AGGREGATION IN SOUFFLÉ

A PREPRINT

Rachel Dowavic
School of Computer Science
The University of Sydney

Xiaowen Hu
School of Computer Science
The University of Sydney

Abdul Zreika
School of Computer Science
The University of Sydney

Martin McGrane
School of Computer Science
The University of Sydney

Sasha Rubin
School of Computer Science
The University of Sydney

Bernhard Scholz
School of Computer Science
The University of Sydney

February 22, 2021

ABSTRACT

Datalog is a database logic programming language that has experienced a resurgence of interest in recent times in domains like static program analysis, network analysis, and security analysis. Since Datalog is a declarative language, it is paramount that the programmer can express problems succinctly and quickly. This includes allowing users to express the concept of aggregate concisely. Aggregates are functions like sum, min, max, $mean$, and $count$ that operate over relations and are useful for creating summary statistics that answer higher-level questions. We present a formal treatment of aggregates' syntax and semantics in Datalog and provide three extensions such that users can express aggregate easily and efficiently. The first extension is a syntax sugar called the *witness rewrite* that allows user to fetch data associated within the aggregate body. The second extension is an optimization technique that stores the result of an aggregate body in-memory so that later the result can be reused by other rules. The final extension is to support nested aggregate as syntax sugar. Overall, this work has increased the functionality, usability, and performance of aggregates in the Soufflé engine.

***K*eywords** Datalog · Aggregation

1 Introduction

Datalog and other logic programming languages [1, 2, 3, 4] have experienced a resurgence of interest in recent years because they provide users with the ability to express real-world problems concisely and easily. Some of these applications include security analysis [5, 6], cloud computing [7, 8], and static program analysis [9, 10]. Because Datalog allows programmers to write programs succinctly and easily, programmers require little knowledge of *how* the program will be executed. Instead, they only concern themselves with *what* the program will compute. It also means that programs of this nature are much easier to maintain, update, and debug. Therefore, it is paramount that the development process is as efficient as possible, and a declarative programming style is said to be more conducive to efficiency since the translation from design brief to working code is less complex.

A common real-world use case in the Datalog program is to summarize statistics over one or more relations [9, 10]. For such cases, Datalog provides *aggregate function* [11, 12] such as sum, min, max, and $count$ that takes input a set of tuples and return the summarization with a single value. Those aggregates provide an easy and effective way for logic programmers to collect and summarize information over relations. However, the question remains on how to correctly and efficiently adapt aggregates into a real-world Datalog engine.

In this work, we define the syntax and semantics for aggregates in a state-of-the-art Datalog engine Soufflé [13]. We provide insight into using syntax sugar to further empower the aggregates' expressiveness with *witness rewrite*, allowing user to implicitly extract information from within an aggregate body. Whatsmore, we present an optimization technique

called *aggregate simplification* that rewrites and caches an aggregate body into a separated relation so that its result can be reused later in the program. We also utilize the simplification technique to support nested aggregates as syntax sugar.

2 Soufflé Syntax and Semantic

Soufflé's syntax extends that of datalog by numeric and string predicates and operations, i.e., A Soufflé *program* p consists of rules R in the form:

$$A(\bar{t}) \leftarrow L_1(\bar{t}_1), \cdots, L_k(\bar{t}_k)$$

where $A(t_0, \cdots, t_n)$ is called an *atom* and A is a relation symbol or interpreted relation with arity of n, each t_i is a *term*. The interpreted relations are of two main types: string and numeric. The string relations include the substring relation, equality between strings, and regular expression matching. The numeric relations include equality and comparison. Terms are defined as usual, closing the constants (strings and numbers), variables, under interpreted operations including addition, subtraction and multiplication of numbers. Furthermore, $L_i(\bar{t}_i)$ is called a *literal*, which is either an *atom* $A(\bar{t})$ or the negation of an atom $\neg A(\bar{t})$. Atoms are also called *positive literals*.

All rules are required to satisfy the additional property that every variable appearing in $\bar{t}$ appears in some $\bar{t}_i$. A literal is *grounded* if all its arguments are constants, and a rule is grounded if all its literals are grounded. A variable x is *limited* in a rule R if

- it appears as an argument in a positive atom $A_i(..., x, ...)$ where A is a relation symbol, or
- it appears in an equality $x = c$ or $c = x$, or
- it appears in an equality $x = y$ or $y = x$ where y is limited.

All rules are required to satisfy the additional property that every variable is limited. This is a safety condition.

A relation symbol A occuring in the program p is in the program's *Extensional Database* (EDB) if it does not occur in the body of any rule of the program, otherwise it is in the program's *Intensional Database* (IDB). In particular, every interpreted relation is in the EDB. Write $EDB(p)$ for the set of ground atoms in the EDB of p.

A program is *semipositive* if for every negative literal $\neg A$ occuring in the program where A is a relation symbol, the symbol A is in the EDB of the program.

The semantics of a semipositive program p is defined by the *immediate consequence operator* T_p as mapping a set W of ground atoms to the set $T_p(W)$ of ground atoms:

$$T_p(W) = W \cup EDB(p) \cup INFER_p(W)$$

where $INFER_p(W)$ is the set of all ground atoms $A(\bar{c})$ for which there is a grounded rule $A(\bar{c}) \leftarrow L_0(\bar{c}_1), \cdots, L_k(\bar{c}_k)$ of p such that if $L_i(\bar{c}_i) = A(\bar{c}_i)$ is positive then $A(\bar{c}_i) \in W$, and if $L_i(\bar{c}_i) = \neg A(\bar{c}_i)$, then $A(\bar{c}_i) \notin W$. If I is a set of EDB atoms, then the sequence $(T_p^n(I))_n$ is non-decreasing, and thus the union $T_p^\infty(I) = \cup_n T_p^n(I)$ is a fix-point of T_p.

A program is *stratified* if it can be partitioned into a sequence of semipositive programs $p_1, \cdots, p_k$, called *strata*, such that

1. every head predicate occurs in exactly one stratum, and
2. if a head predicate of stratum p_i occurs positively (resp. negatively) in the body of a rule in p_j then $i \leq j$ (rep $i < j$).

In other words, we can think that the extensional database of p_i is defined by the intensional databases p_j for $j < i$, as well as the given extensional database of p_1.

The semantics of a stratified program p constructed by a sequence of semipositive programs $p_1, \cdots, p_k$ is defined as:

$$\begin{aligned} M_1 &= T_{p_1}^\infty(\emptyset) \\ M_2 &= T_{p_2}^\infty(M_1) \\ &\vdots \\ M_k &= T_{p_k}^\infty(M_{k-1}) \end{aligned}$$

where M_k is called the *iterated fixed point* of stratified program p.

2.1 Aggregate

The form of a rule with an aggregate atom is defined as follow:

$$H(\overline{Z}) \leftarrow S(\overline{W}),\ y = f\ x : \{A(\overline{T})\}$$

where

1. $\overline{Z}, \overline{W}, \overline{T}$ are lists of terms, $var(\overline{Z}) \subseteq var(\overline{W}) \cup \{y\}$ and $y \notin var(\overline{T})$.
2. f is the aggregate function and x is the cost variable, $x \notin var(\overline{Z} \cup \overline{W})$.
3. $S(\overline{W})$ and $A(\overline{T})$ is a conjunction of atoms and interpreted relations whose terms are $\overline{W}$ and $\overline{T}$.

For example, consider the following rule with an max aggregate:

$$\begin{aligned} TotalCredit(student, year, m) \quad &\leftarrow Enroll(student, year), \\ &\quad m = sum\ c : \{Pass(student, subject, year), Credit(subject, c)\}. \end{aligned}$$

where $H = TotalCredit$, $S = Enroll$ and $A = Pass \wedge Credit$. Their corresponding terms are $\overline{Z} = ((student, year, c))$, $\overline{W} = ((student, year))$ and $\overline{T} = ((student, subject, year), (subject, c))$.

The safeness requirement is that

1. $H(\overline{Z}) \leftarrow S(\overline{W})$ is safe with the assumption that y is limited.
2. x is limited by the aggregate body $A(\overline{T})$.
3. $var(\overline{T})$ must be limited by $A(\overline{T})$ and $S(\overline{W})$.

In addition to that, Soufflé rules with aggregate atoms must be stratified as well — recursive aggregates are not allowed. That is, extend the definition of stratified program p in the previous section with one extra condition: if a rule r from stratum p_i contains aggregate atoms and if the head predicate of r occurs in the body of a rule in p_j then $i < j$.

Variables from the set $var(\overline{T})$ can be further partitioned into two disjoint sets *local* and *injected*. A variable is an injected variable if $v \in var(\overline{W} \cup \overline{Z})$.

An aggregate function essentially represents a mapping from a set of numbers to a single number. With the exception of the `count` aggregate, which is a mapping from a set of tuples to a single number. As a result, the cost variable is omitted in the `count` aggregate.

The semantic of a Soufflé program with aggregate atom is given by extending the immediate consequence operator:

$$INFER_p(W) = \{H(\bar{z}) : H(\bar{z}) \leftarrow S(\bar{n}, \bar{m}), y = f(\{x : \exists x(A(\bar{c}, \bar{n}, x) \in W)\})\}$$

where $\bar{n} \cup \bar{m}$ are the instantiations of the terms in $\overline{W}$, in particular, $\bar{n}$ is the instantiations of those terms that contains variables in the injection set.

3 Aggregation in Soufflé Machinery

Figure 1 gives a general overview of how a Datalog program is compiled and executed in Soufflé. In the first stage, Soufflé parses the input Datalog program, and if no syntax errors are detected, it outputs an Abstract Syntax Tree (AST) representation of the original program. The second stage applies a series of high-level optimizations on the AST. After that, the AST is translated into an intermediate representation – the Relational Abstract Machine (RAM). The RAM is a set of relational operations constructed imperatively. Finally, the RAM is synthesized into an equivalent C++ program.

RAM is a tree structure program containing subroutines and the main program. The main program is the root of the tree, and it calls each subroutine to execute each part of the program. Each subroutine contains a series of relational operations such as projections, membership checks, and imperative constructs such as conditional statements and loops. Together, a RAM program describes a series of executions of the source Datalog.

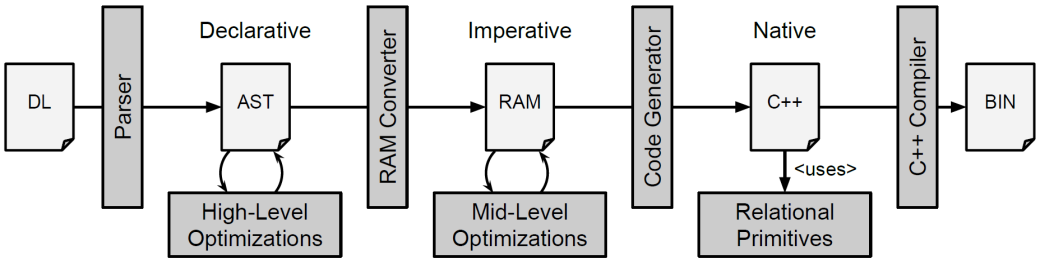

Figure 1: Execution model of Soufflé.

3.1 Aggregation In RAM representation

The syntax of aggregates in Soufflé is defined by the following BNF (Backus–Naur form):

```
<aggregate>           ::= <aggregate-function> <cost-variable>
                          ":{" <aggregate-body> "}"
<aggregate-function>  ::= "count" | "max" | "mean" | "min" | "sum"
<aggregate-body>      ::= [<literal> | <interpreted-relation>]
```

Consider the following Soufflé example that utilities an aggregate atom to summarize the result of a program analysis use case:

$$TotalInvocations(\text{“}foo\text{”}, y) :- y = sum\ c : \{FunctionInvocations(\_, \text{“}foo\text{”}, c)\}$$

The above Soufflé rule states that, for each function call from *any* routine to function foo, let y be the sum of the total number of invocations c and store the result in $TotalInvocations$ as $(\text{“}foo\text{”}, y)$. If the program is given the following EDB:

$$FunctionInvocations = \{(\text{“}A\text{”}, \text{“}foo\text{”}, 1), (\text{“}B\text{”}, \text{“}foo\text{”}, 2), (\text{“}A\text{”}, \text{“}bar\text{”}, 3), (\text{“}C\text{”}, \text{“}bar\text{”}, 2)\}$$

The above Datalog program will produce the following results:

$$TotalInvocations = \{(\text{“}foo\text{”}, 3)\}$$

Under the hood, the Soufflé machine translates the source program into the following RAM representation:

```
1 PROJECT ("A", "foo", 1) INTO FunctionInvocations.
2 PROJECT ("B", "foo", 2) INTO FunctionInvocations.
3 PROJECT ···
4 y=sum t[2] : {FOR ALL t ∈ FunctionInvocations WHERE (t[1] = "foo")}
5   PROJECT ("foo", y) INTO TotalInvocations
```

At line 1 - 3, RAM inserts the initial EDB into the relations. At line 4, RAM extracts all tuples t from $FunctionInvocations$ such that its second column (0-indexed) is a “foo" and sum all their third columns. Finally it assigns the result of summation to y and inserts the new tuple $(\text{“}foo\text{”}, y)$ into relation $TotalInvocations$.

3.2 Injection and Witness Rewrite

The variable injection is a useful technique to control the instantiation of the atom in the aggregate body. In a safe rule, a variable $x \in var(\overline{T})$ is called an injected variable if x also occurs in the outer scope.

A common usage of variable injection is to perform a Database’s "group-by" operation. Consider the same Datalog example given in the above section, say we want to calculate each callees’ total number of invocations instead of just “foo". We can do that by introducing an injected variable to control how tuples in the aggregate body are instantiated:

$$\begin{aligned} TotalInvocations(target, y) \quad &:- FunctionInvocations(\_, target, \_), \\ &y = sum\ c : \{FunctionInvocations(\_, target, c)\}. \end{aligned}$$

The instantiation of variable $target$ is constrained by $FunctionInvocations(\_, target, \_)$, therefore, the aggregate is performed for each $target$ in $FunctionInvocations$.

The translated RAM representation is as follow:

```
1 PROJECT ("A", "foo", 1) INTO FunctionInvocations.
2 PROJECT ("B", "foo", 2) INTO FunctionInvocations.
3 PROJECT ···
4 FOR a ∈ FunctionInvocations
5   y=sum b[2] : {FOR ALL b ∈ FunctionInvocations WHERE (b[1] = a[1])}
6     PROJECT (a[1], y) INTO TotalInvocations
```

At line 4, the RAM chooses to fix a value b from $FunctionInvocations$, and for each fixed value b, it performs the sum aggregate on all tuples a whose second column equals to the second column of the fixed target b

Consider the same EDB, after evaluation, $TotalInvocations$ will yield the following results:

$$TotalInvocations = \{(\text{“}foo\text{”}, 3), (\text{“}bar\text{”}, 5)\}$$

In the context of Soufflé, we call the variable $target$ in the head clause a witness, as it witnesses the summation of variable c in the aggregate body.

We further extend the Soufflé syntax to allow implicit witness — users can omit writing the extra atom in the above example by just leaving it as an unlimited variable in the rule.

A variable x is a potential witness if it 1). Occurs in the aggregate body, and 2). It also occurs elsewhere and unlimitedly in the head or rule body. For example, the above Datalog rule can be implicitly written as:

$$TotalInvocations(target, y) :- \ y = sum\ c : \{FunctionInvocations(\_, target, c)\}.$$

Where $target$ is a potential witness since it occurs in the head of clause and also in the aggregate body, at the same time, $target$ is an unlimited variable in the rule.

We now define formally the semantic of witness by reducing it to normal Datalog:

$$H(\overline{Z}) \leftarrow S(\overline{W}),\ y = f\ x : \{A(\overline{T})\}$$

A variable $v \in var(\overline{Z} \cup \overline{W})$ is a witness if it satisfies the following conditions:

1. $v \in var(\overline{T})$.
2. v is an unlimited variable in $H(\overline{Z}) \leftarrow S(\overline{W})$ with the assumption that y is limited.

If potential witness v exists, we extract the body of the aggregate $A(\overline{T})$ into the body of the rule. Since all variables in $var(\overline{T})$ is limited by $A(\overline{T})$ in a safe rule, $A(\overline{T})$ must limit the potential unlimited variable v in the rule of the body as well.

$$H(\overline{Z}) \leftarrow S(\overline{W}),\ y = f\ x : \{A(\overline{T})\}, A(\overline{T}).$$

3.3 Aggregation Simplification

In many use cases, the same aggregates can appear in more than one rules. To avoid redundant computation, Soufflé rewrite a list of the aggregate body $A(\overline{T})$ into a single positive atom $P(\overline{T})$ so that once an aggregate over $P(\overline{T})$ is computed, all it subsequent usages can avoid recomputing the statistics.

Consider the following example:

$$\begin{aligned} UnsafeCall(target, y) \quad &:- \ y = sum\ c : \{FunctionInvocations(source, target, c), \\ &\quad \neg Safe(target),\ Safe(source)\}. \\ MaxUnsafeCall(target, y) \quad &:- \ y = max\ c : \{FunctionInvocations(source, target, c), \\ &\quad \neg Safe(target),\ Safe(source)\}. \end{aligned}$$

The first rule computes the total number of invocations from a safe subroutine to an unsafe subroutine, while the second rule computes the maximum number of such invocations. Those two rules differ only by the aggregate function, and

their aggregate bodies are the same. Without aggregate simplification, Soufflé produces the following RAM to compute the aggregate body on the fly:

```
FOR a ∈ FunctionInvocations
  IF (a[0] ∈ safe ∧ a[1] ∉ Safe)
    y=sum b[2] : {FOR ALL b ∈ FunctionInvocations WHERE (b[1] = a[1])}
      PROJECT (b[1], y) INTO UnsafeCall
```

This computation appears also in the execution of $MaxUnsafeCall$ with only the aggregate function differs. To save computation cost, Soufflé rewrites the program so that the aggregate body is stored into an individual relation:

$$
\begin{array}{ll}
@aggregate\_atom(source, target, c) & :- FunctionInvocations(source, target, c), \\
 & \neg Safe(target),\ Safe(source). \\
UnsafeCall(target, y) & :-\ y = sum\ c : \{@aggregate\_atom(source, target, c)\}. \\
MaxUnsafeCall(target, y) & :-\ y = max\ c : \{@aggregate\_atom(source, target, c)\}.
\end{array}
$$

As a result of the above rewrite, the body of the aggregate atom is computed only once, and the aggregate function simply iterates over the cached relation. In real-world use cases, where relations contain millions of tuples, the performance cost saved here can be optimistic.

```
FOR a ∈ @aggregate_atom
  y=sum b[2] : {FOR ALL b ∈ FunctionInvocations}
      PROJECT (b[1], y) INTO UnsafeCall
FOR a ∈ @aggregate_atom
  y=max b[2] : {FOR ALL b ∈ FunctionInvocations}
      PROJECT (b[1], y) INTO MaxUnsafeCall
```

We now define formally the semantic of aggregate simplification, for each Datalog rule that contains aggregate atoms, if the body of an aggregate atom does not contain only a single positive atom, it is rewritten into:

$$
\begin{array}{ll}
H(\overline{Z}) & \leftarrow S(\overline{W}),\ y = f\ x : \{P(\overline{V})\}. \\
P(\overline{V}) & \leftarrow S'(\overline{W'}), A(\overline{T}).
\end{array}
$$

Where

1. P is a single positive atom whose terms are $\overline{V}$.
2. $\overline{V}$ is a list of variables where $var(\overline{V})$ contains the union of injected variables in $var(\overline{T})$ and the cost variables x.
3. $S'(\overline{W'}) \subseteq S(\overline{W})$ is a set of conjunction of atoms and interpreted relations that provides limitation to variable in $\overline{V}$.

3.4 Nested Aggregations

We further extend the syntax to allow nested aggregates. Consider the following example where the most called target function is computed by nested aggregates.

$$MostCalled(target) \quad :-\ y = max\ z : \{z = sum\ i : \{FunctionInvocations(\_, target, i)\}\}$$

We define the semantic of the above aggregates by reducing them into a set of rules with only simplified aggregates. Since $MostCalled$ does not contains only a single positive atom, it can be simplified by extracting the cost variable z and injected variable $target$ into a new rule $@aggregate\_atom(z, target)$ and copy its body into the rule:

$$
\begin{array}{ll}
MostCalled(target) & :-\ y = max\ z : \{@aggregate\_atom(z, target)\}. \\
aggregate\_atom(z, target) & :-\ z = sum\ i : \{FunctionInvocations(\_, target, i)\}.
\end{array}
$$

Given the EDB in section 3.1, the program produces following result:

$$
\begin{array}{ll}
@aggregate\_atom & = \{(3, \text{“}foo\text{”}), (5, \text{“}bar\text{”})\} \\
MostCalled & = \{(bar)\}
\end{array}
$$

Formally speaking, we allow the nested aggregates in Soufflé with its semantic defined by keep unfolding the body of the aggregate until it contains only a single positive atom. Consider nested aggregates in the following form:

$$H(\overline{Z}) \quad \leftarrow S(\overline{W}),\ y_0 = f_0\ x_0 : \{A_0(\overline{T_0}), y_1 = f_1\ x_1 : \{A_1(\overline{T_1}), y_2 = \cdots\}\}.$$

where $y_2 = \cdots$ can contain further nested aggregates on the right hand side.

We apply the same simplification technique mentioned in section 3.3 until there is no nested aggregates exists in the program:

$$\begin{array}{ll} H(\overline{Z}) & \leftarrow S(\overline{W}),\ y_0 = f_0\ x_0 : \{P_0(\overline{V_0})\}. \\ P_0(\overline{V_0}) & \leftarrow S'_0(\overline{W'_0}), A_0(\overline{T_0}), y_1 = f_1\ x_1 : \{P_1(\overline{V_1})\}. \\ P_1(\overline{V_1}) & \leftarrow S'_1(\overline{W'_1}), A_1(\overline{T_1}), y_2 = \cdots \\ & \vdots \end{array}$$

We say a rule with nested aggregates is safe if all resulting rules are safe after simplification.

4 Related Work

A clear syntax and semantic definition for aggregates can be found in [11], however, its definition is reduced to *choice-least* and *choice-max* constructs [14] under greedy choice model [15, 16] while Soufflé only consider stratified programs. Similarly, [17], and [18] present and show that aggregates can be computed effectively under greedy fixpoint in a bottom-up logic framework, while our work put the focus on improving the performance of a concrete implementation. The expressiveness of aggregates was investigated in [19]. Semantics and definition of recursive aggregates are considered in [20]; however, Soufflé currently does not allow non-stratified aggregates.

5 Conclusion

In this work, we explored how the usability, functionality, and performance of aggregates in Soufflé can be improved. We have given a refreshed syntax and semantics to the Datalog programming language as it is seen in the Soufflé system, improving and extending upon earlier descriptions of the syntax and semantics of Datalog in [11]. We explain how Soufflé cooperates with Datalog semantic and its RAM machinery regarding aggregate. We introduce a syntax sugar called witness rewrite to improve the usability of the aggregate. Other than this, we provide aggregate simplification to improve performance for complex aggregate body and extend the syntax on top of the simplification technique to allow nested aggregates in Soufflé.

References

[1] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," *Proceedings of Computer Aided Verification*, vol. 28, pp. 422–430, 2016.

[2] M. Madsen, M.-H. Yee, and O. Lhoták, "From datalog to flix: A declarative language for fixed points on lattices," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: ACM, 2016, pp. 194–208. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908096

[3] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn, "Design and implementation of the logicblox system," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1371–1382. [Online]. Available: http://doi.acm.org/10.1145/2723372.2742796

[4] K. Hoder, N. Bjørner, and L. de Moura, "μz– an efficient engine for fixed points with constraints," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 457–462.

[5] W. R. Marczak, S. S. Huang, M. Bravenboer, M. Sherr, B. T. Loo, and M. Aref, "SecureBlox: Customizable secure distributed data processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2010.

[6] X. Ou, S. Govindavajhala, and A. W. Appel, "Mulval: A logic-based network security analyzer," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, ser. SSYM'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 8–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251398.1251406

[7] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears, "Boom analytics: Exploring data-centric, declarative programming for the cloud," in *EuroSys'10 - Proceedings of the EuroSys 2010 Conference*, 2010.

[8] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao, "Efficient querying and maintenance of network provenance at internet-scale," *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pp. 615–626, 2010.

[9] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, 2009.

[10] Y. Smaragdakis and M. Bravenboer, "Using datalog for fast and easy program analysis," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2011.

[11] S. Greco and C. Molinaro, "Datalog and logic databases," *Synthesis Lectures on Data Management*, vol. 10, pp. 47–57, 10 2016.

[12] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases: The Logical Level*, 1st ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[13] B. Scholz, H. Jordan, P. Subotić, and T. Westmann, "On fast large-scale program analysis in datalog," in *Proceedings of CC 2016: The 25th International Conference on Compiler Construction*, 2016.

[14] A. Van Gelder, K. A. Ross, and J. S. Schlipf, "The well-founded semantics for general logic programs," *J. ACM*, vol. 38, no. 3, p. 619–649, Jul. 1991. [Online]. Available: https://doi.org/10.1145/116825.116838

[15] S. Greco and C. Zaniolo, "Greedy algorithms in datalog with choice and negation," in *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming*, ser. JICSLP'98. Cambridge, MA, USA: MIT Press, 1998, p. 294–309.

[16] ——, "Greedy algorithms in datalog," *Theory Pract. Log. Program.*, vol. 1, no. 4, pp. 381–407, 2001. [Online]. Available: https://doi.org/10.1017/S1471068401001090

[17] S. Ganguly, S. Greco, and C. Zaniolo, "Minimum and maximum predicates in logic programming," in *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '91. New York, NY, USA: Association for Computing Machinery, 1991, p. 154–163. [Online]. Available: https://doi.org/10.1145/113413.113427

[18] S. Sudarshan and R. Ramakrishnan, "Aggregation and relevance in deductive databases," in *Proceedings of the 17th International Conference on Very Large Data Bases*, ser. VLDB '91. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991, p. 501–511.

[19] S. Cohen, *Aggregation: Expressiveness and Containment*. Boston, MA: Springer US, 2009, pp. 59–63. [Online]. Available: https://doi.org/10.1007/978-0-387-39940-9_1256

[20] D. B. Kemp and K. Ramamohanarao, "Efficient recursive aggregation and negation in deductive databases," *IEEE Trans. on Knowl. and Data Eng.*, vol. 10, no. 5, p. 727–745, Sep. 1998. [Online]. Available: https://doi.org/10.1109/69.729729