Electronic Communications of the EASST
Volume 35 (2010)

Proceedings of the 10th International Workshop on Automated Verification of Critical Systems (AVoCS 2010)

Evaluation Strategies for Datalog-based Points-To Analysis

Marco-A. Feliú, Christophe Joubert, and Fernando Tarín

18 pages

Guest Editors: Jens Bendisposto, Michael Leuschel, Markus Roggenbach
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Evaluation Strategies for Datalog-based Points-To Analysis

Marco-A. Feliú*, Christophe Joubert, and Fernando Tarín†

mfeliu,joubert,ftarin@dsic.upv.es
Universidad Politécnica de Valencia, DSIC / ELP
Camino de Vera s/n, 46022 Valencia, Spain

Abstract: During the last decade, several hard problems have been described and solved in Datalog in a sound way (points-to analyses, data web management, security, privacy, and trust). In this work, we describe novel evaluation strategies for this language within the context of program analyses. We first decompose any Datalog program into a program where rules have at most two atoms in their body. Then, we show that a specialized bottom-up evaluation algorithm with time and memory guarantees can be described as the on-the-fly resolution of a Boolean Equation System (BES). The resolution computes all ground atoms in an efficient way thanks to a compact data structure with constant time access that has so far not been used in the Datalog or the BES literature. A prototype has been developed and tested on a number of real Java projects in the context of Andersen's points-to analysis. For this specific points-to analysis, experimental results show that our prototype is several orders of magnitude faster than state-of-the-art solvers like BDDBDDB or XSB, and an order of magnitude better that the novel strategy of Liu and Stoller [LS09] both in execution time and memory consumption.

Keywords: datalog; bottom-up evaluation; boolean equation system; program analysis

1 Introduction

The subset of Prolog that is called Datalog was first used in the late 70's [GM78]. It was further popularized in the late 80's by Ullman [Ull89] and Abiteboul [AHV95] with a direct application to database queries. At the time it was defined, Datalog was a very simple language that had a higher expressiveness than other standardized relational languages, such as Sql, which only supported recursive queries in its fourth revision (Sql:1999). Several extensions of the initial Datalog language have been made since then, namely the inclusion of negation, disjunctions in the head of the rules (DLV) [1], constraints, as well as new Datalog-based languages like Axlog [ABM09], Elog [BFG01], and SociaLite [SSN+10].

We have observed a resurgence of Datalog in different computer science communities over the last decade. Datalog has been used in a number of non-trivial analysis problems referenced

* This author is partly sponsored by the Spanish MEC FPU grant AP2008-00608.
† This work has been partially supported by the EU (FEDER), the Spanish MEC/MICINN under grants TIN 2007-68093-C02 and TIN 2010-21062-C02-02 and the Generalitat Valenciana under grant Emergentes GV/2009/024.
[1] http://www.dbai.tuwien.ac.at/proj/dlv/

in [LS09] such as pointer analysis, simplification of regular tree grammar-based constraints, trust management, path queries, model checking, security frameworks, queries of semi-structured data and social networking. Datalog is now popular in companies that currently sell solutions based on Datalog, such as LogicBlox, Semmle Ltd., Lixto, and Exeura[2].

In this paper, we are interested in the use of Datalog to detect critical errors in system software (such as memory leaks or Sql injections) by static analysis of conservative approximate pointer information. Static analysis techniques [ALSU07] like abstract interpretation are a powerful means for finding code errors before actually running the program. They have also been successfully used in conjunction with model checking, by making the program more abstract in order to save time and memory during its verification [GJMS07]. Among the most renowned static analyses are the *pointer analyses*, which compute relationships between program pointers and memory locations. These analyses appear in many program verification and optimization problems. In this paper, we will introduce novel evaluation strategies for solving Datalog programs and apply them to the problem of pointer analysis.

Since the advent of the first *naïve* algorithm to solve Datalog programs, numerous optimizations have appeared in the literature. BDDBDDB [WL04] is a recent tool that uses an implicit BDD-based representation to solve Datalog programs. It is an efficient solver in practice, yet its worst case complexity is linear with the cardinality of the cartesian product of the argument domains. Liu and Stoller have recently defined an evaluation strategy based on an explicit representation of Datalog programs that offers (lower) complexity guarantees [LS09]. The strategy is a generalization of the systematic algorithm development method of Paige et al. [PK82], which transforms extensive set computations like set union, intersection, and difference into incremental operations. Incremental operations are supported by sophisticated data structures with constant time access. An imperative resolution algorithm is derived, and it computes a fixed point over all (preformatted) rules by first considering input predicates, then considering rules with one subgoal, and finally considering rules with two subgoals.
We propose novel evaluation strategies based on the following:

1. A declarative description of Liu and Stoller's bottom-up resolution strategy that is separate from the fixed-point computation. This is achieved by transforming Datalog programs to *Boolean Equation Systems* (BESs) and evaluating the resulting BESs by standard solvers. BESs have been used successfully in the formal verification of asynchronous systems [And94a, VL92, Mad97]. The main features of this formalism are the following: it is concise (simple list of boolean equations); it relies on fixed-point operators; and there exist linear time and memory complexity algorithms to solve alternation-free BESs [Mat06].

2. A simplification of the resulting BES based on the dependency between predicate symbols for a given Datalog program. This predicate order graph allows us to remove various set operations during the construction of the BES.

3. A sophisticated data structure with worst-case constant access and compact representation of the underlying data. This efficient data structure is based on prefix tree structures, lists and bitmaps. This structure has faster look-up keys than binary search trees and imperfect hash tables commonly used in the Datalog literature.

[2] www.logicblox.com, www.semmle.com, www.lixto.com, www.exeura.com

The rest of the paper is organized as follows: Section 2 introduces both the syntax and semantics of the Datalog language considered in this paper. It also gives a running example, namely Andersen's points-to analysis, which illustrates our approach throughout the different sections. Section 3 introduces our BES approach, and Section 4 presents an algorithm that computes an evaluation order over the predicates. Section 5 details the data structure that efficiently supports the evaluation strategy. Section 6 compares the experimental results for the points-to analysis of real Java programs performed by different state-of-the-art Datalog solvers. Finally, Section 7 concludes and gives future directions of research.

2 Datalog Programs

Datalog has a simple and clear syntax and semantics. A Datalog program is composed of a finite set of declarative rules to both describe and query a deductive database.

Definition 1 (Syntax of Rules)

$$p_0(a_{0,1},\ldots,a_{0,n_0}) \ :- \ p_1(a_{1,1},\ldots,a_{1,n_1}),\ \ldots,\ p_m(a_{m,1},\ldots,a_{m,n_m}).$$

where each p_i is a predicate symbol of arity n_i with arguments $a_{i,j}$ ($j \in [1..n_i]$) that are either constant or variable.

The atom $p_0(a_{0,1},\ldots,a_{0,n_0})$ in the left-hand side of the rule is the rule's *head*. The finite conjunction of *subgoals*, also called *hypotheses*, in the right-hand side of the formula is the rule's *body*, *i.e.*, atoms that contain all variables appearing in the head. Following logic programming terminology, a rule with empty body ($m = 0$) is called a *fact*. The set of facts is called the *extensional database*, whereas the set of ground atoms *inferred* from rules is called the *intensional database*. For ease of exposition, we will consider Datalog programs with finite sets of rules and facts, and with rules that do not contain negated hypotheses, although the evaluation strategy would work for rules with stratified negation [LS09]. In the rest of the paper, we will follow the logic programming criteria where variables are described in capital letters and where predicate symbols and constants are described in lower-case letters.

Definition 2 (Fixed point semantics) [Ull89, AHV95] Let R be a Datalog program. The least *Herbrand model* of R is a Herbrand interpretation I of R defined as the least fixed point of a monotonic, continuous operator $T_R : \mathscr{I} \to \mathscr{I}$ known as the *immediate consequences operator* and defined by:

$$\begin{aligned} T_R(I) \quad = \quad & \{q \in B_R \mid \ q :- b_1,...,b_m \text{ is a ground instance of a rule in } R, \\ & \text{with } B_R \text{ the Herbrand base, } \ b_i \in I, i = 1..m, m \geq 0\} \end{aligned}$$

Following the approximation of [LS09], some auxiliary definitions are necessary: a variable that occurs multiple times in a hypothesis is called an *equal card*, and a variable that occurs only once and in only one hypothesis but not in the head of a rule is called a *wild card*. In the remainder of the paper, our derivation of the specialized Datalog evaluation algorithm will be applied to a restricted form of Datalog defined as follows: rules have at most two hypotheses; *equal cards* and *wild cards* can only occur in rules with one hypothesis; and facts must be ground.

There exists a decomposition from any non-restricted Datalog program into a restricted one. This decomposition does not affect the guaranteed worst-case time or space complexities of the original program evaluation. Grouping and reordering of predicate arguments do not affect the complexities, either. We will use this rule format to describe the bottom-up evaluation strategy in terms of BES in the next section.

Definition 3 (Restricted Datalog Rules) Rules and facts have exactly the following forms:

`form 2:` $\mathbf{q_2}(\mathbf{X_{1s}}, \mathbf{X_{2s}}, \mathbf{Y'_s}, \mathbf{c_{3s}}) : -\mathbf{h_1}(\mathbf{X_{1s}}, \mathbf{Y_s}, \mathbf{c_{1s}}), \mathbf{h_2}(\mathbf{X_{2s}}, \mathbf{Y_s}, \mathbf{c_{2s}}).$

`form 1:` $\mathbf{q_1}(\mathbf{Z'_s}, \mathbf{b_s}) : -\mathbf{h}(\mathbf{Z_s}, \mathbf{a_s}).$

`form 0:` $\mathbf{q}(\mathbf{c_s}).$

where each X_{1s},X_{2s},Y_s,Y'_s,Z_s, and Z'_s abbreviates a group of variables; each c_{1s}, c_{2s}, c_{3s}, a_s, b_s, and c_s abbreviates a group of constants; variables in Y'_s and Z'_s are subsets of the variables in Y_s and Z_s respectively.

Example 1 (The Datalog-based Andersen points-to analysis.) *In his thesis [And94b], Andersen gave a type inference system of a flow- and context-insensitive, inclusion-based pointer analysis for* C *programs. This analysis is based on four kinds of assignment statements that involve pointers (object allocations* **vp0***, variable assignments* **a** *and store* **s** *or load* **l** *operations into structures and fields of structures). They can be represented as four declarative rules as follows [ALSU07]:*

```
vp(X, Y) :- vp0(X, Y). (1)
vp(X, Y) :- a(X, Z), vp(Z, Y). (2)
hp(Y, S, T) :- s(X, S, Z), vp(X, Y), vp(Z, T). (3)
vp(Z, T) :- l(X, S, Z), vp(X, Y), hp(Y, S, T). (4)
```

where **vp0**, **a**, **s**, *and* **l** *are extensional predicates and* **vp** *(variables that point to heap locations),* **hp** *(heap locations that point to other heap locations through object fields) are intensional predicates.*
We can note that Datalog *rules (3) and (4) are not in one of the three forms accepted by our formal derivation since their bodies have three hypotheses.*

Rules with more than two hypotheses must be decomposed first into rules with two hypotheses. This can be achieved by repeatedly inserting auxiliary predicates in rules with more than two hypotheses in order to convert them into rules with at most two hypotheses. For a given rule with `h` hypotheses, there exist $(2h-3)!!$ ways of decomposing it by means of a simple algorithm [LS09], with:

$$n!! = \begin{cases} 1, & \text{if } n = 0 \text{ or } n = 1 \\ n \times (n-2)!!, & \text{if } n \geq 2 \end{cases}$$

Time complexity of a Datalog program is given by the sum of firings over all rules. The total number of times a rule is fired can be computed in terms of predicate size, domain size, argument size, and relative argument size. Given a Datalog program with k rules r_i ($i \in [1..k]$) with h_i hypotheses ($h_i > 2$), the time complexity of searching the optimal decomposition process can be bounded by $O(k.(h_{max}!!))$, with h_{max} being the maximum number of hypotheses in a rule of the program.

Assuming that all sets are of the same size, a key aspect in minimizing the evaluation cost is the process of looking up rules of `form 2` based on the set of common variables in the hypotheses. If a decomposition maximizes the size of that set for every rule, then fewer sets have to be checked in order to obtain new solutions. In addition to that, if those sets are in the same order as in the original program, no reordering, also called *views*, of the sets of variables belonging to a hypothesis would be needed to solve the problem. The concept of views will be expanded in section 3 and section 5.

Example 2 *Given the* Datalog*-based Andersen points-to analysis of Example 1, one decomposition that satisfies the strategy presented above, is described as follows:*

```
vp(X, Y) :- vp0(X, Y). (1)
vp(X, Y) :- a(X, Z), vp(Z, Y). (2)
temp1(Z, Y, S) :- s(X, S, Z), vp(X, Y). (3)
hp(Y, S, T) :- temp1(Z, Y, S), vp(Z, T). (4)
temp2(Y, S, Z) :- l(X, S, Z), vp(X, Y). (5)
vp(Z, T) :- temp2(Y, S, Z), hp(Y, S, T). (6)
```

where `temp1` *and* `temp2` *are two newly inserted auxiliary predicates.*
All rules are now either in `form 1` *or* `form 2` *formats. As an example, we have decomposed rule* **hp** *as follows:*

1. *Choose a pair of predicates in* **hp**. *In this case:* $\mathbf{s}(\mathbf{X},\mathbf{S},\mathbf{Z})$ *and* $\mathbf{vp}(\mathbf{X},\mathbf{Y})$.
2. *Create a new rule with the chosen pair as body and a new predicate as header, whose variables are those that are not repeated in both predicates. In this case:* $\mathbf{temp1}(\mathbf{Z},\mathbf{Y},\mathbf{S}) : -\mathbf{s}(\mathbf{X},\mathbf{S},\mathbf{Z}),\mathbf{vp}(\mathbf{X},\mathbf{Y})$.
3. *Substitute the chosen pair with the head of the new created rule. In this case:* $\mathbf{hp}(\mathbf{Y},\mathbf{S},\mathbf{T}) : -\mathbf{temp1}(\mathbf{Z},\mathbf{Y},\mathbf{S}),\mathbf{vp}(\mathbf{Z},\mathbf{T})$.
4. *Repeat the process from step 1 if the rule still has more than two hypotheses.*

Since the original rule **hp** *has only three hypotheses, the rule's decomposition terminates after one iteration. Only one new auxiliary predicate has been created and both new and modified rules have exactly two hypotheses in their body.*

The problem considered in this paper is to efficiently compute the least set of ground atoms that can be inferred using the rules. In the case of the points-to analysis, we want to derive all ground atoms for predicates **vp** and **hp**.

3 BES Approach

In this section, we describe the bottom-up evaluation strategy for any restricted Datalog program at a high level by using parameterized Boolean equation systems (PBESs). We illustrate this evaluation strategy on the example of Andersen's points-to analysis. The PBES is instantiated

into a parameterless BES that is later solved. Finally, we propose an algorithm to compute a predicate evaluation order for a given Datalog program and show its usefulness to optimize the BES resolution.

3.1 Parameterized Boolean Equation System (PBES)

In this paper, we will use a restricted fragment of the PBES formalism [GM98], namely, single equation block PBESs. We first recall the definition of parameterless BES to later introduce its parameterized extension.

Definition 4 (Single block BES) Given $\mathscr{W}$ a set of boolean variables, a *Boolean Equation System* (BES) $B = (W_0, M)$ is defined as follows: $W_0 \in \mathscr{W}$ is a boolean variable whose value is of interest in the context of the local resolution methodology; M is a set of n fixed point equations of the form $W_i \stackrel{\sigma}{=} \phi_i$, where all W_i are different ($i \in [0..n]$). $\sigma \in \{\mu, \nu\}$ is the least (μ) or greatest (ν) fixed point operator. Each W_i is a boolean variable from $\mathscr{W}$ whose value is the value of the respective *boolean formula* ϕ_i. A *boolean formula* ϕ, defined over an alphabet of boolean variables $\mathscr{W}$, is an expression built with the following syntax given in positive form: $\phi ::= \mathsf{true} \mid \mathsf{false} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid W$ where boolean constants and operators have their usual semantics, ϕ_1 and ϕ_2 are boolean formulas, and W is a boolean variable. Empty conjunction (resp. disjunction) corresponds to boolean constant true (resp. false).

Definition 5 (Valuation of boolean variables and formulas) A valuation is a function v from a set of boolean variables $\mathscr{W}$ to $\{\mathsf{true}, \mathsf{false}\}$. It is extended over boolean formulas so that $v(\phi)$, being ϕ a boolean formula, is the value of the formula after substituting each free variable W in ϕ by $v(W)$.

Definition 6 (Semantics of a BES) The semantics of a BES B, denoted by $[\![B]\!]$, is the least (resp., greatest) fixed point valuation of the n boolean formulas ϕ_i of B.

There exist algorithms for solving single block BESs with a temporal and spatial complexity that is linear with respect to the total number of boolean variables and boolean operators of the BES [Mat06]. Thus, BESs allow us to simplify the definition of Datalog evaluation algorithms by only describing declarative aspects of the algorithm, and not operational aspects like fixed-point computation.

Definition 7 (Single block PBES) Single block PBESs are single block BESs where boolean variables have typed value parameters $D \subseteq \mathscr{D}$. Hence, boolean formulae have the following extended syntax given in positive form:

$$\phi, \phi_1, \phi_2 ::= \mathsf{true} \mid \mathsf{false} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid X(e) \mid \forall d \in D.\ \phi \mid \exists d \in D.\ \phi$$

where e is a data term (constant or variable of type D), $X(e)$ denotes the *call* of a boolean variable X with parameter e, and d is a term of type D.

The main advantages of specific symbolic encodings such as Binary Decision Diagrams (BDDs) over explicit encodings such as PBESs in the context of program analysis, is that BDDs

can efficiently compact the program predicates. However, the compaction highly depends on the regularity of the points-to analysis domain, the redundancy of the relations, and the boolean variable ordering. These factors may vary the execution time of the analysis by several orders of magnitude as we observed with the Andersen's points-to analysis with standard BDD and data mining libraries. The Datalog relations can be represented compactly with well adapted data structures (prefix trees structures, lists and bitmaps) for any relation ordering in the Datalog program. Moreover, explicit encodings may facilitate the design of distributed solutions and the integration of further optimizations.

3.2 PBES Evaluation Strategy

The meaning of a restricted Datalog program corresponds to the fixed-point computation of a set of ground atoms over a given set of rules and facts. In this evaluation strategy, we are interested in all the ground atoms that can be inferred from the Datalog program. Hence, our PBES will be described in terms of a greatest fixed-point computation (ν). Rules of the program are in one of the three forms described in Section 2. As in [LS09], we would like to impose a bottom-up evaluation in which only inferred ground atoms of the Datalog program are generated. Therefore, the PBES should start the computation from boolean variable W_0 by generating boolean variables that hold the given facts. This will constitute the first boolean formula of our PBES description. We will represent each ground atom $\mathbf{p}(\mathbf{c_s})$., namely facts and inferred ground atoms, as a parameterized boolean variable $W_1(p : D_p,\ c_s : D_{c_s})$ identified by predicate symbol p defined over a domain D_p of predicates, and constant arguments c_s, defined over a domain $D_{c_s} = D_{a_{p,1}} \times \ldots \times D_{a_{p,p_n}}$, which is the composition of the argument domains. Then, new boolean variables $W_1(q,\ d_s)$, with q, a predicate symbol, and d_s, constant arguments, will be generated for each parameterized boolean variable $W_1(p : D_p,\ c_s : D_{c_s})$ and each rule where predicate p appears as a hypothesis. This will constitute the second boolean formula of our PBES description. Both boolean equations are defined as follows:

$$
\begin{aligned}
W_0 &\overset{\nu}{=} \bigwedge_{\mathbf{q}(\mathbf{c_s}). \in \mathit{Facts}} W_1(q,\ c_s) \\
W_1(p : D_p,\ c_s : D_{c_s}) &\overset{\nu}{=} \bigwedge_{p\ \mathit{of}\ \mathbf{h}(\mathbf{Z_s},\mathbf{a_s})} W_1(q_1,\ (Z'_s, b_s)) \quad (\star 1 \star) \\
&\quad \wedge \bigwedge_{p\ \mathit{of}\ \mathbf{h_1}(\mathbf{X_{1s}},\mathbf{Y_s},\mathbf{c_{1s}})} \ \bigwedge_{\mathbf{X_{2s}} \in \mathbf{h_2}(\mathbf{X_{2s}},\mathbf{Y_s},\mathbf{c_{2s}}).} W_1(q_2,\ (X_{1s}, X_{2s}, Y'_s, c_{3s})) \quad (\star 2 \star) \\
&\quad \wedge \bigwedge_{p\ \mathit{of}\ \mathbf{h_2}(\mathbf{X_{2s}},\mathbf{Y_s},\mathbf{c_{2s}})} \ \bigwedge_{\mathbf{X_{1s}} \in \mathbf{h_1}(\mathbf{X_{1s}},\mathbf{Y_s},\mathbf{c_{1s}}).} W_1(q_2,\ (X_{1s}, X_{2s}, Y'_s, c_{3s})) \quad (\star 3 \star)
\end{aligned}
$$

Boolean formulas on the right hand-side of both equations are conjunctions of distinct W_1 boolean variables. In the first equation, one variable W_1 is generated for each fact of the Datalog program. In the second equation, for each hypothesis $\mathbf{h}$ of the Datalog program that matches the predicate parameter p on the left hand-side of the equation, one variable W_1 is generated for each conclusion q that can be inferred by the given rule. Since there are three types of hypotheses in the Datalog programs, the second boolean equation is divided into three parts:

$(\star 1 \star)$ In rules of `form 1`, $\mathbf{q_1}(\mathbf{Z'_s}, \mathbf{b_s})\ :-\ \mathbf{h}(\mathbf{Z_s}, \mathbf{a_s})$., a new ground atom $\mathbf{q_1}(\mathbf{Z'_s}, \mathbf{b_s})$. is inferred when p appears as the hypothesis $\mathbf{h}$ of the rule, and arguments $(\mathbf{Z_s}, \mathbf{a_s})$ of $\mathbf{h}$ can be sub-

stituted by constant parameters c_s. For each new ground atom, a new boolean variable $W_1(q_1, (Z'_s, b_s))$ is generated.

(⋆2⋆) In rules of `form 2`, $\mathbf{q_2}(\mathbf{X_{1s}}, \mathbf{X_{2s}}, \mathbf{Y'_s}, \mathbf{c_{3s}}) :- \mathbf{h_1}(\mathbf{X_{1s}}, \mathbf{Y_s}, \mathbf{c_{1s}}), \mathbf{h_2}(\mathbf{X_{2s}}, \mathbf{Y_s}, \mathbf{c_{2s}})$., where p appears in the first hypothesis $\mathbf{h_1}$, arguments $(\mathbf{X_{1s}}, \mathbf{Y_s}, \mathbf{c_{1s}})$ of $\mathbf{h_1}$ are substituted by constant parameters c_s. Then, for each value of $\mathbf{X_{2s}}$ such that $\mathbf{h_2}(\mathbf{X_{2s}}, \mathbf{Y_s}, \mathbf{c_{2s}})$. is a previously computed ground atom, a new ground atom $\mathbf{q_2}(\mathbf{X_{1s}}, \mathbf{X_{2s}}, \mathbf{Y'_s}, \mathbf{c_{3s}})$. is inferred, and $W_1(q_2, (X_{1s}, X_{2s}, Y'_s, c_{3s}))$ is generated.

(⋆3⋆) If p appears in the second hypothesis $\mathbf{h_2}$ of a rule of `form 2`, new boolean variables $W_1(q_2, (X_{1s}, X_{2s}, Y'_s, c_{3s}))$ are generated symmetrically.

It can be observed that there is a one-to-one correspondence between boolean variables W_1 and solutions (ground atoms) inferred from the Datalog program. The solution of the greatest fixed-point computation over this PBES always gives that all boolean variables are true. Indeed, this PBES is actually a tautology whose only purpose is to incrementally compute all possible boolean variables of the PBES starting from boolean variable W_0. In order to obtain an efficient implementation of this PBES-based evaluation strategy, expensive set operations like $\mathbf{X_{is}} \in \mathbf{h_i}(\mathbf{X_{is}}, \mathbf{Y_s}, \mathbf{c_{is}})$ (set membership) must be replaced by constant time incremental computation based on auxiliary maps that will be described in Section 5. For instance, our evaluation strategy will construct an auxiliary map (also called view) $h_i(X_{is}, Y_s, c_{is})$ for every hypotheses pertaining to a rule of the `form 2`. The arguments of hypothesis $\mathbf{h_i}$ will be reordered if they are not strictly grouped with the following order: Y_s, X_{is}, c_{is}. Using this approach and a standard BES resolution algorithm, the temporal and spatial complexity of our evaluation strategy is linear with the number of ground atoms. In the rest of the section, we use a canonical form to specify the hypothesis, which is agnostic to any order related to the internal data structures, to ease the exposition of the transformation.

Example 3 (PBES evaluation of Andersen's points-to analysis) *For a given* Datalog *program, the* PBES *evaluation algorithm can be solved by instantiating the parameterized boolean variables over the given predicate domain [GM98].*

Figure 1 describe a simple example in Java*, from which a set of facts are extracted.*

Example a = new Example();	vp0(v_a, h_1).
Example b = new Example();	vp0(v_b, h_2).
b = a;	a(v_b, v_a).
a.x = b;	s(v_a, x, v_b).
c = a.x;	l(v_a, x, v_c).
(a)	(b)

Figure 1: (a) Example of Java program. (b) Corresponding input relations (facts).

In order to efficiently represent the domains, we represent data values as natural numbers. With the example above, we choose the following mapping between domains and values: v_a*=1,* v_b*=2,* v_c*=3* h_1*=0,* h_2*=1* *x=0*
The computed five facts are now written as follows:

```
vp0(1,0).  vp0(2,1).  a(2,1).  s(1,0,2).  l(1,0,3).
```

Given the Datalog*-based Andersen's points-to analysis in Example 2 and the facts above, we can instantiate the* PBES *by incrementally constructing a* BES *from variable* W_0 *[Mat98], which would give the following instantiated* BES*:*

$$
\begin{array}{lcl}
W_0 & \overset{\nu}{=} & W_{1,vp0(1,0)} \wedge W_{1,vp0(2,1)} \wedge W_{1,a(2,1)} \wedge W_{1,s(1,0,2)} \wedge W_{1,l(1,0,3)} \\
W_{1,vp0(1,0)} & \overset{\nu}{=} & W_{1,vp(1,0)} \\
W_{1,vp0(2,1)} & \overset{\nu}{=} & W_{1,vp(2,1)} \\
W_{1,a(2,1)} & \overset{\nu}{=} & \mathsf{true} \\
W_{1,s(1,0,2)} & \overset{\nu}{=} & \mathsf{true} \\
W_{1,l(1,0,3)} & \overset{\nu}{=} & \mathsf{true} \\
W_{1,vp(1,0)} & \overset{\nu}{=} & W_{1,vp(2,0)} \wedge W_{1,temp1(2,0,0)} \wedge W_{1,temp2(0,0,3)} \\
W_{1,vp(2,1)} & \overset{\nu}{=} & \mathsf{true} \\
W_{1,vp(2,0)} & \overset{\nu}{=} & \mathsf{true} \\
W_{1,temp1(2,0,0)} & \overset{\nu}{=} & W_{1,hp(0,0,0)} \wedge W_{1,hp(0,0,1)} \\
W_{1,temp2(0,0,3)} & \overset{\nu}{=} & \mathsf{true} \\
W_{1,hp(0,0,0)} & \overset{\nu}{=} & W_{1,vp(3,0)} \\
W_{1,hp(0,0,1)} & \overset{\nu}{=} & W_{1,vp(3,1)} \\
W_{1,vp(3,0)} & \overset{\nu}{=} & \mathsf{true} \\
W_{1,vp(3,1)} & \overset{\nu}{=} & \mathsf{true}
\end{array}
$$

It can be observed that parameters p *and* c_s *of variable* W_1 *in the* PBES *appear as a subscript in the* BES*, like* $W_{1,vp0(1,0)}$ *with the fact* $\mathbf{vp0}(\mathbf{1},\mathbf{0})$*.* W_0 *is now defined as the conjunction of parameterless boolean variables, one per fact in the* Datalog *program. Then, all boolean variables except* W_0 *are defined by the second equation of the* PBES*. For instance, variable* $W_{1,vp0(1,0)}$ *holds the fact* $\mathbf{vp0}(\mathbf{1},\mathbf{0})$*. Now,* $\mathbf{vp0}$ *is only used in one rule of the* Datalog *program, namely* $\mathbf{vp}(\mathbf{X},\mathbf{Y}) :- \mathbf{vp0}(\mathbf{X},\mathbf{Y})$.*, which is of* `form 1`*, so only the* $(*1*)$ *part of the second boolean equation in the* PBES *applies. From the* Datalog *rule, we can infer only one ground atom, namely* $\mathbf{vp}(\mathbf{1},\mathbf{0})$*. As a result, only one boolean variable* $W_{1,vp(1,0)}$ *is generated on the right hand-side of the equation defining* $W_{1,vp0(1,0)}$ *as shown above. Another example is the boolean variable* $W_{1,a(2,1)}$*. Predicate* $\mathbf{a}$ *is only used in one* Datalog *rule, namely* $\mathbf{vp}(\mathbf{X},\mathbf{Y}) :- \mathbf{a}(\mathbf{X},\mathbf{Z}), \mathbf{vp}(\mathbf{Z},\mathbf{Y})$.*, which is of* `form 2`*. Since* $\mathbf{a}$ *appears as the first hypothesis in this rule, so only the* $(*2*)$ *part of the boolean equation applies. Now, there does not exist any value* $\mathbf{Y}$ *such that* $\mathbf{vp}(\mathbf{1},\mathbf{Y})$*. is a previously computed ground atom. Indeed, only the ground atoms* $\mathbf{vp0}(\mathbf{1},\mathbf{0})$*. and* $\mathbf{vp0}(\mathbf{2},\mathbf{1})$*. have been computed so far. Therefore, variable* $W_{1,a(2,1)}$ *is defined in terms of an empty conjunction, which is by definition the constant* true*. Finally, we can comment on the equation defining variable* $W_{1,hp(0,0,0)}$*. Predicate* $\mathbf{hp}$ *only appears in one rule, namely* $\mathbf{vp}(\mathbf{X},\mathbf{Y}) :- \mathbf{temp2}(\mathbf{Y},\mathbf{S},\mathbf{Z}), \mathbf{hp}(\mathbf{Y},\mathbf{S},\mathbf{T})$.*, which is of* `form 2`*. Since it is the second hypothesis in the rule, only the* $(*3*)$ *part of the boolean equation applies. Now, there exists a value* $\mathbf{Z}$ *such that* $\mathbf{temp2}(\mathbf{0},\mathbf{0},\mathbf{Z})$*. is a previously computed ground atom. Indeed, the ground atom* $\mathbf{temp2}(\mathbf{0},\mathbf{0},\mathbf{3})$*. has been previously generated in the equation that defines variable* $W_{1,vp(1,0)}$*. Hence, the ground atom* $\mathbf{vp}(\mathbf{3},\mathbf{0})$*. can be inferred from the rule and a boolean variable*

$W_{1,vp(3,0)}$ *is generated on the right hand-side of the equation defining* $W_{1,hp(0,0,0)}$.

The least set of ground atoms that can be inferred using the rules of the Datalog*-based points-to analysis is the set of all computed boolean variables, minus* W_0. *In the case of pointer analysis, we are interested in the intentional database, namely the inferred ground atoms for* **vp** *and* **hp**. *These atoms are given by the computed boolean variables* $W_{1,vp(...)}$ *and* $W_{1,hp(...)}$, *from which we directly obtain the solutions* $\mathbf{vp}(\mathbf{1},\mathbf{0})$., $\mathbf{vp}(\mathbf{2},\mathbf{0})$., $\mathbf{vp}(\mathbf{3},\mathbf{0})$., $\mathbf{vp}(\mathbf{2},\mathbf{1})$., $\mathbf{vp}(\mathbf{3},\mathbf{1})$., $\mathbf{hp}(\mathbf{0},\mathbf{0},\mathbf{0})$., *and* $\mathbf{hp}(\mathbf{0},\mathbf{0},\mathbf{1})$. *Since there exists a mapping between the natural number values and the* Java *program's domains, we can describe the solutions in terms of the program's elements as follows:* $vp(v_a, h_1)$. $vp(v_b, h_1)$. $vp(v_c, h_1)$. $vp(v_b, h_2)$. $vp(v_c, h_2)$. $hp(v_a, x, v_a)$. *and* $hp(v_a, x, v_b)$.

4 Evaluation Order based on Predicate Dependency Graphs

Predicate order evaluation has a direct impact on performance results while not affecting time complexity of the evaluation problem. Depending on the predicate sorting, several operations can be discarded in the evaluation algorithm. In this section, we propose an algorithm to compute a predicate evaluation order for a given Datalog program and show its usefulness to optimize the BES resolution. The algorithm constructs a predicate evaluation order from a *predicate dependency graph* (PDG). A PDG is a directed graph that describes how predicates are dependent on each other. Each node is a predicate and each edge indicates that the start node appears as a hypothesis in a rule whose conclusion is the end node.

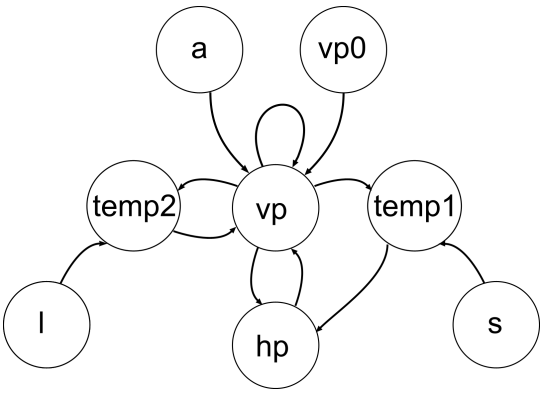

Figure 2: Predicate Dependency Graph for Andersen's Points-to Analysis

Example 4 *Given the decomposed* Datalog *program for Andersen's points-to analysis in Section 1, Figure 2 shows its corresponding* PDG. *In this example, it can be observed that only the extensional predicates, namely* **vp0**, **s**, **l**, *and* **a**, *are not part of a cycle.*

From a PDG, a topological order can be constructed based on the number of incident edges for each node as follows:

1. A node with no incident edges means that it does not appear as the conclusion of any rule in the program. Thus, it is preferable to start the evaluation of the program by propagating the ground atoms of these nodes;

2. Then, all successor nodes that are not part of a cycle, can be ordered and evaluated topologically; and

3. Finally, the remaining nodes that pertain to a cycle cannot be ordered topologically. In order to optimize the execution, we propose to order and evaluate them by decreasing

number of incident edges.

This algorithm forms three *levels* of predicate.

Example 5 *From the* PDG *in Figure 2, predicates* **vp0**, **s**, **l**, *and* **a** *will be evaluated first since the corresponding nodes do not have incident edges. The construction of a topological order over these predicates allows them to be evaluated in any order. All other predicates are part of a cycle. By decreasing the number of incident edges, we get the following predicate order:* **vp**, *and then in any order* **hp**, **temp2** *and* **temp1**.

In the PBES evaluation strategy described in Section 3.2, some expensive computations are performed, like $\mathbf{X_{2s}} \in \mathbf{h_2}(\mathbf{X_{2s}}, \mathbf{Y_s}, \mathbf{c_{2s}})$., which looks up previously computed ground atoms for predicate $\mathbf{h_2}$ with arguments $(\mathbf{X_{2s}}, \mathbf{Y_s}, \mathbf{c_{2s}})$. Another expensive operation is the update of the auxiliary data structure with new ground atoms. This operation does not explicitly appear in the PBES description. The data structure can be updated when a new ground atom is found, or when this ground atom is used as a hypothesis to compute other ground atoms. We will consider this approach later in this paper. By considering the predicate evaluation order described above, we can determine the smallest set of look-up and update operations that should be performed in order to obtain all solutions for a given Datalog program. For instance, if a predicate participates in a rule of `form 2` that generates a predicate that has a higher evaluation order, then the following hold:

- If the other hypothesis in the rule has a lower order or the same order and it has already been processed, then it is only necessary to look up the data structure to test the existence of ground atoms.
- If the other hypothesis in the rule has a higher order or the same order and it has not been processed yet, then it is only necessary to update the data structure by adding the new computed ground atom.

Example 6 *The predicate evaluation order can be used while instantiating the* PBES *described in Section 3.2 for Andersen's points-to analysis. Instead of choosing the facts in an unordered way, we can choose all facts that pertain to a predicate at the same time as indicated in the evaluation order. Then, the simplifications specified in the previous example can be made dynamically to prevent having to look up or update the data structure. For instance, given the fact* $\mathbf{a}(\mathbf{2},\mathbf{2})$., *the predicate* **a** *participates in a rule of* `form 2`, *namely rule* $\mathbf{vp}(\mathbf{X},\mathbf{Y}) :- \mathbf{a}(\mathbf{X},\mathbf{Z}), \mathbf{vp}(\mathbf{Z},\mathbf{Y})$.. *This rule generates a predicate, namely* **vp**, *from another hypothesis which is also* **vp**. *From the evaluation order,* **vp** *is evaluated after* **a**. *Thus, only an update of the data structure with the ground atom* $\mathbf{a}(\mathbf{2},\mathbf{2})$. *is necessary. Thanks to the proposed sorting algorithm, we know in advance that no look-up to previously computed ground atoms in the data structure is necessary. This means that the conjunction* $\bigwedge_{X_{2s} \in \mathbf{h_2}(\mathbf{X_{2s}},\mathbf{Y_s},\mathbf{c_{2s}})}$ *can be simply discarded for facts of predicate* **a**. *The resulting boolean formula that computes all the conclusions inferred from* $\mathbf{a}(\mathbf{X},\mathbf{Y})$ *can be reduced to* true. *The same applies to predicates* **s** *and* **l**, *which appear as hypotheses in rules of* `form 2`. *Hence, we could reduce the instantiated* BES *by the number of facts in* **a**, **s**, *and* **l**, *which would only generate* true *boolean variables. Note that any* BES *resolution algorithm*

could be used to solve our problem. However, in order to capture the preferred evaluation order, a specialized algorithm based on multiple queues is required. Such an algorithm could use one queue for all predicates or one queue per predicate level.

Even though they have less impact on the execution time, there are other kinds of optimizations that are based on the predicate evaluation order such as:

- removing useless rules from strongly connected components in the graph.
- preventing the addition of new ground atoms if there only exists a unique intensional predicate in each strongly connected component of the graph.

5 Efficient and Compact Auxiliary Data Structure

In this section, we describe a sophisticated data structure that has quicker access time and lower memory consumption than the dedicated structure in [LS09]. This efficient and compact data structure plays an essential role in the drastic improvements that can be observed in the experimental results of Section 6.

5.1 Data Structure Representation

We have defined a data structure that is extremely compact in memory while ensuring constant access to previously computed ground atoms and boolean variables. In our evaluation algorithm, we need to perform constant time access over two structures:

1. auxiliary mapping relations, to search all tuples (X_{is}, Y_s, c_{is}) that were previously computed given a set of constant arguments $(\mathbf{Y_s}, \mathbf{c_{is}})$ and a predicate $\mathbf{h_i}$, *i.e.*, operations $\mathbf{X_{is}} \in \mathbf{h_i}(\mathbf{X_{is}}, \mathbf{Y_s}, \mathbf{c_{is}})$ of our PBES evaluation strategy; and
2. result sets, to test if a new generated ground atom has already been computed before adding a new boolean variable.

The auxiliary mapping relations are represented by linked lists, one per node in the data structure representation. This allows us to access all existing ground atoms for a given predicate in a constant time. However, we can no longer use linked lists to retrieve one particular ground atom in a constant time. Instead, result sets are represented by adding bitmaps to the leaf nodes of the data structure representation. These bitmaps are based on digital trees to allow sparse domains. Note that both linked lists and bitmaps grow dynamically.

As a result, our data structure representation is based on the following: linked lists to represent sets; a representation of a `trie` (currently smart digital trees are used) to represent a layered indexed map; and bitmaps to test whether or not a ground atom has been previously computed. In the worst-case, associative access to elements of this data structure are done in O(1) time. Update and look-up operations to the structure are also done in O(1) time. If we relax the constraint about constant access, we could remove the bitmaps of the structure for certain specific problems. An ordered auxiliary mapping relation could be used in their place. In practice, this would save most of the memory used to solve the evaluation problem.

Example 7 *Figure 3 gives the final state of our data structure representation after all the possible ground atoms have been inferred with Andersen's points-to analysis and the five initial facts of Example 3. For example, the ground atom* $\mathbf{hp}(\mathbf{0},\mathbf{0},\mathbf{0})$. *is identified in the trie representation by*

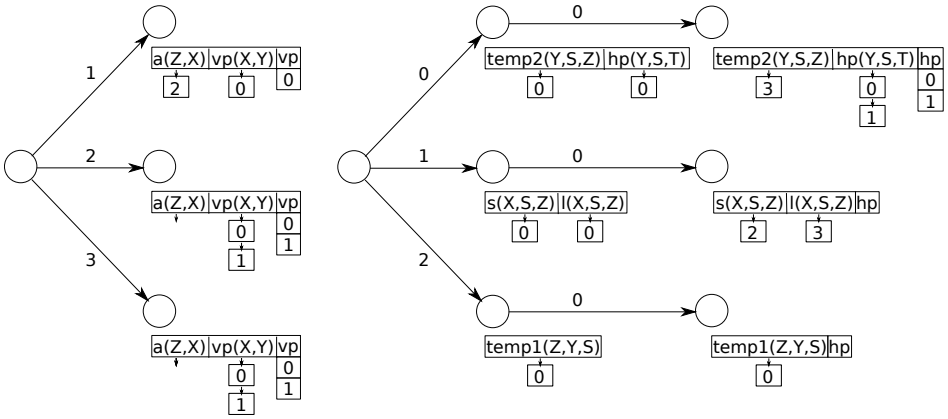

Figure 3: Data structure representation

its prefix, which corresponds to the path $0 \rightarrow 0$ *from root node. In the leaf node, a linked list indicates all values* $\mathbf{X}$ *such that* $\mathbf{hp}(\mathbf{0},\mathbf{0},\mathbf{X})$. *is a ground atom. Actually, there are only two values* (0, 1) *that have been computed for* **hp** *with the prefix* $(\mathbf{0},\mathbf{0})$. *To test whether or not* $\mathbf{hp}(\mathbf{0},\mathbf{0},\mathbf{0})$. *has been previously computed without iterating through the whole linked list, we use the bitmap for* **hp** *in the leaf node as well.*

5.2 Classical Auxiliary Structures

A classical approach for solving a Datalog program is to use `hash tables`. Access in hash tables is done in average, rather than worst-case, O(1) time. Moreover, hash tables have an overhead for computing the hash related functions for every operation. If all keys are known in advance, a perfect hash function can be used to avoid collisions. This enables having constant time look-ups in the worst case. Even if perfect hash tables could be used in our context, since keys are numbers (*i.e.*, no need to use a hash function) and sizes of domains are known a priori, our approach has two advantages. First, our structure is extremely compact, due to the fact that the information is stored by prefixes, which allows memory to be managed in a very efficient way; then, even if domain sizes are known, they are usually so large that they have to be managed dynamically. Therefore, perfect hash tables cannot be used in practice, and regular hash tables cannot assure constant access. Other solutions can be found in the literature, namely [LS02]. This describes a technique to design linked structures that support associative arrays in worst-case O(1) access time with little space overhead for a general class of set-based programs. To achieve this, it uses a representation of an arbitrary number of sets using a base. This approach guarantees a worst-case O(1) access time, but it cannot efficiently manage large sparse domains. Therefore, this structure does not scale for large instances of Datalog programs. Compared to the data structure detailed in [LS09], our structure does not need to manage several sets for one

domain since this information is propagated through the boolean variables. Moreover, our data structure representation does not have the limitation to work with tuples with more than two components. With the structure of [LS09], this would be very inefficient in terms of memory. Finally, in comparison with the usage of tries in XSB [RRS+99], our data structure does not make use of tries in the classical sense. XSB allows to use tries to represent Datalog's tabled subgoals and their answers, while we store dynamic lists and bitmaps to compute solutions to the original Datalog specification in an efficient way.

6 Experimental Results

In this section, we compare an implementation of our novel evaluation strategies with standard Datalog solvers by evaluating Andersen's points-to analysis on the DACAPO [3] benchmark.

Prototype We have extended the DATALOG_SOLVE [AFJV09] prototype, implemented in C, with the new evaluation strategy encoded as BES. Facts are extracted by SOOT [4] from the Java programs of the DACAPO benchmark with JDK 1.6. The extraction time varies between 115 seconds for the `lusearch` Java project and 315 seconds for the `fop` Java project. The complete benchmark is processed by SOOT in 39 minutes and 36 MB. of facts are generated. DATALOG_SOLVE first parses (150 lines of C code) the facts, then calls a BES solver (550 lines of C code) with an implicit description of the BES-based points-to analysis in terms of a successor function. This function is supported by our data structure representation that is connected to the Judy[5] library. Such an architecture brings the following advantages: constant access to check if a solution has already been computed and to browse sets for new solutions; compact domains; and lazy updating. In order to efficiently allocate the memory space that is strictly necessary for the computation, our system is connected to a specific memory pool and set values are put in contiguous memory positions in order to increase cache efficiency.

The new BES solver implements a very simple BES resolution algorithm based on a `queue` and our data structure representation. The `queue` enables the resolution graph associated to the Datalog program to be constructed on-the-fly, whereas our data structure representation is used to store and retrieve the required information to create new boolean variables. Only the boolean variables that are necessary to expand the successors are kept in the `queue`. The BES algorithm starts with boolean variable X_0 for which all successors are enumerated through the successor function provided by DATALOG_SOLVE. Each generated successor is checked in our data structure representation to see if it has already been computed. If it is a new solution, then it is added to the `queue`. Otherwise, it is ignored. Upon completion of the whole evaluation, the DATALOG_SOLVE tool extracts the inferred ground atoms from the computed boolean variables.We can remark that our simple BES solver is not specific to solve Datalog programs. It can be used for any problem that can be represented in terms of a BES with a unique block of conjunctive boolean formulas.

[3] http://voxel.dl.sourceforge.net/sourceforge/dacapobench/dacapo-2006-10-MR2-xdeps.zip
[4] http://www.sable.mcgill.ca/soot
[5] http://judy.sourceforge.net

Experiments We tested the efficiency and feasibility of our implementation by comparing it to three state-of-the-art Datalog solvers BDDBDDB[6], XSB 3.2[7] and the prototype of Liu and Stoller[8], which in the rest of the paper we will call TOPLAS. We also have implemented a pure Python version of our DATALOG_SOLVE prototype to compare it with the TOPLAS prototype (also written in pure Python). This version of the prototype has been developed using built-in types (dictionaries, lists and sets). Therefore, not all the techniques applied to our C prototype have been applied to the Python version. Nevertheless, this allows us to highlight the direct benefits of our new strategies. In Figure 4, performance results are presented in terms of evaluation user time (seconds) and memory consumption (MB.). All experiments were performed on an Intel Core 2 duo E4500 2.2 GHz (only one core used), with 2048 KB cache, 4 GB of RAM, and running Linux Ubuntu 10.04. DATALOG_SOLVE and XSB solvers were compiled using gcc 4.4.1. Python 2.6.4 was used for the TOPLAS solver. The measures do not include the time needed by XSB to precompile the facts. BDDBDDB is executed with the best variable ordering that we have found for the Andersen's analysis example, namely: V V H H F. XSB and BDDBDDB prototypes have been tested on both the original Andersen's analysis with 4 rules and the decomposed analysis with 6 rules. Since their execution time and memory consumption were not better with the decomposed analysis, only experimental results for the original analysis are presented in the figure. The analysis results were verified by comparing the outputs of all solvers.

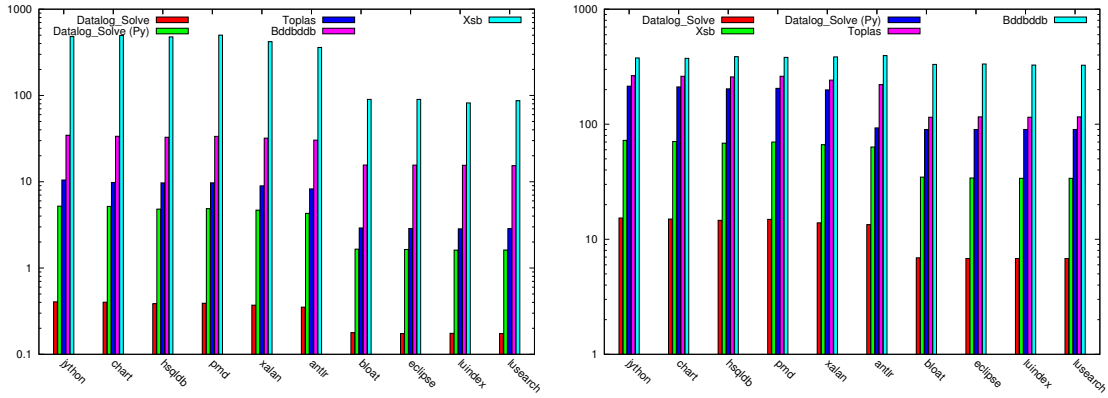

Figure 4: Analysis time (sec.) (left) and memory usage (MB.) (right) on the DACAPO benchmark

DATALOG_SOLVE evaluates the whole benchmark in only 3 seconds with a mean-time of 0.3 seconds per program. On the left part of Figure 4, where the y-axis (sec.) has a logarithmic scale, we can observe that DATALOG_SOLVE is an order of magnitude faster than TOPLAS, two orders faster than BDDBDDB, and three orders faster than XSB. For the *jython* example, XSB evaluated the points-to analysis in 482 seconds, BDDBDDB in 34 seconds, TOPLAS in 10 seconds, DATALOG_SOLVE (Python) in 5 seconds and DATALOG_SOLVE in 0.406 seconds. On the right part of Figure 4, where the y-axis (MB.) has a logarithmic scale, we can

[6] http://bddbddb.sourceforge.net
[7] http://xsb.sourceforge.net
[8] Provided by the authors of [LS09]

observe that DATALOG_SOLVE consumes five times less memory than XSB and an order of magnitude less memory than DATALOG_SOLVE (Python), TOPLAS and BDDBDDB. For the *lusearch* example, BDDBDDB required 326 MB. of memory to solve the analysis, TOPLAS 116 MB., DATALOG_SOLVE (Python) 90 MB., XSB 34 MB, and DATALOG_SOLVE 7 MB. It should be mentioned that the memory results of the DATALOG_SOLVE (Python) prototype can be highly improved avoiding some of the built-in types to build the data structure representation. These performance results show that our novel evaluation strategies scales really well for very large programs in the context of Andersen's points-to analysis.

7 Conclusions and Future Work

This paper presents new evaluation strategies to solve Datalog programs inspired by Liu and Stoller [LS09]. These strategies are based on: the separation of the evaluation strategy from the fixed point computation; an evaluation order over the predicates to avoid useless set operations; and a sophisticated data structure with worst-case constant access and compact representation of the underlying data. The first strategy is based on Boolean Equation Systems (BESs) to derive a specialized bottom-up evaluation algorithm with time and memory guarantees for any Datalog program. The algorithm can then be evaluated by independent optimized fixed-point solvers. The second strategy extracts a topological order over the predicates based on the dependencies of the Datalog program to simplify the BES. Finally, the third strategy exploits the features of prefix tree structures, together with linked lists and bitmaps to enable efficient accesses to inferred tuples whose values are sparse over the finite domains. The overall approach is faster and less memory-consuming, by some orders of magnitude, than state-of-the-art Datalog solvers. It was implemented in the DATALOG_SOLVE tool and was successfully tested on the Datalog-based Andersen points-to analysis of real Java projects.

An interesting future work would be to study the impact of our efficient data structure on more complex applications, such as flow- and context-sensitive pointer analyses, where more than 10^{14} contexts have to be stored and retrieved from memory during the evaluation of the problem. Recently, DOOP[9] became the new state-of-the-art framework for performing context-sensitive pointer analyses of Java programs. It is based on a commercial Datalog solver, called LogicBlox[10]. We would like to compare our prototype with DOOP/LogicBlox over such a set of complex points-to analyses. Other Datalog applications are also foreseen, like model checking and database benchmarks. We are also interested in the adequacy of the BES formalism to distribute the bottom-up Datalog resolution over interconnected machines. Several distributed BES resolution algorithms exist in the literature and have been applied with success for formal verification problems [JM06]. A promising alternative approach to explore would be to distribute the workload directly at the Datalog level by using Map-Reduce-based algorithms such as [AU10].

Acknowledgements We are grateful to Annie Liu and Scott Stoller for providing us with the generated Python code with guaranteed time and memory complexities for Andersen's points-to analysis example.

[9] http://doop.program-analysis.org
[10] http://www.logicblox.com

Bibliography

[ABM09] S. Abiteboul, P. Bourhis, B. Marinoiu. Efficient maintenance techniques for views over active documents. In *Proc. 12th Int'l Conf. on Extending Database Technology EDBT'09*. ACM Int'l Conf. Proc. Series 360, pp. 1076–1087. ACM Press, 2009.

[AFJV09] M. Alpuente, M. A. Feliú, C. Joubert, A. Villanueva. DATALOG_SOLVE: A Datalog-Based Demand-Driven Program Analyzer. *Electr. Notes Theor. Comput. Sci.* 248:57–66, 2009.

[AHV95] S. Abiteboul, R. Hull, V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[ALSU07] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Publishing Co., Inc., Boston, MA, USA, 2007.

[And94a] H. R. Andersen. Model checking and boolean graphs. *Theo. Comp Sci* 126(1):3–30, 1994.

[And94b] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994. DIKU report 94/19.

[AU10] F. Afrati, J. Ullman. Optimizing joins in a map-reduce environment. In *Proc. 13th Int'l Conf. on Extending Database Technology EDBT'10*. ACM Int'l Conf. Proc. Series 426, pp. 99–110. ACM, 2010.

[BFG01] R. Baumgartner, S. Flesca, G. Gottlob. The Elog Web Extraction Language. In *8th Int'l Conf. on Logic for Progr., Artificial Intelligence, and Reasoning LPAR'01*. LNCS 2250, pp. 548–560. Springer, 2001.

[GJMS07] M. Gallardo, C. Joubert, P. Merino, D. Sanán. C.OPEN and ANNOTATOR: Tools for On-the-Fly Model Checking C Programs. In *Proc. 14th Int'l Work. on Model Checking of Software SPIN'07*. LNCS 4595, pp. 268–273. Springer, 2007.

[GM78] H. Gallaire, J. Minker (eds.). *Logic and Data Bases*. Adv. in DB Theo. Plemum, 1978.

[GM98] J. F. Groote, R. Mateescu. Verification of Temporal Properties of Processes in a Setting with Data. In *Proc. 7th Int'l Conf. Algebraic Methodology and Software Technology AMAST'98*. LNCS 1548, pp. 74–90. Springer, 1998.

[JM06] C. Joubert, R. Mateescu. Distributed On-the-Fly Model Checking and Test Case Generation. In *Proc. 13th Int'l Workshop on Model Checking of Software SPIN'06*. LNCS 3925, pp. 126–145. Springer, 2006.

[LS02] Y. A. Liu, S. D. Stoller. Program optimization using indexed and recursive data structures. In *Proc. ACM SIGPLAN Work. on Partial Evaluation and Semantics-Based Program Manipulation PEPM'02*. SIGPLAN Notices 3, pp. 108–118. ACM, 2002.

[LS09] Y. A. Liu, S. D. Stoller. From datalog rules to efficient programs with time and space guarantees. *ACM Trans. Program. Lang. Syst.* 31(6), 2009.

[Mad97] A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. Bertz Verlag, 1997.

[Mat98] R. Mateescu. Local Model-Checking of an Alternation-Free Value-Based Modal Mu-Calculus. In *Proc. 2nd Int'l Workshop on Verication, Model Checking and Abstract Interpretation VMCAI'98*. 1998.

[Mat06] R. Mateescu. CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems. *Springer Int'l Journal on Soft. Tools for Tech. Transfer (STTT)* 8:37–56, 2006.

[PK82] R. Paige, S. Koenig. Finite Differencing of Computable Expressions. *ACM Trans. Program. Lang. Syst.* 4(3):402–454, 1982.

[RRS+99] I. V. Ramakrishnan, P. Rao, K. F. Sagonas, T. Swift, D. S. Warren. Efficient Access Mechanisms for Tabled Logic Programs. *Journal of Logic Prog.* 38(1):31–54, 1999.

[SSN+10] S.-W. Seong, J. Seo, M. Nasielski, D. Sengupta, S. Hangal, S. K. Teh, R. Chu, B. Dodson, M. S. Lam. Preserving Privacy with PrPl: a Decentralized Social Networking Infrastructure. In *Proc. 9th Int'l Symp. on Privacy Enhancing Technologies PETS'10*. LNCS 6205. Springer, 2010.

[Ull89] J. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I and II, The New Technologies*. Computer Science Press, 1989.

[VL92] B. Vergauwen, J. Lewi. A linear algorithm for solving fixed-point equations on transition systems. In *Proc. 17th Colloquium on Trees in Algebra and Progr. CAAP'92*. LNCS 581, pp. 322–341. Springer, 1992.

[WL04] J. Whaley, M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation PLDI'04*. Pp. 131–144. ACM Press, 2004.