Datalog Engine Final Report

Privacy Tools Summer Research Project

August 14, 2015

Student: Aaron Bembenek

Mentors: Stephen Chong and Marco Gaboardi

Summary

Our primary goal for the summer was to address the lack of a good, open-source, Java-based implementation of the logic programming language Datalog. Furthermore, our implementation would ideally take advantage of multiple cores during query evaluation and could easily be extended with both new evaluation engines and new language features. As secondary goals, we also hoped to create a Datalog tutorial that could be used in introductory programming language courses and to implement a language extension, such as types or hypotheticals.

Over the summer we wrote five single-threaded Datalog evaluation engines of varying sophistication, as well as an engine for the extended language DatalogHypothetical. We are currently implementing optimizations for some of the pure Datalog engines and we expect to soon write a concurrent engine that uses the work-stealing thread scheduling technique. We have also created a lightweight graphical user interface for editing and interpreting Datalog code. While it does not have the functionality needed for serious Datalog development, we foresee the GUI being a useful pedagogical tool for such courses as CS 152.

Pure Datalog

The first part of this summer was spent implementing the most popular Datalog evaluation strategies. We felt that having a strong understanding of the existing single-threaded engines would pay dividends when developing our concurrent engine. Existing Datalog evaluation techniques fall into three main categories: bottom-up, top-down, and program rewriting. Bottom-up engines derive all possible facts (i.e. saturate) without reference to any particular query. Top-down engines attempt to only derive facts necessary for answering a specific query. Program rewriting techniques transform a program so that it can be more efficiently processed by one of the other engines. Accordingly, we implemented two bottom-up (naive and semi-naive), two top-down (recursive and iterative query-subquery), and one program rewriting (magic set transformation) engine. These represent the evaluation algorithms most commonly seen in the Datalog literature (see, for instance, [1], [3], and [6]).

All of the engines we wrote implement the same interface, so that client code can easily switch between them. While the bottom-up engines were relatively easy to implement, the top-down engines used surprisingly complicated algorithms and required some extended attention to get right. The magic set transformation engine followed easily from the work done on the top-down engines. We have focused much of our recent attention on improving the performance of the semi-naive engine. As a result of this focus, the top-down engines (and naive engine, although its use is unlikely) would need substantial polishing before any deployment. There are two reasons for this focus: the semi-naive engine is used internally in the magic set transformation engine (which is more streamlined than the top-down engines while having many of their benefits), and we plan to design our concurrent engine so that it will behave like a parallelized semi-naive engine. The concurrent engine will also be used in conjunction with magic set transformation (i.e., a program would first be transformed, and then given to the concurrent engine). While Datalog is P-complete and therefore likely inherently sequential [5], we are hoping that our work-stealing-based concurrent engine performs well on typical Datalog programs and in comparison to some of the pre-existing concurrent engines, which use more complex concurrency designs.

In addition to writing a concurrent engine, there are two other obvious directions for future work: performance-tuning of the engines we currently have, and language extension. In terms of the former, we could optimize our object usage for Java garbage collection, and we could also try to apply some of the optimizations used in query languages based in relational algebra (such as reordering atoms within rules, which corresponds to reordering relational algebra operations). In terms of the latter, there are standard extensions (such as stratified negation) that we could introduce, as well as non-standard extensions such as types.

Additionally, there are some open questions that might lead to interesting future work. For instance, is there any advantage to evaluation engines memoizing results from past queries? Given a Datalog program, are there heuristics for dynamically tweaking the evaluation engine to be a “better fit” for that program? Would it ever make sense to evaluate some queries (or subqueries) using one method and other queries using a different method?

We currently provide three interfaces through which are engines can be accessed. The first one is the Java API itself. The second one is a very basic read-evaluate-print loop (REPL) that allows the user to supply a file containing Datalog code as a command-line argument or enter code directly into the top-level. Once the code is loaded by the interpreter, the user can ask arbitrary queries. The final interface is a graphical user interface that contains an editing window and an interactive query field, and supports basic operations such as opening and saving files containing Datalog code.

DatalogHypothetical

During the second part of the summer we implemented an engine for the evaluation of the language DatalogHypothetical, which extends Datalog with hypothetical reasoning. Whereas in

pure Datalog the body of a rule consists of a lists of atoms, in DatalogHypothetical rule bodies are lists of premises, where a premise is either an atom or a hypothetical. A hypothetical consists of two atoms, one of which is a hypothesis and the other of which is a conclusion. If A and B are atoms, a hypothetical involving A and B could be "A > B", where A is the hypothesis and B is the conclusion. Informally, this hypothetical means, "Treat B as true if it would be true assuming A." A more exact description of the semantics of DatalogHypothetical can be found in [2] and [4].

DatalogHypothetical is a much more powerful language than pure Datalog: it is PSPACE-complete, as opposed to P-complete (by way of comparison, Prolog is Turing-complete) [2]. The semantics are also more complex (especially in the predicate case), and it took some time to come to a reasonable understanding of how the language works. The most naive way to implement a DatalogHypothetical engine would be to use a bottom-up, saturating engine similar to the naive evaluation of pure Datalog except that multiple evaluation frames are used (each frame is like a different evaluation instance). Whenever a hypothetical like "A > B" is encountered in the body of a rule, the current evaluation is paused as a new evaluation frame is pushed in which A has been added to the set of facts. If B is found to be true in this altered frame, we treat it as true in the hypothetical in the original frame. The advantage of this engine is that it only takes polynomial-space, which is what we would hope for. The disadvantage is that it has the potential to perform a lot of redundant work (i.e., the same work could be done in multiple frames).

We took a different tack for our engine. Our engine is based on the semi-naive engine for pure Datalog and only uses one evaluation frame. Instead of computing over facts, our engine computes over "conditional facts," where a conditional fact is a fact and an associated set of assumptions. The disadvantage of this technique is that it has the potential to take exponential space (the only way it can take super-polynomial time is if it takes super-polynomial space, and we know that DatalogHypothetical is inherently super-polynomial in its running time). Since DatalogHypothetical is PSPACE-complete, using super-polynomial space is obviously not optimal. On the other hand, in our limited testing we have found that it performs well in some cases where the first technique struggles. For instance, there are programs that take exponential time using the first technique that only take polynomial time using our technique.

For future work, it would be interesting to further explore the characteristics of the engine we have built. In particular, it would be instructional to find a test case that invokes exponential time (and space) use. Alternatively, we could design an engine that uses the frame technique but with smart optimizations, such as some form of memoization. Another direction would involve adding support for more sophisticated hypotheticals. Notably, our engine currently only supports additive hypotheticals (i.e., what can we infer if we assume this fact is in the database?), while there is also the potential to support deletive hypotheticals (i.e., what can we infer assuming that this fact is *not* in the database?). Datalog extended with both additive and deletive hypotheticals is EXPTIME-complete and allows for a form of counterfactual reasoning that might be helpful in certain situations [2].

We currently offer two interfaces to our DatalogHypothetical engine, the first being the actual Java API and the second being a very basic REPL that allows a user to ask arbitrary queries once she has entered her DatalogHypothetical code directly into the top-level.

Conclusion

During the summer we developed a good Datalog implementation that can be used as a foundation for future work, and in the process we deepened our knowledge of the properties of Datalog and the existing approaches to Datalog evaluation. With this experience, we anticipate that we will be able to make some exciting contributions to the Privacy Tools project. We foresee the main application of Datalog to be to DataTags, and within DataTags, there are two areas that are particularly promising: the declarative encoding of the submission questionnaire and the fine-grained modeling of legislation.

When a user submits a dataset to a repository, he must complete a questionnaire that is used to assign a DataTag to that dataset. This questionnaire is currently encoded procedurally as a decision tree, which makes it hard to maintain and to modify. There is the potential to use Datalog (or some extension thereof) to encode the questionnaire declaratively. That is, a logic programming language like Datalog would allow us to abstract away the actual procedure of evaluating the questionnaire, making it easier to change the questionnaire without worrying about updating the low-level mechanics of evaluation. To make this successful, we would need to figure out a way to make Datalog work with the type of real-time interaction involved with questionnaire processing (i.e. facts are added by querying the user, and then the program needs to decide which question to ask next). Having our own Datalog implementation will make it easier to address these challenges.

A data tag currently gives a broad notion of actions that are permitted and not permitted on a dataset. But, given the specifics of a dataset, there might be certain actions that are allowed even though the tag says they are not. Datalog might allow us to better capture some of these details by modeling privacy legislation. These models, combined with facts about the dataset, would potentially give us the capability to infer that certain actions are in fact permissible. The challenges we face here are the difficulty of formalizing legislation (and how to treat ambiguity in the legislation, or multiple interpretations of the legislation), and extending Datalog in such a way that it can handle the logic necessary for the formulations. Once again, having our own Datalog implementation as a playground will make it easier to investigate solutions to these problems.

Additional documentation for this project can be found in our previous design document and the Javadocs for the code, the latter of which are continually maintained.

References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley, Reading, MA.
2. Anthony J. Bonner. 1990. Hypothetical datalog: complexity and expressibility. *Theor. Comput. Sci.* 76, 1 (October 1990), 3-51. DOI=10.1016/0304-3975(90)90011-6 http://dx.doi.org/10.1016/0304-3975(90)90011-6
3. S. Ceri, G. Gottlob, and L. Tanca. 1989. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. on Knowl. and Data Eng.* 1, 1 (March 1989), 146-166. DOI=10.1109/69.43410 http://dx.doi.org/10.1109/69.43410
4. Henning Christiansen and Troels Andreasen. 1996. A Practical Approach to Hypothetical Database Queries. In *International Seminar on Logic Databases and the Meaning of Change, Transactions and Change in Logic Databases* (ILPS '97), Burkhard Freitag, Hendrik Decker, Michael Kifer, and Andrei Voronkov (Eds.). Springer-Verlag, London, UK, UK, 340-355.
5. Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv.* 33, 3 (September 2001), 374-425. DOI=10.1145/502807.502810 http://doi.acm.org/10.1145/502807.502810
6. Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. 2013. Datalog and Recursive Query Processing. *Found. Trends databases* 5, 2 (November 2013), 105-195. DOI=10.1561/1900000017 http://dx.doi.org/10.1561/1900000017