make it clear that production CodeQL simply scales significantly better than VQ. We argue though that even the one-hour initialization time is acceptable, as we could precompute the results for select "main" branches and compute the results for a PR with incremental updates from those.

The biggest obstacle right now is the memory use. The several tens of GB values for the fully incremental approach are prohibitive. The hybrid approach helps to reduce these numbers significantly, but even ~ 20 GB is too large for a regular GitHub Actions runner. Even with these results, we believe that our study is useful because it demonstrates the potential of incrementalization.

Regarding **Update time (C4.1)**: The incremental update times are within the 1 minute ballpark for both approaches, and the incremental update time is proportional to the commit size.
Regarding **Init time (C4.2)**: The fully-incremental approach requires around 1 hour to perform a from-scratch run, while the hybrid approach requires around 15 minutes. Both of these values are acceptable given that they are not too high in absolute terms, and we can precompute results.
Regarding **Memory use (C4.3)**: The memory use of the fully-incremental approach is very high, but the hybrid approach comes with significantly better memory use. More work would be needed to improve the memory use in a production incremental solver.

7 RELATED WORK

In Section 2.2, we already discussed prior work on the incrementalizability of Datalog-based static analyses. Here, we contrast our work to other related areas.

Studies on the use of static analyses There are several studies in the literature which investigate what developers expect from static analyses. We highlight some key observations here. Christakis and Bird point out that (i) developers in their study care a lot about low false positive rate, (ii) 77 % of them would appreciate if an analysis could be governed to only analyse a change as opposed to the entire codebase, and (iii) long-running analyzers that exceed a few minutes would not be considered by nearly three quarters of developers in their study [9]. Johnson et al. explain that developers expect static analyses to be properly integrated into their workflows,

[8]Turning off all caches does make sense because a realistic scenario is to use CodeQL on GitHub Actions after a push to a branch in a new virtual machine, so caches would not be readily available anyway.

whether that is the IDE or PR reviews [17]. Moreover, it is important that feedback comes in a timely manner before developers switch context because that makes the interpretation of the analysis output more difficult. Sadowski et al. investigate the use of static analyses at Google [27]. They find similar observations as the previous studies in terms of expected run time and workflow integration. However, they also provide the actual list of static analyses that they use at scale. It turns out that all the analyzers perform some sort of linting or dependency analysis. These are obviously helpful analyses, but they are computationally not too expensive, and they immediately "incrementalize" on a file-level granularity. In contrast, our goal is to deliver fast update times for sophisticated data flow analyses that are computationally much more expensive.

Incrementalizing individual static analyses Incrementalization has received a lot of attention both as a means to speed up specific static analyses or as a technique to power entire analysis frameworks dealing with classes of analyses. For example, Lu et al. devise an incremental points-to analysis by formulating it as a graph reachability problem and then making use of well-known techniques for incrementalizing graph reachability itself [20]. Saha and Ramakrishnan design an incremental and demand-driven points-to analysis formulated in Datalog [29]. Demand-driven means that the analysis only computes points-to information that is relevant to the client. They achieve this with the use of magic set transformation [4] which essentially introduces auxiliary predicates in the Datalog program encoding the demand itself (e.g. fixing the pointer variable to something specific). The incremental maintenance in their solution is based on Delete and Rederive (DRed) [15] which is a well-known algorithm for incrementalizing recursive Datalog programs. While iQL is an analysis *framework* that does not make assumptions about the kind of QL analysis it needs to incrementalize, there are definitely connections to these pieces of related work. Magic set transformation is heavily used in the CodeQL compiler to reduce work by inlining information about the input program, so iQL benefits from that, too. DRed is also responsible for the incrementalization in iQL because that is one of the incremental solver algorithms used in VQ.

Incremental static analysis frameworks There are several approaches that incrementalize set-based static analyses [12, 19, 26]. Set-based here refers to how the analyses operate: They propagate sets of data-flow facts along the nodes in the control flow graph using set union or intersection at merge points. Many practically interesting static analyses can be formulated this way (such as liveness or uninitialized read), but this expressive power is not enough for production CodeQL analyses.

The Reviser framework incrementalizes IFDS/IDE analyses [2]. IFDS/IDE is a generic framework for formulating data flow analyses, the former supporting only power set lattice values as data flow facts, while the latter supporting custom lattice values. IFDS/IDE boil down to graph reachability and summarization of the effects of functions. While IFDS/IDE have been used to implement industry-strength static analyses (e.g. Boomerang [30]), the challenge in efficiently employing the technique is that summaries should encode function-local information only, and this is difficult to achieve in practice [6]. It is challenging to compare the expressive power to that of QL because of the different computational models.

Infer is a static analysis framework that is used at scale at Facebook [8]. Infer supports analyses that reason about manipulations of the heap (e.g. null pointer dereference, finding resource leaks). The theoretical foundation is separation logic which allows for efficient summarization of effects of heap manipulations. This approach lends itself to efficient incrementalization because summaries only need to be re-computed for changed code parts, and then the overall analysis result can be composed of the individual summaries.

IncA [32, 33] is an incremental static analysis framework that also builds on VQ. IncA is unique among incremental Datalog-based systems in that it supports user-defined lattices and recursive aggregations over lattices which is an important building block for static analyses. Similar to CodeQL, IncA also defines a higher-level DSL for analysis specification which then gets compiled to the RA format of VQ. Given that QL does not support custom lattices, we also did not make use of this feature in VQ.

Compilation of other languages to Datalog There are several pieces of related work that revolve around the idea that while Datalog has many benefits when it comes to the execution aspect, perhaps it does not have the best design when it comes to the specification aspect. QL itself is a good example because it adds an object-oriented flavor to Datalog with higher-order domain-specific extensions. Pacak et al. design a DSL for defining type checking rules and then compile the DSL code to Datalog to be executed by IncA [25]. The challenge in their work is that Datalog solvers expect the input relations to be finite, but, when it comes to type checking, the typing relation itself can grow infinite. They make heavy use of magic set transformations to get rid of the typing context in the typing relation, thereby eliminating the problem of infinite input. Pacak and Erdweg later go a step further by designing Functional IncA which is still a limited but general-purpose functional language that also compiles to IncA Datalog [24]. Datafun [1] is not strictly speaking an example where the language itself gets compiled to Datalog, but it is a language which marries functional programming and Datalog concepts. Datafun comes with an explicit fixpoint operator that can be used to mark specific functions where fixpoint computation shall happen. This is in contrast to QL or Functional IncA where fixpoint computation is ultimately determined by the dependencies between predicates/functions.

8 CONCLUSIONS

This study successfully showed that incrementalization has potential when it comes to speeding up production CodeQL analyses. First, we showed this empirically with our impact measurements, and then we also demonstrated this by creating our prototype incremental solver. iQL delivers updated feedback in sub-minute run time for commit-style changes. The price of the fast update time is the high initialization time and memory use. While the initialization time is acceptable, the memory use requires further work before incrementalization can make its way to production CodeQL.

ACKNOWLEDGEMENT

Thanks to André Pacak, Ian Wright, Gábor Bergmann, Matthias Plappert, Max Schäfer, Nathan Corbyn, Oege de Moor, Rahul Pandita, and the members of the CodeQL Core team for helpful feedback and discussions on this work.

REFERENCES

[1] Michael Arntzenius and Neelakantan R. Krishnaswami. 2016. Datafun: A Functional Datalog. *SIGPLAN Not.* 51, 9 (sep 2016), 214–227. https://doi.org/10.1145/3022670.2951948

[2] Steven Arzt and Eric Bodden. 2014. Reviser: Efficiently Updating IDE-/IFDS-based Data-flow Analyses in Response to Incremental Program Changes. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 288–298. https://doi.org/10.1145/2568225.2568243

[3] 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) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

[4] Catriel Beeri and Raghu Ramakrishnan. 1991. On the Power of Magic. *J. Log. Program.* 10, 3–4 (mar 1991), 255–299. https://doi.org/10.1016/0743-1066(91)90038-Q

[5] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. 2015. Viatra 3: A Reactive Model Transformation Platform. In *Theory and Practice of Model Transformations*, Dimitris Kolovos and Manuel Wimmer (Eds.). Springer International Publishing, Cham, 101–110.

[6] Eric Bodden. 2018. The Secret Sauce in Efficient and Precise Static Analysis: The Beauty of Distributive, Summary-Based Static Analyses (and How to Master Them). In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops* (Amsterdam, Netherlands) *(ISSTA '18)*. Association for Computing Machinery, New York, NY, USA, 85–93. https://doi.org/10.1145/3236454.3236500

[7] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[8] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6, Article 26 (Dec. 2011), 66 pages. https://doi.org/10.1145/2049697.2049700

[9] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 332–343. https://doi.org/10.1145/2970276.2970347

[10] Oege de Moor. 2013. Doing a Doaitse: Simple Recursive Aggregates in Datalog. In *Liber Amicorum for Doaitse Swierstra*. 207–216. https://webspace.science.uu.nl/~hage0101/liberdoaitseswierstra.pdf

[11] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. 2017. Just-in-Time Static Analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 307–317. https://doi.org/10.1145/3092703.3092705

[12] Michael Eichberg, Matthias Kahl, Diptikalyan Saha, Mira Mezini, and Klaus Ostermann. 2007. Automatic Incrementalization of Prolog Based Static Analyses. In *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages* (Nice, France) *(PADL'07)*. Springer-Verlag, Berlin, Heidelberg, 109–123. https://doi.org/10.1007/978-3-540-69611-7_7

[13] Sebastian Erdweg, Oliver Bracevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. 2015. A co-contextual formulation of type rules and its application to incremental type checking. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 880–897. https://doi.org/10.1145/2814270.2814277

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

[15] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (Washington, D.C., USA) *(SIGMOD '93)*. Association for Computing Machinery, New York, NY, USA, 157–166. https://doi.org/10.1145/170035.170066

[16] Nicholas Hollingum and Bernhard Scholz. 2017. Cauliflower: a Solver Generator for Context-Free Language Reachability. In *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning (EPiC Series in Computing, Vol. 46)*, Thomas Eiter and David Sands (Eds.). EasyChair, 171–180. https://doi.org/10.29007/tbm7

[17] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 672–681.

[18] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 422–430.

[19] Uday Khedker. 1995. *A Generalised Theory of Bit Vector Data Flow Analysis*. Ph. D. Dissertation. Department of Computer Science and Engineering, IIT Bombay.

[20] Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. 2013. An Incremental Points-to Analysis with CFL-Reachability. In *Compiler Construction (Lecture Notes in Computer Science)*, Ranjit Jhala and Koen De Bosschere (Eds.). Springer, Berlin, Heidelberg, 61–81. https://doi.org/10.1007/978-3-642-37051-9_4

[21] Pavel Avgustinov Max Schäfer and Oege de Moor. 2017. Algebraic Data Types for Object-oriented Datalog. https://codeql.github.com/publications/algebraic-data-types.pdf.

[22] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. 2010. Parallel Inclusion-Based Points-to Analysis. *SIGPLAN Not.* 45, 10 (Oct. 2010), 428–443. https://doi.org/10.1145/1932682.1869495

[23] Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. 2014. I3QL: Language-Integrated Live Data Views. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 417–432. https://doi.org/10.1145/2660193.2660242

[24] André Pacak and Sebastian Erdweg. 2022. Functional Programming with Datalog. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 7:1–7:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.7

[25] André Pacak, Sebastian Erdweg, and Tamás Szabó. 2020. A Systematic Approach to Deriving Incremental Type Checkers. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 127 (nov 2020), 28 pages. https://doi.org/10.1145/3428195

[26] L. L. Pollock and M. L. Soffa. 1989. An Incremental Version of Iterative Data Flow Analysis. *IEEE Trans. Softw. Eng.* 15, 12 (Dec. 1989), 1537–1549. https://doi.org/10.1109/32.58766

[27] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 598–608.

[28] Diptikalyan Saha and C. R. Ramakrishnan. 2005. Incremental and Demand-driven Points-to Analysis Using Logic Programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Lisbon, Portugal) *(PPDP '05)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/1069774.1069785

[29] Diptikalyan Saha and C. R. Ramakrishnan. 2005. Incremental and Demand-Driven Points-to Analysis Using Logic Programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Lisbon, Portugal) *(PPDP '05)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/1069774.1069785

[30] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 22:1–22:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.22

[31] Tamás Szabó. 2023. Incrementalizing Production CodeQL Analyses. *arXiv e-prints*, Article arXiv:2308.09660 (Aug. 2023). arXiv:2308.09660 [cs.SE]

[32] Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental Whole-Program Analysis in Datalog with Lattices. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3453483.3454026

[33] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: A DSL for the Definition of Incremental Program Analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 320–331. https://doi.org/10.1145/2970276.2970298

[34] Tamás Szabó. 2021. *Incrementalizing Static Analyses in Datalog*. Ph. D. Dissertation. Mainz. https://doi.org/10.25358/openscience-5613

[35] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. 2010. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *2010 Asia Pacific Software Engineering Conference*. 336–345. https://doi.org/10.1109/APSEC.2010.46

[36] David Zhao, Pavle Subotic, Mukund Raghothaman, and Bernhard Scholz. 2021. Towards Elastic Incrementalization for Datalog. In *23rd International Symposium on Principles and Practice of Declarative Programming* (Tallinn, Estonia) *(PPDP 2021)*. Association for Computing Machinery, New York, NY, USA, Article 20, 16 pages. https://doi.org/10.1145/3479394.3479415