Aggregates in Soufflé

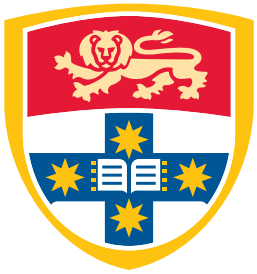
RACHEL DOWAVIC
SID: 450136425

Supervisor: Prof Bernhard Scholz
Associate Supervisor: Dr. Martin McGrane

This thesis is submitted in partial fulfillment of
the requirements for the degree of
Bachelor of Science (Computer Science) (Adv) (Mathematics) (Honours)

School of Information Technologies
The University of Sydney
Australia

15 January 2021



THE UNIVERSITY OF
SYDNEY

Student Plagiarism: Compliance Statement

I certify that:

I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure;

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to the University commencing proceedings against me for potential student misconduct under Chapter 8 of the University of Sydney By-Law 1999 (as amended);

This Work is substantially my own, and to the extent that any part of this Work is not my own I have indicated that it is not my own by Acknowledging the Source of that part or those parts of the Work.

Name: Rachel Dowavic

Signature: **Date**:

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 aggregation in a concise manner. 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 the syntax and semantics of aggregates in Datalog and provide extensions to previous treatments of aggregates such that users can aggregate over an arbitrary conjunction of literals. This implies that nested aggregation is now possible. We also give a semantics to a useful syntax sugar called the *witness construct* that allows users to easily fetch data that is *associated* with the min or max element of a relation. We also parallelise aggregate computation with the prefix sum method and measure a best case $1.1\times$ speedup. For all of the above changes, we implement these in the Soufflé engine. Overall, this thesis has increased the functionality, usability, and performance of aggregates in the Soufflé engine.

Acknowledgements

I want to say thank you to my supervisor Bernhard Scholz for his continued moral and academic support throughout my thesis year, and to Martin McGrane for helping me throughout my undergraduate degree with implementation work on the Soufflé engine. I also want to thank Sasha Rubin for working with me briefly on the semantics of aggregates in datalog. I must also thank the PLANG (Programming Languages) group members Abdul, Sam, Xiaowen, and David for answering my Soufflé-related questions.

CONTENTS

List of Figures

List of Tables

CHAPTER 1

Introduction

Datalog is a programming language that has experienced a resurgence of interest in recent years because it provides users with the ability to express real world problems concisely and easily. Some of these applications include security analysis (Marczak et al., 2010), cloud computing (Alvaro et al., 2010), and static program analysis (Smaragdakis and Bravenboer, 2011; Bravenboer and Smaragdakis, 2009). This is due to the fact that Datalog allows programmers to write programs succinctly and easily since it is declarative. This means that the programmer requires little knowledge of *how* the program will be executed. Instead, they must concern themselves only with *what* the program must do. It also means that programs of this nature are much easier to maintain, update, and debug. In procedural languages like Java, C++, and Python, it is easy to introduce bugs like undefined behaviour and infinite loops. These situations not only cost the programmer precious development time, but they also waste the resources of the businesses that employ programmers. It is therefore of paramount importance that the development process be 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.

Since logic programming languages allow the user to write succinct yet powerful programs, it is also important that the language be equipped with useful functionality. Datalog is not only a logic programming language, but it is also a database logic programming language. This means that *relations* are central objects in the language. Then it is important that users can easily *aggregate* over the contents of relations using functions like sum, min, max, $mean$, and $count$. For example, a user may want to issue a query to find the player of a sport with the minimum number of recent losses. This involves using the min aggregate function. It is also the case that previous treatments of aggregates have not been syntactically flexible. In our implementation vehicle Soufflé, aggregate functions are encoded as terms, for example, $x = mean\ z : \{\ Score(Student, z)\ \}$. This creates the possibility of *nesting*

aggregates within each other which has not previously been treated in the literature. We also consider extending the functionality of aggregates by giving users the possibility of computing aggregates over joins of relations which has also not previously been treated. In fact, we give users the possibility of computing aggregates over any first order formula with some basic restrictions. We also take heed of the fact that users often perform aggregate computations with the intention of retrieving relevant data *associated* with the aggregate result rather than the aggregate result itself. For example, it may be important to find the *names* of students with the highest test scores in their grades, rather than finding the highest test scores on their own. We refer to this construct as the *witness* construct and implement this functionality in the Soufflé engine. We also consider the fact that Soufflé is an engine that is used in applications with potentially billions of tuples, and we therefore seek ways to improve the performance of aggregate computations. We find that aggregates are associative operators, meaning that the order in which the operator is applied to tuples of a relation does not affect the aggregate result. This means that aggregate computations are parallelisable, and we take advantage of this finding in the Soufflé engine by parallelising aggregation computation by using the *prefix sum method* and achieve a $1.1\times$ speedup in the *best* case, and no effect in the worst case. As a result, we offer a new and improved version of Soufflé with aggregates that are far more functional than they were prior to this thesis work.

We now offer a general overview on the literature relating to aggregates and Datalog and highlight the current gaps in the literature for background.

1.1 Literature Review

The most authoritative descriptions of Datalog to date (Ceri et al., 1989; Abiteboul et al., 1995; Greco and Molinaro, 2015) have the limitation that all decidability proofs only consider Datalog terms to be constants or variables. Under this assumption, the termination of any Datalog program P can be easily proven using the finiteness of the Herbrand universe of P. Real world usage of Datalog in the context of Soufflé has outpaced these proofs in the fact that arithmetic functions and aggregates are also used as terms. Programs that use these constructs are known to terminate by the programmer, but the logic that underpins this has not yet been formalised. Work has however been done in this century to reason about the termination of subclasses of logic programs with function symbols under stable model semantics (Greco et al., 2013). In terms of documentation for live Datalog systems, it is the case that the most

thorough description of a live Datalog system is of LDL (Tsur and Zaniolo, 1986) and also of LogicBlox (Aref et al., 2015). Tsur and Zaniolo present the motives underpinning the creation of LDL as well as its overarching features, but the inner workings of the LDL machinery are not exposed.

In terms of the concept of aggregation in database management systems, there has been ample research in this area. Aggregates were originally introduced into implementations of relational databases (Astrahan et al., 1976; Chamberlin and Boyce, 1974; Chamberlin et al., 1976) since there was some demand to answer higher level queries that involved summarising table contents. It was only after their practical introduction that Klug extended relational algebra with an aggregate operator in the interests of assigning aggregate operators a formal semantics (Klug, 1982). This all took place in the context of relational database management systems however. Research into the logic programming paradigm was well underway by the late 80's (Zaniolo, 1986), and these systems originally used sets as primitives in order to capture the concept of aggregation (Tsur and Zaniolo, 1986), but subsequent papers in the cross hairs of logic programming, database management systems, and aggregation encoded aggregates with a `group_by` predicate that only operated over a single relation (Kemp and Stuckey, 1991; Mumick et al., 1990). Research from this point onwards focused heavily on giving both a declarative and an operational meaning to programs with recursive aggregation (Kemp and Stuckey, 1991; Zaniolo et al., 2017), especially since using recursively defined aggregates in some cases was a performance optimisation for common graph queries (Zaniolo et al., 2017).

This thesis aims to answer some of the limitations of previous work, mostly concerning the functionality of aggregates in a Datalog system. We also explore the possibility of parallelising aggregate computation in order to increase the speed at which these computations can take place.

1.2 Contribution

(1) We provide a formal syntax and semantics of aggregates as they appear in the Soufflé engine

(2) We provide the possibility to support recursive parameters in aggregate functions in the Soufflé engine

(3) We provide the possibility to aggregate over any number of joins and any number of constraints in the Soufflé engine

(4) We provide the possibility to arbitrarily nest aggregates within each other with some restrictions

(5) We provide support for a witness syntax sugar, thereby increasing the usability of aggregates in the Soufflé engine

(6) We parallelise aggregate computation in the Soufflé engine and achieve an at best $1.1\times$ speedup

1.3 Organisation

We firstly describe the constituents of a Datalog program in (2.2). These elements are described in a way such that they conform to the specification of the Soufflé engine, and therefore depart from the most up-to-date recounts of Datalog (Greco and Molinaro, 2015). We then describe the Soufflé engine pipeline and this leads us to assign a well-defined syntax and semantics to the intermediate imperative representation of an input Datalog program, RAM. We then assign a syntax and semantics to aggregates that operate over an arbitrary conjunction of literals. We give both a declarative semantics to aggregates as well as an operational semantics to aggregates by showing how they are represented in RAM via syntactic transformations. We then show how a simple syntactic transformation can support the appearance of an aggregate witness syntax sugar, and to finalise this paper, we present the way in which aggregate computation have been parallelised in the Soufflé engine.

1.4 Notation

We use the following notations with subscripts where convenient.

R, A, B, C, D	relation names
X, Y, Z	attribute names
P, G, F	predicate names
e	terms
t, u, v	tuples
x, y	variables
k	constants
i, j, n, m, l, h	indices
f, g	function names
θ	operators
L, J	literals
Φ	constraints, negated atoms
$\bar{A}$	a conjunction or sequence of A's

CHAPTER 2

Datalog

Since this thesis focuses on extending the functionality of aggregates in datalog, we now discuss the concept of datalog, why it is popular, and later give a formal syntax and semantics to datalog as it is realised in the Soufflé engine.

Datalog is a programming language that allows programmers to specify the behaviour of the program based on *rules*. An example of this in the context of a web project can be a statement like "If the user clicks on the red button, then a dropdown menu is displayed". This statement follows the pattern "If X, then Y", and so datalog allows us to encode this as a *rule* where we place Y on the left hand side of the implication symbol $\leftarrow$ and X on the right hand side of the implication symbol.

$$Displayed(dropDownMenu) \leftarrow Clicks(user, redButton).$$

Here, $Displayed$ and $Clicks$ are called *relations*. Being able to specify the behaviour of programs in this manner is incredibly useful because this is how design briefs are initially communicated to the programmer. This means that translating a design brief into its corresponding datalog representation may be much simpler than translating the same brief into the corresponding Java, JavaScript, or C++ representation.

Datalog also has the incredible benefit that recursive queries can be easily expressed. As background, any datalog program is initially loaded with a set of facts that are known to be true and this is called the extensional database. The set of inferences that the program is able to make from this set of facts is called the intensional database. One example where datalog can take advantage of recursion is in answering queries about a family tree. If the extensional database of a datalog program is loaded with facts about immediate relationships, then this program can make deductions about ancestral relationships. For

example, let the extensional database of a datalog program contain the following facts

$$parent(Bavier, Xander).$$

$$parent(Crander, Xavier).$$

These facts mean that Bavier is the parent of Xander, and that Crander is the parent of Xavier. Then, the program could contain the rules

$$ancestor(X, Y) \leftarrow parent(X, Y).$$

$$ancestor(X, Y) \leftarrow parent(X, Z), ancestor(Z, Y).$$

The first rule means that if X is the parent of Y, then X is the ancestor of Y, and the second rule means that X is the ancestor of Y if X is the parent of Z and Z is the ancestor of Y. This second rule is recursive and allows us to find ancestor relations that span any length. Then if we make the query

$$? \leftarrow ancestor(X, Xavier)$$

where X is a variable, then we are effectively asking the datalog program to find all ancestors of Xavier. We should find that the ancestors of Xavier are both Bavier and Crander. This means that the datalog program should have reported the following facts.

$$ancestor(Bavier, Xavier).$$

$$ancestor(Crander, Xavier).$$

It is also worth emphasising how succinct datalog programs can be compared to their imperative counterparts. Take for example a program that computes the transitive closure of a graph. That is, the program computes all pairs of nodes that are reachable on the graph. In datalog, this can be expressed succinctly in a matter of two lines (2.1), whereas the C++ equivalent takes a far higher number of lines to express and is more difficult to understand, maintain, and debug (2.2).

In effect, the datalog equivalent of a C++ program should be easier to develop and maintain. As business needs change, datalog programs can easily be rewritten in order to accommodate these changes.

$$Edge(0,1).\ Edge(0,2).\ Edge(2,3).$$
$$Path(x,y) \leftarrow Edge(x,y).$$
$$Path(x,z) \leftarrow Path(x,y), Edge(y,z).$$

FIGURE 2.1: Datalog Transitive Closure Program

```
void transitiveClosure(int graph[][V])
{
    int path[V][V], i, j, k;
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            path[i][j] = graph[i][j];

    for (k = 0; k < V; k++)
    {
        for (i = 0; i < V; i++)
        {
            for (j = 0; j < V; j++)
            {
                path[i][j] = path[i][j] ||
                  (path[i][k] && path[k][j]);
            }
        }
    }
}
```

FIGURE 2.2: C++ Transitive Closure Program

2.1 Soufflé

Soufflé is a highly performant and highly parallel implementation of datalog. The general pipeline of Soufflé is to translate a datalog program into a corresponding relational algebra machine (RAM) representation, and then into a highly parallel C++ program (2.3). At each stage, optimisations can be made in order to increase the speed of the resulting C++ program.

FIGURE 2.3: The Soufflé Pipeline

Implementation of Soufflé originally began at Oracle Labs in Brisbane where it was used for static program analysis. Soufflé differs from other competing datalog engines since it is able to adapt the representation of relations depending on their use case and is also far more paralellisable. Competitors like LogicBlox and μZ only use one approach for storing all relations (Aref et al., 2015; Scholz et al., 2016). It is also the case for LogicBlox that all datalog programs are run single-threaded. This is why Soufflé is able to exhibit performance characteristics that are equivalent to hand-crafted state-of-the-art tools (Scholz et al., 2016).

In order to prime the reader, we now offer the syntax and semantics of datalog in full as it is realised in the Soufflé engine. We offer an updated syntax of datalog since the traditional description of datalog departs in several aspects from the Soufflé engine, and we also offer a semantics via the immediate consequence operator as well as via the intermediate RAM translation of a datalog program that is used by the Soufflé pipeline. This section can be used for reference.

2.2 Formal Syntax and Semantics of Datalog

We firstly describe the grammar of datalog with arithmetic functions and stratified negation. Datalog is, as can be expected, a context-free language. We do not explicitly present the production rules, but this should become clear through the definition of each syntax element.

2.2.1 Terms

A *term* is recursively defined as being a variable, constant, or where f is an arithmetic function and $e_1, \ldots, e_n$ are all terms of numeric type, then $f(e_1, \ldots, e_n)$ is a term. An *arithmetic function* is a function $f \colon \mathbb{R}^n \to \mathbb{R}$ composed only of operators in $\{+, -, \times, exp, \div, mod\}$. We note that $\mathbb{R}$ as it is used in this thesis does not refer to the real number line, but the much smaller space covered by the 64 bit IEEE 754 standard real numbers. The application order within an arithmetic function can be disambiguated with smooth parentheses. A term is *compound* if it is not a variable or constant. By *numeric type*, we mean all numbers (i.e. where e is a constant in $\mathbb{R}$), and all terms e of the form $f(e_1, \ldots, e_n)$, since the codomain of f is $\mathbb{R}$. We represent numeric type by $\mathbb{R}$ and terms can also be of *symbol type*, meaning that they are not of numeric type. We denote symbol type with $\mathbb{S}$, i.e. $e \in \mathbb{S}$ if

$e \notin \mathbb{R}$. This is a simplification of the type system of Soufflé but we avoid a full treatment in order to stay in scope.

2.2.2 Predicates and Relations

A *first order predicate* $P(x_1, \ldots, x_n)$ is a function with signature $P\colon D_1 \times \ldots \times D_n \to \{true, false\}$, where n is the *arity* of P, and D_i is the domain of the i^{th} *argument* of P, i.e. the domain of x_i. Arguments are always terms, and it turns out that all top-level[1] terms are arguments. Since all terms belong to either $\mathbb{R}$ or $\mathbb{S}$, D_i is always either $\mathbb{R}$ or $\mathbb{S}$ for all $i \in \{1, \ldots, n\}$. We refer to the $x_1, \ldots, x_n$ as the *arguments* of the predicate P. To *evaluate* $P(x_1, \ldots, x_n)$ is to discover the function P's output under the inputs $x_1, \ldots, x_n$. In general, deciding whether a given predicate expression maps to true or false is the responsibility of the evaluation machinery, but some of these are clearly evaluation machinery-invariant and are therefore forwardly specified wherever this is the case. The *negation* of a first order predicate $\neg P(x_1, \ldots, x_n)$ is also a function, where for each input sequence $x_1, \ldots, x_n$, if $P(x_1, \ldots, x_n) \mapsto true$, then $\neg P(x_1, \ldots, x_n) \mapsto false$, else $\neg P(x_1, \ldots, x_n) \mapsto true$. We call this sequence $(x_1, \ldots, x_n)$ a *tuple*, and in general, a tuple $t = (x_1, \ldots, x_n)$ is any sequence of arguments. We refer to the $x_1, \ldots, x_n$ as the *elements* of t, and n is the *size* of the tuple. Since terms can have a type, tuples can have a *sort*. This is simply the sequence of types of its elements. As an example, $(1, 2, 3)$ is a tuple with sort $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$ and size three.

A *relation* $R[X_1\colon D_1, \ldots, X_n\colon D_n]$ is simply a finite set of tuples that coincide on their sort $D_1 \times \ldots \times D_n$, where D_i is always either $\mathbb{R}$ or $\mathbb{S}$ for all $i \in \{1, \ldots, n\}$. The $X_1, \ldots, X_n$ are the *attributes* of R. The expression $R(e_1, \ldots, e_n)$ is a first order predicate that is *backed* by the relation R. This means that $R(e_1, \ldots, e_n) \mapsto true$ if $(e_1, \ldots, e_n) \in R$ and false otherwise. This is of course only deducible if none of $e_1, \ldots, e_n$ contain variables. In general, deciding whether a given tuple belongs to a relation is the responsibility of the compiler for the datalog program and we will address this in (2.3.1). To give an example of a first order predicate that is backed by a relation, imagine that there is a relation $Human[X : \mathbb{S}]$ with one member $(\text{"}Socrates\text{"})$. Then the expression $Human(\text{"}Socrates\text{"})$ is a first order predicate that is backed by the $Human$ relation. We can represent a relation $R[X_1 : D_1, \ldots, X_n : D_n]$ and its

[1] A top-level term is a term that does not have a term as a parent in the abstract syntax tree.

m members $(k_{1,1}, \ldots, k_{1,n}), \ldots, (k_{m,1}, \ldots, k_{m,n})$ in tabular form as is done by Codd (Codd, 1970) and Abiteboul (Abiteboul et al., 1995).

R	X_1	$\ldots$	X_n
	$k_{1,1}$	$\ldots$	$k_{1,n}$
	$\ldots$	$\ldots$	$\ldots$
	$k_{m,1}$	$\ldots$	$k_{m,n}$

FIGURE 2.4: Tabular Representation of a Relation

As an example, take the relation $R[X_1 : \mathbb{R}, X_2 : \mathbb{R}]$ with a sole member $(1, 2)$.

EXAMPLE 2.2.1 (Relation).

R	X_1	X_2
	1	2

Projection is a well-known operator that acts on a relation, and it can be defined as the restriction of each tuple to a sequence of its elements corresponding to a subset of attributes of R. The output of a projection operator is a relation, implying that duplicate tuples are eliminated. We define this operator in the relational calculus as

$$\pi_{X_1,\ldots,X_m}(R) = \{t[X_1, \ldots, X_m] | t \in R\}$$

where $t[X_1, \ldots, X_m]$ denotes the restriction of the tuple t to the attributes $X_1, \ldots, X_m$.

In addition to this, we introduce a relation *restriction* operator, and we write this as $R[\bar{X}]$, where $\bar{X} \subset \{X_1, \ldots, X_n\}$, and $X_1, \ldots, X_n$ are the attributes of R. This differs from a projection since the output of a projection is always a set, and the relation restriction operator preserves duplicate tuples. In this way, it is analogous to sliding a piece of paper over the unwanted columns of R. Take the relation $R[X_1 : \mathbb{R}, X_2 : \mathbb{R}]$ in (2.2.2) as an example.

EXAMPLE 2.2.2 (Relation).

R	X_1	X_2
	1	2
	3	4
	3	2

Then the expression $R[X_2]$ yields the multiset of tuples in (2.2.3).

EXAMPLE 2.2.3 (Relation Restriction).

R	X_2
	2
	4
	2

This relation restriction operator will help us concisely express the semantics of aggregates in datalog in (3.3). In order to support group-by semantics, it will suffice for our purposes to allow a second optional *filtering*. This must be an equality constraint on one of the attributes of R, and we write this $R[\bar{X}|X_i = k]$ where k is some constant in $\mathbb{R}$ or $\mathbb{S}$, and $i \in \{1, \ldots, n\}$. We can also filter by multiple attributes by comma delimiting the constraints, i.e. $R[\bar{X}|X_i = k_1, \ldots, X_j = k_m]$. As an example, the expression $R[X_2|X_1 = 3]$ yields the multiset (2.2.4), where R is the relation in (2.2.2).

EXAMPLE 2.2.4 (Relation Restriction with Filtering).

R	X_2
	4
	2

2.2.3 Literals

A *literal* is either an atom, a negated atom, or a constraint. An *atom* is any first order predicate that is backed by a relation, i.e. $R(e_1, \ldots, e_n)$ is an atom if R is a relation. A *negated atom* $\neg R(e_1, \ldots, e_n)$ is an atom that maps to $true$ if $(e_1, \ldots, e_n) \notin R$, and thus it is the logical negation of an atom. A *constraint* is either $true$, $false$, or a binary predicate $e_1\ \theta\ e_2$ where $\theta \in \{=, \neq, <, \leq, >, \geq\}$. Although constraints have no explicit predicate symbol, we understand that they are equivalent in that we can easily assign them a predicate symbol name based on the operator θ, add a negation symbol if necessary, and convert the infix notation to prefix notation. It is always (somewhat obviously) the case that $true \mapsto true$, $false \mapsto false$ and $e_1\ \theta\ e_2 \mapsto true$ if e_1 and e_2 are constants and the statement is true in $\mathbb{R}$, otherwise $e_1\ \theta\ e_2 \mapsto false$. As an example, $(2 < 4) \mapsto true$, and $(2 < x)$ maps to either $true$ or $false$ depending on how x is *grounded*, but again, this relates to how a datalog program is evaluated, and we will address grounding in (2.3.1). It is worth noting that although we support the negation of an atom, there is no need to syntactically support the negation of a constraint $e_1\ \theta\ e_2$ since $\neg(e_1\ \theta\ e_2)$ can be translated into a semantically equivalent predicate by deleting the negation and replacing the operation θ by its complement, i.e. $\neg(e_1\ \theta\ e_2) \leftrightarrow (e_1\ \theta^c\ e_2)$, and the complement θ^c can be found using (2.5).

θ	θ^c
$=$	$\neq$
$<$	$\geq$
$\leq$	$>$

FIGURE 2.5: Datalog Constraint Operators and their Complements

2.2.4 Formulas

A *formula* is defined recursively. If P is a predicate symbol, then $P(e_1, \ldots, e_n)$ is a formula, assuming that the arity of P is n. If F and G are formulas, then $\neg F$, $F \wedge G$, and $F \leftarrow G$ are also formulas, where the symbols $\neg, \wedge, \leftarrow$ are the logical connectives of propositional logic. It is also the case that if F is a formula and x is a variable, then $\forall x F$ is a formula. We call the expression $\forall x$ the *quantifying clause*, and the *scope* of x is F. The x in $\forall x$ is the *quantifying variable*. We can use smooth parentheses to disambiguate the scope when necessary. We can also aggregate many adjacent quantifying clauses by comma delimiting the variables, i.e. $\forall x \forall y$ becomes $\forall x, y$. In the formula $\forall x F$, all variables occurring in F are *bound* to the variable x in the quantifying clause. To avoid ambiguity, this means that all variables in the quantifying clause must be uniquely named. A variable in F is *free* if it does not appear in the quantifying clause. For example, the variable z in $\forall x, y(x < (y + z))$ is free whereas x and y are bound because only x and y appear in the quantifying clause. If a quantifying clause appears nested within another quantifying clause where the set of variables in the clause is not disjoint, then occurrences the duplicate variables in the inner scope will be bound to the innermost quantifying clause. This is the same as "shadowing" in imperative programming languages. An example of this is in the formula $\forall x((x + 5 < 2) \wedge (\forall x(x < 6)))$, where the occurrence of x in $(x < 6)$ is bound to the innermost quantifying clause.

2.3 Datalog

A datalog *rule* is a first order formula that is an implication on the second highest level. The right hand side of this implication is a conjunction of literals called the *body*, and the left hand side is a single atom called the *head* of the rule. The entire formula is universally quantified by all variables occurring within

it. That is, its general structure is

$$\forall x_1, \ldots, x_n (A \leftarrow L_1 \wedge \ldots \wedge L_m)$$

where A is the head atom, and $L_1, \ldots, L_m$ are the body literals. This is also known as a first order logic horn clause where recursion is permitted. As shorthand, since body literals can only be conjuncted, we replace $\wedge$ with a comma and drop the quantifying clause since it is inferrable by inspecting the set of variables occurring within the formula. This is possible only because neither the head nor the literals of the body can contain nested universal quantifiers, but we will see when we introduce aggregate bodies in (3.1) that variables in the aggregate body are only bound to this quantifying clause under certain conditions. In summary, we write the above formula simply as

$$A \leftarrow L_1, \ldots, L_m.$$

It is possible to capture the concept of a rule with an empty body, but in this case, no implication symbol is needed because the left hand side is invariably true, and is thus appropriately termed a *fact*. If the head atom were to contain variables, i.e. $A(x)$, then this means $\forall x A(x)$. We deal with infinite domains $\mathbb{R}$ and $\mathbb{S}$, and so this translates to an infinite number of facts. The purpose of the compiler is to find all facts that satisfy this rule, but since there are infinitely many, the compiler will not be able to terminate, and therefore we disallow this. Facts cannot contain variables in their arguments. An example of a fact is $R(1, 2)$, assuming that $R[X_1 : \mathbb{R}, X_2 : \mathbb{R}]$ is some relation. A datalog *program* P is described by the tuple $(\bar{R}, \bar{\boldsymbol{R}})$ where $\bar{R}$ is a set of datalog rules and $\bar{\boldsymbol{R}}$ is a set of relations. We assume that each relation either has some fact associated with it, or that its atom is the head of some rule in $\bar{R}$, otherwise the relation will be invariably empty. The intention is for P to be evaluated at some stage using the rules in $\bar{R}$ to derive new facts. Any set of facts that satisfy all rules in P is referred to as a *model* for P. As we prefer not to derive more facts than are strictly necessary, we tend to aim to find a model that is minimal. An even more basic requirement is to find a model that is finite, since this means that the evaluation machinery can terminate. The following section will describe how we can both find a model at all, and how we can ensure that this model is finite. We describe this using the classic Tarski fixpoint operator T_P and also using a translation into Soufflé's RAM (relational algebra machine). Both mechanisms will derive the same model for any fixed P, and we will show that this model is also minimal.

2.3.1 Semantics

The overarching goal of the evaluation machinery is to derive an appropriate model for the given datalog program. Intuitively, this means that all knowledge that *can* be built using the rules and facts of the program are made known by the time the program has been completely executed. It is however always possible to add arbitrary facts to the model if there is no rule in the program to contradict them. We avoid this loophole by stating that we want to operate under the closed world assumption (CWA). This means we want to avoid deriving any facts whose existence is unnecessary to satisfy the program's rules and facts. The consequence of this is that the model that we derive will be minimal. How this minimal model is derived is traditionally captured by the *immediate consequence operator* T_P of P. This is used in any textbook covering the semantics of datalog (Abiteboul et al., 1995; Greco and Molinaro, 2015). Before giving an extended definition of the operator, we need that all variables in each rule of a datalog program occur *groundable*. This means that they appear as the argument of some body atom, i.e. if $x_1, \ldots, x_m$ occur in some rule R, then there is some body atom A with argument x_i for each $i \in \{1, \ldots, m\}$. This is not the only way a variable can become groundable however. If e is either a term containing only groundable variables or a constant, and x is some variable, then the equality constraint literal $x = e$ makes x groundable. To give an example of a rule containing instances of both groundable and ungroundable variables, let the variables x, y, z, w occur in some rule R where R is defined by

$$A(x, y) \leftarrow B(x, y), y < z, w = x.$$

Then x and y occur *groundable* because both x and y appear as arguments in the atom $B(x, y)$, but z does not appear groundable because it only occurs as an argument of the constraint $y < z$. y appears here too, but it also appears in $B(x, y)$, so it is still groundable. Finally, the variable w appears as an argument of an equality constraint where the other argument is a groundable variable, and so w is also groundable. The reason that we impose this restriction of *groundability* is that it makes the process of finding valid assignments of variables during evaluation much easier since we can scan the contents of the relation which gives the variable groundability. I.e., if $A(x) \leftarrow B(x)$ is a rule in some datalog program, then x occurs groundable, and we can find valid assignments of x by scanning the contents of B.

TABLE 2.1: Variable Groundings of a Datalog Rule R

R	x_1	...	x_m
	k_1	...	k_m

Now that we operate under the assumption that all variables occurring in a rule occur groundable, we can now describe the fixpoint operator T_P. First we will describe the intuition of this operator and then give a concise description. Each rule R in a datalog program is in implication form, i.e.

$$A \leftarrow L_1, \ldots, L_n$$

If it can be found that $L_1, \ldots, L_n$ are all true, then A will be true, since it is implied by the truth of $L_1, \ldots, L_n$. This is the definition of the implication connective $\leftarrow$. The T_P operator attempts to make $L_1, \ldots, L_n$ true so that it can derive A. At first, T_P is ignorant to any knowledge, and it instantiates a set of ground atoms $I := \{\}$ to indicate this. Then it must find a way to leverage the rules in $\bar{R}$ of P so that I can grow until a point where it does in fact represent a model for P. We must now consider that $L_1, \ldots, L_n$ is a conjunction of atoms, negated atoms, and constraints, and devise a method to find out whether we can safely say that all of $L_1, \ldots, L_n$ are true. We now describe how this can be achieved. Begin with body literals that are atoms A. If A is grounded, i.e. all of its arguments are constants $A(k_1, \ldots, k_m)$, then if $A(k_1, \ldots, k_m) \in I$, we have satisfied this literal. Otherwise, if $A(k_1, \ldots, k_m) \notin I$, then the body is unsastisfiable and we give up for now. If some of the arguments of A are variables and some are reducible to constants, i.e. $A(x_1, \ldots, x_l, k_{l+1}, \ldots, k_m)$, then we can select a set of atoms $\bar{A} \in I$ such that they coincide on all constants. Every atom in $\bar{A}$ is a candidate. Fix one $A(k_1, \ldots, k_m) \in \bar{A}$ and accept the groundings $x_1 := k_1, \ldots, x_l := k_l$ for the remainder of literals. If at any point while evaluating the truth value of the remainder of the literals under these groundings, it is shown that the literal is unsatisfiable, fix the next atom $A(k_1, \ldots, k_m) \in \bar{A}$. Here is where our assumption about groundability assists us. Each variable occurring in the rule must occur groundable, i.e. as the argument of some atom. Then, once all atoms in the rule have been assigned an atom belonging to I, all variables will have a grounding, and we can refer to these groundings with some convenient map data structure (2.1) when we are evaluating the truth values of the remaining constraints and negated atoms in the rule body.

Now, take all constraint literals $\bar{\Phi}$ in R. Using the groundings we have received, test the truth of Φ for each $\Phi \in \bar{\Phi}$. To make this more clear with an example, let the literal $x_1 < x_2$ occur in R. According to (2.1), these are given groundings by k_1 and k_2, both being in $\mathbb{R}$. Test whether $k_1 < k_2$ is true under $\mathbb{R}$. If this succeeds, continue. Otherwise, we will need to fix new groundings for some subset of the variables in the rule. If we have already cycled through all valid groundings, R is unsatisfiable and we give up. If instead we reach a point where all atoms have been assigned some corresponding atom in I and all constraints have been satisfied, we can now see if the negated atoms appearing in the rule body can be satisfied. This now leads us to discuss the concept of *stratified negation*. For atoms, it is clear that we can simply inspect the current contents of I to see whether the atom is satisfiable. If we use this same approach for a negated atom $\neg A(k_1, \ldots, k_m)$, we should see that the corresponding atom $A(k_1, \ldots, k_m)$ is *not* in I. However, we are currently in the context of evaluating *one* of the rules of P. There may be another rule of which A is the head. If we evaluate this rule first, we may derive the atom $A(k_1, \ldots, k_m)$, and so it is not possible to be sure that $A(k_1, \ldots, k_m)$ cannot be derived at some stage in the future. We now define a mechanism called *stratification* that allows us to define a sensible order of the evaluation of rules in P.

DEFINITION 2.3.1 (Stratification). A stratification of P is a partition $S_0, \ldots, S_n$ of $\bar{\boldsymbol{R}}$ such that the following holds for every rule R in $\bar{R}$ and for every $A, B \in \bar{\boldsymbol{R}}$:

(1) If A is the head atom of R and B appears as an atom in the body of R, then B is in the same stratum as A or a lower stratum than A.
(2) If A is the head atom of R and B appears as a negated atom in the body of R, then B is in a lower stratum than A.

Using this, we can assure that by the time we are trying to ascertain whether $A(k_1, \ldots, k_m)$ cannot be derived, we have already evaluated any rule of which A is the head, and so we have derived all members of A. Then, for all negated atoms $\neg A(k_1, \ldots)$ in the body of R, simply check that $A(k_1, \ldots) \notin I$. Then this counts as a satisfaction of $\neg A(k_1, \ldots)$. At this stage, we have now evaluated the truth value of all literals in the body of the rule. If all steps have been successful, i.e. L_i is *true* for all i then the head atom will join the set of currently derived atoms in I. After repeating this process for every stratum, the conjecture is that T_P will no longer be able to derive further atoms, and therefore the evaluation will be

over. For this to actually be the case, we need some notion of *safety*. Roughly, this means that it will be impossible to derive infinitely many new facts.

2.3.2 Safety

The first condition of safety has already been mentioned, and that is that all variables occurring in a rule must occur *groundable* at some point. The second condition involves the relationship between terms in the head of a rule and variables in the body of a rule. Take the rule R to be defined by

$$A(x+1) \leftarrow A(x).$$

Our current notion of safety does not prohibit us from writing R. However, when this rule is executed, imagining that we have some fact $A(0)$, we will be constructing an infinite sequence $A(1), A(2), \ldots$. Since this sequence is infinite, the model for the program containing R is infinite, and so the evaluation machinery will not be able to terminate. Therefore, we must impose additional safety conditions on rules. In order to do this, we must define the notion of a relation dependency graph.

DEFINITION 2.3.2 (Relation Dependency Graph). Let $\bar{\boldsymbol{R}}$ be the relations of a datalog program P.

(1) Draw a node for each $A \in \bar{\boldsymbol{R}}$.
(2) Fix one $A \in \bar{\boldsymbol{R}}$ and consider the rules in $R \in \bar{R}$ of which A is the head. Fix one of these rules R. For each atom $A_i \in \bar{A}$ appearing in the body of R, draw a directed edge from A to A_i.

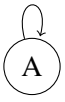
We now show an example of a program P along with its dependency graph.

EXAMPLE 2.3.1 (Counting Program).

$$A(0).$$

$$A(n+1) \leftarrow A(n), n < 100.$$

The corresponding relation dependency graph for P is the following



From this, it is easy to extrapolate that the sequence formed by $a_{n+1} = a_n + 1$ is upper bounded by $a_n < 100$ and so only a finite number of new terms can be constructed since we work along $\mathbb{R}$ where $\mathbb{R}$ only constitutes the real numbers representable in IEE 754.

The next example shows a cycle in the relation dependency graph with length greater than one (2.3.2).

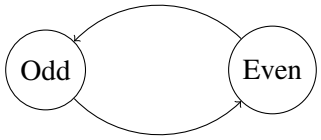
EXAMPLE 2.3.2 (Odd-Even Program).

$$Even(0).$$

$$Odd(n+1) \leftarrow Even(n), n < 10.$$

$$Even(n+1) \leftarrow Odd(n), n < 10.$$

Then, the corresponding relation dependency graph for P is



This also does not lead to the construction of an infinite sequence of terms because in both rules, we apply the function $n \mapsto n + 1$ and n is also upper bounded in both rules. This example makes clear the fact that we must consider each cycle on the dependency graph and the sequence that is constructed along each cycle. Now we introduce an extension of our current safety conditions. Let $f(\bar{x})$ be an argument of the head of a rule R. Then if all variables $x \in \bar{x}$ occur groundable in atoms of a lower stratum in R, then the R is *safe*. This is clear because a new term will only be created once for each element in the cross product of relations backing the atoms which make $\bar{x}$ groundable, and this set is already assumed to be a finite size. Now, let there be a cycle in the dependency graph of P. Consider the sequence $a_0, a_1, \ldots$ constructed by this cycle. If $a_n \mapsto a_{n+1}$ is increasing and upper bounded, or decreasing and lower bounded, then P is also *safe*.

Given that all rules in a program P are safe, this implies that only a finite number of terms can exist. Take the number of terms to be n. The number of relations appearing in P is $|\bar{\boldsymbol{R}}|$, and take the maximum arity of any relation in $\bar{\boldsymbol{R}}$ to be k. Then if all terms appear in all relations, then an upper bound for number of facts that can possibly be derived is $|\bar{\boldsymbol{R}}| \cdot n^k < \infty$. This shows that the model to be derived is finite,

and hence the program will terminate. In order to show that the model is minimal, we argue that since we only derive facts when absolutely necessary, i.e. given a rule $R = A \leftarrow L_1, \ldots, L_m$, we only derive A when $L_1, \ldots, L_m$ are all true. If we were not to derive A, then then I would not be a model, since we would violate the meaning of the implication connective $\leftarrow$.

2.4 RAM

2.4.1 The RAM Language

We will now describe the constituents of a RAM program before describing how a datalog program is transformed into a RAM program. Through this description, it should become clear that the Tarski fixpoint semantics agrees with Soufflé's RAM semantics. We can define a RAM program $P_R = (D_{\bar{\boldsymbol{R}}}, M, \bar{S})$, where M is the main program, $\bar{S}$ represents a set of subroutines, and $D_{\bar{\boldsymbol{R}}}$ is a declaration of the set of relations from P. Let P have $n + 1$ strata. The syntax of P_R is the following

```
PROGRAM
    DECLARATION
        <Relation(s)>
    END DECLARATION
    SUBROUTINE stratum_0
        <Query>
    END SUBROUTINE
    ...
    SUBROUTINE stratum_n
        <Query>
    END SUBROUTINE
    BEGIN MAIN
        CALL stratum_0
        ...
        CALL stratum_n
    END MAIN
```

```
END PROGRAM
```

The purpose of the main routine is to call each subroutine in sequence such that the stratification of P is respected.

RAM relations are written `A(X:D)` where $A[X : D]$ is some datalog relation. We now define and give semantics to RAM queries. Both the main program M and subroutines $\bar{S}$ have a `Statement` as their child, where `Statement` can either be a `Query`, a `CALL` to a subroutine, or a Sequence of `Statements` (a `ListStatement`). A `Query` is a wrapper for a RAM `Operation` and is written

```
QUERY
    <Operation>
```

A RAM `Operation` is either a `Scan`, `Projection`, or `Filter`. A RAM `Scan` iterates over all tuples `t` in a relation `A` and is written

```
FOR t IN A
    <Operation>
```

A RAM `Projection` has the effect of adding a tuple `t` into a relation `A` and is written

```
PROJECT t INTO A
```

In a `Projection` statement, it is often necessary to access a particular component of a tuple, and so the accessing of the i^{th} component of a tuple `t` is written `t.i`. As an inverse operation, tuples can be constructed using smooth parentheses. That is, where `t.i, ..., u.j` is a sequence of tuple elements, `(t.i, ..., u.j)` is a tuple. A RAM `Filter` conditionally executes a RAM `Operation` and it is written

```
IF <RAM Condition>
    <RAM Operation>
```

A RAM `Condition` is either a `Conjunction`, `Negation`, `ExistenceCheck`, or `Constraint`. A `Conjunction` is simply a conjunction of RAM `Conditions` and is written `(C AND D)` where

`C, D` are conditions. A RAM `Negation` is written `(NOT C)` where `C` is a condition, and acts as a logical negation. An `ExistenceCheck` is used to check whether a tuple `t` is a member of the relation `A` and is written `t ∈ A`. The expression returns true if `t` belongs to `A` and false otherwise. A RAM `Constraint` is a binary constraint where the operands are tuple elements `t.i, u.j` and the operator `OP` is an operator in $\{=, \neq, <, >, \geq, \leq\}$. This is written `(t.i OP t.j)`.

Since the `Scan` operation iterates over all tuples of the relation, this is a linear operation. It is not immediately obvious whether an `ExistenceCheck` also requires a full scan of the relation, but in practice, it is true that the C++ translation maps this operation to a B+-Tree index access. The implication is that all relations in P_R for which there is an `ExistenceCheck` must have a corresponding B+-Tree built for it. `Constraints` can be evaluated in constant time assuming that tuple elements can be accessed in constant time, and so can `Negations` and `Conjunctions`, and therefore it is true that any `Condition` of a `Filter` is evaluated in at most logarithmic time. Using this, we can deduce the time complexity of a RAM program by inspection.

We have now described all components of the RAM language that are necessary for executing non-recursive queries. We now describe components of RAM that are necessary for executing recursive queries before detailing the process by which datalog queries are translated into their corresponding RAM representation.

A `Statement` can be executed ad infinitum using the `Loop` construct. A `Loop` is a direct child of a `Statement` on the type hierarchy.

```
LOOP
    <Statement>
END LOOP
```

This construct tends to be paired with an `Exit` statement that evaluates the condition `EXIT (<RAM Condition>)` and exits the outer `Loop` statement in the case where `<RAM Condition>` evaluates to true.

It is also possible to swap the contents of two relations `A, B` with the statement `SWAP (A, B)`, and to clear a relation `A` with the statement `CLEAR A`.

2.4.2 Non-Recursive AST to RAM Translation

Now that we have an understanding of the elements of the RAM language, we can use these in order to translate a non-recursive datalog rule into a RAM query. In order to prepare for the translation, we must make all implicit equality constraints explicit. For example, the rule

$$A(x) \leftarrow B(x), C(x).$$

contains an implicit equality constraint in that the two occurrences of x in the body of this rule are bound to the same quantifying variable, i.e. $\forall x(A(x) \leftarrow B(x), C(x))$. We remove this by extending the body with an equality constraint and renaming the second occurrence of x in the body, i.e.

$$A(x) \leftarrow B(x), C(x_1), x = x_1.$$

This can be achieved for any rule containing implicit equality constraints using the following transformation.

PROCEDURE 2.4.1 (Eliminating Implicit Equality Constraints). Let R be a rule where the variable x occurs $m > 1$ times in the body of R.

(1) Rename each variable $x \mapsto x_{i-1}$ for all $i \in \{2, \ldots, m\}$
(2) Extend the body of R with the literal $x = x_i$ for all $i \in \{1, \ldots, m\}$

We now describe a procedure to translate an arbitrary non-recursive datalog rule R into a RAM `Query`. Let R be defined by

$$A \leftarrow L_1, \ldots, L_n$$

R can only contain atoms $\bar{A}$, negated atoms, and constraints $\bar{\Phi}$, and therefore we can write the body of R as the conjunction of these elements since the conjunction operator is commutative, i.e.

$$A \leftarrow \bar{A}, \bar{\Phi}.$$

Let us also assume that the variables $x_1, \ldots, x_m$ occur in R. If $\bar{A} = \{A_0, \ldots, A_k\}$ is nonempty, then create a `Scan` as the `Operation` of the root `Query` for the first atom A_0, and for subsequent atoms A_i, create a `Scan "FOR ti IN Ai"` to be the `Operation` of the previous `Scan`, i.e.

```
QUERY
    FOR t0 IN A0
        ...
            FOR tk IN Ak
```

At this point, we can construct a mapping between tuple elements and variables in the rule (2.2).

TABLE 2.2: Tuple Element $\leftrightarrow$ Variable Mapping

x_1	...	x_m
tj.h	...	ti.k

If $\bar{\Phi}$ is nonempty, then create a `Filter` as the `Operation` of the last instantiated element, i.e. assuming that $\bar{A}$ and $\bar{\Phi}$ are nonempty, the `Query` will take the shape

```
QUERY
    FOR t0 IN A0
        ...
            FOR tk IN Ak
                IF <Condition>
```

If $\bar{\Phi} = \{\Phi_0, \ldots, \Phi_l\}$ contains more than one element, instantiate $l-1$ `Conjunctions`. Then, for each negated atom in $\bar{\Phi}$, instantiate the `Negation` of an `ExistenceCheck`, using the variable to tuple element mapping as needed. Then, for each constraint in $\bar{\Phi}$, instantiate a `Constraint`. All of these elements should become direct children of a `Conjunction`. If $\bar{\Phi}$ only contains one element, then the child of the `Filter` will either be a `Negation` of an `ExistenceCheck` or a `Constraint`. As a first example, let $\bar{\Phi} = \{\neg A(x), x < 5\}$. The resulting `Condition` of the `Filter` will take the shape as in (2.6).

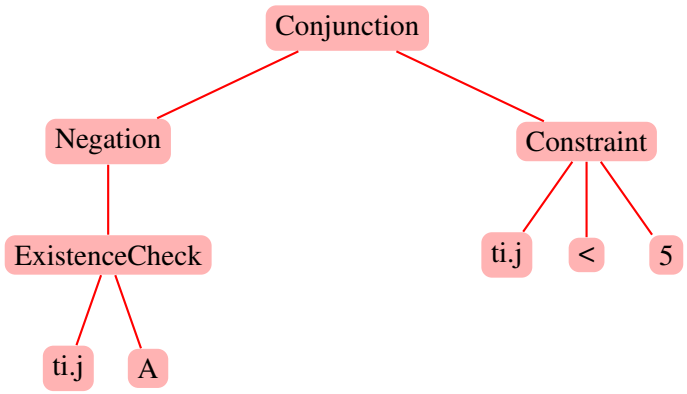


FIGURE 2.6: Example RAM Condition

Here we assume that x maps to the tuple element `ti.j`. Now, create a `Project` as the `Operation` of the RAM element that is still missing a child. This will either be a `Filter`, a `Scan`, or a `Query` in the case of a fact. The `Project` statement takes both a relation and a tuple as arguments, so we take the head of the rule `A` and the tuple `(e1, ..., eh)` corresponding to the terms that occur in the head. These terms may involve variables, in which case, we refer to the variable to tuple element mapping in order to mirror these terms in RAM. Now, assuming that both $\bar{A}$ and $\bar{\Phi}$ are nonempty, the completed `Query` will take the appearance

FIGURE 2.7: Non-Recursive RAM Query Syntax

```
QUERY
    FOR t0 IN A0
        ...
            FOR tk IN Ak
                IF <Condition>
                    PROJECT (e1, ..., eh) INTO A
```

Assuming that we have already decided a stratification of the program in which this rule R occurs, and that stratum is S_i, then all rules where the relation A is the head will be transformed into `Queries`, collected into a `ListStatement` in the case where the number of queries is more than one, and this `ListStatement` will be the `Operation` of `SUBROUTINE stratum_i`. This can be visualised in the following way.

```
SUBROUTINE stratum_i
QUERY
```

```
    FOR t0 IN A0
        ...
            FOR tk IN Ak
                IF <RAM Condition>
                    PROJECT (e1, ..., eh) INTO A
QUERY
    ...
...
END SUBROUTINE
```

We now present an example non-recursive datalog program P along with its RAM translation P_R.

$$A(0).$$

$$B(1).$$

$$A(x) \leftarrow B(x).$$

Then a stratification of P is $\{B\}, \{A\}$ and so the translation of P into RAM is

```
PROGRAM
    DECLARATION
        A(x:number)
        B(x:number)
    END DECLARATION
    SUBROUTINE stratum_0
        QUERY
            PROJECT (1) INTO B
    END SUBROUTINE
    SUBROUTINE stratum_1
        QUERY
            PROJECT (0) INTO A
```

```
        QUERY
            FOR t0 IN B
                PROJECT (t0.0) INTO A
    END SUBROUTINE
    BEGIN MAIN
        CALL stratum_0
        CALL stratum_1
    END MAIN
END PROGRAM
```

Now that we have found a general strategy to translate any non-recursive datalog rule, we now focus on devising a strategy to translate any recursive datalog rule.

2.4.3 Recursive AST to RAM Translation

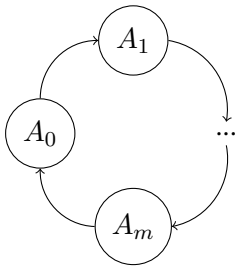
A datalog rule R of P is recursive whenever the head atom of R appears in a cycle of the relation dependency graph (2.3.2) of P. In order to translate a recursive rule, we follow a procedure that is equivalent to semi-naive evaluation. As a refresher, we briefly explain the process of naive and semi-naive evaluation with an example. Let there be a rule R for B as in the following program.

$$B(0).$$

$$B(n+1) \leftarrow B(n), n < 10.$$

The relation dependency graph for this program has a self-loop on the relation B. Then we must run the RAM Query equivalent to the rule body $B(n), n < 10$ until no new facts can be derived. This is coined *naive* evaluation. However, it is obvious that once we use a set of facts from B in order to derive a new member of B, we cannot use these again to find a new member. Therefore, it is only worth looking at members of B that have not yet been analysed. If we place ourselves into the context of a loop whose child is a `Scan` over B, we must only inspect members of B that have been derived on the previous round, and we name this δB. Then we insert these tuples into a set of tuples B' that will join B once the `Scan` for this iteration of the while loop has terminated. We can terminate this while loop when no new tuples have been derived, or equivalently, $B' = \emptyset$. This is coined *semi-naive* evaluation.

We now detail the process of semi-naive evaluation for an arbitrary recursive datalog query as it is translated into RAM. Let there be mutually recursive relations $A_0, \ldots, A_m$ as in the following depiction.



(1) Insert all current contents of $A_0, \ldots, A_m$ into `delta_A0, ..., delta_Am`, i.e.

```
FOR t0 in Ai
    PROJECT (t0.0, ..., t0.n) INTO delta_Ai
```

(2) Find the non-recursive translation for each recursive rule in P of which any of $A_0, \ldots, A_m$ is the head. Replace the occurrences of any of `A0, ..., Am` with `delta_A0, ..., delta_Am`, and project into `new_A0, ..., new_Am` rather than `A0, ..., Am`. Group these `Queries` together as a `ListStatement` assuming that there is more than one, and make this the child of a `Loop`.

(3) Exit if `new_A0, ..., new_Am` are all empty, i.e.

```
EXIT ((new_A0 = ∅) AND ... AND (new_Am = ∅))
```

(4) At the end of one iteration of the while loop, project the contents of `new_A0, ..., new_Am` into `A0, ..., Am`, i.e.

```
FOR t in new_Am
    PROJECT (t.k, ..., t.j) INTO Am
```

(5) Swap the contents of `new_Ai` and `delta_Ai` for each `Ai`, i.e. `SWAP (delta_Ai, new_Ai)`.

(6) Clear the contents of `new_Ai` for each `Ai`, i.e. `CLEAR A_i`.

We now give an example of a recursive rule that is translated using this framework. Take the program P to be defined by

EXAMPLE 2.4.1 (Transitive Closure Program).

$$Edge(0,1).\ Edge(0,2).\ Edge(2,3).$$

$$Path(x,y) \leftarrow Edge(x,y).$$

$$Path(x,z) \leftarrow Path(x,y), Edge(y,z).$$

All non-recursive rules for $Path$ must precede all recursive rules for $Path$. We have the non-recursive query corresponding to the rule $Path(x,y) \leftarrow Edge(x,y)$.

```
FOR t0 IN Edge
    PROJECT (t0.0, t0.1) INTO Path
```

Then for the recursive rule $Path(x,z) \leftarrow Path(x,y), Edge(y,z)$, we have the following RAM translation.

```
QUERY
    FOR t0 in Edge
        PROJECT (t0.0, t0.1) INTO Path
QUERY
    FOR t0 in Path
        PROJECT (t0.0, t0.1) INTO delta_Path
LOOP
    QUERY
        FOR t0 in delta_Path
            FOR t1 in Edge
                PROJECT (t0.0, t1.1) INTO new_Path
EXIT (new_Path = ∅)
QUERY
    FOR t0 in new_Path
        PROJECT (t0.0, t0.1) INTO Path
SWAP (new_Path, delta_Path)
```

```
CLEAR new_Path
END LOOP
```

We can see now how both recursive and non-recursive rules are translated into RAM Queries. These RAM `Queries` will find their way into `SUBROUTINE stratum_i` given that their head relation is on stratum i, and it is the case that all non-recursive queries will precede recursive queries. Now we have a complete picture of how a given datalog program P is translated into its corresponding RAM representation P_R. In the following section, we will extend the syntax and semantics of datalog to handle aggregation over an arbitrary conjunction of literals.

2.5 Chapter Summary

We have now given motivation for datalog as well as its formal syntax and semantics as it is realised in the Soufflé engine. This chapter has given the necessary background in order to understand the new developments for aggregate functionality in the Soufflé engine.

CHAPTER 3

Aggregates in Datalog

Now that we have defined a clear syntax and semantics of the Soufflé engine as well as given motivation via its usage in real world applications, we now consider the functionality of aggregates in the engine and how this can be improved upon. Aggregates like $sum, count, min, mean$, and max are typically used for summarising large amounts of data into a single numerical result. This is useful for report writing. A use case of aggregates in the traditional relational model is an SQL query where the user wishes to find the maximum grade in a subject (3.1). An equivalent query can be encoded in the form of a *rule* in a Soufflé program as in (3.2). In the SQL case, the aggregate expression yielding the numerical result is `MAX(grade)`, and in the Soufflé case, the aggregate expression is $max\ g : \{\ Students(g, name, "Maths")\ \}$. Here, we select tuples of the $Students$ relation where g is maximised. Imagining that the $Students$ relation has the entries in (3.3), the evaluation machinery should iterate through all tuples in (3.3) and find that 74 is the highest score of any student in $Maths$. Then the variable $grade$ should be assigned the value 74. This means that the relation $HighestMathsGrade$ will appear as in (3.4).

Now that an intuition about aggregation has been established, we will now give a formal syntax and semantics to Soufflé aggregates.

```
SELECT MAX(grade) FROM Students
WHERE subject = "Maths";
```

FIGURE 3.1: Regular SQL Query invoking the Aggregate Construct

$$HighestMathsGrade(grade) \leftarrow grade = max\ g : \{\ Students(g, name, "Maths")\ \}.$$

FIGURE 3.2: Soufflé Basic Aggregate Example

$Students$	$Grade$	$Name$	$Subject$
	74	$John$	$Maths$
	80	$Mary$	$Science$
	65	$Matthew$	$Maths$

FIGURE 3.3: Tabular Representation of the $Students$ Relation

$HighestMathsGrade$	$Grade$
	74

FIGURE 3.4: Tabular Representation of the $HighestMathsGrade$ Relation

3.1 Syntax

Aggregates are an extension of the definition of a term (2.2.1) in Soufflé and they are written

$$f\ z : \{\ L_1, \ldots, L_n\ \}$$

where f is the aggregate function. This is either min, max, sum, or $mean$. z is the *target variable* of the aggregate, and $\{\ L_1, \ldots, L_n\ \}$ is the aggregate *body*. The symbols $L_1, \ldots, L_n$ are literals as in (2.2.3). As an example, consider the aggregate term $sum\ cp : \{\ Takes(student, subject, cp)\ \}$. Here, cp is the target variable, $\{\ Takes(student, subject, cp)\ \}$ is the aggregate body, and $Takes(student, subject, cp)$ is the only literal in the body.

The output of every aggregate function, except degenerate aggregates, which will be discussed soon, is some value in $\mathbb{R}$, where $\mathbb{R}$ is the set of real numbers representable in 64 bit IEEE 754. Take for another example the aggregate α

$$sum\ z : \{\ A(z, w), z = 5\ \}$$

The target variable of α is z and the body of α is $\{\ A(z, w), z = 5\ \}$. We can also have an aggregate of the appearance $count : \{\ L_1, \ldots, L_n\ \}$ but this can be reduced to $sum\ z : \{\ L_1, \ldots, L_n, z = 1\ \}$ This transformation permits the use of a universal syntax for an aggregate. We can also have a compound term e acting as the *target expression* of an aggregate, for example, $f\ e : \{\ L_1, \ldots, L_n\ \}$ but again, an aggregate in this form can be reduced to $f\ z : \{\ L_1, \ldots, L_n, z = e\ \}$ where z does not occur elsewhere in the aggregate body. An example of this transformation in action is taking the aggregate $min\ (2 * x) : \{\ B(w, x)\ \}$ to $min\ z : \{\ B(w, x), z = 2 * x\ \}$. Therefore, all aggregates can be written with a single target variable z and as a safety condition, this target variable must occur groundable in at

$$creditPointTotal(student, creditPoints) \leftarrow \\ Takes(student, s, c), \\ creditPoints = sum\ cp : \{\ Takes(student, subject, cp)\ \}.$$

FIGURE 3.5: Aggregates as Terms

least one of the literals $L_1, \ldots, L_n$. We say that the aggregate body has its own *scope* and that any other part of the rule body besides other adjacent aggregate bodies constitutes the *outer scope*. For example, take the following rule.

$$A(x, y) \leftarrow B(x), y = sum\ z : \{\ C(z, x)\ \}.$$

The variable z is *local* to the scope of the aggregate $sum\ z : \{\ C(z, x)\ \}$ and the variable x originates from the *outer scope* of this rule. Let us take the rule (3.5) and consider the set of variables occurring in the rule for $creditPointTotal$. The variable $student$ and $creditPoints$ occur in the outer scope, but the variables $subject$ and cp only occur in the aggregate body, and therefore constitute local variables.

The target variable of an aggregate is always *local* to the scope of the aggregate, and any occurrence of the same variable in the outer scope will be shadowed by the target variable. All variables occurring exclusively within the aggregate body, not considering the variables occurring in inner aggregate bodies, are *local* to the scope of the aggregate whereas variables that also occur groundable in the outer scope are *injected*. These variables have the same behaviour as grouping variables. Now we will give some examples of rules with aggregates and the classification of each variable occurring within the given rule. Beginning with a simple example, take R_1 to be defined by

$$A(x, y) \leftarrow B(x), y = sum\ z : \{\ C(z, x, u)\ \}.$$

The variable x occurs groundable in the outer scope (via the atom $B(x)$), and the variable z is the target variable of the aggregate, making it local to the aggregate body. Finally, the variable u occurs exclusively in the aggregate body, and so it is local. Take another rule R_2 to be defined by

$$A(x, y, z) \leftarrow D(z), B(x), y = sum\ z : \{\ C(x, z), z < 10\ \}.$$

The target variable z is local to the aggregate $sum\ z : \{\ C(x, z), z < 10\ \}$, and the variable z occurs in the outer scope, so the occurrence of z in the outer scope is *shadowed* by the occurrence of z as the

target variable. This means that the z occurring in the atom $C(x, z)$ is *bound* to the target variable. The variable x occurs groundable in the outer scope in the atom $B(x)$, and it also occurs in the aggregate body atom $C(x, z)$, and so it acts as an injected variable in this scope.

3.2 Local and Injected Variables

We have described some notion already of *local* and *injected* variables. Intuitively, a local variable is a variable that is only available in the scope of the aggregate. Take for example the following rule that computes the lowest score for each player of some sport.

$$LowestScore(player, score) \leftarrow Player(player), score = min\ z : \{\ Score(player, z)\ \}.$$

The variable z is local to the aggregate because different assignments of z will be cycled through in order to deduce the minimum z. The concept of the injected variable captures the group-by semantics of regular aggregation. That is, the aggregate here should be computed for each player, rather than over all players. It is often useful to gather statistics for each grouping of objects rather than over all objects. In RAM, the previous rule is translated to

```
FOR t0 in Player
    t1.0=min t1.1 ∈ Score WHERE t1.0 = t0.0
        PROJECT (t0.0, t1.0) INTO LowestScore
```

Here, `t0.0` represents the injected variable $player$. In this way, the notion of *group-by* is given for free by the outer for loop.

Now that an understanding of variable classifications has been established, we now outline the restrictions placed on the literals $L_1, \ldots, L_n$. No literal L_i can be an atom that is on the same stratum as the head of the rule in which this aggregate term occurs. This is why we say that this version of datalog only supports *stratified aggregation*. We will give an argument for why this is the case after giving the full semantics of aggregates in datalog. The second restriction is that all local variables that occur in the body must also appear groundable in the body. Also, for convenience, we define an *aggregate literal* (3.6) as an aggregate that appears as an argument in an equality constraint where the other argument is some variable x.

$$x = f\ z : \{\ L_1, \ldots, L_n\ \}$$

FIGURE 3.6: Aggregate Literal Form

We can translate any aggregate term into this form using the following procedure that is a close adaption of (2.4.1).

PROCEDURE 3.2.1 (Aggregate Literal Transformation). Let R be some datalog rule, and let e_α be an aggregate term that appears in a literal of R that is not an aggregate literal.

(1) Replace every occurrence of e_α with a *new* variable z.
(2) Extend the body of R with the aggregate literal $z = e_\alpha$.

One example of where this transformation must be invoked is where an aggregate term appears in a constraint that is not an equality constraint, i.e. $x < 5 + f\ z : \{\ L_1, \ldots, L_n\ \}$. This literal will be converted into two literals $x < 5 + y, y = f\ z : \{\ L_1, \ldots, L_n\ \}$. This transformation will also be applied where an aggregate term appears as an argument to an atom, i.e. $A(f\ z : \{\ L_1, \ldots, L_n\ \})$ will become $A(y), y = f\ z : \{\ L_1, \ldots, L_n\ \}$. From this point onwards, we assume that all aggregates occur in aggregate literal form.

3.3 Semantics

The semantics of the aggregate agrees with intuition. In the relational model, an aggregate simply applies the operator f to a fixed column of each row of a table, outputting a single numeric result per grouping. For example, take the following relation $R[X_1 : \mathbb{R}, X_2 : \mathbb{R}]$ shown below. We can express the sum of X_2 values as $sum(R[X_2])$, where we are invoking the relation restriction operator defined in (2.2.2). The input to the aggregate function is then a multiset of tuples, and the output is some value in $\mathbb{R}$. The output of $sum(R[X_2])$ is then 4.

EXAMPLE 3.3.1 (Aggregating over a Relation).

R	X_1	X_2
	1	2
	3	2

We can relate this relational interpretation of an aggregate function to support datalog's aggregate semantics. For this, we assume that all injected variables have been given a fixed grounding by the outer scope by the time we begin to evaluate the aggregate function. For all variables occurring in the aggregate body, besides those occurring in nested aggregate bodies, these will have a set of valid grounding configurations, where each configuration can be represented as a tuple. Let the variables $z_1, \ldots, z_n$ occur in the body of an aggregate $f\ z_1 : \{\ L_1, \ldots, L_m\ \}$, excluding nested aggregate bodies. Referring to our set of currently known facts I, given that we find atoms in I that give a grounding to the variables $z_1, \ldots, z_n$, and all of these substitutions make the resulting conjunction of literals $L_1, \ldots, L_m$ resolve to $true$, we include this configuration in the following table $R[Z_1, \ldots, Z_n]$ as shown below.

R	Z_1	$\ldots$	Z_n
	$k_{1,1}$	$\ldots$	$k_{1,n}$
	$k_{p,1}$	$\ldots$	$k_{p,n}$

FIGURE 3.7: Finding Groundings for an Aggregate's Variables

Then, since z_1 is the target variable of the aggregate and the corresponding attribute of R for this is Z_1, the output of aggregate $f\ z_1 : \{\ L_1, \ldots, L_m\ \}$ will be $f(R[Z_1])$. If however the aggregate body contains injected variables $z_2, \ldots, z_{k-1}$, we will instead calculate $f(R[Z_1 | Z_2 = c_1, \ldots, Z_k = c_{k-1}])$ for each combination of valid groundings $z_2 := c_1, \ldots, z_k := c_{k-1}$.

We now give an example of a program with an aggregate (3.8) along with the corresponding table R.

$$A(0).$$
$$A(n+1) \leftarrow A(n), n < 10.$$
$$B(s) \leftarrow s = sum\ z : \{\ A(z), z < 5\ \}.$$

FIGURE 3.8: Triangular Sum Program

Then R is

R	Z
	0
	1
	2
	3
	4

FIGURE 3.9: Groundings for Counting Sum Program

and $sum(R[Z])$ is clearly $\sum_{i=1}^{i=4} i = \frac{(4)(5)}{2} = 10$.

We must quickly also define the semantics of *degenerate* aggregates, that is, aggregates whose bodies have length zero, i.e. $f\ z : \{\ \}$. If $f \in \{\ min, max, mean\ \}$, then the aggregate is unsatisfiable, and so the rule evaluation must fail. If f is sum, then $sum\ z : \{\ \} = 0$. This also covers the case of $count$.

There is one large restriction on the datalog aggregates discussed in this paper however, and this is that all aggregates must be stratified. An aggregate is stratified if all atoms appearing in its body belong to a lower stratum than the head of the rule in which the aggregate appears. Formally, let R be a rule with aggregate literal α, where the body of α contains the atoms $\bar{A}$ and non-atoms $\bar{J}$.

$$A \leftarrow \bar{L}, x = f\ z : \{\ \bar{A}, \bar{J}\ \}.$$

R is stratified with respect to aggregation if all atoms in $\bar{A}$ belong to a lower stratum than A. We explain why this restriction has been created using an example. Let there be a rule R defined by

$$B(x) \leftarrow x = sum\ z : \{\ A(z)\ \}.$$

If A is on a lower stratum than B, then this implies that all members of A have already been derived by the time we begin to evaluate R. This is referred to as A having been *saturated*. If A is on the same stratum as B, then there is no guarantee that there will not be additional members of A that are inserted into I in the future. This means that the sum we will have calculated at this stage is only partial. It is therefore seen as a sensible restriction to only allow aggregates to be calculated over relations that are completely saturated. There are implementations of datalog like DeALS (Zaniolo et al., 2016) that support unstratified aggregation, but this is only in cases where the value being aggregated over is monotonically increasing or decreasing. For example, take this datalog program P (Ganguly et al., 1995) that calculates the shortest paths between all nodes in a graph.

EXAMPLE 3.3.2 (Ganguly et al.'s Shortest Path Program).

$$path(x, y, cost) \leftarrow arc(x, y, cost). \tag{3.1}$$

$$path(x, y, cost) \leftarrow shortestpath(x, z, c_1),\ arc(z, y, c_2),\ cost = c_1 + c_2. \tag{3.2}$$

$$shortestpath(x, y, cost) \leftarrow path(x, y, cost), cost = min\ c : \{\ path(x, y, c)\ \} \tag{3.3}$$

This is not stratified with respect to aggregation because the aggregate in (3.3) contains the atom $path$, so $path$ should be in a lower stratum than $shortestpath$, but (3.2) forces $shortestpath$ to be in a stratum lower than or equal to $path$. Despite this, the intended meaning of P is clear. Using $shortestpath$ instead of $path$ in (3.2) makes the shortest path computation mimic Dijkstra's in that longer subpaths are discarded since they will not be used in the $shortestpath$ relation. Non-monotonic aggregates have only appeared where their use optimises the performance of the input program. It may be an area of future work to hide this from the programmer's perspective and have the compiler infer that such an optimisation can be made since datalog programs are intended to be declarative.

We now give the semantics of aggregates by incorporating aggregate literals into the immediate consequence operator T_P as it is described in (3.3). Let R be a rule where the variables $x_1, \ldots, x_m$ occur in the outermost scope, and let us fix an aggregate literal α. Then, let us already be at a stage where the injected variables $\bar{x} \subseteq \{\ x_1, \ldots, x_m\ \}$ occurring in α have received groundings $x_1 := k_1, \ldots$ from the atoms of R as in (3.10).

x_1	$\ldots$	x_m	z_1	$\ldots$	z_p
k_1	$\ldots$	k_m	k_{m+1}	$\ldots$	k_{m+p}

FIGURE 3.10: Groundings for Injected and Local Variables

Our rule R takes the shape

$$\forall x_1, \ldots, x_m (A \leftarrow L_1, \ldots, L_n, \alpha(\bar{x}))$$

where $\alpha(\bar{x})$ denotes the occurrence of the injected variables $\bar{x}$ in α. Now, in the case that α contains nested aggregate literals, we can represent α and its nested aggregate literals as a tree structure (3.11).

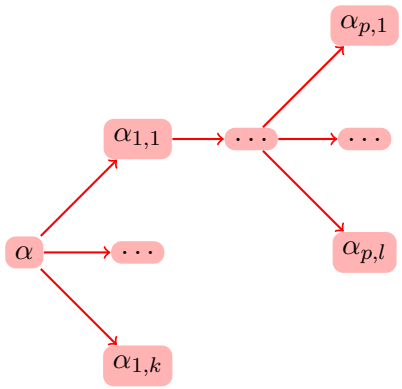


FIGURE 3.11: Nested Aggregation Dependency Graph

It is clear that we will need to evaluate the leaves of (3.11) before evaluating the inner nodes, since the child aggregates may give groundings to variables that are necessary in order to evaluate some constraint. Due to this, we say that an aggregate is always a *groundable* term except for the degenerate min and max case. We can also have dependencies between aggregates on the same level. Assuming that no aggregate is degenerate, in which case the rule will automatically fail, the aggregate literals must be arrangeable as a directed acyclic graph such that we can compute sibling aggregate literals in a sensible order. As an example, take the rule R to be defined by

$$A(x, y) \leftarrow x = sum\ z : \{A(z)\}, y = sum\ w : \{A(w), w < x\}.$$

Then clearly $x = sum\ z : \{A(z)\}$ must be computed before $y = sum\ w : \{A(w), w < z\}$ since x must have a grounding at the time of evaluating the second aggregate. If sibling aggregate literals are not arrangeable in a directed acyclic graph, then it is impossible to assign groundings in any order, and so we must disallow this.

It is possible however that there are injected variables that originate from lower level aggregates. These must receive a grounding before we can evaluate the leaves of this tree. For all atoms at any level of nesting, fix groundings for the variables occurring as arguments in these atoms, and cycle through all possible groundings as in (3.3). If at any point we deduce that some literal in any aggregate body is false, then we will cycle to the next valid set of groundings.

So far, we have given a semantics to nested aggregation through the fixpoint operator, but we have not seen how this can be dealt with by the relational algebra machine program produced by the input program. The general syntax of an aggregate in the relational algebra machine language is as follows

```
ti.0=f ti.j ∈ A WHERE <Condition>
```

This syntax only allows for an aggregate to operate over a single relation. Therefore there must be work done either at the datalog syntax level or at the RAM level in order to compute aggregates over an arbitrary conjunction of literals $L_1, \ldots, L_n$.

3.4 Converting Arbitrary Aggregates to Single-atom Aggregates

Previous to the work of this thesis, the Soufflé engine could handle aggregates like $min\ z : \{\ Score(player, z)\ \}$ and $mean\ mark : \{\ TestScore(student, mark)\ \}$. These aggregates compute statistics over a single relation $Score$ and $TestScore$. Computing statistics over more than one relation has not before been possible, and there are many useful queries belonging to this class. For example, a user may want to calculate the average test score of all students who take both Maths and Physics. This query could be made using the aggregate expression

$$mean\ score : \{\ Takes(student, "Maths"), Takes(student, "Physics"), Score(student, score)\ \}$$

and we now explore ways in which this can be achieved.

Historically, aggregation has only taken place over a single relation A and zero or more constraints or negated atoms $\bar{\Phi}$. This is why RAM can already handle aggregates of the following form:

$$x = f\ z : \{\ A, \bar{\Phi}\ \}$$

FIGURE 3.12: Single-atom Aggregate Literal Form

We refer to aggregates that can be written in this way as *single-atom* aggregates. These can be directly translated into a RAM `Aggregate` statement as follows

```
ti.0=f ti.j FOR ALL ti ∈ A WHERE <Condition>
```

FIGURE 3.13: RAM Aggregate with Conditions Syntax

Here, x maps to `ti.0`, A maps to `A`, and $\bar{\Phi}$ maps to `<Condition>`. $\bar{\Phi}$ can be translated into a RAM `Condition` in the same manner as described in (2.4.2). If $|\bar{\Phi}| = 0$, we can omit the `Condition` and simply write

```
ti.0=f ti.j FOR ALL ti ∈ A
```

FIGURE 3.14: RAM Aggregate Syntax

It is now clear that all single-atom aggregates can be easily handled at the RAM level, but any aggregate containing more than one atom cannot.

One method to compute aggregates with more than one atom is to abandon the RAM `Aggregate` syntax and instead translate the body of the aggregate in a similar fashion to how the body of a non-recursive rule is translated. This is possible because we work under the assumption that our program is stratified with respect to aggregation. We will now show how an arbitrary rule R can be translated using what is termed the *cactus method*. Let R be defined by

$$A \leftarrow \bar{L}, x = f\ z : \{\ \bar{J}\ \}.$$

Then, assuming that both $\bar{L}$ and $\bar{J}$ contain at least one atom and one negated atom or constraint only for ease of visualisation, the RAM translation corresponding to R is

```
QUERY
    ti.j=<initial>
    FOR t0 IN A0
        ...
            IF <Condition1>
                FOR tk in B0
                    ...
                        IF <Condition2>
```

```
ti.j=f(ti.j, tk.l)
```

assuming that $A_0 \in \bar{L}$ and $B_0 \in \bar{J}$, and `f` is the accumulator function associated with the aggregate function f. For example, the accumulator function for sum is $+$. If R were to contain more than one aggregate literal, then these could be placed adjacently under `<Condition1>`. For example,

```
FOR tk in B0
    ...
        IF <Condition2>
            ti.j=f(ti.j, tk.l)
FOR th in C0
    ...
        IF <Condition3>
            tf.p=f(tf.p, tu.v)
```

Nested aggregates could also be dealt with in a straightforward manner by simply deepening the loop nest even further. This is why this approach is called the cactus method. There can be many adjacent branches of execution, and certain branches can spawn other branches. The downside to this approach is that it does not lend itself well to parallelisation. We can spawn threads to work on divisions of the outermost loop, but this means that there will be multiple points of synchronisation, for example, when the aggregate result is referenced. To overcome the overhead of synchronisation, we prefer to refactor deeply nested queries into several shallowly nested queries. One approach that will achieve this is to push all nested aggregates and atoms into a new synthesised relation, preserving injection relationships. Then it will be possible to leave the RAM language untouched and implement aggregation over an arbitrary conjunction of literals through syntactic transformations of the input datalog program only. We now present a syntax-level transformation to take an aggregate over a conjunction of arbitrary literals to a single-atom aggregate. Through the course of this transformation, we will discover the restrictions that must be placed on the aggregate as a result of this method.

Let us take an aggregate literal α where α is $x = f\ z : \{\ \bar{J}\ \}$ and α appears in the following rule R.

$$A \leftarrow \bar{L}, x = f\ z : \{\ \bar{J}\ \}$$

We can rewrite the body of α as

$$f\ z : \{\ \bar{A}, \bar{\Phi}, \bar{\alpha}\ \}$$

where $\bar{A}$ is a conjunction of atoms, $\bar{\Phi}$ is a conjunction of negated atoms and constraints excluding aggregate literals, and $\bar{\alpha}$ is a conjunction of nested aggregate literals. Then we should synthesise a new relation B with a sole rule as follows:

$$A \leftarrow L_1, \ldots, L_n, x = f\ z : \{\ B, \bar{\Phi}\ \} \tag{3.4}$$

$$B \leftarrow \bar{A}, \bar{\alpha}. \tag{3.5}$$

The name of the new relation can be any name that does not appear elsewhere in the program. It is possible that $\bar{\alpha}$ contains further nested aggregate literals. In this case, we must repeat this process until all aggregates appear in single-atom aggregate form, or equivalently, until a fixpoint is reached.

This does not specify however which variables occurring in the body should appear in the head of the rule for B and how, if at all, injection relationships can be preserved. To begin, we need the target variable z to appear in the head of B to preserve the groundability of z. Then we must consider the subset of local variables that must appear in the head. We will have a correct result if we include all local variables $\bar{y}$ in the head. In the case of min and max, no local variable except for the target variable needs to appear, but for sum and $count$, the issue is that we must not accidentally eliminate duplicate column elements. The only way to ensure that duplicates are not eliminated is to include all local variables in the head. This is because the aggregate must operate over the multiset of elements in that column, and if we eliminate any column, we will be taking the projection of this relation, thereby eliminating duplicate tuples, as is the semantics of a projection. We can unify our approach then to all aggregates by placing all local variables $\bar{y}$ into the head, and the target variable z is a local variable, so we do not need to include z separately. It is also worth noting that we can safely ignore the local variables of any inner aggregates at this stage. We also must not forget that all injected variables $\bar{x}$ appearing in the aggregate, or any nested aggregate, must also be included in the head. This will preserve the injection relationship. Then, in summary, the revised auxiliary rule that is headed by B is now

FIGURE 3.15: Single-Atom Aggregate Transformation

$$D \leftarrow L_1, \ldots, L_n, x = f\ z : \{\ B(\bar{y}, \bar{x}), \bar{\Phi}\ \} \tag{3.6}$$
$$B(\bar{y}, \bar{x}) \leftarrow \bar{A}, \bar{\alpha}. \tag{3.7}$$

It may not yet be immediately obvious how an injected relationship will survive once the transformed datalog program is translated into RAM. Take for example the following program P with an injection relationship that spans more than one level.

EXAMPLE 3.4.1 (Reaching Injected Variable).

$$B(4).\ A(1).\ D(3).$$
$$C(x, y) \leftarrow D(y), x = sum\ z : \{\ A(z), z < sum\ w : \{\ B(w), w < y\ \}\ \}.$$

The variable y appears groundable in the outermost scope, and it also appears in the aggregate $sum\ w : \{\ B(w), w < y\ \}$. The transformation described in (3.15) dictates that we should transform this program into the following:

$$B(4).\ A(1).\ D(3). \tag{3.8}$$
$$C(x, y) \leftarrow D(y), x = sum\ z : \{\ E(z, y, k), z < k\ \} \tag{3.9}$$
$$E(z, y, k) \leftarrow A(z), k = sum\ w : \{\ B(w), w < y\ \}. \tag{3.10}$$

However, now the variable y in (3.10) is no longer groundable. We might handle this situation by "pulling in" an atom that will make y groundable, i.e. we pull in the atom $D(y)$ into the rule for E,

$$E(z, y, k) \leftarrow A(z), D(y), k = sum\ w : \{\ B(w), w < y\ \}.$$

but this approach is not always safe. We work under the assumption that all atoms occurring in the body of C are of a lower or equal stratum to C. If D is on the same stratum as C, then pulling D into the rule for E will mean that the aggregate $sum\ z : \{\ E(z, y, k), z < k\ \}$ will no longer be stratified. Therefore we disallow reaching injected variables. A variable can be injected as long as the relationship only spans

a single level since the outer scope will give a grounding to an ungroundable occurrence of the injected variable.

3.5 Recursive Parameters

It has been the case so far in the Soufflé engine that recursive parameters for aggregate bodies with more than one atom have not been functional. We now give an example of a program with an aggregate with recursive parameters. This rule counts the number of points that are far left enough, i.e. their y-position is at least y_0 or less.

$$Point(1,1).\ Point(1,2).\ Point(-1,-1).\ Point(-2,1). \tag{3.11}$$

$$NumPoints(n,(y_0+1)) \leftarrow NumPoints(k,y_0), \tag{3.12}$$

$$n = count\ :\{\ Point(x,y), y \leq y_0\ \}. \tag{3.13}$$

Here, the parameter y_0 is recursive because it is made groundable by the relation $NumPoints$. This relation is on the same stratum as the head $NumPoints$ since they are the same relation.

The rule for $NumPoints$ is roughly translated into the corresponding RAM as

```
LOOP
FOR t0 IN delta_NumPoints
    t1.0=count FOR ALL t0 ∈ Point WHERE t0.1 <= t0.1
        PROJECT (t1.0, (t0.1 + 1)) INTO new_NumPoints
EXIT (new_NumPoints = ∅)
...
END LOOP
```

Here, the recursive parameter y_0 maps to `t0.1` and this is sourced from the `delta_NumPoints` relation. Over the span of the entire outer loop, all possible groundings of y_0 will be given, and so recursive parameters are in this sense handled for free by the RAM translation.

$StudentScores$	$Score$	$Name$	$Subject$
	74	$John$	$Maths$
	80	$Mary$	$Science$
	65	$Matthew$	$Maths$

FIGURE 3.16: The $StudentScores$ Relation

$HighestScore$	$Name$	$Score$
	$John$	74

FIGURE 3.17: The $HighestMathsGrade$ Relation

Query	Witness
Find the name of the student with the highest grade.	The student's name
Find the names of the friends of the tallest student.	The friends' names
Find the location of the cafe that is closest to me.	The location of the cafe
Find the teachers of the student(s) with the lowest average grades	The teachers' names

TABLE 3.1: Examples of queries involving the witness construct

3.6 Witnesses

Before defining the witness construct formally, we will now give an informal introduction.

It is sometimes useful to retrieve values that are associated with some min or max criteria. For example, a user may want to retrieve the maximum test score of a class, but it may be even more useful to retrieve the *name* of the student with the maximum test score. Let us imagine that there is a relation $StudentScores[Score, Name, Subject]$ that holds all test scores (3.16).

Then we could express the desire to fetch the $name$ of the student with the highest score in the following way:

$$HighestScore(name, score) \leftarrow score = min\ z : \{\ StudentScores(z, name, s)\ \}.$$

Not only are we projecting the minimum score into the $HighestScore$ relation, but we are also projecting the $name$ of the student with the minimum score into this relation. Then the resulting tabular representation of $HighestScore$ would be as in (3.17).

We give further examples of queries that invoke the witness construct in (3.1).

We now formally extend aggregates to support a *witness* syntax sugar. A *witness* is a variable w that occurs groundable in the immediate scope of a min or max aggregate and ungroundable in the outermost scope of a rule R. That is, w occurs in the body of some aggregate as well as either in the head of R, or in a negated atom or constraint in the body of R. Below is an example of a rule R containing a witness variable w.

$$A(x, w) \leftarrow x = min\ z : \{\ B(z, w)\ \}.$$

The variable w appears groundable in the scope of the aggregate due to the atom $B(z, w)$, and since w occurs in the head of R and the head atom of a rule never gives groundability, w is a witness variable. We now give a formal semantics to this construct. Aggregate functions min and max select a row from the table R in (3.7), i.e. if row l contains the min or max value, we take the tuple $(k_{l,1}, \ldots, k_{l,n})$, and assuming that the first column Z_1 corresponds to the target variable, we return the value $k_{l,1}$. A *witness* is any other element of this tuple, i.e. $k_{l,2}, \ldots, k_{l,n}$. In this way, a witness is some value associated with the min or max value. It is worth making explicit the fact that there may be more than one row with this witness property. This means that we must consider the set of tuples $\{t \in R | t[Z_1] = k_{l,1}\}$, and so there will always be as many witnesses as there are tuples satisfying the min or max property. An example of this the following rule that returns the *name* of the student with the highest grade, as opposed to returning the highest grade, where the latter does not invoke the witness construct.

EXAMPLE 3.6.1 (Witness).

$$TopStudent(class, name, grade) \leftarrow Class(class),$$
$$grade = max\ g : \{\ Grade(class, name, g)\ \}$$

In (3.6.1), we first fix a class with the atom $Class(class)$, then we calculate the maximum value present in the third column of the $Grade$ relation where the first column has value $class$. This is equivalent to calculating the maximum grade grouping by class. We then retrieve the $name$s associated with these maximum values and export them to the head.

The two examples given thus far do not explore the full capability of witness semantics since the ungrounded occurrence of the witness is always in the head atom of the rule. In the following rule, we find the value(s) associated with the minimum over the first column of A, and then check that these values do not appear in B, and then finally that the minimum value does not exceed the witness value.

$$C(y) \leftarrow y = min\ z : \{\ A(z, w)\ \}, \neg B(w), y < w.$$

We can support the semantics of the witness construct with a simple syntactic transformation. Take R to be the following rule where $f \in \{\ min, max\ \}$, $\bar{L}$ and $\bar{J}$ represent a conjunction of literals in the outer and inner scope of the aggregate respectively, and the witness variable w occurs groundable in $\bar{J}$, but ungroundable in A or $\bar{L}$.

$$A \leftarrow \bar{L}, x = f\ z : \{\ \bar{J}\ \}.$$

Then all we must do is copy all literals in $\bar{J}$ to the rule body, replacing the aggregate target variable z with the output of the aggregate function x, and then rename the witness in the aggregate body to a new name w' to avoid w appearing as an injected variable.

$$A \leftarrow \bar{L}, x = f\ z : \{\ \bar{J}\ \}. \tag{3.14}$$
$$A \leftarrow \bar{L}, x = f\ z : \{\ \bar{J}[w := w']\ \}, \bar{J}[z := x]. \tag{3.15}$$

FIGURE 3.18: Witness Variable Transformation

We now apply this transformation to an example for clarity. Let there be a rule defined by

$$MostPopular(student, nFriends) \leftarrow$$
$$nFriends = max\ n : \{\ Student(student), n = count\ : \{Friend(student, other)\}\ \}.$$

Then in order to capture the semantics of the witness variable, we must copy the body literals $Student(student), n = count\ : \{Friend(student, other)\}$ into the outermost rule scope, i.e.

$$MostPopular(student, nFriends) \leftarrow$$
$$nFriends = max\ n : \{\ Student(student), n = count\ : \{Friend(student, other)\}\ \},$$
$$Student(student), nFriends = count\ : \{Friend(student, other)\}.$$

and then rename all occurrences of the witness variable $student$ to $student'$ to avoid the appearance of $student$ being an injected variable.

$$MostPopular(student, nFriends) \leftarrow$$
$$nFriends = max\ n : \{\ Student(student'), n = count\ : \{Friend(student', other)\}\ \},$$
$$Student(student), nFriends = count\ : \{Friend(student, other)\}.$$

We have not yet however considered the appearance of a witness variable w in an aggregate that is nested more than one level deep. We only allow the exporting of witnesses to the outermost scope. This implies that the variable x must occur in the outermost scope because of the renaming $\bar{J}[z := x]$. Then it is necessary that the nested aggregate literal be copied into the outermost scope. This implies that a nested aggregate literal that contains a witness variable can *only* contain injected variables that originate from the outermost scope. We present the witness transformation for an aggregate that is two levels deep but this can be generalised for any depth. Take the rule

$$A \leftarrow \bar{L}, x = f\ z : \{\ \ \bar{J}, x_0 = g\ z_0 : \{\ \bar{K}\ \}\ \ \}.$$

Then we must copy the aggregate literal $x_0 = g\ z_0 : \{\ \bar{K}\ \}\ \ \}$ into the outermost scope, renaming the target variable to the aggregate result x_0, and renaming occurrences of the witness w in $\bar{K}$ to w'.

$$A \leftarrow \bar{L}, x = f\ z : \{\ \ \bar{J}, x_0 = g\ z_0 : \{\ \bar{K}\ \}\ \ \}. \tag{3.16}$$
$$A \leftarrow \bar{L}, x = f\ z : \{\ \ \bar{J}, x_0 = g\ z_0 : \{\ \bar{K}\ \}\ \ \}, x_0 = g\ z_0 : \{\ \bar{K}[w := w']\ \}, \bar{K}[z_0 := x_0]. \tag{3.17}$$

FIGURE 3.19: Nested Witness Variable Transformation

Now that the transformation is clear, we give an example of a witness transformation on the $TopStudent$ rule where each relation is renamed to a single letter for brevity.

$$T(c, n, g) \leftarrow C(c), g = max\ g_0 : \{\ G(c, n, g_0)\ \} \tag{3.18}$$

$$T(c, n, g) \leftarrow C(c), g = max\ g_0 : \{\ G(c, n', g_0)\ \}, G(c, n, g) \tag{3.19}$$

We also show a more complex example where the witness is nested within an outer aggregate body. We first prepare the body of the outer aggregate by transforming the inner aggregate term into an aggregate literal (3.21).

$$A(x, y, z, w) \leftarrow B(x, y), z = sum\ x_0 : \{\ A(x, x_0), x_0 < max\ n : \{\ C(n, w)\ \}\ \} \tag{3.20}$$

$$A(x, y, z, w) \leftarrow B(x, y), z = sum\ x_0 : \{\ A(x, x_0), x_0 < y_0, y_0 = max\ n : \{\ C(n, w)\ \}\ \} \tag{3.21}$$

$$A(x, y, z, w) \leftarrow B(x, y), z = sum\ x_0 : \{\ A(x, x_0), x_0 < y_0, y_0 = max\ n : \{\ C(n, w')\ \}\ \}, C(y_0, w) \tag{3.22}$$

All of these transformations have assumed that there is only a single witness, but this approach can easily be extended to support multiple witnesses. Let the witnesses $w_0, \ldots, w_k$ occur in $\bar{J}$. We must still copy the contents of the aggregate $\bar{J}$, replacing the target variable with the aggregate result x, and then we must rename all witnesses $w_i \mapsto w_i'$.

$$A \leftarrow \bar{L}, x = f\ z : \{\ \ \bar{J}\ \ \}. \tag{3.23}$$

$$A \leftarrow \bar{L}, x = f\ z : \{\ \ \bar{J}[w_0 := w_0', \ldots, w_k := w_k']\ \ \}, \bar{J}[z := x]. \tag{3.24}$$

FIGURE 3.20: Multiple Witness Variables Transformation

If there are witnesses on multiple levels, we must begin at the leaves, as with the evaluation of nested aggregate literals. Once leaf witnesses have been handled, they will be demoted to the status of a local variable, and outer witnesses will have a well-defined semantics.

We now give an argument to support the correctness of the basic witness transformation with one variable inasmuch as it agrees with the witness semantics. The conjunction of literals $\bar{J}$ can be represented by the table of valid groundings of variables occurring in $\bar{J}$ (3.7). Assuming that the target variable z corresponds to Z_1, the minimum or maximum value of this aggregate will be stored in x. Substituting x into $\bar{J}$ with $\bar{J}[z := x]$ leads us to recover all tuples or rows $(k_{l,1}, \ldots, k_{l,n})$ of (3.7) such that $k_{l,1}$ is the max value of $R[Z_1]$. Then the grounding of the witness w will be restricted to one of $(k_{l,2}, \ldots, k_{l,n})$. This proof is directly generalisable to aggregates with more than one witness as well as nested aggregates.

The common relational approach to solving the witness problem is to sort the results by a particular column, and then limit the result table to size one, revealing a single row in which the minimum or maximum element is found. We show the general form of this query given in the most general SQL dialect. R corresponds to the table name from which we wish to export a witness, and $Z1$ corresponds to the attribute on which to take the minimum or maximum.

```
SELECT * FROM R
ORDER BY Z1 [DESC]
LIMIT 1;
```

FIGURE 3.21: Non-Deterministic Relational Witness Retrieval

It is worth noting that this retrieves *all* possible witnesses since it returns the entire tuple $(k_{l,1}, \ldots, k_{l_n})$ via the asterisk $(*)$ operator, but the output of this query is non-deterministic since there is no telling *which* of the candidate tuples will be returned that satisfy the min or max property if there is more than one. Our witness approach is deterministic, and is therefore preferable. It is analogous to the following SQL query which contains an embedded query within it:

```
SELECT *
FROM R
WHERE Z1 = (
    SELECT MIN(Z1) FROM R
)
```

FIGURE 3.22: Deterministic Relational Witness Retrieval

Aggregate	Accumulator
min	$y = min(y, y_i)$
max	$y = max(y, y_i)$
sum	$y \mathrel{+}= y_i$
$count$	$y{+}{+}$
$mean$	use sum and $count$, dividing as a final step

TABLE 3.2: Aggregate Functions and their Accumulators

The computational work of this approach is therefore also higher since we must perform two scans over R instead of sorting R once and then truncating R to a fixed length, but we believe this is a sensible compromise given that the semantics are well-defined.

3.7 Parallel Aggregate Computation

We now discuss the possibility of parallelising the computation of aggregates in the Soufflé engine. The first important observation is that each aggregate function maps to some accumulator function. We can take y as the current value in the iteration, and y_i to be the next value in the input relation.

The next observation is that each aggregate function is *associative*, meaning that we can apply the accumulator function in any order and this will not affect the final result. Instead of calculating the aggregate result in a single-threaded manner then, it is possible to use many threads where each accumulates the result over a particular subset of the columns of the input relation. These threads can then combine their results to form the final result and this is visualised in the case of sum in (3.23).

Internally, this prefix calculation can be achieved with a `#pragma omp for reduction` statement. We now test the input program given in (3.24) and show the performance results as we extend the range N.

We find that for small relations, i.e. where the number of tuples is one million, computing the aggregate over A in parallel is not beneficial to performance in terms of speed. For programs where the size of A is larger, i.e. where A contains one billion tuples, we can enjoy a speedup of $1.1\times$, since the time taken for one thread over a billion tuples is $11s$, and the time taken for more than one thread over a billion tuples is $10s$.

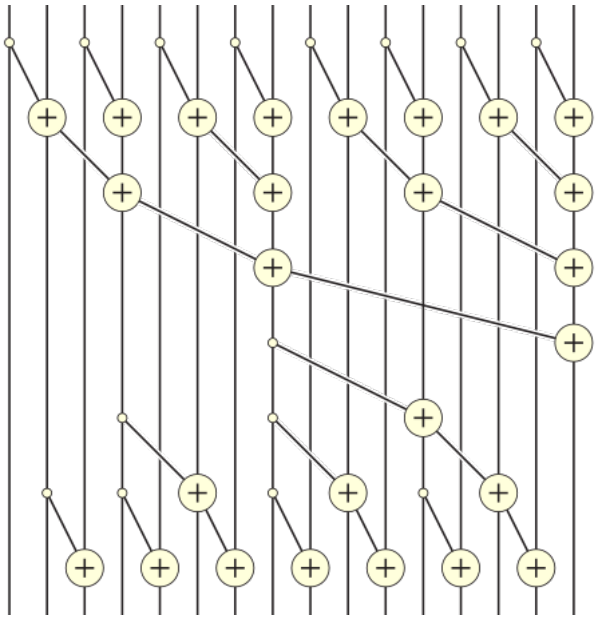


FIGURE 3.23: Prefix Sum Calculation

$$A(x) \leftarrow x = range(0, N).$$
$$B(x) \leftarrow x = min\ y : \{\ A(y)\ \}.$$

FIGURE 3.24: Soufflé Aggregation Program

3.8 Chapter Summary

In this chapter, we have extended the functionality, usability, and performance of aggregates in the Soufflé engine. We have improved the functionality of aggregates by allowing users to aggregate over more than one relation as well as to aggregate over other aggregates since the ability to aggregate over an arbitrary conjunction of literals is now possible with some restrictions. We have improved the usability of aggregates because the Soufflé engine now supports a witness syntax sugar and this is also achieved via program rewriting techniques. We have then as a final step improved the performance of aggregate computations in the Soufflé engine by allowing aggregates to be computed using multiple threads, having the intermediate results combined via the prefix sum method.

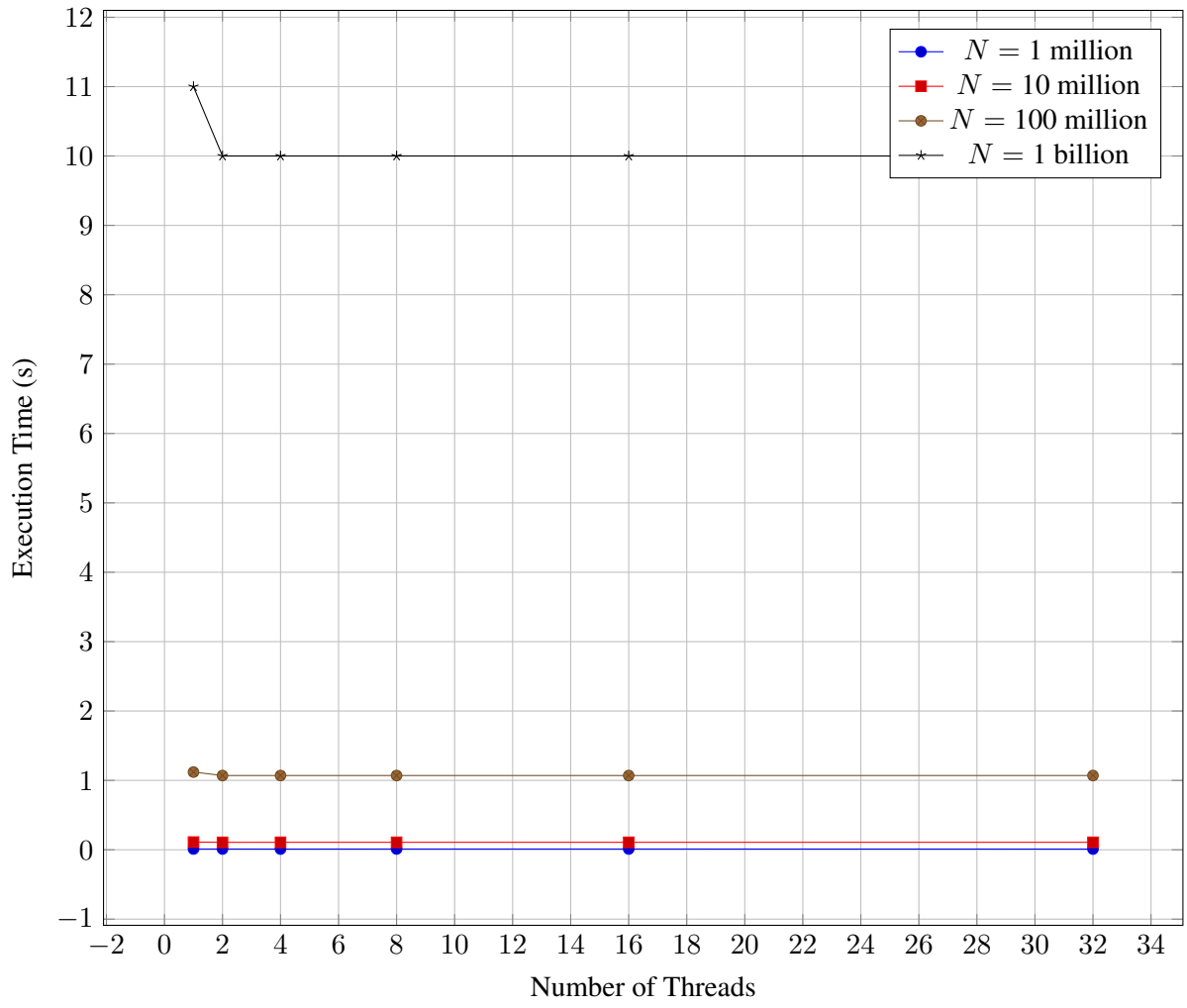


FIGURE 3.25: Execution Time of Program (3.24) as N increases

CHAPTER 4

Conclusion

In this thesis, we have explored the ways in which the usability, functionality, and performance of aggregate functions in Soufflé can be improved. We have achieved an improvement in functionality by describing the program rewriting rules that are necessary in order to implement aggregation over an arbitrary conjunction of literals with the restriction that variables cannot be injected across more than one level unless a grounding is given to the injected variable in the inner scope. We have improved the usability of aggregates by introducing the witness syntax sugar, and we have also improved the performance of aggregates using the prefix sum method such that aggregate computations can now take advantage of multi-threading.

Other than this, we have given a refreshed syntax and semantics to the datalog programming language as it is seen in the Soufflé system, improving upon earlier descriptions of the syntax and semantics of datalog in (Greco and Molinaro, 2015) and (Abiteboul et al., 1995). We have also given a thorough treatment of the backbone of datalog; RAM. This is the imperative translation of an input Soufflé program.

4.1 Future Work

In the process of developing this thesis, we have found that there is one major improvement that could be made in order to widen the notion of groundability such that programs that are computable and do terminate are registered as such, and therefore can be compiled and run. With the current notion of groundability, the following rule is not permissible:

$$A(x, y) \leftarrow B(y), x = 2 * y.$$

The variable y does not occur groundable, since the `ResolveAliases` transformer will convert this rule to

$$A(x, 2 * x) \leftarrow B(2 * x).$$

An argument can be made that this rule can be rewritten to

$$A(\frac{x}{2}, x) \leftarrow B(x)$$

but Soufflé is purported as a declarative language, and as such, the programmer should be concerned with only *what* the program does, and not *how* the compiler or interpreter may react to particular arrangements of inputs that are logically equivalent. This gap can easily be filled by introducing an *assignment operator* into the RAM intermediate representation. An equality constraint can be read as an assignment operator whenever one argument of the equality constraint is a variable and an invocation of `ResolveAliases` would cause a groundability issue. An equality constraint of this form is trivially satisfied because the right hand side of the assignment operator is a function. The previous rule can be translated into RAM as

```
FOR t0 in B
    t1.0=number(2*t0.0)
    PROJECT (t0.0, t1.0) INTO A
```

An appropriate ordering will need to be deduced such that all variables occurring on the opposing side of the assignment operator are already in scope. The result of this is that Soufflé will become more attractive to programmers who may feel confused by errors related to groundability because they were not able to write the program in a permissible form according to the compiler. It is possibly less important that notions of groundability be narrowed and made more accurate as long as programmers feel confident that they understand when they are writing programs that will or will not terminate. Then, proofs for termination can be left as a theoretical exercise only.

Bibliography

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. 2010. Boom analytics: Exploring data-centric, declarative programming for the cloud. In *EuroSys'10 - Proceedings of the EuroSys 2010 Conference*.

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 ACM SIGMOD International Conference on Management of Data*.

M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. 1976. System R: Relational approach to database management. *ACM Transactions on Database Systems (TODS)*.

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*.

Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering*.

Donald D. Chamberlin, Morton M. Astrahan, Kapali P. Eswaran, Patricia Priest Griffiths, Raymond A. Lorie, James W. Mehl, Phyllis Reisner, and Bradford Warren Wade. 1976. SEQUEL 2: A UNIFIED APPROACH TO DATA DEFINITION, MANIPULATION, AND CONTROL. *IBM Journal of Research and Development*.

Donald D. Chamberlin and Raymond F. Boyce. 1974. Sequel: A structured english query language. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*.

Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. 1995. Extrema predicates in deductive databases. *Journal of Computer and System Sciences*.

Sergio Greco and Cristian Molinaro. 2015. *Datalog and Logic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers.

Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. 2013. Bounded programs: A new decidable class of logic programs with function symbols. In *IJCAI International Joint Conference on Artificial Intelligence*.

David B. Kemp and Peter J. Stuckey. 1991. Semantics of logic programs with aggregates.

Anthony Klug. 1982. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *Journal of the ACM (JACM)*.

William R. Marczak, Shan Shan Huang, Martin Bravenboer, Micah Sherr, Boon Thau Loo, and Molham Aref. 2010. SecureBlox: Customizable secure distributed data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. 1990. The magic of duplicates and aggregates. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, page 264–277. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On fast large-scale program analysis in datalog. In *Proceedings of CC 2016: The 25th International Conference on Compiler Construction*.

Yannis Smaragdakis and Martin Bravenboer. 2011. 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)*.

Shalom Tsur and Carlo Zaniolo. 1986. LDL: a logic-based data-language.

Carlo Zaniolo. 1986. PROLOG: A DATABASE QUERY LANGUAGE FOR ALL SEASONS.

Carlo Zaniolo, Mohan Yang, Ariyam Das, and Matteo Interlandi. 2016. The magic of pushing extrema into recursion: Simple and powerful datalog programs. In *CEUR Workshop Proceedings*.

Carlo Zaniolo, Mohan Yang, Ariyam Das, Alexander Shkapsky, Tyson Condie, and Matteo Interlandi. 2017. Fixpoint semantics and optimization of recursive Datalog programs with aggregates. *Theory and Practice of Logic Programming*.