Datalog Implementation in JastAdd (OUTLINE)

Hampus Balldin
C13, Lund University, Sweden
dat12hba@student.lu.se

Abstract

Text of abstract ....

1 Introduction

Datalog is a syntactically simple declarative language that allows the expression and evaluation of certain first-order logic propositions. From its inception in the nineteen-eighties it saw substantial interest from the academic community into the early nineteen-nineties[6]. The effort's primary drive were to enable knowledge based systems that allowed the generation of new facts based on rules stated using the logic programming paradigm. At the time, this had applications in both artificial intelligence as well as a complement to the traditional relational database querying systems such as SQL[4][2].

After a time of cooling interest, Datalog has emerged again as an attractive way to express complex inter-dependencies[6]. A notable example is from Program Analysis where frameworks such as Doop[8] make use of Datalog to derive e.g. call-graph and points-to information, both of which typically have mutually recursive dependencies in languages using dynamic dispatch.

There are currently many Datalog implementations, including Souffle[7], LogiQL[1], Iris[3], and BDDBDDB[9]. The implementations provide different evaluation methods and different extensions to the core Datalog language. This paper describes a common front-end that allows for source-to-source compilation. A common front-end enables a convenient way to compare the performance and expressive power of different Datalog implementations. An additional goal is to provide a library that can be used to implement and evaluate new inference schemes.

JastAdd [5] is a meta-compilation system that enables the expression of arbitrary graphs on top of an abstract syntax tree (AST). Information is propagated through the AST using so-called Reference Attribute Grammars[5]. JastAdd also supports aspects, which allow the weaving of methods and class fields from different source locations into a single generated class. This enables easy extension of the generated AST classes with additional properties. In particular, it nicely permits the incremental addition of support for source-to-source compilation to different Datalog implementations.

Course paper, EDAN70, Lund University, Sweden
December 5, 2018.

1.1 Core Language

There are many flavors of the Datalog language but they all build on a common core.

A *program* P consists of a set of *Horn clauses* $H_1, \dots H_n$. A horn clause has a *head* and a *body*. The head is a single *atom* and the body is a sequence of atoms. An atom is identified by a *predicate symbol* and a sequence of *terms*. An example of a propositional rule (i.e. a rule with atoms that have no terms, such a term-less atom thus represents a relation of arity zero) is shown below:

$$A \text{ :- } B_1, B_2, \dots B_m$$

Above we have a single horn clause (hereafter called a *rule*). It has head A and body $B_1, B_2, \dots B_m$. The intuitive meaning of the above rule is that if the conjunction of all the atoms in the body are true then we conclude A.

Datalog deals not only with propositional rules, but allows a restricted range of first-order propositions where each atom is associated with a sequence of terms. A term is either *variable* or *constant*. An atom that contain only constant terms is called a *ground atom*. We further partition the predicates into extensional (EDB) and intensional (IDB). The EDBs are all predicates that are taken as input from an external database. The IDBs are the predicates that are not EDB and are intensionally defined through rules. The EDBs introduce *facts*, i.e. ground atoms. One may additionally introduce facts by declaring a rule with a ground atom head that has an empty body.

- The set of all constants in all facts is called the *domain* and is denoted Ω.
- The set of all facts is called the *active database instance* and is denoted I.

There are three main semantic interpretations of Datalog: model-, fixpoint-, and proof-theoretic semantics. [6]. A brief overview is given below.

Model-theoretic Semantics

A *model* of a Datalog program P is a consistent (satisfying all rules of P) extension of the initial EDB facts. Each rule is interpreted as a universally quantified rule. For example, below is given a rule and its' corresponding semantic interpretation.

$$A(x, y, "C") \text{ :- } B_1(x, "C"), B_2("C", y), B_3(x, y)$$

$$\forall c_1 \in \Omega.\ \forall c_2 \in \Omega.\ \ B_1(c_1, "C"), B_2("C", c_2), B_3(c_1, c_2) \implies A(c_1, c_2, "C")$$

An inference algorithm attempts to find the *minimal model*, i.e. a model m of P such that for any other model m' of

P, all facts of m are in m'. In practice this means that an inference algorithm should only add a fact if it is required by the semantics of a rule (even if adding the fact may lead to an extended model of P).

Fixpoint-theoretic Semantics

Begin with the set of all facts in the active database instance I^0. The set of new facts that can be derived (under model-theoretic semantics) using the rules of a program P and the existing facts in I^i is denoted Δ_i. We get the following inductive definition of I:

$$I^0 = \{\text{EDB Facts in } P\}$$
$$I^{i+1} = I^i \cup \Delta_i$$

It can be shown(CITE) that the minimal model is computed as I^n for n such that $I^n = I^{n+1}$. Since $I^i \subseteq I^{i+1}$ (monotonically increasing) and with the practical assumption of a finite domain, the fix-point algorithm is guaranteed to terminate.

Proof-theoretic Semantics

Consider a ground atom $A(C_1, \ldots, C_n)$. A query for the ground atom asks for a proof that $A(C_1, \ldots, C_n)$ is in the minimal model of a program P. A proof can be visualized as an *and-or tree* T. T has the proposition (ground atom) to prove as the root. At an OR-node, all possible rules are tested. If any of them succeed then the proposition has been proven. At an AND-node, all the child propositions need to be proven for the node to become true. An example is shown in figure 1.

$$r_1 : A \text{ :- } B, C$$
$$r_2 : A \text{ :- } B, D$$

When traversing the tree the model is updated with new

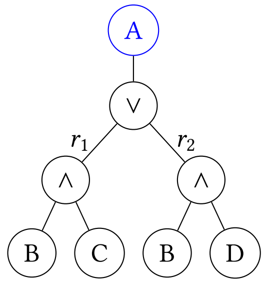

Figure 1. An and-or tree for rules $r_1 : A \text{ :- } B, C, r_2 : A \text{ :- } B, D$

facts that are needed to prove the root proposition. Those facts then become part of the extended model.

1.2 Security Features and Time Complexity

Above a fact was stated to be a ground atom that is true in a given model. An axiomatic fact can be declared as a rule with no body: $A(t_1, \ldots, t_n)$. Datalog disallows axiomatic facts that contain variables. This is implied by the following more general rule: all variables occurring in the head of a rule must also occur in the body of the rule. This is called the *range restriction property* [6].

2 Core Language Implementation

The current query evaluation mechanism is a bottom-up naive (BUN) [6] evaluation. It is based on the fixpoint-theoretic semantics. The implementation is best introduced through an example.

$$Order(1, 2),\ Order(2, 3). \qquad [r_1]$$
$$Order(x, z) \text{ :- } Order(x, y),\ Order(y, z). \qquad [r_2]$$

Rule $[r_1]$ states that the orders $1, 2$ and $2, 3$ holds. Rule $[r_2]$ states that the binary $Order$ relation is transitive. The BUN evaluation proceeds as follows. For each rule that is evaluated, an artificial body-relation B is introduced. The body-relation will be incrementally populated and extended through the evaluation of the atoms in the body.

Initially, the relation associated with each atom is unnamed, i.e. the columns of the relation have no name-restrictions. Given a rule r, we order the atoms from the body of the rule as $A_1, \ldots, A_n$. We will consider each atom in turn and use the body relation B as an accumulator. The equations describing the rule-evaluation process is as follows (the notation is explained below)

$$B^0 \leftarrow \top$$
$$B^{i+1} \leftarrow B^i \bowtie \sigma_{Term(A_i)}\ (A_i),\ i = 0, \ldots, n-1$$
$$H \leftarrow \Pi_{Term(H)}(B_n)$$

The body relation is initially assigned to $\top$ denoting an unknown relation: $\top \bowtie R = R \bowtie \top = R$. A selection is then done for the atom. The selection operator is denoted σ as is usual in relational algebra (CITE). The selection result is *joined*(CITE) with the current body relation B^i to form the next body relation B^{i+1}. The join-operator is denoted $\bowtie$. The $Term$ function gives the terms in the given atom. The project function Π projects the corresponding columns into a new relation. In our example (assuming that r_1 has been evaluated) we get:

$$B^1 \leftarrow \sigma_{x,y}\ (Order) = \{(1, 2), (2, 3)\}_{x,y}$$
$$B^2 \leftarrow B^1 \bowtie (\sigma_{y,z}\ (Order) = \{(1, 2), (2, 3)\}_{y,z})$$
$$= \{(1, 2), (2, 3)\}_{x,y} \bowtie \{(1, 2), (2, 3)\}_{y,z}$$
$$= \{(1, 2, 3)\}_{x,y,z}$$

Finally we project the final result of the body relation into the head of the rule:

$$Order \leftarrow \Pi_{x,z}(\{(1, 2, 3)\}_{x,y,z}) = \{(1, 3)\}$$

The process is iterated until a fix-point is found, i.e. until no new tuples can be derived from the set of rules. In our example, iterating $[r_2]$ again gives no new tuples and so the $Order$ relation has been computed as: $\{(1, 2), (1, 3), (2, 3)\}$.

2.1 Mutual Dependencies and Predicate Ordering

3 Language Extensions

3.1 Type System

3.2 Negation

3.3 Object Creation

3.4 Built-in Predicates

4 Evaluation

I want to use the following evaluation criteria

- Front-end extensibility
- Front-end source-to-source coverage
- Back-end implementations performance evaluation

5 Related work

5.1 LogicBlox

5.2 Iris

5.3 Souffle

5.4 BDDBDDB

6 Conclusion

Acknowledgments

Text of acknowledgments . . . .

References

[1] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1371–1382. https://doi.org/10.1145/2723372.2742796

[2] Francois Bancilhon and Raghu Ramakrishnan. 1986. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data (SIGMOD '86)*. ACM, New York, NY, USA, 16–52. https://doi.org/10.1145/16894.16859

[3] Barry Bishop and Florian Fischer. [n. d.]. IRIS- Integrated Rule Inference System. ([n. d.]).

[4] 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. https://doi.org/10.1109/69.43410

[5] Torbjörn Ekman and Görel Hedin. 2007. The Jastadd Extensible Java Compiler. *SIGPLAN Not.* 42, 10 (Oct. 2007), 1–18. https://doi.org/10.1145/1297105.1297029

[6] 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

[7] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-scale Program Analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. ACM, New York, NY, USA, 196–206. https://doi.org/10.1145/2892208.2892226

[8] Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for Fast and Easy Program Analysis. In *Proceedings of the First International Conference on Datalog Reloaded (Datalog'10)*. Springer-Verlag, Berlin, Heidelberg, 245–251. https://doi.org/10.1007/978-3-642-24206-9_14

[9] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Proceedings of the Third Asian Conference on Programming Languages and Systems (APLAS'05)*. Springer-Verlag, Berlin, Heidelberg, 97–118. https://doi.org/10.1007/11575467_8