Datalog and Its Modern Extensions: Semantics, Convergence, and Optimizations

Suggested Citation: (2024), "Datalog and Its Modern Extensions: Semantics, Convergence, and Optimizations", : Vol. XX, No. XX, pp 1–251. DOI: XXX.

now
the essence of knowledge
Boston — Delft

Contents

ABSTRACT

Datalog, proposed in the mid 1980s, is a declarative logic programming language that enjoys several elegant properties: it has a simple syntax and declarative semantics; the solution to a Datalog program can be computed by a simple fixpoint iteration; and it admits a couple of powerful optimization techniques, semi-naïve evaluation and magic set transformation.

Pure Datalog is not the answer to modern needs, because it only supports monotone queries over sets. For example, most tasks in data science today require the interleaving of recursion and aggregate computation. Aggregates are not monotone under set inclusion, and therefore they are not supported by the framework of pure datalog.

There have been various extensions to Datalog proposed in the literature over decades to address these shortcomings, including a few very recent ones. In this monograph, we review the semantics, convergence, and optimization techniques of Datalog and its modern extensions. We also review a number of applications of Datalog and its extensions. We conclude with a discussion of open problems.

1

Introduction

What is this book about? Why are we writing it?

- There have been related works and sometimes re-discovery of Datalog or its generalizations from a variety of communities – amazingly they can be captured with the same formalism. Pure Datalog is simple and clean, but applications require lots of features that pure datalog does not support. There are a few references for pure Datalog (Vianu, 2021; Abiteboul *et al.*, 1995), we refer to them for more formal treatment and more historical perspective.
- This monograph quickly reviews classic materials, in order to motivate more modern treatments that address shortcomings of pure datalog, while retaining the simplicity and elegance of pure datalog.
- Explain why we present even classic techniques like magic set optimization and seminaive optimization in some details: we are able to generalize them to more modern settings, and we have new proof techniques that are more general and more elegant.
- List of existing Datalog surveys / books; and how we are different from them

- Motivating the modern datalog angle
- New algorithms, new theory for modern datalog
- A couple of (running?) examples
- An outline of the book

2

Pure Datalog

Need a high-level chapter overview paragraph here.

This chapter introduces the syntax and semantics of pure Datalog. In later chapters, we will extend this core language with advanced constructs such as negation and semiring aggregation.

- Example of grounding of a Datalog program
- Remy keeps Datalog-style consistent and change commas to $\wedge$
- Introducing the dependency graph? (chapter 3)
- Bold-face means tuple of IDBs (Remy)
- Add a more concrete example of data and how it works on actual data
- Semantics paragraphs need some work, to be consistent with the later chapters
- IVM part needs work
- Also a related work section on other optimization techniques
- The further reading section needs to be updated

2.1 Syntax

We first introduce necessary notation used in this book. This notation is overviewed in Figure 2.1.

We use uppercase letters such as X and Y to denote variables and lowercase letters such as x and y to denote constants. Each variable X has a discrete domain $\mathsf{Dom}(X)$ of constants. A *schema* $\mathbf{X}$ is a tuple $(X_1, \ldots, X_n)$ of variables. A tuple of constants $\mathbf{x}$ over schema $\mathcal{X}$ is an element from $\mathsf{Dom}(\mathbf{X}) = \mathsf{Dom}(X_1) \times \cdots \times \mathsf{Dom}(X_n)$. A *relation* R over schema $\mathbf{X}$ is a set of tuples over the same schema. A *database* is a set of relations.

Datalog rules A pure Datalog program is a collection of rules of the following form:

$$A \text{ :- } B_1 \wedge \cdots \wedge B_k \tag{2.1}$$

The above rule has the *head* A and the *body* $B_1 \wedge \cdots \wedge B_k$, where A, $B_1, \ldots, B_k$ are called *atoms*. The notation for an atom is that of a function, where the function name is called the *predicate symbol* and the argument is a tuple of variables and constants. For instance, the head atom A can be $Q(\mathbf{X})$, where Q is the predicate symbol and the argument $\mathbf{X}$ is a tuple of variables. A *ground atom* is an atom whose argument is a tuple of constants. For instance, $R(\mathbf{x})$ is a ground atom expressing that the tuple $\mathbf{x}$ of constants is in the relation corresponding to the predicate symbol R. A *ground instance* of a rule is obtained by grounding its atoms, i.e., by replacing the variables with constants in its atoms.

Every rule in a pure Datalog program must satisfy the *range restriction property*: Each variable in the head atom must also occur in some body atom. The symbol $\wedge$ in the body means conjunction (logical "and"): The body is thus a conjunction of atoms. The symbol :- between the head and the body means logical implication. The rule thus reads as follows: If the conjunction of the atoms in the body holds, then the head atom also holds. If the body is empty, then the head, which is a ground atom, holds unconditionally.

(Union of) Conjunctive Queries Rules of the form (2.1) are commonly used in the logic programming literature. The variables in the body of the rule that are not in the head are implicitly existentially quantified. In particular, if atom B_i is $R_i(\boldsymbol{X}_i)$, for some predicate symbol R_i and tuple of variables $\boldsymbol{X}_i$, then the rule is equivalent to the following conjunctive query:

$$Q(\boldsymbol{Y}) \text{ :- } \exists \boldsymbol{X}\ R_1(\boldsymbol{X}_1) \wedge \cdots \wedge R_k(\boldsymbol{X}_k) \tag{2.2}$$

where $\boldsymbol{X}$ is the tuple of variables that do not occur in the head atom $A = Q(\boldsymbol{Y})$. There may be many rules deriving into the same head atom. Semantically, this becomes a *union of conjunctive queries* (UCQ), as the following example illustrates..

EDBs and IDBs The predicates are either *extensional database (EDB) predicates* or *intensional database (IDB) predicates*. EDB predicates represent relations in the input database. IDB predicates represent relations that are computed by the Datalog program, so they correspond to predicates in the heads of rules.

In some cases (especially in Chapter 3), for technical convenience, we define EDB predicates to *also* be predicates that are computed by the Datalog program, using only rules with empty bodies. In this setting, there is no EDB, only IDBs. In case the body is empty, i.e., it has no atoms, then the head atom cannot have variables due to the range restriction property and is therefore a ground atom. In this case, the rule is called a *fact rule*. For instance, to load the input database with EDB predicate E, we can use the following fact rules:

$$E(1,2) \text{ :- } .$$
$$E(2,3) \text{ :- } .$$
$$E(3,1) \text{ :- } .$$

In the rest of this chapter, however, we separate EDBs and IDBs, operating under the mental model that Datalog programs compute IDBs, while EDBs are part of the input.

Example 2.1.1 (Transitive closure)**.** The classic example of a Datalog program is the "transitive closure" program that, given an input edge

relation E of a directed graph, computes the relation $R(u, v)$ indicating whether u can reach v via a directe path in the graph.

$$\begin{aligned} R(U, V) &\text{ :- } E(U, V) \\ R(U, V) &\text{ :- } R(U, W) \wedge E(W, V) \end{aligned} \tag{2.3}$$

E is an EDB relation, and R is an IDB relation. The rules above can be written equivalently with explicit quantification and a disjunction:

$$R(U, V) \text{ :- } E(U, V) \vee \exists W \ R(U, W) \wedge E(W, V) \tag{2.4}$$

For fixed values u and v in the domains of U and V, $E(u, v)$ is a ground atom.

Example 2.1.2 (Binary Transitive closure)**.** Another way to express transitive closure is its binary form:

$$R(U, V) \text{ :- } E(U, V) \vee \exists W \ R(U, W) \wedge R(W, V) \tag{2.5}$$

Using logic-programming notations, the program is written as:

$$\begin{aligned} R(U, V) &\text{ :- } E(U, V) \\ R(U, V) &\text{ :- } R(U, W) \wedge R(W, V) \end{aligned}$$

2.2 Semantics

There are several ways to give a formal meaning to a Datalog program, i.e. how we can tell if a solution is a valid solution to a Datalog program. A candidate solution to a Datalog program is a set of IDB facts. For example, in the transitive closure program of Example 2.1.1, a candidate solution is a list of facts of the form $R(u, v)$ for some constants $(u, v) \in \mathsf{Dom}(U) \times \mathsf{Dom}(V)$.

In this section, we give such formal meanings by means of two semantics: the *model-theoretic* and the *fixpoint-theoretic* semantics. The former is based on first-order logic and sees the Datalog rules as constraints that hold on the database instance. The latter is operational and gives a simple procedure for the evaluation of a pure Datalog program. These two semantics are equivalent in the sense that they both assign the same meaning to any given Datalog program (Emden and Kowalski, 1976). Further semantics are discussed in Section 3.1.

2.2.1 Model-theoretic Semantics

The model-theoretic semantics follows the natural interpretation of each rule as an implication from its body to its head. For example, the rule $R(U, V)$:- $E(U, V)$ in Example 2.1.1 can be interpreted as "if $E(u, v)$ holds, then $R(u, v)$ holds" (i.e. $R(u, v)$ is a fact); and the rule $R(U, V)$:- $R(U, W) \wedge E(W, V)$ can be interpreted as "if $R(u, w)$ and $E(w, v)$ hold, then $R(u, v)$ holds". EDB facts are facts that were given as part of the input database. Thus, we are only interested in what IDB facts hold.

Definition 2.2.1 (Model)**.** A *model* of a Datalog program P is a collection of facts that satisfy the rules in P.

While some models may be larger than others, it can be shown that there exists a *minimal model* for P under set containment (Ullman, 1989). This minimal model is defined to be the semantic of the Datalog program P, under minimal-model semantics.

While one can in principle compute the minimal model of a Database program using the above definition, with bruteforce search, this is not practical. The fixpoint-theoretic semantic defined in the next section provides a more practical way to compute the minimal model.

2.2.2 Fixpoint-theoretic Semantics

The fixpoint-theoretic semantics is defined via fixponts of a certain map, formalized as follows. Let L denote the set of all possible facts (**aka Herbrand base etc, and introduce the notation**).[1] Then, the input database instance I is a subset of L, consisting of all the IDB facts that are true in the input database. By applying the implication rules defined by P, we obtain a new set of facts denoted by $T_P(I)$, where $T_P : 2^L \to 2^L$. is a mapping from the Boolean algebra lattice 2^L to itself. This mapping is called the *immediate consequence operator* (ICO) corresponding to the program P.

It is straightforward to show that T_P is *monotoe*, i.e. for any $I, J \subseteq L$, if $I \subseteq J$ then $T_P(I) \subseteq T_P(J)$. (See more discussion in the next

[1] For example, in the transitive closure program of Example 2.1.1, L is the set of all facts of the form $R(u, v)$ for $(u, v) \in \mathsf{Dom}(U) \times \mathsf{Dom}(V)$.

section below.) Since L is a complete lattice[2], from Knaster-Tarski theorem (Davey and Priestley, 2002), we know T_P has a *least fixed point*, which is defined to be the so solution of the program P. This is its fixpoint-theoretic semantics.

It is a remarkable property, see (Emden and Kowalski, 1976), that the two semantics coincide:

Theorem 2.2.2. For any pure Datalog program P, the least fixpoint of T_P is the minimal model of P.

2.3 Convergence

The fixpoint-theoretic semantics yields a constructive procedure to evaluate a pure Datalog program P, which also gives an *operational semantic* to P. In order to define this procedure, we fix some new notations. We use $\boldsymbol{R}$ to denote the set of IDBs in the database. Overloading notations, the collection of IDB facts is also denoted by $\boldsymbol{R}$: there is one set of fact for every IDB in $\boldsymbol{R}$. For example, in the transitive closure program of Example 2.1.1, $\boldsymbol{R}$ has only one IDB R, and the set of facts are captured by the tuples in R.

In this setup, T_P takes $\boldsymbol{R}$ as input, and produces a (potentially new) tuple of IDBs $T_P(\boldsymbol{R})$ as output. Note that the EDBs $\boldsymbol{E}$ are technically part of the input to T_P, but we can also view them as part of T_P itself. Thus, we do not need to explicitly write $T_P(\boldsymbol{E}, \boldsymbol{R})$.

A ground atom A is an *immediate consequence* of P and an IDB instasnce $\boldsymbol{R}$, denoted by $A \in T_P(\boldsymbol{R})$, if A is a ground atom in $\boldsymbol{R}$ or there is a ground instance $A \text{ :- } B_1 \wedge \cdots \wedge B_k$ of a rule in P, such that the ground atoms $B_1, \ldots, B_k$ are in $\boldsymbol{R} \cup \boldsymbol{E}$. The operator T_P maps an input IDB instance $\boldsymbol{R}$ to a new instance $T_P(\boldsymbol{R})$ that includes the ground atoms like A that are an immediate consequence of the ground atoms in $\boldsymbol{R}$ and the rules in P.

For a pure Datalog program P, T_P is *monotone*; namely, for two instances $\boldsymbol{R}$ and $\boldsymbol{S}$, if $\boldsymbol{R} \subseteq \boldsymbol{S}$, then $T_P(\boldsymbol{R}) \subseteq T_P(\boldsymbol{S})$. Monotonicity implies inflation: $\boldsymbol{R} \subseteq \boldsymbol{R}'$. Indeed, T_P may derive new ground atoms and may not delete existing atoms in its input database instance.

[2] A lattice is complete if every subset has a supremum and an infimum.

$\boldsymbol{R}_0 \leftarrow \emptyset$;
for $t \leftarrow 1$ **to** ∞ **do**
  $\boldsymbol{R}_t \leftarrow T_P(\boldsymbol{R}_{t-1})$;
  if $\boldsymbol{R}_t = \boldsymbol{R}_{t-1}$ **then**
    return $\boldsymbol{R}_t$; `// Fixpoint solution` $T_P^\omega(\emptyset) = \boldsymbol{R}_t$
  end
end

Algorithm 1: Naïve evaluation of a pure Datalog program P.

The fixpoint-theoretic semantics yields a naïve evaluation for pure Datalog programs, as captured by Algorithm 1. Starting from the empty instance $\boldsymbol{R}_0 = \emptyset$, by repeatedly applying the ICO T_P, we obtain a nested sequence: $\boldsymbol{R}_t \subseteq \boldsymbol{R}_{t+1} = T_P(\boldsymbol{R}_t)$. If we get to a time t for which the sequence converges: $\boldsymbol{R}_{t+1} = T_P(\boldsymbol{R}_t) = \boldsymbol{R}_t$, then $\boldsymbol{R}_t$ is a fixpoint of the program P, denoted by $T_P^\omega(\emptyset)$. We can show that this is the least fixpoint of the operator T_P.

Theorem 2.3.1. $T_P^\omega(\emptyset)$ is the least fixpoint of T_P, for any pure Datalog program P.

Proof. Let $\boldsymbol{R}$ be any fixpoint of T_P, i.e. $\boldsymbol{R} = T_P(\boldsymbol{R})$. We show by induction that $\boldsymbol{R}_t := T_P^t(\emptyset) \subseteq \boldsymbol{R}$. This follows from the fact that $\boldsymbol{R}_0 = \emptyset \subseteq \boldsymbol{R}$ and that $\boldsymbol{R}_{t+1} = T_P(\boldsymbol{R}_t) \subseteq T_P(\boldsymbol{R}) = \boldsymbol{R}$ because T_P is monotone. Thus, at convergence $T_P^\omega(\emptyset)$ is a least fixpoint of T_P. □

The fact that T_P is inflationary implies that, before convergence, every iteration gains at least one new fact. Thus, the number of iterations before convergence is bounded by the maximum number of facts possible, which is a polynomial in the input domain size if the program P is fixed. This is why pure Datalog is in PTIME. However, this rough analysis is unsatisfactory in practice.

Consider, for example, the (linear) transitive closure program of Example 2.1.1. We know that the number of iterations is bounded by the diameter of the input graph, which is linear in the input size. On the other hand, for the binary transitive closure formulation of Example 2.1.2, the number of iterations is bounded by the logarithm

of the input edge relation. In Chapter 5, we will see some examples of analysis techniques based on context-free grammars that can be used to give finer bounds on the number of iterations.

Furthermore, the number of iterations does not tell the whole story. While the binary formulation of transitive closure has exponentially less number of iterations, it tends to perform worse in practice because every iteration does a much greater amount of work.

The above claim is not true about naive evaluation, as demonstrated by the three examples below. However, it is guaranteed under semi-naive evaluation. –Mahmoud.

Example 2.3.2. For example, consider the line graph $1 \to 2 \to \cdots \to N$. The linear formulation in Example 2.1.1 will take $\Theta(N)$ iterations to compute the transitive closure. At the tth iteration, it will derive new facts by joining facts of the form $(u-k, u)$ with $(u, u+1)$ for $u \in [N]$ and $1 \leq k < t$. Thus, the total amount of work is in the order of $\sum_{t=1}^{N} \sum_{u=1}^{N} \min(u, t) = \Theta(N^3)$. With the binary formulation in Example 2.1.2, the number of iterations is $\Theta(\log N)$. At iteration t, it will derive new facts by joining facts of the form $(u-k, u)$ and $(u, u+j)$, for $u \in [N]$ and $1 \leq k, j < 2^t$. Thus, the total amount of work is in the order of $\Theta(N^3)$.

Example 2.3.3. Consider now a grid graph with N vertices (i, j) for $i, j \in [\sqrt{N}]$ and edges $(i, j) \to (i+1, j)$ for $i \in [\sqrt{N}-1], j \in [\sqrt{N}]$ along with edges $(i, j) \to (i, j+1)$ for $i \in [\sqrt{N}], j \in [\sqrt{N}-1]$. The total number of edges is $\Theta(N)$. Because the longest path in the graph has length $2\sqrt{N}$, the linear formulation in Example 2.1.1 takes $\Theta(\sqrt{N})$ iterations. At iteration t, it joins paths $(i-k, j-l) \to (i, j)$ of length less than t (i.e. where $1 \leq k+l < t$) with edges $(i, j) \to (i+1, j)$ and $(i, j) \to (i, j+1)$. The total amount of work is $\sum_{t \in \sqrt{N}} \sum_{i,j \in \sqrt{N}} \sum_{k,l:k+l<t} \Theta(1) = \Theta(N^{2.5})$. The binary formulation takes $\Theta(\log N)$ iterations. At iteration t, it joins paths $(i-k, j-l) \to (i, j)$ with paths $(i, j) \to (i+k', j+l')$ where $1 \leq k+l, k'+l' < 2^t$. The total amount of work is thus $\Theta(N^3)$.

Example 2.3.4. Now consider a graph consisting of $N+1$ disjoint paths: One path of length $N^{1/3}$ along with N paths of length 2 each. The linear formulation takes $\Theta(N^{1/3})$ iterations. Just like in Example 2.3.2, the

total amount of work it spends on the $N^{1/3}$-path during all iterations is $\Theta(N)$. However during each iteration, it will also spend $\Theta(N)$ time rediscovering paths of length 2. Thus, the total amount of work is $\Theta(N^{4/3})$. In contrast, the binary formulation takes $\Theta(\log N)$ iterations. Similar to Example 2.3.2, it also spends a total amount of work of $\Theta(N)$ on the $N^{1/3}$-path across all iterations. However, during each iteration, it will additionally spend $\Theta(N)$ time rediscovering paths of length 2. Thus the total amount of work is $\Theta(N \log N)$.

Mahmoud to fix the example; also move it to seminaive section – Hung.

Question: Under naive evaluation, either one of linear and binary TC could end up doing less amount of work than the other (or they could do the same), as demonstrated by the above three examples. On the other hand, under semi-naive evaluation, linear TC is always guaranteed to do no more mork than binary TC. The question is: Do we want to keep the above three examples about naive evaluation, considering that no one usually evaluates TC using naive? Or should we just stick to comparing their amount of work under semi-naive instead? – Mahmoud.

2.4 Optimizations

2.4.1 Semi-naïve optimization

The naïve evaluation algorithm, as given in Algorithm 1 is really "naïve," as it repeats a lot of work unnecessarily. For example, in the linear transitive closure program, every iteration t rediscovers all pairs (u, v) of vertices reachable within a walk of length $\leq t$. This can be improved by reducing the amount of redundant work between subsequent iterations. This is the basis of *semi-naïve evaluation*, also called the *differential approach*, from the works of (Bancilhon, 1985; Balbin and Ramamohanarao, 1986; Balbin and Ramamohanarao, 1987). The optimization is presenting in Algorithm 2.

The main idea of this optimization is the observation that we can break the main update step $\boldsymbol{R}_t \leftarrow T_P(\boldsymbol{R}_{t-1})$ of Algorithm 1 into two

$\boldsymbol{R}_0 \leftarrow \emptyset$;
$\delta\boldsymbol{R}_0 \leftarrow \emptyset$;
for $t \leftarrow 1$ **to** ∞ **do**
  $\delta\boldsymbol{R}_t \leftarrow T_P(\boldsymbol{R}_{t-1}) \setminus \boldsymbol{R}_{t-1}$; `// differential rule`
  $\boldsymbol{R}_t \leftarrow \boldsymbol{R}_{t-1} \cup \delta\boldsymbol{R}_t$;
  if $\delta\boldsymbol{R}_t = \emptyset$ **then**
    return $\boldsymbol{R}_t$; `// Fixpoint solution` $\boldsymbol{R}^* = \boldsymbol{R}_t$
  end
end

Algorithm 2: Semi-naïve optimization of pure Datalog

steps:

$$\delta\boldsymbol{R}_t = T_P(\boldsymbol{R}_{t-1}) \setminus \boldsymbol{R}_{t-1} \qquad \boldsymbol{R}_t = \boldsymbol{R}_{t-1} \cup \delta\boldsymbol{R}_t$$

This two-step process is correct, thanks to the fact that T_P is inflationary, where $\boldsymbol{R}_{t-1} \subseteq T_P(\boldsymbol{R}_{t-1})$ implies $\boldsymbol{R}_{t-1} \cup (T_P(\boldsymbol{R}_{t-1}) \setminus \boldsymbol{R}_{t-1}) = T_P(\boldsymbol{R}_{t-1})$.

In words, the application of the immediate consequence operator does not remove ground atoms once they are derived. This holds for pure Datalog and, as we will see in later chapters, for monotone and even partially inflationary Datalog programs.

Computing the differential rule The key observation of semi-naïve optimization is that, the two-step computation may lead to *faster* evaluation overall if the differential rule for computing $\delta\boldsymbol{R}_t$ can be evaluated quickly. Let us further investigate this computation step:

$$\begin{aligned}\delta\boldsymbol{R}_t = T_P(\boldsymbol{R}_{t-1}) \setminus \boldsymbol{R}_{t-1} &= T_P(\underbrace{\boldsymbol{R}_{t-2}}_{\boldsymbol{A}} \cup \underbrace{\delta\boldsymbol{R}_{t-1}}_{\boldsymbol{\delta}}) \setminus T_P(\underbrace{\boldsymbol{R}_{t-2}}_{\boldsymbol{A}}) \\ &= T_P(\boldsymbol{A} \cup \boldsymbol{\delta}) \setminus T_P(\boldsymbol{A})\end{aligned}$$

Explaining how the difference $T_P(\boldsymbol{A} \cup \boldsymbol{\delta}) \setminus T_P(\boldsymbol{A})$ can be computed quicly is the main reason why we introduced the notation $\boldsymbol{R}$, a collection of IDBs. In particular, T_P is a map from a tuple of IDBs to a tuple of IDBs. We will treat different copies of the same relation symbol in P as different arguments to T_P. For example, in the binary transitive closure

program of Example 2.1.2, the program is a function of two IDBs R and R. See Example 2.4.2 below for a concrete example.

A key observation is that for pure Datalog the immediate consequence operator satisfies the following *"coordinate-wise linearity"* property.

$$\begin{aligned} &T_P(A_1, A_2, \cdots, A_{i-1}, A_i \cup \delta, A_{i+1}, \cdots, A_m) \\ &= T_P(A_1, A_2, \cdots, A_{i-1}, A_i, A_{i+1}, \cdots, A_m) \\ &\cup T_P(A_1, A_2, \cdots, A_{i-1}, \delta, A_{i+1}, \cdots, A_m). \end{aligned} \tag{2.6}$$

When $\boldsymbol{A} = (A_1, \cdots, A_m)$ and $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_m)$ are collections of predicates, the differential rule can be computed by applying the identity (2.6) m times, to obtain:

$$\begin{aligned} f(\boldsymbol{A} \cup \boldsymbol{\delta}) \setminus f(\boldsymbol{A}) &= f(A_1 \cup \delta_1, A_2 \cup \delta_2, \ldots, A_m \cup \delta_m) \setminus f(\boldsymbol{A}) \\ &= f(A_1 \cup \delta_1, A_2 \cup \delta_2, \ldots, A_{m-1} \cup \delta_{m-1}, \delta_m) \\ &\cup f(A_1 \cup \delta_1, A_2 \cup \delta_2, \ldots, \delta_{m-1}, A_m) \\ &\vdots \\ &\cup f(A_1 \cup \delta_1, \delta_2, A_3, \ldots, A_{m-1}, A_m) \\ &\cup f(\delta_1, A_2, \ldots, A_{m-1}, A_m) \setminus f(\boldsymbol{A}) \end{aligned} \tag{2.7}$$

The above derivation gives a efficient way of evaluating the differential rule, making semi-naïve evaluation sometimes faster than naïve evaluation. To summarise, the semi-naïve algorithm thus shows how to achieve the fixpoint incrementally, by only accounting for the changes between the iteration steps.

Example 2.4.1 (Linear transitive closure)**.** A particularly simple application of semi-naïve optimization is when the Datalog program is *linear*, i.e. every rule has only one IDB in the body of the rule. In this case, δR_t is a function of δR_{t-1}, according to (2.7). For example, consider the linear transitive closure formulation in Example 2.1.1. In this case R only appears once on the right-hand-side of (2.4); hence, the program is linear. The differential rule to compute $\boldsymbol{R}_{t+1}$, after the first iteration, is

$$\delta R_{t+1}(U, V) = \exists W \ \delta R_t(U, W) \wedge E(W, V).$$

This makes sense: after initializing $\delta R_1 = E$, for the rest of the iterations, the fact that u can reach v can be discovered by seeing that u can reach w in the previous iteration and $w \to v$ is an edge. In this way, we will not have to repeat so much work as dictated by the naïve algorithm.

Example 2.4.2 (Binary transitive closure)**.** Consider the binary-recursion formulation of transitive closure (2.1.2). As mentioned earlier, the ICO T_P has two duplicate arguments R and R, in order for us to apply the optimization rule (2.7). The semi-naïve algorithm with the differential rule (2.7) proceeds as follows. It initializes $R_0 = \emptyset$ and $\delta R_0 = E$, then repeats the following instructions for $t = 1, 2, \ldots$

$$\begin{aligned}\delta R_{t+1}(U,V) &= (\exists W\ R_t(U,W) \wedge \delta R_t(W,Z)) \\ &\quad \cup (\exists W\ \delta R_t(U,W) \wedge R_{t-1}(W,V)) \\ &\quad \setminus R_t(U,V) \\ R_{t+1}(U,V) &= R_t(U,V) \cup \delta R_{t+1}(U,V)\end{aligned}$$

In practice, depending on the data and the shape of the program P, it is not always the case that semi-naïve evaluation is faster than naïve evaluation.

Under data complexity, the above claim is not true: Semi-naïve is always guaranteed to have a lower data complexity than naïve. – Mahmoud.

The claim isn't about asymptotic complexity; it's "in practice" – Hung.

Furthermore, one may not want to implement the rule (2.7) as is, because that rule requires keeping two consecutive versions of each IDB (e.g. R_t and R_{t-1} in Example 2.4.2), which may be expensive in terms of space complexity. Identity (2.6) is the key property, which we can use to evaluate the differential rule without keeping two versions of each IDB, at the expense of a more complex query to compute the differential.

Example 2.4.3. Consider the line graph from Example 2.3.2. Suppose that we want to compute its transitive closure using the semi-naïve evaluation of the linear transitive closure from Example 2.4.1. This evaluation will take $\Theta(N)$ iterations. At iteration t, we will discover new

facts by joining facts of the form $(u-t, u)$ with $(u, u+1)$ for $u \in [N]$. The total amount of work is thus $\sum_{t\in[N]} \sum_{u\in[N]} \Theta(1) = \Theta(N^2)$. On the other hand, the semi-naïve evaluation of the binary transitive closure from Example 2.4.2 takes $\Theta(\log N)$ iterations. At iteration t, it joins facts $(u-k)$ with $(u+l)$ where either $1 \le k \le 2^t \wedge 2^{t-1} < l \le 2^t$ or $2^{t-1} < k \le 2^t \wedge 1 \le l \le 2^{t-1}$. Across all iterations, each fact (u, v) where $u < v$ will be rediscovered $v - u - 1$ times. The total amount of work is thus $\Theta(N^3)$.

Example 2.4.4. Consider the $\sqrt{N} \times \sqrt{N}$ grid graph from Example 2.3.2. Using a similar analysis as above, we can show that the linear semi-naïve formulation from Example 2.4.1 takes $\Theta(N^2)$ total amount of work while the binary semi-naïve one from Example 2.4.2 takes $\Theta(N^3)$.

Example 2.4.5. On the graph from Example 2.3.4, both formulations take time $\Theta(N)$.

2.4.2 Magic-set Optimization

Magic-set optimization (or transformation) (Beeri and Ramakrishnan, 1991; Balbin *et al.*, 1991) is another powerful technique for optimizing Datalog programs. It was observed that, in some practical settings, users define IDBs via a Datalog program only to access a portion of the IDB. A simple example is the following:

Example 2.4.6. Consider the transitive closure query, where we ask "which vertices are reachable from vertex 1?":

$$\begin{aligned} R(X,Y) &\text{ :- } E(X,Y) \vee \exists Z\ R(X,Z) \wedge E(Z,Y) \\ Q(Y) &\text{ :- } R(1,Y) \end{aligned}$$

Computing R and *then* Q is resource intensive, both in terms of the time it takes to compute R, and the space needed to store the relation R, only to throw away quadratically most of it to get to Q in the end. As a human programmer, we could have formulated our question by "pushing" Q into the recursion for computing R, thereby reformulating the computation of Q as a recursive Datalog program directly, without going through R:

$$Q(Y) \text{ :- } E(1,Y) \vee \exists Z\ Q(Z) \wedge E(Z,Y). \tag{2.8}$$

The reformulation says: 1 can reach Y if either there is an edge from 1 to Y, or if we knew 1 can reach Z, and there is a $Z \to Y$ edge. The magic-set optimization was designed to automate this type of reformulation.

There are several ways to describe the magic set transform. Our presentation here follows a combination of (Mascellani and Pedreschi, 2002; Beeri and Ramakrishnan, 1991). For simplicity, we assume that the input consists of a Datalog program P_{in} computing IDB predicates in $\boldsymbol{R}$ (a tuple of IDB predicates). Also, there is a query Q that accesses a predicate $T \in \boldsymbol{R}$ by setting a subset of variables in T to be specific constants. In particular, Q is of the form

$$Q(\boldsymbol{Y}) \text{ :- } T(\boldsymbol{X}, \boldsymbol{Y}) \wedge \boldsymbol{X} = \boldsymbol{x}. \tag{2.9}$$

We can further generalize the formulation of this "query" (i.e. the way users access portions of IDBs defined via a Datalog program); however, we refrain from doing so to get the main ideas across more clearly.

The Magic-set transform The magic set transform is a (Datalog) program-to-program transformation that, given a recursive Datalog program P_{in} and a query Q of the form (2.9), produces a new Datalog program P_{out} that computes the same output Q. There are three main steps:

1. Construct a "sideways information passing strategy" (SIPS).
2. From the SIPS and the input program, construct an "adorned program".
3. From the adorned program, perform the magic set transform to obtain P_{out}.

The SIPS (defined below) is a parameter to the magic-set transform; different SIPSs lead to different P_{out}, which may have vastly different performance characteristics. The adorned program is an intermediate representation constructed from the SIPS in order to guide the construction of P_{out}.

We start with describing the notion of "adornments".

Definition 2.4.7 (Adornment)**.** Given an arity-k relation symbol R, an *adornment* for R (also called *binding pattern* or *moding* in the literature) is a tuple of k letters in $\{\mathsf{b}, \mathsf{f}\}^k$. Here, b stands for "bound" and f stands for "free". For example, if R is the edge relation of a (non-hyper) graph, then bf, bb, fb, and ff are possible adornments for R.

An adornment is an annotation to indicate the fact that the variables at the b-positions in R may have their domains restricted to be a "small" subset. For example, when we are querying for for $R(1, Y)$ in Example 2.4.6, we may assign R with the adornment bf, because we are interested in the restriction onto the value 1 of the first variable in R.

Creating the magic-set transformed program We start with describing the 3rd step, where we assume that every IDB R of the input program was already given an adornment $\boldsymbol{p} \in \{\mathsf{b}, \mathsf{f}\}^k$. The only requirement we have for the adornment is that T is adorned with the pattern $\boldsymbol{p}$ where the bound positions of $\boldsymbol{p}$ are precisely the positions where the query $Q(Y) \coloneq T(\boldsymbol{x}, Y)$ sets to be constants. We sometimes write $R^{\boldsymbol{p}}$ to signify the fact that R was adorned with pattern $\boldsymbol{p}$.

We next create for each IDB predicate R two new predicate symbols: R^o and R', where R^o has the same arity as R, and R' has the same arity as the number of b's in $\boldsymbol{p}$. Roughly speaking, the variables of R' are the variables of R in the bound positions of $\boldsymbol{p}$, and the variables of R^o are the same as those of R. [3] For instance, for $R^{\mathsf{bf}}(X, Y)$, we will have $R^o(X, Y)$ and $R'(X)$.

The magic-set transform produces P_{out} by adding the following rules:

- For each rule adorned rule $A \coloneq B_1 \wedge \cdots \wedge B_n$, add the following rules to P_{out}:
  - The rule $A^o \coloneq A' \wedge B_1^o \wedge \cdots \wedge B_n^o$
  - The rule $B_i' \coloneq A' \wedge B_1^o \wedge \cdots \wedge B_{i-1}^o$ for each $i \in [n]$
- The rule $T'(\boldsymbol{X}) \coloneq \boldsymbol{X} = \boldsymbol{x}$

[3] In the literature, R' is sometimes denoted by R^{magic}; these are the "magic sets". R^o stands for R "optimized," which we will need to prove the correctness of the transform in a later chapter.

- The rule $\boldsymbol{Q}(\boldsymbol{Y})$:- $T^o(\boldsymbol{x}, \boldsymbol{Y})$

Example 2.4.8. Consider the query from Example 2.4.6. Suppose R was adorned with the bf adornment, i.e. the adorned program (in logic-programming notation) is

$$\begin{aligned} R^{\mathsf{bf}}(X,Y) &\text{ :- } E(X,Y) \\ R^{\mathsf{bf}}(X,Y) &\text{ :- } R^{\mathsf{bf}}(X,Z) \wedge E(Z,Y) \\ Q(Y) &\text{ :- } R^{\mathsf{bf}}(1,Y) \end{aligned}$$

Then, the magic-set transformed program reads:

$$\begin{aligned} R'(X) &\text{ :- } X = 1 \\ R^o(X,Y) &\text{ :- } R'(X) \wedge E(X,Y) \\ R^o(X,Y) &\text{ :- } R'(X) \wedge R^o(X,Z) \wedge E(Z,Y) \\ Q(Y) &\text{ :- } R^o(1,Y) \end{aligned}$$

This is effectively the same formulation as the hand-written rule (2.8).

Sideways information passing (SIPS) The main intuition behind doing a magic set transform is that, if the output query such as $Q(Y)$:- $R(1,Y)$ is only interested in the truth-values over a small'ish subset of all possible facts, then we may not need to compute all possible facts during the recursion evaluation. Stated in another way, if we are only interested in the truth-values over some small sub-domain (such as only $X = 1$), then this output domain restriction can sometimes be "pushed" inside the recursion so that there is also a domain restriction during recursion evaluation. The idea is essentially the typical query optimization idea of "pushing selection through joins", except for the fact that with magic-set transformation we are pushing the selection into a recursion and thus we need to be very careful about preserving the semantics of the answer.

A *sideways information passing strategy* (SIPS) (Beeri and Ramakrishnan, 1991) is a formal specification of how the domain restriction from one predicate can be "passed" to another predicate in the datalog program.

Formally, a SIPS is a collection of *SIPS arcs*. Every SIPS arc specifies how domain restriction can be "pushed" from one IDB to another via a subset of predicates in a rule. In particular, given a datalog rule A :- $B_1 \wedge \cdots \wedge B_n$ where A was already adorned with an adornment $\boldsymbol{p}$, a SIPS arc is a triple $(\mathcal{L}, B, \boldsymbol{q})$ where $\mathcal{L} \subseteq \{A^{\boldsymbol{p}}, B_1, \ldots, B_n\}$, $B \in \{B_1, \ldots, B_n\}$, and $\boldsymbol{q} \in \{\mathsf{b}, \mathsf{f}\}^{\mathsf{arity}(B)}$ is an adornment. The variables of B in the b-positions in $\boldsymbol{q}$ are a subset of all variables appearing in the predicates in $\mathcal{L}$.

The SIPS arc is meant to capture the idea that, if predicates in $\mathcal{L}$ are domain restricted, then when put in a conjunction with B, we can also restrict the domains of the variables in B which are in the bound positions of the adornment $\boldsymbol{q}$. Initially, for the head A of the rule, we have the domain restriction specified by $\boldsymbol{p}$. This way, information about what we are interested in from the head is passed through SIPS arcs into the predicates in the body.

Creating an adorned program Start with $\mathcal{N} = \{T^{\boldsymbol{a}}\}$, where $\boldsymbol{a}$ is the demand-pattern on the target IDB T.[4] While $\mathcal{N} \neq \emptyset$, repeat the following steps:

- Remove an adorned predicate $A^{\boldsymbol{p}}$ from $\mathcal{N}$
- For every rule r in P_{in} with A in the head, A :- $B_1 \wedge \cdots \wedge B_m$
  - For every SIPS arc $(\mathcal{L}, B, \boldsymbol{q})$ corresponding to the adorned $A^{\boldsymbol{p}}$ (obtained from SIPS)
    - Create an adorned predicate $B^{\boldsymbol{q}}$. If $B^{\boldsymbol{q}}$ is seen for the first time, then add it to $\mathcal{N}$.
    - Add the (by now adorned) rule r to the output, where every predicate is replaced with its adorned version (including the head)

Note the crucial property that, in the above procedure, the same predicate may be given multiple demand-patterns. (We will have to "remember" the SIPS arc creating $B^{\boldsymbol{q}}$, which will be used to transform demand

[4] In the general case, if there were more than one pattern that were demanded, or more than one IDB that was used, then we start with that entire set.

into $B^{\boldsymbol{q}}$ in the magic-set transform step. Note the crucial property that, for the same $B^{\boldsymbol{q}}$, there may be multiple SIPS arcs with demands to be pushed into it. In that case, there will be multiple rules with $B^{\boldsymbol{q}}$ in the head.)

Which SIPS are good? As aforementioned, the chosen SIPS has an enormous effect on the characteristics of the output program. To the best of our knowledge, we do not have a satisfactory theory of which SIPS are good. However, there are some heuristics that are often used in practice. For example, in order to avoid circular information passing, the SIPS arcs corresponding to a given rule A :- $B_1 \wedge \cdots \wedge B_n$ must satisfy an "acyclicity" condition: there is a total order on the set $\{A', B_1, \ldots, B_n\}$ where A' is first in the order, and, for every arc $(\mathcal{L}, B, \boldsymbol{p})$, the predicates in $\mathcal{L}$ precede B in the total order. All predicates that do not appear in the SIPS are listed last in the order.

To illustrate the main issues, we present next some examples of good SIPS and bad SIPS for the same program.

Mahmoud, can you please type up some examples: good SIPS, bad SIPS, SIPS leading to infinite recursion, etc. thanks. –Hung.

More TODOs. (1) give an example; same generation (2) say that proof of correctness is deferred to a later chapter, where we can generalize the rules to semiring setting –Hung.

2.4.3 Incremental View Maintenance

A common problem in practice is to maintain the materialization or output of a Datalog program under data updates. Updates are expressed as deletions and insertions of tuples from and respectively to the extensional predicates. An immediate approach is to re-evaluate the program after each batch of updates. However, this may be too costly, as the updates may only affect a small part of the program output. A better approach is to only recompute those parts of the output that are affected by the updates. This amounts to incrementally maintaining the program output, hence the general term of *incremental view maintenance* or IVM for short.

The ideal approach would only take time proportional to the amount of the change in the program output, yet this is generally difficult or even impossible to achieve. In this section, we discuss two well-established approaches for the incremental maintenance of pure Datalog programs: DRed and FBF. Later in the book, we will revisit the incremental maintenance problem and consider more sophisticated solutions that rely on counting and program stratification.

DRed: The Derive/ReDerive Algorithm

Given a set Δ_- of facts to delete and a set Δ_+ of facts to insert, DRed proceeds in three steps. In the *over-deletion* step, it deletes each fact in each (intensional and extensional) predicate derivable in zero or more steps from a fact in Δ_-. In the *re-derivation* step, it re-derives those facts that remain re-derivable after the previous step. In the *insertion* step, it further computes all facts that are derivable using Δ_+ and the predicates updated after the previous step. As the names of the first two steps suggest, DRed's main weakness is that it may over-delete facts that have several possible derivations, where some, yet not all, of these derivations are invalidated by the deletes in Δ_-. These facts need to be re-derived back. This redundant computation may cost non-trivial time. The benefit is that DRed remains simple and needs no bookkeeping of the number of derivations for each fact.

We next exemplify DRed using a simple recursive Datalog program and then show how to specify the DRed maintenance declaratively. We conclude with historical notes.

Example 2.1. Consider the following program (Motik *et al.*, 2019):

$$T(X,Y) \text{ :- } E_1(X,Y) \tag{2.10}$$
$$T(X,Y) \text{ :- } R(X,Y) \wedge E_2(X) \tag{2.11}$$
$$T(X,Y) \text{ :- } S(X,Y) \wedge E_3(X) \tag{2.12}$$
$$V(X) \text{ :- } E_4(X) \tag{2.13}$$
$$V(Y) \text{ :- } T(X,Y) \wedge V(X) \tag{2.14}$$

Now consider the following facts in the extensional database:

$$E_1 = \{(a,b),(b,c),(c,d),(d,c),(e,c),(f,g),(g,c)\}$$
$$R = \{(b,e)\} \qquad S = \{(b,f)\} \qquad E_2 = \{(b)\} \qquad E_4 = \{(a)\}$$

The materialization of the program yields the following derived facts:

$$T = E_1 \cup \{(b,e)\}$$
$$V = E_4 \cup \{(b),(c),(d),(e)\}$$

The new fact $T(b,e)$ is derived from $R(b,e)$ and $E_2(b)$ using Rule (2.10). The new facts in V are derived by applying repeatedly Rule (2.14) with the facts in T and the facts in V.

Consider now the delete $\Delta_- = \{E_2(b)\}$ followed by the insert $\Delta_+ = \{E_2(b)\}$. In the over-deletion step, we gather all facts that can be derived from $E_2(b)$ and the existing facts:

$$T(b,e) \text{ :- } R(b,e) \wedge E_2(b) \tag{2.15}$$
$$V(e) \text{ :- } T(b,e) \wedge V(b) \tag{2.16}$$
$$V(c) \text{ :- } T(e,c) \wedge V(e) \tag{2.17}$$
$$V(d) \text{ :- } T(c,d) \wedge V(c) \tag{2.18}$$
$$V(c) \text{ :- } T(d,c) \wedge V(d) \tag{2.19}$$

The facts $T(b,e)$, $V(e)$, $V(c)$, and $V(d)$ are thus deleted. While Rule (2.19) fires as it is triggered by the derivation of $V(d)$ in Rule (2.18), the derived fact $V(c)$ was already deleted, so it cannot be deleted again. There are no further facts that can be derived in this first step.

In the re-derivation step, we aim to re-derive all facts that were deleted in the previous step, using the state of the program materialization as left by the previous step. It turns out that some of these facts admit alternative derivations:

$$V(c) \text{ :- } T(b,c) \wedge V(b) \tag{2.20}$$

I am still working on this example –DanO.

Historical Notes (Ceri and Widom, 1991)

(Gupta *et al.*, 1993)
(Staudt and Jarke, 1996)
(Motik *et al.*, 2019)

FBF: The Forward/Backward-Forward Algorithm

2.4.4 Other recursion optimization techniques

write this one as a related work type of section; cite papers but do not need to describe details –Hung.

Various stratification work from Ken Ross, Zaniolo, etc. Perhaps examples such as (Ross, 1994) don't belong to this section?

2.5 Further Readings

Notation	Meaning
X, Y, Z	Variables
x, a, b	Constants
R, S, T	Predicate symbols
$\boldsymbol{R}$, $\boldsymbol{S}$, $\boldsymbol{T}$	Tuples of predicate symbols
$\mathbf{X}, \mathbf{Y}, \mathbf{Z}$	Predicate schemas, tuple of variables
$\mathbf{x}, \mathbf{y}, \mathbf{z}$	Tuples of constants
A, B	Atoms
$R(\mathbf{x})$	Ground atoms
A :- $B_1 \wedge \cdots \wedge B_k$	Datalog rule with head atom A and conjunction of body atoms $B_1, \ldots, B_k$
P	Datalog program, a collection of Datalog rules
I	Database instance
T_P	Immediate consequence operator for program P
M	a (partial) interpretation or model of a program
$M(\Phi)$	truth value of formula Φ in interpretation M
$\mathsf{ground}(P, D)$	the grounded program of P relative to domain D
$\mathsf{ground}(P)$	the grounded program when D is clear from the context
$HB(P, D)$	the Herbrand Base of P relative to domain D, i.e., all ground atoms that can be formed with symbols in P and D.
lfp	least fixed point (is it "fixpoint" or "fixed point"?)
gfp	greatest fixed point
$P \cup I$	to denote that I is integrated into the program, i.e., we identify edb atoms A with rules of the form A :- .
$T_P^\omega(\emptyset)$	Compute the lfp starting from $\emptyset$ (rather than $T_P^\omega(I)$) and consider the edb as rules with empty body.
r: ...	use r (possibly with subscript) to denote a rule
p, q	possibly with subscript: to denote propositional atoms

Figure 2.1: Notation used in this book.

3

Extensions Based on Logic

This chapter is almost complete; need to write an intro paragraph here. The Further Reading section needs to be updated. And, XY-stratification needs to be updated

3.1 Adding Negation

Between the mid-1980s and early 1990s, intensive research in both, the logic programming and the Database Theory communities, were aiming at defining an intuitive and useful semantics of logic programs with negation. These activities are witnessed by numerous publications at the primary venues of these communities such as ICLP and PODS. Below we briefly recall the most important approaches.

We are now dealing with so-called *normal* Datalog programs, where rules are of the form $A \leftarrow A_1, \ldots, A_k \neg B_1 \ldots \neg B_\ell$ with $k, \ell \geq 0$. The resulting language is referred to as Datalog$^\neg$.

3.1.1 Stratification

The simplest form of Datalog$^\neg$ occurs if negation is only allowed for edb-atoms in the body of a rule, i.e., all B_j's are edb-atoms. The language of

such programs is referred to as *semipositive* Datalog$^\neg$ by Apt *et al.*, 1988. For a semipositive Datalog$^\neg$ program P, the immediate consequence operator T_P is monotone, where T_P is defined analogously to Datalog programs, i.e.: $A \in T_P(I)$ iff there exists a rule in ground(P) with head A whose body is `true` in I (Apt *et al.*, 1988). Hence, the semantics of such a program P is defined as the lfp of T_P, which in turn can be computed as $T_P^\omega(\emptyset)$.

This idea is then extended in (Apt *et al.*, 1988) to stratified programs. Intuitively, negation is only applied to *known* relations or, equivalently, there is no recursion through negation. The idea is formalized by the following definition.

Definition 3.1. Consider a Datalog$^\neg$ program P. A *stratification* σ of the predicate symbols occurring in P is a mapping that assigns 0 to the edb predicates and a positive integer to every idb predicate, such that the following condition holds:

For every rule in P of the form $A \leftarrow A_1, \ldots, A_k \neg B_1 \ldots \neg B_\ell$, such that Q is the predicate symbol of the head atom A, we have:

- if R is the a predicate symbol of a positive body atom A_i, then $\sigma(Q) \geq \sigma(R)$;
- if R is the a predicate symbol of a negated body atom B_i, then $\sigma(Q) > \sigma(R)$.

If such a stratification exists, then P is called *stratified*.

In the above definition, we may assume w.l.o.g., that a stratification σ assigns values $\{1, \ldots, n\}$ for some $n \geq 1$ to the idb predicates occurring in P. Then σ gives rise to a partitioning $P = P_1 \cup \cdots \cup P_n$ of program P, such that P_i consists of those rules whose head predicates are assigned value i by σ. Each P_i is referred to as a *stratum* of P. Intuitively, each stratum defines relations in terms of itself only positively and in terms of relations from previous strata positively or negatively. Stratification can be easily checked via the *dependency graph*, namely: a program is stratifiable iff its dependency graph has no cycle containing a negative edge (for details, see (Apt *et al.*, 1988)).

The semantics of a stratified program $P = P_1 \cup \cdots \cup P_n$ over an edb I is defined by layerwise application of the semantics of semipositive programs:

$$\begin{aligned} M_1 &= T^{\omega}_{P_1}(\emptyset) && \text{// with edb } I \\ M_2 &= T^{\omega}_{P_2}(\emptyset) && \text{// with edb } I \cup M_1 \\ &\vdots \\ M_n &= T^{\omega}_{P_{n-1}}(M_{n-1}) && \text{// with edb } I \cup \bigcup_{i_1}^{n-1} M_i \end{aligned}$$

That is, each P_i is considered as a semipositive program that is evaluated over the edb obtained by the evaluation of P_{i-1}.

Example 3.1. Consider the following program, which defines the complement of the transitive closure in the predicate CTC for a graph given as edb with an Edge and Vertex relation.

$$\begin{aligned} \mathsf{TC}(\mathsf{X},\mathsf{Y}) &\text{ :- } \mathsf{Edge}(\mathsf{X},\mathsf{Y}) \\ \mathsf{TC}(\mathsf{C},\mathsf{Y}) &\text{ :- } \mathsf{TC}(\mathsf{X},\mathsf{Z}), \mathsf{Edge}(\mathsf{Z},\mathsf{Y}) \\ \mathsf{CTC}(\mathsf{X},\mathsf{Y}) &\text{ :- } \mathsf{Vertex}(\mathsf{X}), \mathsf{Vertex}(\mathsf{Y}), \mathsf{X} \neq \mathsf{Y}, \neg\, \mathsf{TC}(\mathsf{X},\mathsf{Y}) \end{aligned}$$

That is, the program first defines the transitive closure TC with the obvious two rules and then defines CTC in terms of TC. Clearly, CTC depends on TC but not vice versa. So the program is stratified and can be evaluated by first saturating TC via the first two rules and then computing CTC. ◇

Importantly, the following property holds.

Theorem 3.1. Let P be stratified program. Then all stratifications are equivalent. In other words, the output of the semantics (**??**) remains the same if we consider a different stratification of the program P.

Stratified programs can be generalized to *locally stratified* programs (Przymusinski, 1988a) and yet further to *weakly stratified* programs

(Przymusinska and Przymusinski, 1988). In both cases, one considers the grounded program ground(P). The idea of *local stratification* is to assign every ground atom A in ground(P) to a stratum, denoted $\sigma(A)$, such that, analogously to Definition 3.1, if ground(P) contains a rule of the form $A \leftarrow A_1, \ldots, A_k \neg B_1 \ldots \neg B_\ell$, then $\sigma(A) \geq \sigma(A_i)$ and $\sigma(A) > \sigma(B_j)$ for every i, j. It is straightforward to extend the layerwise evaluation of stratified programs to locally stratified ones simply by considering the grounded program ground(P). In case of Datalog$^\neg$, local stratification gives almost no additional power compared with stratification. The following relationship was shown by Bidoit, 1991;

Lemma 3.2. Let P be a Datalog$^\neg$ program such that no constant occurs in the head of a non-unit rule in P. Then the following equivalence holds: P is stratified if and only P is locally stratified.

Moreover, it easily follows that, for every locally stratified Datalog$^\neg$ program P, there exists an equivalent stratified Datalog$^\neg$ program P' (see Exercise 15.14 in Abiteboul *et al.*, 1995). However, in logic programming, where function symbols are allowed, local stratification does make a difference. The prototypical example is the following program, which specifies the even natural numbers:

Example 3.2. (Przymusinski, 1988a) Let $s(\cdot)$ be a unary function symbol (denoting the successor function on the natural numbers) and let the natural numbers be represented as $\{0, s(0), s^2(0), s^3(0), \ldots\}$. Then the following program defines the predicate even, which is `true` precisely for the even numbers.

```
even(0) :-
even(s(X)) :- ¬even(X)
```

Weakly stratified programs further generalize locally stratified programs based on the observation that local stratification may possibly be destroyed by rules (or atoms occurring in rules) that never "fire" or are redundant. Przymusinska and Przymusinski, 1988 thus introduce a program transformation intertwined with layerwise evaluation. We start with program $P_0 = ground(P)$ and define M_0 as the interpretation that sets the atoms contained in the input instance I to `true` and all other ground atoms with an edb-predicate to `false`.

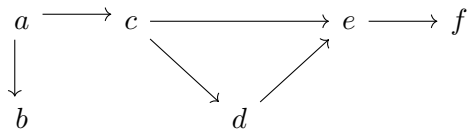


Figure 3.1: Example graph with edges $\mathsf{Edge} = \{(a,b),(a,c),(c,d),\ (c,e),\ (d,e),\ (e,f)\}$, used for the win-move game in Example 3.3.

Then, for $i > 0$, we use M_{i-1} to transform P_{i-1} into P_i by removing all "irrelevant" atoms and rules, i.e., remove all rules whose body contains a literal which is `false` in M_{i-1}, remove from the remaining rules those body literals which are `true` in M_{i-1}. Moreover, if a rule is left with empty body, then all other rules with the same head atom may be deleted. In the reduced program, we proceed with stratification but only determine the bottom layer $L(P_i)$. Provided that it contains no negation, we compute $M_i = M_{i-1} \cup \mathsf{lfp}(T_{L(P_i)})$. We iterate this process of further reducing the remaining program and computing the lfp of the bottom layer until P has been reduced to the empty set of rules. If this procedure succeeds, then the program is *weakly stratified* and the last model computed in this process is the *weakly perfect model*. If the procedure fails (i.e., at some stage the bottom layer of the program contains negation), then the program is not weakly stratified.

We illustrate these ideas by the so-called win-move game, which has been extensively studied in the literature as a prototypical Datalog$^\neg$ program with negation that cannot be stratified (Abiteboul *et al.*, 1995)

Example 3.3. The win-move game is the following pebble game played by two players on a graph $\mathsf{Edge}(\mathsf{X},\mathsf{Y})$. The first player places the pebble on some node. Players take turn and move the pebble, which can only be moved along an edge. A player who cannot move loses. The problem is to compute the set of winning position for the first player. This set is expressed concisely by the following rule:

$$\mathsf{Win}(\mathsf{X}) \text{ :- } \mathsf{Edge}(\mathsf{X},\mathsf{Y}), \neg\, \mathsf{Win}(\mathsf{Y})$$

Suppose that this game is played on the graph displayed in Figure 3.1. Clearly, the program is not stratified and, for an arbitrary directed graph, grounding does not yield a weakly or locally stratified program. However, for the acyclic

graph of Figure 3.1, we get the following ground program P_0.

$$\begin{aligned}
\mathsf{Win(a)} &\text{ :- } \mathsf{Edge(a,a)}, \neg\,\mathsf{Win(a)}\\
\mathsf{Win(a)} &\text{ :- } \mathsf{Edge(a,b)}, \neg\,\mathsf{Win(b)}\\
\mathsf{Win(a)} &\text{ :- } \mathsf{Edge(a,c)}, \neg\,\mathsf{Win(c)}\\
\mathsf{Win(a)} &\text{ :- } \mathsf{Edge(a,d)}, \neg\,\mathsf{Win(d)}\\
\mathsf{Win(a)} &\text{ :- } \mathsf{Edge(a,e)}, \neg\,\mathsf{Win(f)}\\
\mathsf{Win(b)} &\text{ :- } \mathsf{Edge(b,a)}, \neg\,\mathsf{Win(a)}\\
&\vdots\\
\mathsf{Win(f)} &\text{ :- } \mathsf{Edge(f,f)}, \neg\,\mathsf{Win(f)}
\end{aligned}$$

As model M_0 of the edb we get $M_0 = \{\mathsf{Edge(a,b)}, \mathsf{Edge(a,c)}, \mathsf{Edge(c,d)}, \mathsf{Edge(c,e)}, \mathsf{Edge(d,e)}, \mathsf{Edge(e,f)}\}$ Clearly, as it stands, the program P_0 is not locally stratified, since it contains many cyclic dependencies with negation, e.g.: already the first rule would require a stratification σ to satisfy $\sigma(\mathsf{Win(a)}) > \sigma(\mathsf{Win(a)})$. However, we may transform program P_0 into program P_1 by deleting all rules which contain a body atom `false` in M_0 and deleting the Edge-atoms (i.e., those which are `true` in M_0) from the remaining rules. We thus get the following program P_1:

$$\begin{aligned}
\mathsf{Win(a)} &\text{ :- } \neg\,\mathsf{Win(b)}\\
\mathsf{Win(a)} &\text{ :- } \neg\,\mathsf{Win(c)}\\
\mathsf{Win(c)} &\text{ :- } \neg\,\mathsf{Win(d)}\\
\mathsf{Win(c)} &\text{ :- } \neg\,\mathsf{Win(e)}\\
\mathsf{Win(d)} &\text{ :- } \neg\,\mathsf{Win(e)}\\
\mathsf{Win(e)} &\text{ :- } \neg\,\mathsf{Win(f)}
\end{aligned}$$

It turns out the P is weakly stratified, since the program P_1 can be stratified by the following stratification σ: $\sigma(\mathsf{Win(b)}) = \sigma(\mathsf{Win(f)}) = 1$, $\sigma(\mathsf{Win(e)}) = 2$, $\sigma(\mathsf{Win(d)}) = 3$, $\sigma(\mathsf{Win(c)}) = 4$, and $\sigma(\mathsf{Win(a)}) = 5$. Layered evaluation yields the following sequence of models, where we only show idb atoms up to a given stratum and with truth value `true` (i.e, edb atoms and idb atoms of higher strata or with truth value `false` are left out):

$\mathsf{M}_1 = \{\}$ // i.e., $\mathsf{Win(b)}$ and $\mathsf{Win(f)}$ are `false`
$\mathsf{M}_2 = \{\mathsf{Win(e)}\}$
$\mathsf{M}_3 = \{\mathsf{Win(e)}\}$ // i.e., we now also have that $\mathsf{Win(d)}$ is `false`
$\mathsf{M}_4 = \{\mathsf{Win(e)}, \mathsf{Win(c)}\}$
$\mathsf{M}_5 = \{\mathsf{Win(e)}, \mathsf{Win(c)}, \mathsf{Win(a)}\}$

$M = M_5$ is the weakly perfect model of program P over the edb given in Figure 3.1. Note that the program on layer 1 (i.e., restricted to the atoms $\{\mathsf{Win(b)}, \mathsf{Win(f)}\}$ is the special case of the empty program. In particular, this means that it contains no rule with head atom either $\mathsf{Win(b)}$ or $\mathsf{Win(f)}$. That is why these atoms are set to `false` in model M_1. The same happens on layer 3 where atom $\mathsf{Win(d)}$ is set to `false` for the same reason.

Finally, as a middle ground between locally stratified and weakly stratified programs, (Ross, 1994) defines *modularly stratified* programs. The motivation for this additional form of stratification is that it generalizes local stratification while avoiding computational problems caused by weak stratification. For details, see (Ross, 1994).

Apart from the procedural definition of the semantics for various forms of stratification, there also exists a declarative approach to a semantics definition of stratified and locally stratified programs in the form of *perfect models* (Przymusinski, 1988b; Przymusinski, 1989b). The approach is inspired by Mc Carthy's prioritized circumscription from the Artificial Intelligence domain (McCarthy, 1984). The strata of a locally stratified program are thus considered as priority levels and the goal is to find a model of the given program by always trying to minimize the atoms set to `true` in a higher priority level (= a lower stratum) even if this means increasing the number of atoms set to `true` in a lower priority level (= a higher stratum).

Formally, analogously to the definition of strata in Definition 3.1, we can define relations $<$ and $\leq$ on ground atoms as follows: $A < B$ means that A must lie in a stratum below B and $A \leq B$ means that A must lie in the same stratum as B or in a stratum below. We can then use these relations to define a preference relation on models as follows.

Definition 3.2. Let N, M denote models of a locally stratified program P. We say that N is *preferable* to M (denoted $N \ll M$) if for every ground atom A in $N \setminus M$, there exists a ground atom B in $M \setminus N$ with $A < B$. A model is called *perfect* if there is no model preferable to M.

We obviously have that $N \subset M$ implies $N \ll M$. Hence, in particular, perfect models are minimal and every locally stratified program has a unique perfect model – namely the one obtained by the above mentioned layerwise evaluation.

Note that in (Przymusinska and Przymusinski, 1988), perfect models are generalized to *weakly perfect models* to cover also weakly stratified programs. However, these *weakly perfect models* are introduced in a procedural fashion – by the above mentioned intertwining of layerwise program evaluation and reduction. No declarative definition is given.

3.1.2 Stable Model Semantics

We have described in the previous section several methods for stratification. However, there exists Datalog$^\neg$ programs where no form of stratification applies, and this has motivated researchers to propose more powerful approaches for reasoning about negation in Datalog programs. For example, consider the program consisting of two rules:

$$\begin{aligned} p &\leftarrow \neg q \\ q &\leftarrow \neg p \end{aligned} \tag{3.1}$$

where p, q are distinct ground atoms. This program has two models $M_1 = \{p\}$ and $M_2 = \{q\}$. Since this program is symmetric in p and q, there is no justifiable criterion to put p and q into different strata or to prefer one model over the other (Abiteboul *et al.*, 1995).

There exist several approaches to defining a useful semantics for general Datalog$^\neg$ programs. We first inspect the so-called *stable model semantics*. There are many equivalent definitions of the stable model semantics of Datalog$^\neg$ programs (Lifschitz, 2008). We follow here the original definition from (Gelfond and Lifschitz, 1988) via reducts. In the remainder of Section 3.1, it is convenient to consider *ground* programs, i.e., we take $ground(P)$ instead of P.

Definition 3.3. Let P be a (ground) Datalog$^\neg$ program and let $M \subseteq HB(P, D)$ denote a set of ground atoms. The Gelfond-Lifschitz reduct of P w.r.t. M (denoted P^M) is obtained from P as follows:

- delete from P all rules that contain a body literal $\neg B$ with $B \in M$;
- the remaining rules only contain negative body literals $\neg B$ with $B \notin M$; we now delete all these negative body literals.

That is, P^M is a Datalog program, i.e., it contains no negation anymore. We say that M is a *stable model* of P, if M is the unique minimal model of the Datalog program P^M.

Intuitively, M can be considered as a "guess" of the truth values of the negative literals occurring in P. The Gelfond-Lifschitz reduct thus replaces every negative literal by this "guessed" truth value. Clearly, if at least one negative body literal in a rule gets truth value `false`, then the entire rule gets `false` and we may not use it in a derivation. We therefore delete it. On the other hand, negative literals evaluating to `true` in M may be deleted from the rule bodies since, in a sense, we have thus already verified (based on our guess M) that they are `true`. Then a model M of P is a *stable model*, if this guess of the truth values of the negative literals was "correct", so to speak.

The minimal model of a Datalog program or the perfect model of a locally stratified Datalog$^\neg$ program always exists and is unique. In contrast, a general Datalog$^\neg$ program may have 0, 1, or several stable models.

The prototypical program with 2 stable models is the program (3.1) shown above. It has two stable models, $M_1 = \{p\}$ and $M_2 = \{q\}$. We verify that M_1 is a stable model. In model M_1, the negative literal $\neg q$ is `true` while $\neg p$ is `false`. Hence, in the Gelfond-Lifshitz reduct P^{M_1} of P w.r.t. M_1, we delete the second rule and the body literal $\neg q$ from the first rule and obtain the single rule:

$$p \leftarrow$$

Its minimal model is $M_1 = \{p\}$, which proves that M_1 is a stable model. The case of M_2 is symmetric.

The protoypical program with no stable model at all is the program P consisting of a single rule $p \leftarrow \neg p$. The only model of this rule is $M = \{p\}$. However, the Gelfond-Lifschitz reduct of P w.r.t. M is the empty program and, therefore, the minimal model of P^M is the empty set (and not M). Hence, M is the only model of P but it is not stable.

A drawback of stable model semantics is that it is computationally harder than the other approaches dealing with unrestricted negation that we are going to discuss in Sections 3.1.3 and 3.1.4 and of course also

than any form of stratified negation discussed in the previous section. For instance, deciding if a propositional program with negation has at least one stable model is an NP-complete problem (see (Marek and Truszczynski, 1991)).

3.1.3 Fitting's Three-Valued Semantics

(Fitting, 1985) introduced a 3-valued semantics of logic programs (with or without negation) based on ideas of (Kripke, 1976) for modeling "truth-value gaps" using Kleene's 3-valued logic from (Kleene, 1952). In addition to `true` and `false`, atoms (and, hence, formulas) can take the truth value `undef`, later denoted $\bot$ in (Fitting, 1991), with the intended meaning of being "undefined" or "undetermined". The three values are ordered by $\texttt{false} < \bot < \texttt{true}$ (think of them as representing $0, \frac{1}{2}, 1$ respectively), and $\vee, \wedge, \neg$ are $\max, \min, 1 - x$ respectively. For example, $\neg\bot = \bot$, $\texttt{true} \wedge \bot = \bot$, $\texttt{false} \wedge \bot = \texttt{false}$, $\texttt{true} \vee \bot = \texttt{true}$, etc.

A *partial* (or *3-valued*) *interpretation* for a logic program (likewise, for a Datalog$^\neg$ program) thus assigns one of the 3 possible truth values to each ground atom. A *partial* (or *3-valued*) *model* is a partial interpretation I that satisfies every rule of the program, that is: if $A \leftarrow body$ is a ground instance of a rule, then $I(A) \geq I(body)$ holds.

There are several notations for partial interpretations in the literature. We mention three of them here:

- In (Fitting, 1991), a partial interpretation I is represented as a mapping from $HB(P, D)$ to $\{\texttt{false}, \bot, \texttt{true}\}$. That is, for every ground atom, we specify its truth value in I.

- In (Hitzler and Wendt, 2005), a partial interpretation I is represented as a consistent set of ground literals $S \subseteq \{A, \neg A \mid A \in HB(P, D)\}$. Here, "consistent" means that S may not contain both A and $\neg A$ for any ground atom A. A ground atom B is `true` (resp. `false`) in I, if $B \in S$ (resp. $\neg B \in S$). If neither $B \in S$ nor $\neg B \in S$ holds, then B has truth value $\bot$ in I.

- Similarly, in (Fitting, 2002), a partial interpretation I is represented as a pair of disjoint sets of ground atoms (T, F) with the

intended meaning that a ground atom B is `true` in I if $B \in T$; it is `false` in I if $B \in F$; and it is $\bot$ in I otherwise.

In this section, we will adopt the notation via consistent sets of ground literals from (Hitzler and Wendt, 2005). Note that, if for every atom $A \in HB(P, D)$ either $A \in S$ or $\neg A \in S$, then S is actually a *complete* (or two-valued) interpretation.

Obviously, for a Datalog$^\neg$ program P, the ICO T_P is not monotone and, hence, the existence of a fixpoint is not guaranteed. The simplest example is program P consisting of a single rule $p \leftarrow \neg p$, where $T_P^n(\emptyset)$ oscillates between setting p to `false` and `true`. When talking about "non-monotonicity", we have the *truth order* $\leq_t$ in mind with `false` $\leq_t$ $\bot \leq_t$ `true`, which extends to (partial or complete) interpretations I, J as $I \leq_t J$ if $I(A) \leq_t J(A)$ for every ground atom A.

Fitting remedies the problem of non-monotonicity by introducing an additional order $\leq_k$ on truth values, referred to as *knowledge order* with $\bot \leq_k$ `true` and $\bot \leq_k$ `false` being the only order relationships. Note that, under this order, negation becomes monotone. This is due to the fact that `true` and `false` are incomparable under $\leq_k$ and $\neg\bot = \bot$. As a consequence, also the immediate consequence operator, to be defined next, becomes monotone.

First, this order is naturally extended to partial interpretations as $I \leq_k J$ if $I(A) \leq_k J(A)$ for every ground atom A. By using the representation via sets of ground literals, the order $I \leq_k J$ simply comes down to $I \subseteq J$. Then, Fitting defines the following ICO Φ_P as the natural extension of the ICO T_P to partial interpretations. More specifically, for a (ground) Datalog$^\neg$ program P, we set $\Phi_P(I) = J$, such that for a ground atom $A \in HB(P, D)$ we have:

- J assigns `true` to A (i.e., $A \in J$), if there is a rule $A \leftarrow B_1, \ldots, B_n$ in P, such that all literals B_i are `true` in I, i.e, $B_i \in I$;

- J assigns `false` to A (i.e., $\neg A \in J$), if for every rule r with head atom A in P, the body of r evaluates to `false` in I, i.e., for $r\colon A \leftarrow B_1, \ldots, B_n$, at least one B_i is `false` in I (this includes the case that no rule with head atom A exists in P);

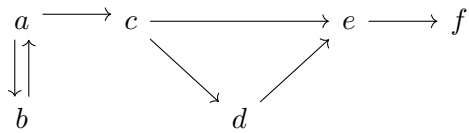


Figure 3.2: Example graph with edges $Edge = \{(a,b),(a,c),(c,d),\ (b,a),\ (c,e),\ (d,e),\ (e,f)\}$, used for the win-move game in Example 3.4.

- otherwise, J assigns $\bot$ to A (i.e., $A \notin J$ and $\neg A \notin J$).

Under the partial order $\leq_k$ on the set of partial interpretations of a program P, the ICO Φ_P is *monotone*. Hence, the least fixpoint of Φ_P exists and it is chosen as semantics of logic programs by (Fitting, 1985). Fitting refers to it as the "Kripke-Kleene semantics" but it is later commonly called *Fitting semantics* (Hitzler and Wendt, 2005). We use the latter naming here.

We illustrate the Fitting semantics by revisiting the win-move game from Example 3.3, but now on a graph that is not acyclic.

Example 3.4. Recall that the win-move game is realized by a single Datalog$^\neg$ rule:

$$\mathsf{Win(X)} \text{ :- } \mathsf{Edge(X,Y)}, \neg\, \mathsf{Win(Y)}$$

We now consider the edb given by the graph shown in Figure 3.2. Then iterating the ICO Φ_P computes the following sequence of IDB instances, $\mathsf{Win}^{(0)} \leq_\mathsf{k} \mathsf{Win}^{(1)} \leq_\mathsf{k} \mathsf{Win}^{(2)} \leq_\mathsf{k} \cdots$, shown here. It is convenient to abbreviate Win to W; moreover, we write 0 for `false` and 1 for `true`:

	W(a)	W(b)	W(c)	W(d)	W(e)	W(f)	
$\mathsf{W}^{(0)} =$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	
$\mathsf{W}^{(1)} =$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	0	
$\mathsf{W}^{(2)} =$	$\bot$	$\bot$	$\bot$	$\bot$	1	0	
$\mathsf{W}^{(3)} =$	$\bot$	$\bot$	$\bot$	0	1	0	
$\mathsf{W}^{(4)} =$	$\bot$	$\bot$	1	0	1	0	$= \mathsf{W}^{(5)}$

A 3-valued interpretation M is a model of a (ground) program P, if $M(A) \geq_t M(body)$ holds for every rule $A \leftarrow body$ of P. Now assume

that we transform P into program P' by combining all rules with the same head atom into a single rule by allowing disjunction in the body. Then a fixpoint M of Φ_P satisfies $M(A) = M(body)$ for every rule $A \leftarrow body$ of P. Therefore, every fixpoint of Φ_P is a model of P.

It should be noted that the Fitting semantics differs from the semantics introduced in Section 2 when applied to Datalog programs without negation. For instance, consider program P consisting of a single rule $p \leftarrow p$. Here, the $\mathsf{lfp}(T_P)$ is the *complete* interpretation $I = \emptyset$, which assigns truth value `false` to p. In contrast, the $\mathsf{lfp}(\Phi_P)$ is the *partial* interpretation $J = \emptyset$, which assigns truth value $\bot$ to p. Fitting, 2002 poses the question as to "which is the 'right' choice". He argues that a "good case can be made for either" and the preference of one interpretation over the other depends on the application context.

In (Fitting, 1991), Fitting extends the truth value space to *bilattices* with Belnap's 4-valued logic FOUR as the simplest case. In addition to the truth values `false`, `true`, $\bot$, we now also have $\top$. One possible interpretation of these 4 truth values is as subsets of $\{$`false`, `true`$\}$, where $\{$`false`$\}$ and $\{$`true`$\}$ correspond to the "classical" truth values `false` and `true`, $\bot = \emptyset$ means no knowledge and $\top = \{$`false`, `true`$\}$ stands for contradictory information. The knowledge order $\leq_k$ then corresponds to subset-inclusion. The 4-valued logic FOUR forms a complete bilattice based on the two orders $\leq_t$ and $\leq_k$ with `false` $\leq_t$ $\bot, \top \leq_t$ `true` and $\bot \leq_k$ `false`, `true` $\leq_k \top$. Moreover, the meet and join operator in each lattice is monotone w.r.t. the order in the other lattice, e.g.: if $a \leq_k b$ and $c \leq_k d$, then $a \wedge b \leq_k c \wedge d$. One way of looking at the orders is that greater w.r.t. $\leq_k$ means "more knowledge" while greater w.r.t. $\leq_t$, it means "less false or more true". They are visualized in Figure 3.3 (cf. (Fitting, 1991), Figure 1):

One motivation of interpreting logic programs in a bilattice is to deal with inconsistencies. (Fitting, 1991) mentions as an example a distributed environment in which program portions on different sites may possibly assign contradicting truth values. The use of a bilattice is meant to allow useful derivations for parts not affected by the contradiction. The second motivation in (Fitting, 1991), which is further elaborated in (Fitting, 1993), is to chart the territory of stable model semantics and well-founded semantics in terms of least fixpoints and greatest fixpoints

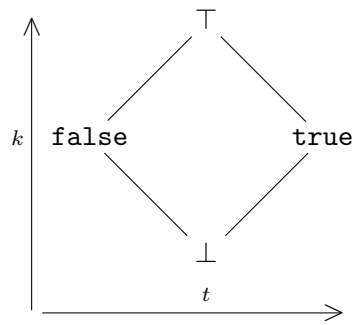


Figure 3.3: Orders $\leq_k$ and $\leq_t$ in the bilattice FOUR

of appropriately defined operators w.r.t. the orders $\leq_t$ and $\leq_k$. We will come back to this point in Section 3.1.5.

3.1.4 Well-Founded Semantics

In (Gelder *et al.*, 1988; Gelder *et al.*, 1991), the well-founded semantics is introduced via the crucial concept of *unfounded sets*. For instance, consider the Datalog program $P = \{a \leftarrow b; b \leftarrow a\}$. The classical semantics of Datalog (cf. Chapter 2) sets both (ground) atoms a and b to `false`. Intuitively, this is due to the fact that, in the fixpoint computation, a derivation of a would require b to be derived first and vice versa. As a consequence, none of them can be derived. The definition of unfounded sets generalizes this idea of mutual dependence between atoms and, hence, the impossibility to derive any of them. As such, it also matches the "Negation as Failure" paradigm of logic programming.

In this section, we mostly consider programs to be *grounded*. For partial interpretations, we use the notation via sets of positive and negative ground literals as in Section 3.1.3.

Definition 3.4. Consider a (ground) Datalog$^\neg$ program P and a partial interpretation I of P. A set $U \subseteq HB(P, D)$ of ground atoms is an *unfounded set* of P w.r.t. I, if, for every $A \in U$ it holds that every rule r with head atom A of P satisfies (at least) one of the following conditions:

1. some *positive or negative* body literal of r is `false` in I;
2. some *positive* atom B in the body of r occurs in U.

The intuition of unfounded sets is about the inability to derive some ground atom to be `true`. Suppose that we already know for sure the truth value of some ground atoms. This knowledge is given by the partial interpretation I. Then we can ignore all rules that contain a body literal with truth value `false`, since these rules will never fire. This is captured by item 1 in Definition 3.4. Item 2 in Definition 3.4 captures the fact that, if there is a circular dependency between ground atoms, then it is impossible to fire any of the rules because none of them can go first. An obvious example is the set of rules $\{a \leftarrow b, b \leftarrow a\}$, which has already been mentioned above. The notion of unfounded sets generalizes this kind of circular dependencies between ground atoms to arbitrarily complex cycles.

The well-founded semantics uses the conditions of Definition 3.4 to infer that all atoms in U are `false`. The partial interpretation I contains the knowledge that we already have about the intended model of P. That is, if I assigns one of the truth values `false` or `true` to a ground atom A in $HB(P, D)$, then this is the truth value of A in the intended model of P. On the other hand, as more knowledge about the intended model is obtained, we may possibly change the truth value of some ground atom from $\perp$ to either `false` or `true`.

Note that in Definition 3.4, which is taken from (Gelder *et al.*, 1988; Gelder *et al.*, 1991), no restriction on the relationship between I and U is imposed. However, as we will see below, the partial interpretations I and unfounded sets U considered in the definition of the well-founded semantics, will always be disjoint. That is, it can never happen that a ground atom A is contained both, in I (which means, that we want to set it to `true`) and also in U (which means, that we want to set it to `false`).

Formally, the well-founded semantics is defined via the following fixpoint computation w.r.t. the $\leq_k$-ordering. This computation uses the fact that the union of unfounded sets is again an unfounded set. Hence, given program P and partial interpretation I, there exists a greatest unfounded set of P w.r.t. I.

Definition 3.5. Consider a (ground) Datalog$^\neg$ program P and a partial interpretation I of P. The operators T_P, U_P, and W_P are defined as follows.

- A ground atom A is in $T_P(I)$, if there exists a rule r in P, such that A is the head of r and every literal in the body of r is `true` in I;
- $U_P(I)$ is the greatest unfounded set of P w.r.t. I;
- $W_P(I) = T_P(I) \cup \neg U_P(I)$, where $\neg U_P(I) = \{\neg B \mid B \in U_P(I)\}$.

The *well-founded semantics* of program P is defined as the least fixpoint (w.r.t. $\leq_k$) of W_P.

Theorem 3.3. Given a (ground) Datalog$^\neg$ program P, the well-founded semantics of P is a partial model of P.

Proof. Let $I_0 = \emptyset$ and $I_{n+1} = W_P(I_n)$ and let $I = W_P^\omega(\emptyset)$. We have to verify for every rule $A \leftarrow \mathit{body}$ in P, that $I(A) \geq_t I(\mathit{body})$ holds:

- For $I(A) = \texttt{true}$, there is nothing to prove.
- For $I(A) = \bot$, we have to show $I(\mathit{body}) \neq \texttt{true}$. We claim that $I_n(\mathit{body}) \neq \texttt{true}$ holds for every n, from which the property $I(\mathit{body}) \neq \texttt{true}$ follows. Suppose to the contrary that $I_n(\mathit{body}) = \texttt{true}$ for some n. Then $I_{n+1}(A) = \texttt{true}$ and, by monotonicity of T_P, also $I(A) = \texttt{true}$, a contradiction.
- Finally, if $I(A) = \texttt{false}$, then there exists n where $I_n(A)$ is set to `false`, i.e., $A \in U_P(I_{n-1})$. This means that body contains a literal B satisfying $I_{n-1}(B) = \texttt{false}$ or $B \in U_P(I_{n-1})$ (actually, it means that every rule with head atom A has this property). In either case, $I_n(B) = \texttt{false}$ and, by monotonicity of W_P w.r.t. $\leq_k$, we have $I(B) = \texttt{false}$ and, therefore, also $I(\mathit{body}) = \texttt{false}$.

That is, every rule in P evaluates to `true` in I, i.e., I is indeed a model of P. □

A natural question arises as to what is the complexity of computing the lfp of W_P; and here, in particular, the complexity of computing $U_P(I)$ given P and I. Iterating through all possible sets of ground atoms and checking for each if it is unfounded is clearly not computationally attractive. (Przymusinski, 1989a) was the first to prove that the well-founded semantics is computable in polynomial time data complexity. This result is obtained by Przymusinski, 1989a by providing an equivalent definition of the well-founded semantics by a slightly different formalism. Above all, Przymusinski, 1989a shows that (1) the well-founded semantics can be obtained by a "dynamic stratification", which can be considered as a generalization of the usual ("static") stratification, and that (2) it can be computed by *iterated fixpoint computation*, i.e., a fixpoint computation that involves yet another fixpoint iteration at each step. Similarly, (Gelder *et al.*, 1991), extend their definition of the well-founded semantics to an *iterated fixpoint computation* (in this case, a nested fixpoint iteration to compute $U_P(I_n)$ at each stage n) to also obtain a polynomial-time upper bound. Below, we briefly recall the *iterated fixpoint computation* of (Gelder *et al.*, 1991).

The idea of the computation of $U_P(I)$ for given ground program P and partial interpretation I is to determine its complement $HB(P,D) \setminus U_P(I)$, that is, determine all ground atoms that are *not* part of the greatest unfounded set. This is achieved by the following operator $NU_{P,I}$ on sets of ground atoms $J \subseteq HB(P,D)$. A ground atom A is in $NU_{P,I}(J)$ if there exists a rule $A \leftarrow body$ in P, s.t. no literal in $body$ is `false` in I and all positive atoms in $body$ are in J. Clearly, $NU_{P,I}$ is monotone. Hence, the lfp of $NU_{P,I}$ exists and is reached after polynomially many steps. Moreover, it is shown in (Gelder *et al.*, 1991), that $U_P(I) = HB(P,D) \setminus NU^{\omega}_{P,I}(\emptyset)$ holds.

Dan, does the explanation below help to make things clearer? – Reinhard.

Intuitively, the operator $NU_{P,I}$ tries to derive all ground atoms, which evaluate to `true` or $\bot$ if we assume the partial truth assignment I. Assuming a (in this case, partial) interpretation I in this process resembles a bit the Gelfond-Lifschitz reduct discussed in Section 3.1.2. However, in contrast to the Gelfond-Lifschitz reduct, now the assumed

(partial) interpretation I is applied to *all* atoms in a rule body and not only the negative ones.

To illustrate the well-founded semantics and the iterated fixpoint computation of U_P, we revisit Example the win-move game from Example 3.4.

Example 3.5. Recall that the win-move game is realized by a single Datalog$^\neg$ rule:

$$\mathsf{Win(X)} \text{ :- } \mathsf{Edge(X,Y)}, \neg\, \mathsf{Win(Y)}$$

As in Example 3.4, we apply this program to the (cyclic) graph in Figure 3.2. As in Example 3.3, when grounding the program, we restrict ourselves to those rules which do not have an Edge-atom in the body that is `false`. Moreover, we may of course also drop the Edge-atoms with truth value `true` from the rule bodies. We thus get the following ground Datalog$^\neg$ program:

$$\begin{aligned}
\mathsf{Win(a)} &\text{ :- } \neg\, \mathsf{Win(b)}\\
\mathsf{Win(a)} &\text{ :- } \neg\, \mathsf{Win(c)}\\
\mathsf{Win(b)} &\text{ :- } \neg\, \mathsf{Win(a)}\\
\mathsf{Win(c)} &\text{ :- } \neg\, \mathsf{Win(d)}\\
\mathsf{Win(c)} &\text{ :- } \neg\, \mathsf{Win(e)}\\
\mathsf{Win(d)} &\text{ :- } \neg\, \mathsf{Win(e)}\\
\mathsf{Win(e)} &\text{ :- } \neg\, \mathsf{Win(f)}
\end{aligned}$$

When tracing the partial interpretations produced by the computation of the well-founded model, we restrict our attention to the Win-atoms. For the Edge-atoms, it is clear that only the ones in the edb are `true` and all others are `false`.

Computation of I_1. We start with $I_0 = \emptyset$, i.e., all Win-atoms are `undef`. Hence, also $T_P(I_0) = \emptyset$. Moreover, we compute $U_P(I_0)$ via the NU_{P,I_0}-operator as follows:
$J_0 = \emptyset$.
$\mathsf{J_1} = \{\mathsf{Win(a)}, \ldots \mathsf{Win(e)}\}$ (the rules with these head atoms have no positive body literals and I_0 sets all negative literals to $\bot$; in particular, none of these body atoms is `false` in I_0).
Then $NU^{\omega}_{P,I_0} = J_2 = J_1$ and we get $U_P(I_0) = HB(P,D) \setminus J_1 = \{\mathsf{Win}(f)\}$ and, therefore, $I_1 = W_P(I_0) = \{\neg\mathsf{Win}(f)\}$.

Computation of I_2. We have $T_P(I_1) = \{\mathsf{Win}(e)\}$. Moreover, $U_P(I_1)$ is computed exactly as $U_P(I_0)$, i.e., $J_0 = \emptyset$, $J_1 = \{\mathsf{Win}(a), \ldots \mathsf{Win}(e)\} = J_2 =$

NU^{ω}_{P,I_1}. Hence, we get $U_P(I_1) = \{\mathsf{Win}(f)\}$ and, therefore, $I_2 = W_P(I_1) = \{\mathsf{Win}(e), \neg\mathsf{Win}(f)\}$.

Computation of I_3. We have $T_P(I_2) = \{\mathsf{Win}(e)\}$. Moreover, $U_P(I_2)$ is computed as follows:
$J_0 = \emptyset$, $J_1 = \{\mathsf{Win}(a), \mathsf{Win}(b), \mathsf{Win}(c), \mathsf{Win}(e)\} = J_2 = NU^{\omega}_{P,I_2}$ and, hence, $U_P(I_1) = \{\mathsf{Win}(d), \mathsf{Win}(f)\}$. In total, this yields $I_3 = W_P(I_2) = \{\mathsf{Win}(e), \neg\mathsf{Win}(d), \neg\mathsf{Win}(f)$

Computation of I_4. By the analogous consideration as above, we get $T_P(I_3) = \{\mathsf{Win}(c), \mathsf{Win}(e)\}$ and $U_P(I_3) = U_P(I_2)$. Hence, in total, we have $I_4 = W_P(I_3) = \{\mathsf{Win}(c), \mathsf{Win}(e), \neg\mathsf{Win}(d), \neg\mathsf{Win}(f)\}$.

Computation of I_5. It turns out that the computation of I_5 yields the same result as I_4. Hence, $I_5 = I_4 = \{\mathsf{Win}(c), \mathsf{Win}(e), \neg\mathsf{Win}(d), \neg\mathsf{Win}(f)\}$. is the well-founded semantics of program P.

Note that the well-founded semantics computed in Example 3.6 coincides with the Fitting semantics computed in Example 3.4. Of course, this is not the case in general, since the Fitting semantics even deviates from the standard semantics of Datalog (without negation), as was mentioned in Section 3.1.3

In (Przymusinski, 1990), an alternative way of defining the well-founded semantics is presented. It is via an extension of stable models to *3-valued stable* models, which in turn is based on an extension of Datalog programs to "3-extended Datalog programs" and of the ICO T_P to the 3-valued ICO 3-T_P:

Definition 3.6. A *3-extended Datalog program* is a Datalog program (i.e., no negation), where the truth constants `false`, $\bot$, `true` are allowed to occur in rule bodies.

Given a 3-extended Datalog program P, the *3-valued immediate consequence operator* 3-T_P is defined as follows: 3-$T_P(I) = J$ such that, for every ground atom $A \in HB(P, D)$, we have:

- $J(A) =$ `true` if P contains a rule with head atom A, s.t. the body evaluates to `true` in I.
- $J(A) =$ `false` if for every rule with head atom A in P the body evaluates to `false` in I.
- $J(A) = \bot$ otherwise.

The 3-valued ICO 3-T_P is very similar to Fitting's ICO Φ_P discussed in Section 3.1.3. The main difference is that we apply 3-T_P only to 3-extended Datalog programs. That is, these programs no longer contain negation and, therefore, 3-T_P is now also monotone w.r.t. $\leq_t$. Hence, the lfp (w.r.t. $\leq_t$) of 3-T_P exists. It is now possible to extend the Gelfond-Lifschitz reduct and stable models to partial interpretations.

Definition 3.7. Given a Datalog$^\neg$ program P and partial interpretation I, the *positivized ground program of P given I*, denoted $pg(P, I)$, is the 3-extended Datalog program obtained from $ground(P, DB)$ by replacing every negative literal in the body of every rule r in $ground(P, DB)$ by its truth value in I. Moreover, I is a *3-stable model* of P, if it is the unique minimal 3-valued model of $pg(P, I)$, i.e., it is the lfp of 3-$T_{pg(P,I)}$.

Clearly, if I is a complete interpretation, then the introduction of truth constants `false` and `true` in $pg(P, I)$ has the same effect as the Gelfond-Lifschitz reduct. In particular, in this case, the truth constant $\bot$ does not occur in $pg(P, I)$ and also the definition of 3-stable models corresponds to stable models in the Gelfond-Lifschitz sense.

A Datalog$^\neg$ program P can have many 3-stable models. It is shown in (Przymusinski, 1990) that at least one 3-stable model exists and that the well-founded semantics of P coincides with the smallest 3-stable model w.r.t. $\leq_k$, which can be obtained as the intersection of all 3-stable models of P.

In (Gelder, 1989), yet a further way of defining and computing the well-founded model is presented. It works via "alternating fixpoints". Throughout the computation, this approach deals with complete interpretations. Only at the very end, when the two alternating fixpoints have been obtained (which are complete interpretations), `undef` truth values are introduced by computing the meet w.r.t. $\leq_k$ of the two alternating fixpoints.

Definition 3.8. Given a (ground) Datalog$^\neg$ program P, we define the operator $conseq_P$ on complete interpretations as follows: let P' denote the Gelfond-Lifschitz reduct of P w.r.t. I. Then $conseq_P(I)$ is the lfp (w.r.t. $\leq_t$) of $T_{P'}$.

We can now use $conseq_P$ to define an *alternating sequence* of (complete) interpretations $(I_n)_{n\geq 0}$ as follows:

- Let I_0 be the minimal interpretation w.r.t. $\leq_t$, i.e., all ground atoms are set to `false`.

- For $n \geq 0$, set $I_{n+1} = conseq_P(I_n)$.

We ultimately want to compute the positive and negative facts that can be derived by P. Intuitively, I_0 overestimates the negative facts (and hence, the truth values of negative literals in P). Therefore, I_1 overestimates the positive facts. But then, I_2 is again an overestimation of the negative facts, etc. More formally, the operation $conseq_P$ is anti-monotone, i.e., if $I \leq_t J$ then $conseq_P(J) \leq_t conseq_P(I)$. Together with the fact that I_0 is defined as the smallest model w.r.t. $\leq_t$, we thus get:

$$I_0 \leq_t I_2 \leq_t \cdots \leq_t I_{2i} \leq_t \cdots \leq_t I_{2i+1} \leq_t I_{2i-1} \leq_t \cdots \leq_t I_3 \leq_t I_1$$

That is, the even sequence of interpretations is increasing while the odd sequence is decreasing. Hence, there exist limits I_* of the even sequence and I^* of the odd sequence with the following properties: $I_* \leq_t I^*$ and $conseq_P(I_*) = I^*$ and $conseq_P(I^*) = I_*$. In general, I_* and I^* are distinct. Gelder, 1989 refers to them as "alternating fixpoints". Moreover, he defines the "*alternating fixpoint partial model*" (of P) as

$$\begin{aligned} I_*^* = &\{A \in HB(P,D) \mid I_*(A) = I^*(A) = \texttt{true}\} \cup \\ &\{\neg A \mid A \in HB(P,D) \text{ and } I_*(A) = I^*(A) = \texttt{false}\} \end{aligned}$$

The main result in (Gelder, 1989) is the following:

Theorem 3.4. Let P be a (ground) Datalog$^\neg$ program. The alternating fixpoint partial model of P is identical to the well-founded semantics of P.

We revisit the win-move game once again and compute the well-founded semantics by applying the alternating fixpoint computation.

Example 3.6. Recall the win-move game from Examples 3.4 and 3.6 and again apply the program to the (cyclic) graph in Figure 3.2. Moreover, recall from Example 3.6, that after grounding and some simplifications,

we get the following Datalog$^\neg$ program:

$$\begin{aligned}
\mathsf{Win(a)} &\text{ :- } \neg\,\mathsf{Win(b)}\\
\mathsf{Win(a)} &\text{ :- } \neg\,\mathsf{Win(c)}\\
\mathsf{Win(b)} &\text{ :- } \neg\,\mathsf{Win(a)}\\
\mathsf{Win(c)} &\text{ :- } \neg\,\mathsf{Win(d)}\\
\mathsf{Win(c)} &\text{ :- } \neg\,\mathsf{Win(e)}\\
\mathsf{Win(d)} &\text{ :- } \neg\,\mathsf{Win(e)}\\
\mathsf{Win(e)} &\text{ :- } \neg\,\mathsf{Win(f)}
\end{aligned}$$

The alternating fixpoint computation of Gelder, 1989 yields the following sequence of IDB instances. As in Example 3.4, we abbreviate Win to W and we write 0 for `false` and 1 for `true`:

	$W(a)$	$W(b)$	$W(c)$	$W(d)$	$W(e)$	$W(f)$	
$I^{(0)} =$	**0**	**0**	**0**	**0**	**0**	**0**	
$I^{(1)} =$	**1**	**1**	**1**	**1**	**1**	**0**	
$I^{(2)} =$	**0**	**0**	**0**	**0**	**1**	**0**	
$I^{(3)} =$	**1**	**1**	**1**	**0**	**1**	**0**	
$I^{(4)} =$	**0**	**0**	**1**	**0**	**1**	**0**	
$I^{(5)} =$	**1**	**1**	**1**	**0**	**1**	**0**	$= I^{(3)}$
$I^{(6)} =$	**0**	**0**	**1**	**0**	**1**	**0**	$= I^{(4)}$

That is, we get the sequence of inclusions $I^{(0)} \subseteq I^{(2)} \subseteq I^{(4)} \subseteq \ldots \subseteq I^{(5)} \subseteq I^{(3)} \subseteq I^{(1)}$. Moreover, $I* = I^{(3)}$ and $I_* = I^{(4)}$ are the alternating fixpoints with $conseq_P(I_*) = I^*$ and $conseq_P(I^*) = I_*$. Note that, from $I^{(3)}$ on, the truth values of Win(c), Win(e), Win(d), Win(f) stabilize, while the truth values of Win(a), Win(b) keep switching between `true` and `false` as we continue to apply the $conseq_P$-operator. Hence, the alternating fixpoint partial model I^*_* is obtained as $I^*_* = \{\mathsf{Win}(c), \mathsf{Win}(e), \neg\mathsf{Win}(d), \neg\mathsf{Win}(f)\}$. As expected, it is identical to the well-founded model that we computed in Example 3.6 via the operator W_P.

3.1.5 Fitting Semantics vs. Well-Founded Semantics

Fitting, 1993 presented a connection between his semantics based on the Φ_P operator and the well-founded semantics. To be precise, it is a connection between the extension of his semantics to the four-valued logic FOUR and multi-valued *stable models* in general.

When dealing with four-valued interpretations, it is convenient to use the mapping-notation for interpretations. That is, we use valuations

v that assign one of the 4 truth values $\{\bot, \texttt{false}, \texttt{true}, \top\}$ in FOUR to every ground atom. Recall from Section 3.1.3 and, in particular, Figure 3.3, that FOUR forms a bilattice with the truth order $\leq_t$ and the knowledge order $\leq_k$. In the bilattice FOUR, we use $\wedge, \vee$ and $\otimes, \oplus$, to denote the meet and join w.r.t. the $\leq_t$ and $\leq_k$ order, respectively. The operators $\wedge, \vee, \otimes, \oplus$ are extended to valuations by pointwise application to the values of ground atoms, e.g., $(v_1 \oplus v_2)(A) = v_1(A) \oplus v_2(A)$.

Throughout this section, we restrict ourselves to grounded programs and we assume that all rules with the same (ground) head atom are combined to a single rule by using disjunction in the rule body. Given a program P, an interpretation v is a model of P if every rule $A \leftarrow \mathit{body}$ in P satisfies $v(\mathit{body}) \leq_t v(A)$. The ICO Φ_P introduced in Section 3.1.3 for 3-valued logic is readily extended to 4-valued logic: we set $\Phi_P(v) = w$, such that for every ground atom A we have either (1) $w(A) = \texttt{false}$, if P contains no rule with head A, or we have (2) $w(A) = v(\mathit{body})$ otherwise, where $A \leftarrow \mathit{body}$ is the unique rule with this head atom.

Recall from Sections 3.1.1 and 3.1.4 the idea of using a separate interpretation for the negative literals in the Gelfond-Lifschitz reduct and also in Przymusinki's definition of the well-founded semantics via 3-valued stable models. Analogously, Fitting, 1993 introduces the notion of "pseudo-valuations" as a way of treating unnegated and negated atoms differently. Given two (4-valued) valuations v_1 and v_2, the pseudo-valuation $v_1 \Delta v_2$ is defined as

$$v_1 \Delta v_2(A) = v_1(A) \quad \text{and} \quad v_1 \Delta v_2(\neg A) = \neg v_2(A)$$

for arbitrary (ground) atom A. The combined valuation $(v_1 \Delta v_2)$ is extended to arbitrary ground formulas by interpreting $\wedge$ and $\vee$ in the usual way. Then the *extended ICO* Ψ_P is defined as follows:

Definition 3.9. For arbitrary (ground) Datalog$^\neg$ program P and valuations v_1, v_2, define $\Psi_P(v_1, v_2) = w$, such that

- if P contains no rule with head atom A, then set $w(A) = \texttt{false}$;
- otherwise, let $A \leftarrow \mathit{body}$ be the unique rule with head atom A in P. (Recall that we are assuming that all rules with the same head atom are combined to a single rule.) In this case, we set $w(A) = v_1 \Delta v_2(\mathit{body})$.

That is, Ψ_P extends Φ_P by treating positive and negative information differently. Clearly, without this distinction of positive and negative information, we are back to Φ_P. That is, $\Psi_P(v, v) = \Phi(v)$. The extended ICO Ψ_P has the following crucial monotonicity properties:

Theorem 3.5. For arbitrary (ground) Datalog$^\neg$ program P, we have:

1. $\Psi_P(v, w)$ is monotone in both arguments v and w w.r.t. $\leq_k$;
2. $\Psi_P(v, w)$ is monotone in the first argument v w.r.t. $\leq_t$;
3. $\Psi_P(v, w)$ is anti-monotone in the second argument w w.r.t. $\leq_t$, i.e., for arbitrary v, if $w_1 \leq_t w_2$, then $\Psi_P(v, w_2) \leq_t \Psi_P(v, w_1)$.

By the monotonicity of $\Psi_P(v, w)$ in the first argument v w.r.t. $\leq_t$, the following notion of *stability operator* Ψ'_P is well-defined:

Definition 3.10. For arbitrary (ground) Datalog$^\neg$ program P and valuation w, we define Ψ'_P as follows:

$$\Psi'_P(w) = \text{ the lfp, with respect to } \leq_t \text{ of } (\lambda v)\Psi_P(v, w)$$

The monotonicity of $\Psi_P(v, w)$ in the second argument w.r.t. $\leq_k$ carries over to $\Psi'_P(w)$. Hence, Ψ'_P has fixpoints, which are called *4-valued stable interpretations* of P. Fitting, 1993 shows that every fixpoint of Ψ'_P is a model of P. It is therefore justified to refer to the fixpoints of Ψ'_P as *4-valued stable models*. The proof is straightforward by combining several simple facts: Suppose that s is a fixpoint of Ψ'_P, i.e., (1) $\Psi'_P(s) = s$. By definition of Ψ', we have that $\Psi'_P(s)$ is a fixpoint of $(\lambda v)\Psi_P(v, s)$, i.e., (2) $\Psi'_P(s) = \Psi_P(\Psi'_P(s), s)$. And, finally, by definition of Ψ_P, we have (3) $\Psi_P(s, s) = \Phi_P(s)$. In order to show that a fixpoint s of Ψ'_P is a model of P it suffices to show that s is also a fixpoint of Φ_P. Indeed, this holds by the following chain of equalities:

$$\Phi_P(s) \overset{(3)}{=} \Psi_P(s, s) \overset{(1)}{=} \Psi_P(\Psi'_P(s), s) \overset{(2)}{=} \Psi'_P(s) \overset{(1)}{=} s.$$

In principle, Ψ'_P is defined on 4-valued interpretations. If a fixpoint of Ψ'_P is 2-valued (i.e., it never takes on $\bot$ or $\top$), then it is a stable model in the Gelfond-Lifschitz sense (cf. Section 3.1.2). Likewise, if a fixpoint of Ψ'_P is 3-valued (i.e., it never takes on $\top$), then it is a

3-valued stable model according to Przymusinski's approach to defining the well-founded semantics (see Section 3.1.4). We will come back to this point below. The advantage of using the bilattice FOUR is that there always exists a least fixpoint (lfp) *and* a greatest fixpoint (gfp) both, w.r.t. $\leq_t$ and $\leq_k$. This is in contrast to Fitting's 3-valued approach recalled in Section 3.1.3, where the existence of a gfp w.r.t. $\leq_k$ is not guaranteed.

We now also make use of the complete-lattice structure w.r.t. $\leq_t$. As was mentioned above, the monotonicity of $\Psi_P(v, w)$ in the second argument w.r.t. $\leq_k$ carries over to $\Psi'_P(w)$. The same holds true for the anti-monotonicity w.r.t. $\leq_t$. On the other hand, carrying out an anti-monontone operation f twice gives a monontone operation f^2. Using the Knaster-Tarski theorem, it is straightforward to conclude that an anti-monotone operation f in a complete lattice has a unique pair of extreme oscillation points (cf. (Fitting, 1993), Theorem 9.5). That is, there exist a, b with $a \leq b$, $f(a) = b$, $f(b) = a$ and, for every pair x, y with $f(x) = y$, $f(y) = x$, we have $a \leq x, y \leq b$. This allows us to establish the following relationship between $\leq_t$ and $\leq_k$ in the context of Ψ'.

Theorem 3.6. For arbitary (ground) Datalog$^\neg$ program P, define the following fixpoints of Ψ'_P:

- let $s^k_P = \mathsf{lfp}$ of Ψ' w.r.t. $\leq_k$ and $S^k_P = \mathsf{gfp}$ of Ψ' w.r.t. $\leq_k$;
- let s^t_P and S^t_P with $s^t_P \leq_t S^t_P$ be the two extreme oscillation points of Ψ' w.r.t. $\leq_t$.

Then the following relationships hold:

- $s^k_P = s^t_P \otimes S^t_P$ and $S^k_P = s^t_P \oplus S^t_P$
- $s^t_P = s^k_P \wedge S^k_P$ and $S^t_P = s^k_P \vee S^k_P$

It is easy to verify that the 4-valued interpretations s^k_P, S^k_P, s^t_P, S^t_P in Theorem 3.6 constitute boundaries of the space of 4-valued stable models in the sense that every 4-valued stable model M of P satisfies

- $s^k_P \leq_k M \leq_k S^k_P$ and

- $s_P^t \leq_t M \leq_t = S_P^t$.

The inequality $s_P^k \leq_k M$ for every 4-valued stable model M, of course, also holds for every 3-valued stable model M (being a special case of a 4-valued stable model). Moreover, the construction of the well-founded model via 3-stable models proposed by Przymusinski, 1990 implicitly guarantees the existence of 3-valued stable models. Hence, in order to satisfy $s_P^k \leq_k M$ for every 4-stable model M, s_P^k is actually 3-valued. In other words, obtaining s_P^k as the lfp of Ψ' w.r.t. $\leq_k$ corresponds to the construction of the well-founded model according to Przymusinski, 1990. Alternatively, s_P^k can be obtained as $s_P^k = s_P^t \otimes S_P^t$ from the two extreme oscillation points of Ψ' w.r.t. $\leq_t$, which corresponds to the construction of the well-founded model via alternating fixpoints proposed by Gelder, 1989.

3.1.6 Further Relationships between various Semantics

We have seen in Section 3.1.3 that the Fitting semantics may differ even on pure Datalog from the semantics via the $\mathsf{lfp}(T_P)$. The question arises, how the other semantics of Datalog$^\neg$ relate to each other when restrictions are imposed on negation or when negation is omitted altogether. The following result from (Przymusinska and Przymusinski, 1988) answers this question:

Theorem 3.7. For weakly stratified programs, the (unique) weakly perfect model is also the well-founded model and the unique stable model. Moreover, for Datalog programs, they all coincide with $\mathsf{lfp}(T_P)$.

Note that the procedural definition of the weakly perfect model in (Przymusinska and Przymusinski, 1988) can also be applied to programs that are not weakly stratified. In this case, the procedure described in Section 3.1.1 fails because at some stage, the remaining program P_i does not have a bottom layer without negation. We then define the *partial weakly perfect model* by taking the model M_{i-1} from the last successful iteration and assign $\bot$ to all atoms in P_i. In (Hitzler and Wendt, 2005), the following relationship between different partial models is shown (see (Hitzler and Wendt, 2005), Corollary 6.7):

Theorem 3.8. Let P be a normal logic program with Fitting model M_F, weakly perfect model M_{WP}, and well-founded model M_{WF}. Then $M_F \subseteq M_{WP} \subseteq M_{WF}$ holds. (As in Section 3.1.3, we are assuming the notation of partial interpretations by sets of ground literals, i.e., the $\subseteq$-order corresponds to the knowledge order $\leq_k$).

Moreover, (Hitzler and Wendt, 2005) give an example where these inclusions are strict (see (Hitzler and Wendt, 2005), Examples 6.3 and 6.7):

Example 3.7. Let program P be defined as

$$P = \{a \leftarrow \neg b; \quad b \leftarrow c, \neg a; \quad b \leftarrow c, \neg d; \quad c \leftarrow b, \neg e; \quad d \leftarrow e; \quad e \leftarrow d\}$$

Then we get $M_F = \emptyset$, $M_{WP} = \{\neg d, \neg e\}$, and $M_{WF} = \{a, \neg b, \neg c, \neg d, \neg e\}$. A key difference between the Fitting semantics and the other two is how the subset of rules $\{d \leftarrow e; \quad e \leftarrow d\}$ is treated. In the Fitting semantics, d and e are assigned the truth value $\bot$. For the other two semantics, this subset of rules is a prototypical case of an unfounded set. There, both atoms are set to `false`. More generally, $M_F = \emptyset$ is easy to see: since there are no facts in the program or an edb, the rule bodies will always keep the truth value $\bot$, i.e., no rule will ever assign a value different from $\bot$ to any atom.

For M_{WF}, we observe that $\{d, e\}$ is an unfounded set. Hence, we may set these atoms to `false`. For this assignment, now also $\{b, c\}$ is an unfounded set. Hence, we also set these two atoms to `false`. But then the first rule fires and we may set a to `true`.

It remains to consider M_{WP}: stratification of program P yields as bottom layer the atoms $\{d, e\}$. The rules $\{d \leftarrow e; \ e \leftarrow d\}$ yield the model $M_1 = \{\neg d, \neg e\}$. Now we can use these truth values of d and e to reduce P and we get the program $P' = \{a \leftarrow \neg b; \quad b \leftarrow c, \neg a; \quad b \leftarrow c; \quad c \leftarrow b\}$. Here, stratification does not allow us to get rid of negation in the bottom layer. Hence, the process stops with $M_{WP} = M_1 = \{\neg d, \neg e\}$.

3.1.7 A Unifying Approach via Level Mappings

In this and the next section, we revisit two unifying approaches to defining various semantics of Datalog$^\neg$ or, more generally, of logic programs with negation. Hitzler and Wendt, 2005 propose such a unifying

approach via so-called *level mappings*, i.e., assigning natural numbers to (a subset of) the ground atoms in the Herbrand Base. Then, by imposing varying restrictions on the level mappings, different semantics of Datalog and Datalog$^\neg$ can be captured. Below we illustrate this approach for Datalog programs (i.e., no negation) and for Datralog$^\neg$ programs with stable model semantics. The other semantics will only be briefly mentioned.

In the sequel, it is convenient to consider an empty edb. That is, all ground atoms A of the edb are turned into rules $A \leftarrow$ with empty body and are thus considered part of the program. Then, for the minimal models of Datalog programs, we get the following characterization in terms of level mappings. Intuitively, the level mapping reflects the order in which ground atoms are derived by the ICO.

Theorem 3.9. Let P be a Datalog program. Then the minimal model $T_P^\omega(\emptyset)$ of P can be characterized as the unique model M of P for which there exists a level mapping $l\colon HB(P,D) \to \mathbb{N}$ such that for every atom $A \in M$ there exists a clause $A \leftarrow A_1, \ldots, A_n$ in $ground(P)$ with $A_i \in M$ and $l(A) > l(A_i)$ for every $i \in \{1, \ldots, n\}$.

Proof. First, let M be the minimal model $T_P^\omega(\emptyset)$ of P. We define a level mapping l be setting $l(A) = \min\{k \mid A \in T_P^{k+1}(\emptyset)\}$ for all $A \in M$ and choosing an arbitrary value $l(A)$ for all $A \notin M$. It remains to show that this level mapping has the desired property.

Let $A \in M$. That is, A is eventually derived by the ICO T_P. Hence, there exists k, such that $A \notin T_P^k(\emptyset)$ and $A \in T_P^{k+1}(\emptyset)$ (it is convenient to set $T_P^0(\emptyset) = \emptyset$). By our definition of the level mapping, we thus have $l(A) = k$. On the other hand, since A is derived by the $(k+1)$-st iteration of the ICO, there exists a clause $A \leftarrow A_1, \ldots, A_n$ in $ground(P)$ with $A_i \in T_P^k(\emptyset)$ (hence also $A_i \in M$) for every i. Moreover, by our definition of the level mapping, $l(A_i) < k$ and, therefore, $l(A_i) < l(A)$ holds. That is, the level mapping l satisfies the desired property according to the theorem.

Conversely, suppose that M is a model of P for which such a level mapping exists. We show that then $M = T_P^\omega(\emptyset)$ holds, which also proves the uniqueness of M. We clearly have $T_P^\omega(\emptyset) \subseteq M$, since $T_P^\omega(\emptyset)$ is the minimal model of P. It remains to show the opposite inclusion

$T_P^\omega(\emptyset) \supseteq M$. The proof is by induction on the level $l(A)$ for atoms $A \in M$:

First consider an atom A with minimum level $l(A) = \min$. The property of the level mapping means that A must be the head of a ground clause in $ground(P)$ with empty body. Hence, A is derived by the ICO T_P in the first iteration and, therefore, $A \in T_P^1(\emptyset) \subseteq T_P^\omega(\emptyset)$ clearly holds.

For the induction step, consider an atom A with $l(A) = i > \min$ and, as induction hypothesis, suppose that, for all $B \in M$ with $l(B) < l(A)$, we have $B \in T_P^\omega(\emptyset)$. The property of M means that there exists a clause $A \leftarrow A_1, \ldots, A_n$ in $ground(P)$ with $A_i \in M$ and $l(A) > l(A_i)$ for every $i \in \{1, \ldots, n\}$. By the induction hypothesis, we conclude that $A_i \in T_P^\omega(\emptyset)$ for every $i \in \{1, \ldots, n\}$. Hence, by yet another implication of the ICO T_P, also $A \in T_P^\omega(\emptyset)$ holds. □

This idea is readily extended to stable models next.

Theorem 3.10. Let P be a Datalog$^\neg$ program. Then a two-valued model M of P is a stable model of P if and only there exists a level mapping $l\colon HB(P, D) \to \mathbb{N}$ such that for every atom $A \in M$ there exists a clause $A \leftarrow A_1, \ldots, A_n, \neg B_1, \ldots, \neg B_m$ in $ground(P)$ with $A_i \in M$, $B_j \notin M$ and $l(A) > l(A_i)$ for every $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$.

Remark. Note that the theorem does not impose any restrictions on the levels of the negated atoms $B_1, \ldots B_m$ occurring in a rule body.

Proof. First, let M be a stable model of P, i.e., M is the minimal model of the Gelfond-Lifschitz reduct P^M. In particular, M is also a model of P. Moreover, P^M is a definite program, i.e., without negation. Hence, we may apply Theorem 3.9 to P^M and conclude that there exists a level mapping $l\colon HB(P^M, D) \to \mathbb{N}$, such that for every atom $A \in M$ there exists a clause $A \leftarrow A_1, \ldots, A_n$ in P^M with $A_i \in M$ and $l(A) > l(A_i)$ for every $i \in \{1, \ldots, n\}$. Moreover, the existence of clause $A \leftarrow A_1, \ldots, A_n$ in P^M implies that there exists a clause $A \leftarrow A_1, \ldots, A_n, \neg B_1, \ldots, \neg B_m$ in $ground(P)$ with $B_j \notin M$ for every $j \in \{1, \ldots, n\}$. Hence, we get the desired level mapping $l\colon HB(P, D) \to \mathbb{N}$ according to the theorem by extending l to the atoms in $HB(P, D) \setminus HB(P^M, D)$ (i.e., ground atoms that must be `false` in M) in an arbitrary way.

Conversely, suppose that M is a model of P for which such a level mapping exists. Consider an arbitrary ground atom A with $A \in M$. Then, there exists a clause $A \leftarrow A_1, \ldots, A_n, \neg B_1, \ldots, \neg B_m$ in $ground(P)$ with $A_i \in M$, $B_j \notin M$ and $l(A) > l(A_i)$ for every $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$. In particular, this also means that there is a clause $A \leftarrow A_1, \ldots, A_n$ in P^M with $A_i \in M$ and $l(A) > l(A_i)$ for every $i \in \{1, \ldots, n\}$. By Theorem 3.9 we may conclude that $M = T^{\omega}_{P^M}(\emptyset)$ holds. That is, M is a stable model. □

Fitting models, well-founded models, and weakly stratified models require quite involved conditions on the level mappings to make this approach work. We only mention the intuition of the level-mapping-based approach for these semantics here.

For Fitting's semantics and for the well-founded semantics, we have to consider 3-valued interpretations. Recall from Section 3.1.3, that the Fitting model is defined as the least (w.r.t. the knowledge ordering $\leq_k$) fixpoint of the ICO Φ_P. Hitzler and Wendt, 2005 give an alternative definition of the Fitting model in terms of a level mapping which reflects for both, positive atoms and negated atoms, the number of iterations of the ICO Φ_P needed to derive them. Similarly, recall from Section 3.1.4 that the well-founded semantics is defined as the least fixpoint of the ICO W_P. Then the well-founded model can be defined in terms of a level mapping which reflects for both, positive atoms and negated atoms, the number of iterations of the ICO W_P needed to derive them. Finally, Hitzler and Wendt, 2005 also give a characterization of the weakly perfect model of a weakly stratified Datalog$^{\neg}$ program in terms of a level mapping that is chosen consistently with the weak stratification and, again, the ordering $l(A) < l(B)$ reflects the order in which the truth values of the ground atoms A and B is determined by the layerwise program evaluation sketched in Section 3.1.1.

We conclude this section by briefly mentioning that level mappings have also been used for a completely different purpose. In his PhD thesis, Hitzler, 2001 studied conditions under which logic programs with negation allow for a unique supported model. Recall that supported models (originally studied by Apt *et al.*, 1988) are simply the fixpoints of the ICO. Of course, in case of Datalog$^{\neg}$, neither the existence nor the

uniqueness of a fixpoint is guaranteed. Hitzler, 2001 therefore revisits several alternative fixpoint theorems such as the Banach contraction mapping theorem which is stated as follows:

Theorem 3.11. Let (X, d) be a complete metric space, let $0 \leq \lambda < 1$ and let $f\colon X \to X$ be a function which is a *contraction* with *contractivity factor* λ, i.e., it satisfies $d(f(x), f(y)) \leq \lambda d(x, y)$ for all $x, y \in X$ with $x \neq y$. Then f has a unique fixpoint which can be obtained as the limit of the sequence $(f^n(x))$ for any $x \in X$.

In order to make this and other fixpoint theorems applicable to logic programs, we thus need do define a metric on interpretations of programs. This can be achieved via level mappings: let $l\colon HB(P, D) \to \mathbb{N}$ be a level mapping and let $\mathcal{L}_\alpha$ denote the ground atoms of $HB(P, D)$ on level α. Then, for two interpretations I, J over $HB(P, D)$, we can define

$$d(I, J) = \begin{cases} \inf(\{2^{-\alpha} | I \cap \mathcal{L}_\alpha \neq J \cap \mathcal{L}_\alpha\}) & \text{if } I \neq J \\ 0 & \text{if } I = J \end{cases}$$

That is, the distance between two mappings is smaller the later in the level mapping there is a discrepancy between the mappings. It is easy to verify that the above defined d is even an ultrametric, i.e., it satisfies the strong triangle inequality $d(x, z) \leq \max(d(x, y), d(y, z))$ for all elements x, y, z of the metric space. Indeed, let $I, J, K \subseteq HB(P, D)$ with $d(I, J) = 2^{-\alpha}$ and $d(I, J) = 2^{-\beta}$ (the case that one of the distances is 0 and, hence, two of the sets are identical, is trivial). W.l.o.g., let $\alpha \leq \beta$. That is, I and J coincide on all atoms up to level $\alpha - 1$. By $\alpha \leq \beta$, also J and K coincide on all atoms up to level $\alpha - 1$. Hence, also I and K coincide on all atoms up to level $\alpha - 1$, i.e., $d(I, K) \leq 2^{-\alpha} = \max(\{d(I, J), d(J, K)\})$.

Now suppose that a program P satisfies the condition that there exists a level mapping $l\colon HB(P, D) \to \mathbb{N}$, such that for each clause $A \leftarrow L_1, \ldots, L_n$ in ground(P) and for all $i \in \{1, \ldots, n\}$ we have $l(A) \geq l(L_i)$. Then P is called *acyclic*. In this case, T_P is actually a contraction mapping w.r.t. the metric d defined above with contractivity factor 0.5. To see this, let $I, J \subseteq HB(P, D)$ with $d(I, J) \leq 2^{-\alpha}$ and consider an arbitrary atom $A \in T_P(I)$ with $l(A) \leq \alpha$. That is, there exists a rule $A \leftarrow L_1, \ldots, L_n$ in $ground(P)$ such that L_i is `true` in I and

$l(L_i) < l(A)$. But I and J coincide on all ground atoms up to level $\alpha - 1$. Hence, this rule also fires on J and we have $A \in T_P(J)$. By symmetry, also every atom $A \in T_P(J)$ with $l(A) \leq \alpha$ is contained in $T_P(I)$. Hence, $d(T_P(I), T_P(J)) \leq 2^{-(\alpha+1)}$, i.e., $d(T_P(I), T_P(I)) \leq d(I, J)/2$.

By the Banach fixpoint theorem, we may therefore conclude that, for acyclic P, the ICO T_P has a unique fixpoint (i.e., a unique supported model), which can be reached by iteration of T_P. Hitzler, 2001 studies several variants of metrics (ultrametrics, quasi-metrics, pseudometrics, etc.) together with further fixpoint theorems (Prieß-Crampe and Ribenboim, Matthews, Rutten-Smyth) that guarantee the existence of a unique fixpoint under certain conditions. These results are then applied to the ICO T_P of logic programs P with negation by imposing appropriate restrictions on the (level mappings for) programs as was illustrated by the simplest case of acyclic programs recalled here.

3.1.8 A Unifying Approach via Assumptions on Predicates

Liu and Stoller, 2020 introduce two new semantics of Datalog$^\neg$: the 3-valued founded semantics and the 2-valued constraint semantics, which capture (and even generalize) the major semantics of Datalog$^\neg$ programs recalled in Sections 3.1.1 – 3.1.4. The new semantics are motivated by the observation that, in general, different assumptions may apply to different predicates. For instance, an application may have full information on some predicates but not on others. More specifically, the following assumptions may be applied to the predicates in a program.

- *certain* vs. *uncertain*: a predicate can be certain (= 2-valued) or uncertain (= 3-valued). For a certain predicate, everything that is `true` must be given or inferred; the rest is `false`. In an uncertain predicate, everything that is `true` again has to be given or inferred as such; the rest is `false` or `undef` depending on further assumptions mentioned next. A predicate may only be declared certain, if (in the dependency graph) it neither depends on its own negation nor on any uncertain predicate.

- *complete* or not: an uncertain predicate may be declared as complete. In this case, completion rules are added to explicitly infer

the truth value `false`. Hence, only those facts for which we can neither derive `true` nor `false` are set to `undef`. This can be further refined by the next assumption.

- *closed* or not: an uncertain and complete predicate may be declared as closed. Similarly to unfounded sets in the well-founded semantics, a predicate is assumed to be `false` if inferring it requires itself to be `true`. In other words, the set of facts with `undef` truth value is further decreased.

We now present the definition of the founded and constraint semantics without closed predicates. The *founded semantics* of a program P (again we consider the edb as part of the program) is defined via the following steps, which we illustrate by revisiting the win-move game from Example 3.3. Actually, Liu and Stoller, 2020 make quantification explicit and, as we will see below, they also allow universal quantification. The only rule with non-empty body in case of the win-move game becomes

$$\mathsf{Win}(\mathsf{X}) \text{ :- } \exists \mathsf{Y}\ \mathsf{Edge}(\mathsf{X},\mathsf{Y}), \neg\, \mathsf{Win}(\mathsf{Y})$$

In addition, the program contains rules of the form $\mathsf{Edge}(\mathsf{a},\mathsf{b})$:- to encode the edges (a, b), etc. Let us assume that the Edge predicate is defined as certain and the Move predicate is uncertain and complete but not closed.

Step 1. Completion. For all predicates that are defined as uncertain and complete, we add completion rules to the program. This is done by first combining all rules with the same (uncertain and complete) head predicate into a single rule (the head atoms are made identical and the bodies of different rules are combined by disjunction). Then, for each rule $A \leftarrow body$, the "negation" $\neg A \leftarrow \neg body$ is added. In case of the above rule of the win-move game, we thus add the completion rule

$$\neg\, \mathsf{Win}(\mathsf{X}) \text{ :- } \forall \mathsf{Y} \neg\, \mathsf{Edge}(\mathsf{X},\mathsf{Y}) \vee\ \mathsf{Win}(\mathsf{Y})$$

Step 2. Elimination of negation by introducing new predicates. Every literal of the form $\neg P(X)$ is replaced by $n.P(X)$, where $n.P$ is a new predicate that stands for the negation of predicate P. Now the win-move program contains the following two rules:

$$\mathsf{Win}(\mathsf{X}) \text{ :- } \exists \mathsf{Y}\ \mathsf{Edge}(\mathsf{X},\mathsf{Y}), \mathsf{n}.\mathsf{Win}(\mathsf{Y})$$
$$\mathsf{n}.\mathsf{Win}(\mathsf{X}) \text{ :- } \forall \mathsf{Y}\ \mathsf{n}.\mathsf{Edge}(\mathsf{X},\mathsf{Y}) \vee\ \mathsf{Win}(\mathsf{Y})$$

Step 3. Fixpoint computation per SCC. After the previous step, negation has been completely removed and the program is monotone. Note that this also applies to the second rule in the above example, which contains universal quantification. However, during the process of program evaluation, the program (including the single-atom rules encoding the edb) and the active domain are fixed. Hence, the rule only fires if indeed all ground instantiations of the body have been derived as `true`.

We can now determine the strongly connected components in the dependency graph and stratify the program in this dependency order (arranging incomparable SCCs in arbitrary order). Hence, analogously to stratified programs discussed in Section 3.1.1, we can now process the strata in this order and compute the least fixpoint of the ICO for each stratum. To this fixpoint, for each certain predicate, negated facts $\neg A$ are added whenever A could not be derived.

Step 4. The final step consists in renaming predicates $n.P$ back to $\neg P$. In case of the win-move game, we thus derive precisely the same truth values of the Win-atoms as for the Fitting semantics (see Example 3.4) and the well-founded semantics (see Example 3.6).

The *constraint semantics* defines *constraint models* M of a program P as follows: M is a 2-valued model of P extended by the completion rules, such that M is obtained from the founded model by trying out all possible ways of replacing the truth value `undef` by `true` or `false`.

Closed predicates (that is uncertain, complete and closed) are integrated into the founded semantics as follows: analogously to the well-founded semantics (see Section 3.1.4), after computing the founded semantics M of a program as described above, the greatest unfounded set U w.r.t. M is determined for those predicates which are defined as uncertain, complete, and closed, i.e., for every fact in U, the grounded rules with a head atom from U have a body that evaluates to `false` in M or that contains an atom from U. All facts in U are set to `false` and the 4 steps for computing the founded semantics of the program plus the negative literals $\{\neg A \mid A \in U\}$ is started again, resulting in a new model M of the program. This process of fixpoint iteration followed by determining the greatest unfounded set U w.r.t. M is iterated analogously to the computation of the well-founded semantics. The resulting (unique) model is referred to as the *founded-closed* semantics of a program. The models of the *constraint-closed* semantics are again obtained from the *founded-closed* semantics by trying out all possible combinations of `true` and `false` for the ground atoms with `undef` truth value and checking that we indeed get a model M of the program plus completion rules, such that all atoms in the greatest unfounded set w.r.t. M are set to `false`.

Liu and Stoller, 2020 obtain the following relationship of the new semantics with previous ones:

- If in a stratified program all predicates are defined as certain, then the perfect model semantics coincides with both the founded and the constraint semantics.
- If all edb predicates are defined as certain and all idb predicates are defined as uncertain and complete but not closed, then the founded semantics coincides with Fitting's 3-valued semantics.
- If predicates are defined as certain whenever possible and all other predicates are defined as uncertain, complete and closed, then the founded-closed semantics coincides with the well-founded semantics and the constraint-closed semantics coincides with the stable-model semantics.

3.2 Adding Aggregation

The plan is very extensive. I think it is worth it, since our entire paper is about adding numerical computations to datalog. The challenge is to distill the key ideas, without making this section too long. (I am not familiar with any of the references below.) –Dan.

We now turn our attention to Datalog extended by aggregates, referred to as Datalog^{agg} in the sequel. Unless stated otherwise, Datalog^{agg} also includes the extension by negation. In fact, negation can be considered as a special case of aggregation, since we can express a negated atom of the form "$\neg \mathsf{P}(\mathsf{X})$" as the comparison atom[1] "$\mathsf{count}\{\mathsf{X}\colon \mathsf{P}(\mathsf{X})\} = 0$". There is a huge body of literature on aggregation in logic programming and Datalog. Similarly to the extension of Datalog with negation, the introduction of aggregation poses non-trivial challenges concerning the semantics. In principle, the approaches to incorporating aggregates are quite similar to the incorporation of negation: they either stick to the usual fixpoint iteration with the ICO of the program and require some form of stratification or monotonicity to guarantee a unique fixpoint. Or they adapt more powerful semantics such as stable model semantics or the 3-valued Fitting's and well-founded semantics. However, it turns out that the situation of aggregation is yet more complex than with negation for the following reasons:

- As we will see in this section, the main approaches from Datalog$^{\neg}$ also apply to Datalog with aggregates (definite programs, stratification, Fitting's 3-valued semantics, well-founded semantics, stable model semantics). However, additional sub-classes of programs through various

[1] We are using here the terminology and syntax of aggregates proposed by Liu and Stoller, 2022. We will formally introduce this syntax in Section 3.2.1.

restrictions on programs are defined so as to guarantee termination of the fixpoint iteration. In particular, the delineation between monotone and non-monotone programs gets more nuanced in the presence of aggregates (see Section 3.2.2).

- Also inside each main approach, several variations are possible, e.g.: for stable model semantics of programs with negation, the definition of a reduct is generally agreed. In contrast, for programs with aggregates, several definitions of a reduct (and hence, of stable models) have been proposed in the literature (see Section 3.2.5).
- In case of Datalog$^{\neg}$, differences concerning the notation of rules are negligible. In contrast, for aggregate atoms, a wide variety of notations exists in the literature. There have been attempts of unifying approaches (see e.g., recent works by Liu and Stoller, 2022 and Alviano *et al.*, 2023) – including a unifying notation. However, in our review of various approaches, we will inevitably have to deviate from such a uniform notation.

In this section, we shall therefore proceed in "reverse order" compared with Section 3.1 and start with an extension of the unifying approach of Liu and Stoller, 2020 for programs with negation to programs with aggregates proposed by Liu and Stoller, 2022. This will allow us to provide an overview of the various approaches to incorporating aggregates into logic programs or Datalog.

3.2.1 A Unifying Approach via Assumptions on Predicates

I find Liu and Stoller's framework in 3.2.1 very complicated. –Dan.

I could only make sense of this 2022 paper after reading the 2020 paper by the same authors. So my question is: is Section 3.2.1 understandable after reading Section 3.1.8? If yes, then maybe it is acceptable that Section 3.2.1 is difficult to read by itself? If no, then Section 3.2.1 probably has to be re-thought. At any rate, I have extended some explanations a bit hoping that, together with Section 3.1.8, things are clearer now. –Reinhard.

Recall from Section 3.1.8, that Liu and Stoller, 2020 introduced a formalism that allowed them to capture the principal approaches to handling negation in logic programs. This was achieved by making different assumptions on predicates: (i) declaring them as certain vs. uncertain, (ii) declaring uncertain predicates as complete or not and (iii) declaring uncertain and complete

predicates as closed or not. This formalism is extended by Liu and Stoller, 2022 to programs with aggregates.

To this end, Liu and Stoller, 2022 first of all extend the syntax of programs by allowing *comparison atoms* of the form $aggS \odot k$, where $aggS$ is an *aggregation*, $\odot$ is a comparison operator, and k is either a variable or a numeric constant. An aggregation $aggS$ consists of an *aggregation operator* $agg \in \{\texttt{count}, \texttt{sum}, \texttt{min}, \texttt{max}\}$ and a set expression S of the form $\{X_1, \ldots, X_n : B\}$, where the body B has the same form as a rule body and each X_i is a variable in B. The comparison operator can be an equality $=$ or an inequality from $\{\neq, <, \leq, >, \geq\}$. We have already used this syntax above when presenting the comparison atom $\mathsf{count}\{\mathsf{X} : \mathsf{P}(\mathsf{X})\} = 0$ as a way of expressing negation via aggregates.

A famous example of logic programs or Datalog with aggregation is the Company Control problem, which has been studied since the earliest works on this subject (see Kemp and Stuckey, 1991; Mumick *et al.*, 1990; Ross and Sagiv, 1992; Gelder, 1992). In the above described syntax, it looks as follows:

Example 3.8. Consider edb predicates $\mathsf{company}(\mathsf{C})$ denoting that C is a company and $\mathsf{ownsStk}(\mathsf{C1}, \mathsf{C2}, \mathsf{P})$ denoting that company $\mathsf{C1}$ owns percentage P of shares of company $\mathsf{C2}$. We want to define a predicate $\mathsf{controls}(\mathsf{C1}, \mathsf{C2})$ that indicates that company $\mathsf{C1}$ controls company $\mathsf{C2}$. By this we mean that the percentage of shares of $\mathsf{C2}$ owned either directly by $\mathsf{C1}$ or indirectly via companies controlled by $\mathsf{C1}$ is more than 50. The following program defines the target predicate $\mathsf{controls}(\mathsf{C1}, \mathsf{C2})$ and the auxiliary predicate $\mathsf{controlsStk}(\mathsf{C1}, \mathsf{C2}, \mathsf{C3}, \mathsf{P})$ denoting that company $\mathsf{C1}$ controls P percent of company $\mathsf{C3}$ via company $\mathsf{C2}$:

$$
\begin{aligned}
\mathsf{controlsStk}(\mathsf{C1}, \mathsf{C2}, \mathsf{C2}, \mathsf{P}) &\text{ :- } \mathsf{owns}(\mathsf{C1}, \mathsf{C2}, \mathsf{P}) \\
\mathsf{controlsStk}(\mathsf{C1}, \mathsf{C2}, \mathsf{C3}, \mathsf{P}) &\text{ :- } \mathsf{company}(\mathsf{C1}) \wedge \mathsf{controls}(\mathsf{C1}, \mathsf{C2}) \wedge \mathsf{owns}(\mathsf{C2}, \mathsf{C3}, \mathsf{P}) \\
\mathsf{controls}(\mathsf{C1}, \mathsf{C3}) &\text{ :- } \mathsf{company}(\mathsf{C1}) \wedge \mathsf{company}(\mathsf{C3}) \wedge \\
&\qquad \mathsf{sum}\{\mathsf{P}, \mathsf{C2} : \mathsf{controlsStk}(\mathsf{C1}, \mathsf{C2}, \mathsf{C3}, \mathsf{P})\} > 50
\end{aligned}
$$

We will give a formal definition of the semantics of comparison atoms below. Intuitively, the $\mathsf{company}$-atoms in the third rule provide bindings of the variables $\mathsf{C1}$ and $\mathsf{C3}$. The set expression is then evaluated by first determining all pairs $(\mathsf{P}, \mathsf{C2})$ for which the body $\mathsf{controlsStk}(\mathsf{C1}, \mathsf{C2}, \mathsf{C3}, \mathsf{P})$ (for the current bindings of $\mathsf{C1}$ and $\mathsf{C3}$) evaluates to `true` followed by computing the sum of the P-values for these pairs.

The company control example is very nice, but it is unclear why one needs the "undefined" value, and why we need to declare the predicates as certain/uncertain, or complete/incomplete. I understand that these are needed for negation, but Company Control doesn't have negation. I'm confused why Liu and Stoller use undefined/certain/uncertain/complete/incomplete for programs with aggregates. My guess is that programs with aggregates but without negation don't need them, while programs without aggregates and with negation use them in the same way as you explained in 3.1.8. –Dan.

In general, aggregates can also introduce non-monotonicity. So the entire theory of certain/uncertain, complete/incomplete is also applicable here. I have therefore added below that the Company Control example is particularly simple because here we indeed have monotonocity: new derivations of controls-atoms and controlsStk facts can only lead to further derivations but they never invalidate a previous derivation. However, in general, aggregates can have non-monotone behavior (as in the above example of expressing negation via aggregates). –Reinhard.

Note that the Company Control program is particularly simple in the sense that all predicates behave in a monotone way. That is, deriving new atoms will never invalidate previous derivations. In particular, this is also true for the comparison atom in the body of the last rule if we assume that the attribute P can only take non-negative values. However, in general, aggregates can have non-monotone behavior (as in the above example of expressing negation via aggregates). This distinction between monotone and non-monotone behavior of predicates plays an important role for the definition of the semantics of Datalog^{agg}. And it also restricts the assumptions we are allowed to make on the predicates, e.g., only monotone predicates are allowed to be assumed "certain"; all others must be assumed "uncertain". We will provide more details on the semantics definition for Datalog^{agg} by Liu and Stoller, 2022 and on the evaluation of the Company Control program below.

Above all, we have to provide a formal definition of the semantics of a comparison atom $aggS \odot k$, where S is a set expression of the form $\{\boldsymbol{X} : B\}$ with $\boldsymbol{X} = X_1, \ldots, X_n$. Let $G(S)$ denote the set of ground instances $(\boldsymbol{X}\sigma, B\sigma)$ obtained by instantiating the variables occurring in S with any constants via the ground substitution σ. For a given (3-valued) interpretation I and truth value $t \in \{\texttt{true}, \texttt{false}, \texttt{undef}\}$, we define the set $G(S, I, t)$ of combinations of constants for which the body B of S evaluates to t. Formally, we thus have

$$G(S, I, t) = \{\boldsymbol{X}\sigma \mid (\boldsymbol{X}\sigma, B\sigma) \in G(S) \text{ and } B\sigma \text{ has truth value } t \text{ in } I\}$$

An aggregation operator $agg \in \{\texttt{count}, \texttt{sum}, \texttt{min}, \texttt{max}\}$ is applied to the set $G(S, I, t)$ in the natural way: `min`, `max` determine the minimum and maximum, respectively, of the tuples in $G(S, I, t)$ provided that $\boldsymbol{X}$ consists of a single variable and its values are numerical[2]. `count` returns the cardinality of $G(S, I, t)$ and `sum` sums up the first components of the tuples in $G(S, I, t)$, provided that they are numerical values.

Now the question is, when can a comparison atom $aggS \odot k$ be derived in an interpretation I of a program π, denoted $\pi, I \vdash aggS \odot k$. Here, in addition to $G(S, I, \texttt{true})$ also $G(S, I, \texttt{undef})$ may have to be taken into account. Intuitively, $\pi, I \vdash aggS \odot k$ holds if $aggS \odot k$ is `true` regardless of whether atoms with truth value `undef` are set to `true` or `false`. For instance, $\pi, I \vdash \texttt{count}\, S > k$ holds iff $|G(S, I, \texttt{true})| > k$. That is, tuples in $G(S, I, \texttt{undef})$ can make the count only bigger and can, therefore, not destroy the $>$ relationship. However, for $=$ and $<$ we have

$$\pi, I \vdash \texttt{count}\, S = k \Leftrightarrow |G(S, I, \texttt{true})| = k \wedge G(S, I, \texttt{undef}) = \emptyset$$
$$\pi, I \vdash \texttt{count}\, S < k \Leftrightarrow |G(S, I, \texttt{true}) \cup G(S, I, \texttt{undef})| < k$$

Similarly, $\pi, I \vdash \texttt{max}\, S > k$ iff there exists $i \in G(S, I, \texttt{true})$ with $i > k$. In contrast, for $=$ and $<$, we have to take $G(S, I, \texttt{undef})$ into account. Actually, in case of `max` (likewise `min`), the inequality $\neq$ requires special treatment, since a comparison atom with $\neq$ is not simply the negation of the corresponding comparison atom with $=$.

$$\begin{aligned}
\pi, I \vdash \texttt{max}\, S = k \Leftrightarrow\ & k \in G(S, I, \texttt{true}) \wedge \\
& \forall i \in G(S, I, \texttt{true}) \cup G(S, I, \texttt{undef}) \colon i \leq k \\
\pi, I \vdash \texttt{max}\, S \neq k \Leftrightarrow\ & k \notin G(S, I, \texttt{true}) \cup G(S, I, \texttt{undef}) \vee \\
& \exists i \in G(S, I, \texttt{true}) \colon i > k \\
\pi, I \vdash \texttt{max}\, S < k \Leftrightarrow\ & \exists i \in G(S, I, \texttt{true}) \wedge \\
& \forall i \in G(S, I, \texttt{true}) \cup G(S, I, \texttt{undef}) \colon i < k
\end{aligned}$$

Aggregates make the distinction between monotonicity and non-monotonicity significantly more complex. For instance, in the Company Control example recalled above, the comparison atom $\texttt{sum}\{\ldots\} > 50$ is apparently monotone w.r.t. the atom $\mathsf{controls}(\mathsf{C1}, \mathsf{C2}, \mathsf{C3}, \mathsf{P})$, provided that all summands are guaranteed to be non-negative. This would not be the case if we changed the comparison atom to $\texttt{sum}\{\ldots\} < k$. Liu and Stoller, 2022 define various (sufficient) conditions that ensure that a comparison atom is monotone w.r.t. some predicate atom A, i.e., A is of the form $P(\cdot)$. More specifically, this is the

[2] As is mentioned by Liu and Stoller, 2022, it is straightforward to extend this definition to non-numerical types and also to tuples of values by considering lexicographical ordering.

case if A is a positive literal in the set expression S of a comparison literal $\texttt{count}\, S \geq k$, $\texttt{count}\, S > k$, $\texttt{max}\, S \geq k$, $\texttt{max}\, S > k$, $\texttt{min}\, S \leq k$, $\texttt{min}\, S < k$. Likewise, this is the case if A is a negative literal in the set expression S of a comparison literal $\texttt{count}\, S \leq k$, $\texttt{count}\, S < k$, $\texttt{max}\, S \leq k$, $\texttt{max}\, S < k$, $\texttt{min}\, S \geq k$, $\texttt{min}\, S > k$. If any of these conditions is satisfied, A is said to occur positively in the comparison atom. More generally, whenever a comparison atom is monotone w.r.t. a predicate atom A, we say that A occurs positively, as was the case with $\mathsf{controls}(\mathsf{C1}, \mathsf{C2}, \mathsf{C3}, \mathsf{P})$ in the above mentioned example in case of non-negative summands.

We can now adapt the definition of the dependency graph so as to take positive and not positive occurrences of predicate atoms into account: for predicates Q and R occurring in a program P, (1) there is a positive edge from Q to R, if P contains a rule with head predicate Q and R is the predicate of an atom occurring positively in the body; and (2) there is a negative edge from Q to R, if P contains a rule with head predicate Q and R is the predicate of an atom with a non-positive occurrence in the body. We say that a predicate Q has a *circular non-positive dependency* in P, if the dependency graph contains a cycle through Q with a non-positive edge. As we will see next, circular non-positive dependencies have to be taken into account when making assumptions on predicates.

In principle, the declaration of predicates as certain, uncertain, complete, and closed now works analogously to programs with negation discussed in Section 3.1.8. However, the restriction as to when a predicate may be declared as certain now hinges on the adapted definition of the dependency graph. That is, if a predicate has a circular non-positive dependency, then it must be declared as uncertain. Other than that, the definition of certain, uncertain, complete, and closed predicates works as in Section 3.1.8: A certain predicate can only take truth values `true` and `false`, whereas an uncertain predicate can also take truth value `undef`. If an uncertain predicate is defined as complete, then we add completion rules to the program. And if an uncertain, complete predicate is defined as closed then an atom with this predicate is set to `false` if its derivation by the given rules and facts would require to assume itself to be `true`.

Also the formal definition of the founded and constraint semantics is, in principle, analogous to programs with negation discussed in Section 3.1.8 - with some additional complications, though, which we will briefly mention below.

- First, the *founded semantics* of a program with aggregates (in the form of comparison atoms) is defined via the same 4 steps as in case of programs with negation in Section 3.1.8, that is: (1) *Completion* (adding completion rules for uncertain and complete predicates), (2) *Monotone ICO* (by introducing new predicates $n.P(\cdot)$ to positively

represent $\neg P(\cdot)$), (3) *Fixpoint computation* (which again proceeds SCC-wise along a dependency order), and (4) *Renaming back* of predicates (i.e., replacing $n.P$ by $\neg P$).

- The models of the constraint semantics of a program are again obtained by trying out all possible combinations of `true` and `false` for the ground atoms with truth value `undef`.
- Finally, also closed predicates are taken into account by an appropriate definition of unfounded sets (referred to as "self-false" by Liu and Stoller, 2022) to define the founded-closed and constraint-closed semantics.

As mentioned above, some steps become a bit more involved when allowing aggregates compared with negation. For instance, Step 2 ("Monotone ICO") has to take into account that, in a comparison atom, a positive atom may in fact occur non-positively and a negated atom may occur positively. Hence, if $\neg P(\cdot)$ occurs non-positively, it makes sense to replace it by $n.P(\cdot)$, but a positive occurrence of a negated atom is left unchanged. And we even have to replace literals of the form $P(\cdot)$ by $\neg n.P(\cdot)$ if $P(\cdot)$ occurs non-positively inside a comparison atom. At any rate, in principle, the philosophy underlying the possible assumptions on predicates and of the various steps summarized above should be the same as in case of Datalog$^\neg$.

Let us revisit now the Company Control program from Example 3.8. We have already mentioned, that all atoms occur positively in this program. It is therefore allowed (and, for this application, also intuitive) to assume all predicates to be "certain".

Example 3.9. Consider again the Company Control program from Example 3.8 and suppose that the edb contains the following facts: {owns(a, b, 60), owns(a, c, 15), owns(a, d, 25), owns(b, c, 40), owns(c, d, 30)}.

Recall that we are assuming all predicates to be "cetain". The fixpoint computation of the ICO restricted to the idb predicates controlsStk and controls allows us to derive the following facts. The completion rules and completion facts play no role for the derivation of controlsStk(C1, C2, C3, P) and controls(C1, C2) facts. They are therefore omitted.

The first application of the ICO yields the facts controlsStk(a, b, b, 60), controlsStk(a, c, c, 15), controlsStk(a, d, d, 25), controlsStk(b, c, c, 40), controlsStk(c, d, d, 30). After that, exactly one additional fact is derived by each application of the ICO. That is, the following facts are derived in this order: controls(a, b), controlsStk(a, b, c, 40), controls(a, c), controlsStk(a, c, d, 30), controls(a, d).

Similarly to programs with negation treated in (Liu and Stoller, 2020), Liu and Stoller, 2022 state that also for programs with negation, the founded

and constraint semantics capture several previously defined semantics for Datalog^{agg}:

- If in a stratified program all predicates are defined as certain, then the founded semantics captures the extension of the stratified semantics to Datalog^{agg}, which was originally studied by Mumick *et al.*, 1990. We will discuss this one and several further approaches to defining an intuitive semantics of Datalog^{agg} via stratification and/or monotonicity in Section 3.2.2.

- If all edb predicates are defined as certain and all idb predicates are defined as uncertain and complete, then the founded semantics coincides with the extension of Fitting's 3-valued semantics to Datalog^{agg}, which was originally studied by Pelov *et al.*, 2007. We will look into this approach in Section 3.2.3.

- If predicates are defined as certain whenever possible and all other predicates are defined as uncertain, complete and closed, then the founded-closed semantics yields an extension of the well-founded semantics to Datalog^{agg}, which was originally studied by Kemp and Stuckey, 1991 and Gelder, 1992 and later also by Pelov *et al.*, 2007. We will discuss these approaches in Section 3.2.4.

- If predicates are defined as certain whenever possible and all other predicates are defined as uncertain, complete and closed, then the constraint-closed semantics yields an extension of the stable model semantics to Datalog^{agg}. There have been several extensions of stable model semantics to Datalog^{agg}, e.g., by Kemp and Stuckey, 1991; Denecker *et al.*, 2001; Marek and Remmel, 2004; Pelov *et al.*, 2007; Liu *et al.*, 2010; Ferraris, 2011; Faber *et al.*, 2011; Gelfond and Zhang, 2019. We will discuss several of these approaches in Section 3.2.5.

3.2.2 Stratified Aggregation and Monotonicity

In this section, we recall some of the most basic approaches of incorporating aggregates into logic programs or Datalog. The common theme of these approaches is that they aim at some form of monotonicity of the ICO in order to guarantee the existence of a least fixpoint. One way of reaching this goal is to extend the idea of stratification from Datalog$^{\neg}$ to Datalog^{agg} and to guarantee monotonicity of the ICO in each stratum. This and other ways of achieving monotonicity are the topic of this section.

Aggregate stratification

In a pioneering work, Mumick *et al.*, 1990 have investigated Datalog^{agg} (or, more generally, logic programs with aggregates) from the point of view of stratification and monotonicity. Syntactically, they allow the use of aggregates in rule bodies by introducing *group_by subgoals*. These are body literals of the form $group\_by(R(\bar{t}), GL, AL)$, where $R(\bar{t})$ is the "goal" (i.e., an atom with predicate symbol R and attribute list $\bar{t}$), $GL = [Y_1, \ldots, Y_m]$ is the "grouping list" (i.e., a list of variables occurring in $\bar{t}$, which are used for grouping), and AL is the "aggregation list". That is, AL is essentially (with some minor extensions) a list of equalities of the form $Z_i = A_i(E_i)$, where A_i is an aggregate function and E_i are variables occurring in $\bar{t}$. For instance, the last rule in the Company Control program in Example 3.8 would look as follows:

$$\begin{aligned}\mathsf{controls(C1, C3)} \text{ :- } & \mathsf{company(C1)}, \mathsf{company(C3)},\\ & \mathsf{group\_by(controlsStk(C1, C2, C3, P), [C1, C3], [A = SUM(P)]},\\ & \mathsf{A} > 50\end{aligned}$$

There is overlap in the syntax between Mumick's group_by(R, GL, A) and Liu and Stoller's G(S,I,true); we could use a unified syntax. –Dan.

I am hesitating to translate this and the next 2 examples from Mumick's syntax to the syntax of Liu and Stoller. I admit, the translation is not too difficult. Nevertheless it seems that some care is required when translating also the definitions of "aggregate stratified" and "group stratified" so as to really get an equivalent definition when using the other syntax. Also statements like "following program given by Mumick et al." are not precisely correct anymore. Sure, all this can be mitigated. But I would prefer not to do it. –Reinhard.

A rule in a Datalog^{agg} program is defined as *monotone* if adding new tuples to the relations of predicates occurring in the rule body can only add tuples to the head (i.e., it cannot invalidate a previous derivation). A program is monotone if all rules are. In this case, the usual fixpoint iteration of the ICO yields the desired minimal model also in the presence of aggregates. However, Mumick *et al.*, 1990 observe that, in general, a group_by subgoal introduces non-monotonicity. So the goal is to identify conditions under which an intuitive minimal model can be computed by fixpoint iteration of the ICO.

The first type of programs thus obtained are *aggregate stratified* programs. They are defined via the following extension of the dependency graph: a

predicate P depends on predicate Q "through a grouping operation" if the program contains a rule with head predicate P and a group_by subgoal in the body where Q is the predicate of the goal. If this is the case, we draw an edge from P to Q that is marked as such. A Datalog^{agg} program is *aggregation stratified* if it contains no cycle with an edge representing a dependency through a grouping operation. Then stratification and the perfect model can be defined for such a program analogously to stratification in case of Datalog$^{\neg}$ (see Section 3.1.1). One of the most common examples in the literature on logic programs with aggregates is the All-Pairs-Shortest-Path (APSP) program. In the syntax proposed by Mumick *et al.*, 1990, it looks as follows:

Example 3.10. Consider an edb predicate $\mathsf{arc}(\mathsf{X},\mathsf{Y},\mathsf{C})$ indicating that there is an arc from X to Y of length C. Then the following program defines the idb predicates $\mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{C})$ and $\mathsf{spath}(\mathsf{X},\mathsf{Y},\mathsf{C})$ indicating that there is a path from X to Y of length C and that the shortest path from X to Y has length C.

$$\begin{aligned}\mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{C}) &\text{ :- } \mathsf{arc}(\mathsf{X},\mathsf{Y},\mathsf{C}).\\ \mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{C}) &\text{ :- } \mathsf{path}(\mathsf{X},\mathsf{Z},\mathsf{Cxz}),\mathsf{arc}(\mathsf{Z},\mathsf{Y},\mathsf{Czy}),\mathsf{C}=\mathsf{Cxz}+\mathsf{Czy}\\ \mathsf{spath}(\mathsf{X},\mathsf{Y},\mathsf{C}) &\text{ :- } \mathsf{group\_by}(\mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{Cxy}),[\mathsf{X},\mathsf{Y}],\mathsf{C}=\mathsf{MIN}(\mathsf{Cxy})).\end{aligned}$$

Clearly, the predicate spath depends on path via aggregation. Hence, the dependency graph contains an edge from path to spath labeled as "through a grouping operation". However, the dependency graph does not contain a cycle involving this edge; so the program is aggregate stratified.

Intuitively, aggregate stratification requires that a predicate must not be defined by grouping over itself. This restriction can be relaxed in the sense that a group (that is, a particular instantiation of the group_by-variables) must not depend on itself. Programs satisfying this condition are called *group stratified* by Mumick *et al.*, 1990. Then group stratification and the perfect model of such programs can be defined analogously to weak stratification in case of Datalog$^{\neg}$ (see Section 3.1.1). The following program given by Mumick *et al.*, 1990 (which is a variant of the so-called "Bill-Of-Material" program), computes the number of units of a particular subpart contained directly or indirectly in some part. Of course, a subpart C can be (indirectly) contained in a part A via different subpart-paths (i.e., if the subpart-relationship is not a tree but a DAG). In this case, the program is supposed to sum up the occurrences of C from the different subpart-paths. Later in Section 3.2, we will encounter variants of the "Bill-Of-Material" program with a slightly different behavior.

I'm also confused by Example 3.10, since I don't know what it's supposed to compute. If the same sub-sub-...-sub-part occurs on multiple paths, do we want to count it repeatedly? Or just once? The program seems to count it repeatedly, for example if the DAG is Subpart(A,B1), Subpart(A,B2), Subpart(B1,C), Subpart(B2,C) then A contains 2 copies of C. But in that case I don't see the need the predicate contains(P,S,U,C), i.e. a single rule suffices count_contains(P,S,C) = groupby(subpart(P,U,C1), count_contains(P,U,C2), C=sum(C1*C2)). –Dan.

Ad semantics of the program: I have added above an explanation what the program is supposed to do. As you mention, it should count subparts on different subpart-paths multiple times. Ad simplification to a single rule: Above all, the syntax of Mumick et al. requires that the relation over which we aggregate is defined by a single atom. So we need the auxiliary relation contains(P,S,U,C) for the join of the two atoms subpart(P,U,C1), count_contains(P,U,C2). This means one additional rule. And then I think that the base case is missing in the one-rule formulation. Whence, another additional rule. That's why we end up here with 3 rules. –Reinhard.

Example 3.11. Consider an edb predicate subpart(P, S, C) indicating that a part P contains C units of subpart S. Then the following program defines the idb predicates contains(P, S, U, C) and count_contains(P, S, C). Here contains(P, S, U, C) means that P contains C units of subpart S via direct subpart U (that is U is a direct subpart of P and S is a direct or indirect subpart of U). If no such intermediate subpart U exists, then this is indicated by the constant "null" which is not allowed to occur anywhere in the database. By count_contains(P, S, C) we express that P contains in total, directly or indirectly, C units of S.

should "null" below be S? –Dan.

According to Mumick et al., "null" is okay. I have extended the explanation above. –Reinhard.

$$\begin{aligned}
\mathsf{contains}(P, S, \mathsf{null}, C) &:\text{-} \ \mathsf{subpart}(P, S, C).\\
\mathsf{contains}(P, S, U, C) &:\text{-} \ \mathsf{subpart}(P, U, C1),\\
&\quad\ \mathsf{count_contains}(U, S, C2), C = C1 * C2.\\
\mathsf{count_contains}(P, S, C) &:\text{-} \ \mathsf{group_by}(\\
&\quad\ \mathsf{contains}(P, S, U, M), [P, S], C = \mathsf{SUM}(M)]).
\end{aligned}$$

Does this program compute anything different from the standard bill of material? Or does it compute the same output, but in a more complicated way in order to illustrate stratification? –DAN.

As mentioned above, I think this version is slightly more "intelligent" than our usual Bill-of-Material program: normally, we just sum up the costs of the subparts without taking the number of units of each subpart into account. Here, we are not interested in costs but we correctly compute the number of units of each subpart contained in another part. And because of the syntax defined by Mumick, I don't think that the program is unnecessarily complicated. But you are right, in the syntax of Liu und Stoller, we could contract the second and third rule to a single rule. Do you think we should mention this? Or (as in a comment further above), do you think we have to present everything in this section in the syntax of Liu and Stoller anyway? –REINHARD.

The program is not aggregation stratified since the dependency graph contains the cycle $\mathsf{contains} \to \mathsf{count\_contains} \to \mathsf{contains}$. On the other hand, it is easy to verify that the program is group stratified, provided that the subpart relation is acyclic. Indeed, a dependency $\mathsf{contains}(a, \_, \_, \_) \to \mathsf{count\_contains}(c, \_, \_)$ only holds if c is, directly or indirectly, a subpart of a. And a dependency $\mathsf{count\_contains}(a, \_, \_) \to \mathsf{contains}(c, \_, \_, \_)$ only holds if $a = c$ holds. Hence, for an acyclic subpart relation, there can never be a path from $\mathsf{contains}(a, \_, \_, \_)$ back to $\mathsf{contains}(a, \_, \_, \_)$.

It is also noted by Mumick *et al.*, 1990 that the Company Control program (which is actually formulated slightly differently than in Example 3.8 and referred to as "Company Takeover" program by Mumick *et al.*, 1990) is, in general, not even group stratified. Indeed, a cyclic dependency containing group_by subgoals with the same grouping values could arise, if for some values a, b, c, the following dependencies hold: $\mathsf{controls}(a, b) \to \mathsf{controlsStk}(a, c, b, \_) \to \mathsf{controls}(a, c) \to \mathsf{controlsStk}(a, b, c, \_) \to \mathsf{controls}(a, b)$. Nevertheless, the

program is clearly monotone (i.e., the derivation of new tuples never invalidates the derivation of previously derived tuples) and has an intuitive minimal model, which can be computed by the usual fixpoint iteration of the ICO. The idea of exploiting monotonicity to get an intuitive semantics of programs with aggregation is significantly extended in the approach of Ross and Sagiv, 1992, which we will discuss next.

Interpreting programs over a complete lattice

Ross and Sagiv, 1992 propose an approach that aims at making a program monotone by considering the order in a complete lattice rather than the order by set inclusion of Herbrand interpretations. Syntactically, aggregates may occur as *aggregate subgoals* of the following form

$$C = \mathcal{F}\, D : p(X_1, \ldots, X_n, Y_1, \ldots, Y_m, D),$$

where C is a variable, $\mathcal{F}$ is a function that maps non-empty multisets over D to values in some ordered set R, and $p(X_1, \ldots, X_n, Y_1, \ldots, Y_m, D)$ is a "cost atom" with grouping variables $X_1, \ldots, X_n$, local variables $Y_1, \ldots, Y_m$, and "cost argument" D. Note that the variables do not necessarily have to be arranged in this order. The grouping variables are those which occur outside the aggregate subgoal while the local variables only occur inside the aggregate subgoal. The following example will help to illustrate these definitions.

Example 3.12. Consider an edb predicate $\mathsf{record}(\mathsf{S}, \mathsf{C}, \mathsf{G})$ indicating that a student S has scored grade G (expressed as a percentage) in course C. Then the idb predicates $\mathsf{s-avg}(\mathsf{S}, \mathsf{G})$, $\mathsf{c-avg}(\mathsf{C}, \mathsf{G})$, and $\mathsf{all-avg}(\mathsf{G})$, indicating the average grades of each student, the average grades of each course, and the overall average grades, respectively, can be defined by the following rules:

$$\begin{aligned}
\mathsf{s-avg}(\mathsf{S}, \mathsf{G}) &\text{ :- } \mathsf{G} = \mathsf{averageG}' : \mathsf{record}(\mathsf{S}, \mathsf{C}, \mathsf{G}') \\
\mathsf{c-avg}(\mathsf{C}, \mathsf{G}) &\text{ :- } \mathsf{G} = \mathsf{averageG}' : \mathsf{record}(\mathsf{S}, \mathsf{C}, \mathsf{G}') \\
\mathsf{all-avg}(\mathsf{G}) &\text{ :- } \mathsf{G} = \mathsf{averageG}' : \mathsf{record}(\mathsf{S}, \mathsf{C}, \mathsf{G}')
\end{aligned}$$

Ross and Sagiv, 1992 consider a wide range of aggregate functions, including min, max, sum, product, and count. In case of count, the cost argument in the final position of the aggregate atom can be omitted.

Now Ross and Sagiv, 1992 assume that the *domain* D and the *range* R are equipped with partial orders $\sqsubseteq_D$ and $\sqsubseteq_R$. The partial order $\sqsubseteq_D$ is extended to multisets on D by defining $I \sqsubseteq_D I'$ if there is an injective map m from elements of I to elements of I' such that $i \sqsubseteq_D m(i)$ holds for all $i \in I$. Moreover, the

aggregate function $\mathcal{F}$ is requested to be monotone in the sense that $I \sqsubseteq_D I'$ implies $\mathcal{F}(I) \sqsubseteq_R \mathcal{F}(I')$ for any two non-empty multisets I, I' over D. Finally, $\langle R, \sqsubseteq_R \rangle$ is requested to be a complete lattice. Intuitively, the $\sqsubseteq$-relationship holds between two multisets if the greater one is obtained from the smaller one either by adding a new element or by increasing an old element.

A Herbrand interpretation I for a program P with aggregates is a subset of the Herbrand base of P such that no two atoms in I differ only on the cost-argument and such that interpreted predicates (such as $<$, $+$, $=$) are given their standard interpretation. Note that Ross and Sagiv, 1992 assume that the cost arguments are functionally dependent on the other arguments. The definition of Herbrand interpretations ensures this. A Herbrand interpretation I is a *Herbrand model* of P if every rule in P is satisfied by I.

For ground cost atoms $A = P(a_1, \ldots, a_n, c)$ and $A' = P(a'_1, \ldots, a'_n, c')$ we define $P(a_1, \ldots, a_n, c) \sqsubseteq P(a'_1, \ldots, a'_n, c')$ iff $a_i = a'_i$ for every i and $c \sqsubseteq c'$. In case the ground atoms A, A' have no cost argument, we define $A \sqsubseteq A'$ iff $A = A'$. We can then also define an order on Herbrand interpretations J_1, J_2, by defining $J_1 \sqsubseteq J_2$ if for every atom $A_1 \in J_1$ there exists an atom $A_2 \in J_2$ with $A_1 \sqsubseteq A_2$. One can show that, if $\sqsubseteq$ is the partial order of a complete lattice R then also the set of Herbrand interpretations forms a complete lattice w.r.t. $\sqsubseteq$.

Ross and Sagiv, 1992 define the semantics of a program P by considering one component of mutually recursive predicates at a time and proceeding along a component order. For a given program component, let I be an interpretation for the predicates not appearing in the heads of rules in P (i.e., these predicates are defined in a lower component) and let J be an interpretation for the predicates appearing in the heads of rules in P. Then the ICO T_P for fixed I is defined as

$$T_P(J, I) = \{A \colon A \leftarrow B_1, \ldots, B_n \text{ is a ground instance of a rule in } P \text{ with } I \cup J \models B_1 \wedge \cdots \wedge B_n\}$$

We say that a program P is *monotone* if $T_P(J, I)$ is monotone in J for every fixed I. In this case, the least fixpoint M_I^P of $T_P(\cdot, I)$ for arbitrary but fixed interpretation I exists and M_I^P is a model of P. Consider a component order $P_1, \ldots, P_m$ of a monotone program P and set $M_0 = \emptyset$. Then we compute a sequence of interpretations $M_1, \ldots, M_m$ as $M_i = M_{I_{i-1}}^{P_i}$ for every $i \in \{1, \ldots, m\}$ and $M = M_m$ is the desired model of P.

The following examples from Ross and Sagiv, 1992 illustrate the appropriateness of the above definitions. Recall the All-Pairs-Shortest-Path (APSP) problem from Example 3.10. Ross and Sagiv, 1992 give a slightly different program for this problem:

Example 3.13. Consider an edb predicate $\mathsf{arc}(\mathsf{X},\mathsf{Y},\mathsf{C})$ denoting that there exists an arc from X to Y of weight C. Then the following program defines the idb predicates $\mathsf{path}(\mathsf{X},\mathsf{Z},\mathsf{Y},\mathsf{C})$ and $\mathsf{spath}(\mathsf{X},\mathsf{Y},\mathsf{C})$. Here $\mathsf{path}(\mathsf{X},\mathsf{Z},\mathsf{Y},\mathsf{C})$ indicates that there is a path from X to Y of weight C such that Z is the node immediately before Y on this path. This slightly non-standard definition of the path predicate is chosen to make sure that the cost (in the last argument) is functionally dependent on the other arguments. Actually, it would also be useful if one wishes to construct the actual shortest paths. By $\mathsf{spath}(\mathsf{X},\mathsf{Y},\mathsf{C})$ we denote that the shortest path from X to Y has weight C. Assume that direct is a constant not occurring anywhere in the database (analogously to the constant null used by Mumick *et al.*, 1990). Then the two idb predicates can be defined as follows:

$$\begin{aligned}\mathsf{path}(\mathsf{X},\mathsf{direct},\mathsf{Y},\mathsf{C}) &\text{ :- } \mathsf{arc}(\mathsf{X},\mathsf{Y},\mathsf{C})\\ \mathsf{path}(\mathsf{X},\mathsf{Z},\mathsf{Y},\mathsf{C}) &\text{ :- } \mathsf{spath}(\mathsf{X},\mathsf{Z},\mathsf{C1}),\mathsf{arc}(\mathsf{Z},\mathsf{Y},\mathsf{C2}),\mathsf{C}=\mathsf{C1}+\mathsf{C2}\\ \mathsf{spath}(\mathsf{X},\mathsf{Y},\mathsf{C}) &\text{ :- } \mathsf{C}=\mathsf{min}\ \mathsf{D}:\mathsf{path}(\mathsf{X},\mathsf{Z},\mathsf{Y},\mathsf{D})\end{aligned}$$

As domain $\langle D,\sqsubseteq_D\rangle$ and range $\langle R,\sqsubseteq_R\rangle$ we can, for instance, choose $\langle \mathbb{N}\cup\{\infty\},\geq\rangle$ or $\langle \mathbb{R}^+\cup\{\infty\},\geq\rangle$ or even $\langle \mathbb{R}\cup\{\pm\infty\},\geq\rangle$. However, in the latter case, the lfp of the ICO can be reached by finitely many iterations of the ICO only if the underlying graph contains no cycles with negative weight.

Another common example in the literature on Datalog^{agg} is the Party Invitations program. Below we recall the variant of Party Invitations studied by Ross and Sagiv, 1992.

Example 3.14. Consider edb predicates $\mathsf{knows}(\mathsf{X},\mathsf{Y})$ and $\mathsf{requires}(\mathsf{X},\mathsf{K})$, where the former indicates that X knows Y and the latter indicates that X only attends the party if at least K persons X knows will come to the party. The program below defines an idb predicate $\mathsf{coming}(\mathsf{X})$ indicating that X will come and and an idb predicate $\mathsf{kc}(\mathsf{X},\mathsf{Y})$ indicating that X knows Y and Y will come.

$$\begin{aligned}\mathsf{coming}(\mathsf{X}) &\text{ :- } \mathsf{requires}(\mathsf{X},0)\\ \mathsf{coming}(\mathsf{X}) &\text{ :- } \mathsf{requires}(\mathsf{X},\mathsf{K}),\mathsf{N}=\mathsf{count}:\mathsf{kc}(\mathsf{X},\mathsf{Y}),\mathsf{N}\geq\mathsf{K}\\ \mathsf{kc}(\mathsf{X},\mathsf{Y}) &\text{ :- } \mathsf{knows}(\mathsf{X},\mathsf{Y}),\mathsf{coming}(\mathsf{Y})\end{aligned}$$

Note that the kc predicate is only needed here due to the specific syntax defined by Ross and Sagiv, 1992, where aggregation is only allowed over a relation defined by a single predicate. No such auxiliary predicate would be required by the syntax of Liu and Stoller, 2022 recalled in Section 3.2.1.

We conclude this section by noting that the definition of monotone programs based on arbitrary complete lattices by Ross and Sagiv, 1992 clearly generalizes the notion of monotonicity in (Mumick *et al.*, 1990). For instance,

there is no obvious formulation of the shortest path program that would qualify as *monotone* in the sense of (Mumick *et al.*, 1990): subsequent iterations of the program may detect shorter paths and should therefore replace an atom $\mathsf{s}(\mathsf{X},\mathsf{Y},\mathsf{C})$ by a new one $\mathsf{s}(\mathsf{X},\mathsf{Y},\mathsf{C}')$ with $\mathsf{C}' < \mathsf{C}$. However, retracting $\mathsf{s}(\mathsf{X},\mathsf{Y},\mathsf{C})$ contradicts monotonicity in terms of set inclusion.

Monotone aggregates

In a series of works (Mazuran *et al.*, 2013; Shkapsky *et al.*, 2015; Zaniolo *et al.*, 2017; Zaniolo *et al.*, 2019), Zaniolo et al. have approached Datalog^{agg} by splitting the computation of aggregates into a monotone part and a non-monotone part. Apart from the implications for the semantics definition, this distinction is also important for stream processing, since the non-monotone part is blocking, whereas the monotone part is not, as noted by Zaniolo *et al.*, 2019. The distinction of monotone vs. non-monotone aggregates is illustrated by the the following version of the All-Pairs-Shortest-Paths (APSP) program given in (Shkapsky *et al.*, 2015).

$$
\begin{aligned}
\mathsf{spaths}(\mathsf{X},\mathsf{Y},\mathsf{mmin}\langle\mathsf{D}\rangle) &\text{ :- } \mathsf{edge}(\mathsf{X},\mathsf{Y},\mathsf{D})\\
\mathsf{spaths}(\mathsf{X},\mathsf{Y},\mathsf{mmin}\langle\mathsf{D}\rangle) &\text{ :- } \mathsf{spaths}(\mathsf{X},\mathsf{Z},\mathsf{D1}),\mathsf{edge}(\mathsf{Z},\mathsf{Y},\mathsf{D2}),\mathsf{D}=\mathsf{D1}+\mathsf{D2}\\
\mathsf{shortestpaths}(\mathsf{X},\mathsf{Y},\mathsf{min}\langle\mathsf{D}\rangle) &\text{ :- } \mathsf{spaths}(\mathsf{X},\mathsf{Y},\mathsf{D})
\end{aligned}
$$

Even without formally introducing the concrete syntax used in (Shkapsky *et al.*, 2015), the idea of the above program should be clear: the computation of the shortest paths is split into a monotone min operation mmin in the first two rules, which writes a new tuple to the output if the value of D is smaller than the currently known minimum value of D for the given combination of vertices X, Y. This operation is monotone in that it does not retract any previously derived tuples. In contrast, the min operation in the third rule is the usual non-monotone min operation that only selects the tuple with the minimum value of D. In (Zaniolo *et al.*, 2017), the non-monotone nature of the min operation is highlighted by expressing it as "there does not exist a shorter one". In our example, the third rule would thus be rewritten to Datalog$^{\neg}$ as follows:

$$
\begin{aligned}
\mathit{shortestpaths}(X,Y,D) &\text{ :- } \mathit{spaths}(X,Y,D),\neg\,\mathit{lesser}(X,Y,D)\\
\mathit{lesser}(X,Y,D) &\text{ :- } \mathit{spaths}(X,Y,D),\mathit{spaths}(X,Y,D1),D1<D
\end{aligned}
$$

Similarly, also count and sum can be split into a monotone part and a non-monotone part. For instance, in (Zaniolo *et al.*, 2017), the following variant of the Party Invitation program is given, which slightly deviates from Example 3.14.

Example 3.15. Consider edb predicates organizer(X) and friend(X, Y) where the former indicates that X is one of the organizers of a party and the latter means that Y is a friend of X. Now assume that organizers attend the party for sure while all other persons attend the party only if at least 3 of their friends attend the party. The following program defines the idb predicates attends(X) and cntfriends(X, N), where the former means that X attends the party and the letter indicates that (at least) N friends of X attend the party.

$$\begin{aligned} \mathsf{attends}(\mathsf{X}) &\text{ :- } \mathsf{organizer}(\mathsf{X}) \\ \mathsf{attends}(\mathsf{X}) &\text{ :- } \mathsf{cntfriends}(\mathsf{X}, \mathsf{Nfx}), \mathsf{Nfx} \geq 3 \\ \mathsf{cntfriends}(\mathsf{X}, \mathsf{N}) &\text{ :- } \mathsf{attend}(\mathsf{Y}), \mathsf{friend}(\mathsf{Y}, \mathsf{X}), \mathsf{mcount}((\mathsf{Y}), (\mathsf{X}), \mathsf{N}) \end{aligned}$$

Here, mcount((Y), (X), N) is the monotone version of the count operation, where the first argument (Y) is the group-by argument, the second argument (X) is the count argument (i.e., how many (X) values do there exist for given (Y)?), and the third argument is the result of the count aggregate. Clearly, the entire program is monotone. If it turns out that K friends of some person X attend the party, then the program will derive facts cntfriends(X, J) for every $\mathsf{J} \in \{1, \ldots, \mathsf{K}\}$. Moreover, as soon as the fact cntfriends(X, 3) has been derived, also attends(X) will be derived by the second rule.

Indeed, there is no need for non-monotonicity. This need only arises, if we want to compute also an idb predicate fcount(X, N) to determine the precise number N of friends of X attending the party. In this case, we would have to apply the non-monotone max operation to cntfriends(X, N) or, alternatively, we could use a ¬greater literal analogously to the ¬lesser literal in case of the shortestpaths predicate discussed before.

Of course, also the use of monotone aggregates raises the question of formally defining an intuitive semantics. A systematic approach towards this goal is presented in (Mazuran *et al.*, 2013) for monotone count and sum. In (Zaniolo *et al.*, 2019), a similar approach is applied to monotone min and max. The crux of the semantics definition is to extend Datalog by Prolog-style lists, where [ ] denotes the empty set, $[X]$ denotes the list containing the single element X, and $[X|S]$ denotes a list where the first element is X (the head) followed by the list S (the tail). We primarily concentrate here on the approach of Mazuran *et al.*, 2013, who deal with counting in the first place. However, the idea of using lists as a conceptual tool to formally define the semantics of aggregates is also applicable to the other aggregates as we will briefly mention later.

Mazuran *et al.*, 2013 introduce an extension of Datalog with monotone count (which is then also further extended to monotone sum). This extension of Datalog is referred to as Datalog^{FS}, which stands for Datalog with *frequency support goals*. Syntactically, Datalog^{FS} allows the formulation of so-called

running FS goals of the form $\mathsf{K}\colon [\mathsf{expr}]$, where K is a constant or variable and expr is a conjunction of positive atoms. The intuition of a running FS-goal is to express count with a "$\geq$"-condition. The following example, which determines the authors whose H-index is ≥ 13, illustrates the use of this construct.

Example 3.16. Consider edb predicates $\mathsf{author}(\mathsf{A}, \mathsf{P})$ and $\mathsf{refer}(\mathsf{Pfrom}, \mathsf{Pto})$, where the former expresses that A is an author of paper P and the latter means that there is a reference from paper Pfrom to paper Pto. The following program defines idb predicates $\mathsf{atleast13}(\mathsf{P})$ indicating that paper P has at least 13 citations and $\mathsf{hindex13}(\mathsf{A})$ indicating that author A has authored at least 13 of these papers.

$$\begin{aligned} \mathsf{atleast13}(\mathsf{Pto}) &\text{ :- } 13 : [\mathsf{refer}(\mathsf{Pfrom}, \mathsf{Pto})] \\ \mathsf{hindex13}(\mathsf{A}) &\text{ :- } 13 : [\mathsf{author}(\mathsf{A}, \mathsf{P}), \mathsf{atleast13}(\mathsf{P})] \end{aligned}$$

The semantics of a running-FS goal is defined via the following rewriting to Datalog with lists: Consider a running FS-goal $\mathsf{K} : [\mathsf{expr}(\mathsf{X}, \mathsf{Y}]$, where X, Y are vectors of variables, such that X are the global variables (i.e., occurring outside the running-FS goal) and Y are the local variables. Then the rewriting replaces the subgoal $\mathsf{K} : [\mathsf{expr}(\mathsf{X}, \mathsf{Y}]$ in the rule body by $\mathsf{con}(\mathsf{K}, \mathsf{X}, \_)$, where the con predicate is defined by the following rules:

$$\begin{aligned} \mathsf{con}(1, \mathsf{X}, [\mathsf{Y}]) &\text{ :- } \mathsf{expr}(\mathsf{X}, \mathsf{Y}) \\ \mathsf{con}(\mathsf{N1}, \mathsf{X}, [\mathsf{Y}|\mathsf{T}]) &\text{ :- } \mathsf{expr}(\mathsf{X}, \mathsf{Y}), \mathsf{con}(\mathsf{N}, \mathsf{X}, \mathsf{T}), \mathsf{notin}(\mathsf{Y}, \mathsf{T}), \mathsf{N1} = \mathsf{N} + 1 \end{aligned}$$

For every running FS-goal in the program, a separate con-predicate has to be defined. Intuitively, for given value of X, it collects all (pairwise distinct) values Y satisfying $\mathsf{expr}(\mathsf{X}, \mathsf{Y})$ in a list and simultaneously counts the number of elements in this list. The notin-predicate is defined without negation as follows:

$$\begin{aligned} \mathsf{notin}(\mathsf{Z}, [\,]) &\text{ :-} \\ \mathsf{notin}(\mathsf{Z}, [\mathsf{V}|\mathsf{T}]) &\text{ :- } \mathsf{Z} \neq \mathsf{V}, \mathsf{notin}(\mathsf{Z}, \mathsf{T}) \end{aligned}$$

The above rewriting of running-FS goals into a pure logic program proves that the running-FS construct is indeed monotone and can thus be freely used in recursion without destroying the minimal model semantics of logic programs. For convenience, Mazuran *et al.*, 2013 allow two further constructs in Datalog^{FS}, namely *final-FS goals* (to express count with an "="-condition) and *FS-assert terms* (to express that some atom has support $\geq$ some constant or variable). Both constructs can be replaced by running-FS goals and negation.

A *final-FS goal* has the form "$\mathsf{K} =! [\mathsf{expr}(\mathsf{X}, \mathsf{Y})]$", which can be rewritten to "$\mathsf{K} : [\mathsf{expr}(\mathsf{X}, \mathsf{Y})], \neg\mathsf{morethan}(\mathsf{X}, \mathsf{K})$", where the morethan predicate is defined as

$$\mathsf{morethan}(\mathsf{X}, \mathsf{K}) \text{ :- } \mathsf{K1} : [\mathsf{expr}(\mathsf{X}, \mathsf{Y})], \mathsf{K1} > \mathsf{K}$$

An *FS-assert term* is of the form $\mathsf{q}(\mathsf{X1}, \ldots, \mathsf{Xn})$:K and can be used in a rule head to express that, when it comes to counting, the atom $\mathsf{q}(\mathsf{X1}, \ldots, \mathsf{Xn})$ should actually contribute the value K (rather than just 1) to the overall count. For instance, an edb fact $\mathsf{ref}('\mathsf{MousaviZ11}')$:6 (incorporated into the program as a rule with empty body) means that, whenever an atom $\mathsf{ref}(\mathsf{X})$ occurs in an FS-support goal, then $\mathsf{ref}('\mathsf{MousaviZ11}')$ counts as 6. Again, there is a straightforward rewriting also for FS-assert terms. Suppose that we have a rule of the form $\mathsf{q}(\mathsf{X1}, \ldots, \mathsf{Xn}) : \mathsf{K}$:- Body. Then this rule is rewritten to

$$\mathsf{q}'(\mathsf{X1}, \ldots, \mathsf{Xn}, \mathsf{J}) \text{ :- } \mathsf{leq}(\mathsf{J}, \mathsf{K}), \mathsf{Body}$$

where q' is a fresh predicate symbol and J does not occur in Body. That is, if the atom $\mathsf{q}(\mathsf{X1}, \ldots, \mathsf{Xn})$ can be derived in the original program, we actually derive Ã¢ll atoms of the form $\mathsf{q}'(\mathsf{X1}, \ldots, \mathsf{Xn}, \mathsf{J})$ with $1 \leq \mathsf{J} \leq \mathsf{K}$ in the rewritten program. That is, K atoms in total. Clearly, occurrences of an atom $\mathsf{q}(\mathsf{X1}, \ldots, \mathsf{Xn})$ in a rule body of the program now have to be replaced by $\mathsf{q}'(\mathsf{X1}, \ldots, \mathsf{Xn}, \mathsf{J})$, where J does not occur elsewhere in the rule. And the "less-than-or-equal" predicate $\mathsf{leq}(\mathsf{J}, \mathsf{K})$ is readily defined as follows:

$$\begin{aligned} \mathsf{leq}(1, \mathsf{K}) &\text{ :- } \mathsf{K} \geq 1 \\ \mathsf{leq}(\mathsf{J1}, \mathsf{K}) &\text{ :- } \mathsf{leq}(\mathsf{J}, \mathsf{K}), \mathsf{K} > \mathsf{J}, \mathsf{J1} = \mathsf{J} + 1 \end{aligned}$$

Crucially, a Datalog^{FS} program is called stratified if its rewriting into a logic program is negation-stratified. Mazuran *et al.*, 2013 prove that stratified Datalog^{FS} is strictly more expressive than stratified Datalog$^{\neg}$ by showing that a classical game query, which is not expressible by stratified Datalog$^{\neg}$ according to Kolaitis, 1991 can actually be formulated in Datalog^{FS}.

Zaniolo *et al.*, 2019 generalize the idea of rewriting to logic programs with lists to any aggregates. In particular, the authors present generic programs with lists that realize monotone count, sum, min, max. Moreover, it is shown that the only piece missing to turn these monotone aggregates into the corresponding non-monotone ones is the *final count*, i.e., the cardinality of the set over which the aggregate is defined. This final count can be defined by a rule using negation. On the other hand, if the final count is retrieved from an external source (e.g., an EOF marker), then the entire aggregation becomes monotone. The following example illustrates the definition of the monotone and non-monotone max of an arbitrary predicate p. Zaniolo *et al.*, 2019 refer to monotone aggregates as *continuous* and to non-monotone ones as *final*.

Example 3.17. Consider an arbitrary unary predicate p(X). The following program using lists defines the *continuous maximum* cmp(M, C, L) of values satisfying p, where M is the current maximum, L is the list of values that at some point where the current maximum, and C is the cardinality of L. The program makes use of the idb predicate notin(X, L) defined above and the idb predicate larger(X, Y, Z) that binds Z to the maximum of the two values X and Y.

$$\begin{aligned}
\mathsf{cmp}(\mathsf{X}, 1, [\mathsf{X}]) &\text{ :- } \mathsf{p}(\mathsf{X}) \\
\mathsf{cmp}(\mathsf{X}, \mathsf{C1}, [\mathsf{X}|\mathsf{L}]) &\text{ :- } \mathsf{p}(\mathsf{X}), \mathsf{cmp}(\mathsf{M}, \mathsf{C}, \mathsf{L}), \mathsf{larger}(\mathsf{M}, \mathsf{X}, \mathsf{X}), \mathsf{notin}(\mathsf{X}, \mathsf{L}), \mathsf{C1} = \mathsf{C} + 1 \\
\mathsf{larger}(\mathsf{X}, \mathsf{Y}, \mathsf{X}) &\text{ :- } \mathsf{X} > \mathsf{Y} \\
\mathsf{larger}(\mathsf{X}, \mathsf{Y}, \mathsf{Y}) &\text{ :- } \mathsf{X} \leq \mathsf{Y}
\end{aligned}$$

Now the *final maximum* fmp can be defined by the following program that makes use of the *final count* fcountp of predicate p.

$$\begin{aligned}
\mathsf{fcountp}(\mathsf{C}) &\text{ :- } \mathsf{cmp}(_, \mathsf{C}, _), \mathsf{C1} = \mathsf{C} + 1, \neg\, \mathsf{cmp}(_, \mathsf{C1}, _) \\
\mathsf{fmp}(\mathsf{M}) &\text{ :- } \mathsf{cmp}(\mathsf{M}, \mathsf{C}, _), \mathsf{fcountp}(\mathsf{C})
\end{aligned}$$

It should be noted that the rewriting into logic programs with lists is primarily used to provide a formal foundation of the semantics of monotone aggregates. In principle, this rewriting could be used to allow for a Prolog-style top-down evaluation of the resulting programs. However, Zaniolo et al. propose extensions of Datalog optimizations such as semi-naïve evaluation and magic set optimization instead, see (Mazuran *et al.*, 2013; Shkapsky *et al.*, 2015; Zaniolo *et al.*, 2017; Zaniolo *et al.*, 2019). An important aspect of the optimization techniques thus proposed is the question as to whether the aggregation may be pushed into the recursion, which will be discussed next.

Pushing aggregation into the recursion

Recall that stratification was one of the first techniques to define an intuitive and clean semantics of programs with aggregates. This was illustrated by the APSP program in Example 3.10. However, for the actual evaluation of a Datalog^{agg} program, strictly following this stratification may turn out to be highly inefficient or it may even render the usual fixpoint iteration of the ICO impossible. Indeed, if the underlying graph contains a cycle of non-zero length, then the path predicate in Example 3.10 gives rise to an infinite model. That is, if node a is part of a cycle of length c, then the minimal Herbrand model of the program contains all atoms of the form path(a, a, i ∗ c) for all integers i ≥ 1. Consequently, already early works on optimized evaluation of Datalog^{agg}

programs (see, e.g., (Sudarshan and Ramakrishnan, 1991; Ganguly *et al.*, 1991) aimed at pushing the aggregation into the recursion so as to reduce the set of candidates for the actual minimization or maximization. A systematic study of this topic was initiated by Zaniolo et al. in a series of papers (Zaniolo *et al.*, 2017; Condie *et al.*, 2018; Zaniolo *et al.*, 2018). The crucial concept introduced by Zaniolo *et al.*, 2017 is the *pre-mappability* of a constraint γ, which can, for instance, be a minimum or maximum constraint.

Definition 3.11. In a Datalog program P, let P' be a set of rules defining some predicate(s). Moreover, let T denote the immediate consequence operator of P' and let γ be a constraint, i.e., a mapping to be applied to interpretations I of P'. We say that γ is *pre-mappable* to P', if $\gamma(T(I)) = \gamma(T(\gamma(I)))$ holds for every interpretation I of P'.

In case of the APSP program in Example 3.10, the first two rules constitute the set of rules P' defining the predicate path. The constraint γ in this case is the minimization of the third component C for given pair (X, Y) over all atoms $\mathsf{path}(\mathsf{X}, \mathsf{Y}, \mathsf{C})$ in the lfp of the ICO T. Pre-mappability would mean that we may interleave the fixpoint iteration T with the minimization. Indeed, let T_γ denote the application of T followed by γ, i.e., $T_\gamma(I) = \gamma(T(I))$ for arbitrary interpretation I. Then pre-mappability ensures the following property:

Theorem 3.12. In program P, let T denote the ICO of the standard Datalog rules of P. Let γ be a constraint that is pre-mappable to T and suppose that the least fixpoint of T exists and is reachable by a finite number of iterations of T, i.e., there exists n with $T^n(\emptyset) = T^{n+1}(\emptyset))$. Then $T_\gamma^n(\emptyset)$ is the lfp of T_γ; in particular, we have $\gamma(T^n(\emptyset)) = T_\gamma^n(\emptyset)$.

Proof. We prove by induction on k that the equality $\gamma(T^k(\emptyset)) = T_\gamma^k(\emptyset)$ holds for every integer $k \geq 1$. For $k = 1$, the equality $\gamma(T^1(\emptyset)) = T_\gamma^1(\emptyset)$ is simply the definition of T_γ. For arbitrary $k \geq 1$, we have the following chain of equalities:

$$\gamma(T^{k+1}(\emptyset)) = \gamma(T(T^k(\emptyset))) \overset{(1)}{=} \gamma(T(\gamma(T^k(\emptyset)))) \overset{(2)}{=} T_\gamma(\gamma(T^k(\emptyset))) \overset{(3)}{=} T_\gamma(T_\gamma^k(\emptyset)) = T_\gamma^{k+1}(\emptyset),$$

where equality (1) holds by pre-mappability, (2) holds by the definition of T_γ, and (3) holds by the induction hypothesis. □

We illustrate the idea of pushing aggregation into the recursion by revisiting the APSP program from Example 3.10. However, it is convenient to switch to the syntax used by Zaniolo *et al.*, 2017. We thus avoid a slightly clumsy formulation of the rule combining recursion with minimization, which would be required by the syntax of Mumick *et al.*, 1990 that only allows the application of an aggregate to a relation defined by a single atom. Zaniolo *et al.*, 2017

allow the use of $\mathsf{is\_min}((\mathsf{Y}),(\mathsf{D}))$ and $\mathsf{is\_max}((\mathsf{Y}),(\mathsf{D}))$ atoms in rule bodies, where Y is the list of group-by variables and D is the *cost argument* that has to be minimized or maximized, respectively. Then the aggregation stratified APSP program has the following form:

$$\begin{aligned}\mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{C}) &\text{ :- } \mathsf{arc}(\mathsf{X},\mathsf{Y},\mathsf{C}).\\ \mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{C}) &\text{ :- } \mathsf{path}(\mathsf{X},\mathsf{Z},\mathsf{Cxz}),\mathsf{arc}(\mathsf{Z},\mathsf{Y},\mathsf{Czy}),\mathsf{C}=\mathsf{Cxz}+\mathsf{Czy}\\ \mathsf{spath}(\mathsf{X},\mathsf{Y},\mathsf{C}) &\text{ :- } \mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{C}),\mathsf{is\_min}((\mathsf{X},\mathsf{Y}),\mathsf{C}).\end{aligned}$$

Theorem 3.12 tells us that we may transform the program as follows, provided that minimization is pre-mappable to the first 2 rules.

$$\begin{aligned}\mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{C}) &\text{ :- } \mathsf{arc}(\mathsf{X},\mathsf{Y},\mathsf{C}).\\ \mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{C}) &\text{ :- } \mathsf{path}(\mathsf{X},\mathsf{Z},\mathsf{Cxz}),\mathsf{arc}(\mathsf{Z},\mathsf{YZ},\mathsf{Czy}),\mathsf{C}=\mathsf{Cxz}+\mathsf{Czy},\mathsf{is\_min}((\mathsf{X},\mathsf{Y}),\mathsf{C})\\ \mathsf{spath}(\mathsf{X},\mathsf{Y},\mathsf{C}) &\text{ :- } \mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{C}).\end{aligned}$$

This naturally raises the question how we can determine if minimization or maximization is pre-mappable to a single rule or a set of rules in a program. Zaniolo *et al.*, 2017 propose the following test for minimization. The property thus checked is referred to as "deflation preserving". Intuitively, it means that, if we instantiate the variables to be minimized to smaller values in the body of the rule, then we get a smaller value for the variable to be minimized in the head. In this case, minimization is pre-mappable to this rule. Analogously, we get a sufficient condition for the pre-mappability of maximization to a rule, if the rule is "inflation preserving", i.e., if increasing the values to be maximized in the body also increases the value in the head. Zaniolo *et al.*, 2017 show that, if a rule is deflation preserving (resp. inflation preserving), then minimization (resp. maximization) may be pushed into this rule.

For instance, let us inspect the second rule of the stratified program above and consider two instances of this rule:

$$\begin{aligned}\mathsf{path}(\mathsf{x},\mathsf{y},\mathsf{c}') &\text{ :- } \mathsf{path}(\mathsf{x},\mathsf{z},\mathsf{cxz}'),\mathsf{arc}(\mathsf{z},\mathsf{y},\mathsf{czy}),\mathsf{c}'=\mathsf{cxz}'+\mathsf{czy}\\ \mathsf{path}(\mathsf{x},\mathsf{y},\mathsf{c}'') &\text{ :- } \mathsf{path}(\mathsf{x},\mathsf{z},\mathsf{cxz}''),\mathsf{arc}(\mathsf{z},\mathsf{y},\mathsf{czy}),\mathsf{c}''=\mathsf{cxz}''+\mathsf{czy}\end{aligned}$$

We only want to minimize over the third component of the path predicate. So the two instances r' and r'' of the rule coincide on all positions except for the third positions of the path predicate. Suppose that $\mathsf{cxz}'' \leq \mathsf{cxz}'$ holds. Then, for the equalities $\mathsf{c}' = \mathsf{cxz}' + \mathsf{czy}$ and $\mathsf{c}'' = \mathsf{cxz}'' + \mathsf{czy}$ to be satisfied, also $\mathsf{c}'' \leq \mathsf{c}'$. Hence, the rule is deflation preserving and, by Theorem 3.12, minimization is

pre-mappable to the second rule. In other words, pushing minimization into this rule as was done in the non-stratified program above is indeed allowed.

Zaniolo *et al.*, 2018 propose yet another sufficient criterion for pre-mappability, referred to as *drop-in goal*. Again consider a (recursive) rule with head predicate Q into which we want to push an is_min atom (the case is_max is analogous). A drop-in goal would mean that we add (that is, we "drop in") yet another is_min atom after all atoms with predicate Q in the body. If these drop-in goals do not change the meaning of the rule, then minimization is pre-mappable to this rule. Intuitively, this corresponds precisely to checking if the pre-mappability condition $\gamma(T(I)) = \gamma(T(\gamma(I)))$ holds. That is, we can consider the ICO of the original rule as T, pushing an is_min atom into the rule corresponds to the outer application of γ, and finally, the inner application of γ corresponds to the drop-in goals. In case of APSP, we would thus compare the following two rules

$$\begin{aligned}
\mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{C}) \text{ :- } & \mathsf{path}(\mathsf{X},\mathsf{Z},\mathsf{Cxz}),\\
& \mathsf{arc}(\mathsf{Z},\mathsf{YZ},\mathsf{Czy}), \mathsf{C} = \mathsf{Cxz} + \mathsf{Czy}, \mathsf{is_min}((\mathsf{X},\mathsf{Y}),\mathsf{C})\\
\mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{C}) \text{ :- } & \mathsf{path}(\mathsf{X},\mathsf{Z},\mathsf{Cxz}), \mathsf{is_min}((\mathsf{X},\mathsf{Z}),\mathsf{Cxz}),\\
& \mathsf{arc}(\mathsf{Z},\mathsf{YZ},\mathsf{Czy}), \mathsf{C} = \mathsf{Cxz} + \mathsf{Czy}, \mathsf{is_min}((\mathsf{X},\mathsf{Y}),\mathsf{C})
\end{aligned}$$

It is easy to verify that the two rules are equivalent, i.e., the drop-in goal $\mathsf{is\_min}((\mathsf{X},\mathsf{Z}),\mathsf{Cxz})$ does not change the ICO mapping of the rule. We may thus conclude that pushing the $\mathsf{is\_min}((\mathsf{X},\mathsf{Y}),\mathsf{C})$ constraint into the recursive rule is allowed.

Rewriting aggregates

Ganguly *et al.*, 1991 concentrate on the min and max aggregates. They show that programs with these aggregates can be rewritten to Datalog$^\neg$ and, hence, have a unique well-founded model. This leads to two challenges: first, to establish conditions under which the well-founded model is 2-valued and, second, to provide an efficient evaluation method of the rewritten program. We illustrate the main idea of the rewriting by recalling the APSP program from Example 3.10. There, the $\mathsf{spath}(\mathsf{X},\mathsf{Y},\mathsf{C})$ predicate is defined in the third rule of the program by minimization over the C-values in the $\mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{C})$ predicate. This minimization constraint could, in principle, be captured by a Datalog$^\neg$ rule of the following form[3]:

$$\mathsf{spath}(\mathsf{X},\mathsf{Y},\mathsf{C}) \text{ :- } \mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{C}), \neg\,(\mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{C1}), \mathsf{C1} < \mathsf{C})$$

[3]We will see a slightly more elaborate rewriting below.

As far as the efficient evaluation of Datalog^{agg} programs is concerned, Ganguly *et al.*, 1991 observe that a purely stratified approach by first fully evaluating the path predicate before evaluating spath is highly inefficient (and does not even terminate if the graph contains cycles). Analogously to Example 3.13, the authors therefore propose to push the min aggregate into the recursion. We have already discussed above this line of research, which was later continued and extended by the work on pre-mappability. Here, we concentrate on the other challenge; that is, finding conditions under which the well-founded model of the rewritten program is 2-valued. We thus first recall the syntax and semantics definition of Ganguly *et al.*, 1991. Note that the entire paper only deals with min. But it is mentioned (and clear) how to carry these results over to max.

For the syntax of Datalog^{agg} programs, Ganguly *et al.*, 1991 assume the existence of two predicate symbols: a unary predicate d (= domain) and a binary predicate $<_d$ (= total order on the domain). These two predicates define the *cost domain*. Then a "special atom" is of the form $\mathsf{min}(\mathsf{C}, \mathsf{S}, \mathsf{Q})$, where S is the set of "grouping variables", $\mathsf{C} \notin \mathsf{S}$ is a variable that takes values only from the cost domain, and Q is an atom containing all variables in $\mathsf{S} \cup \{\mathsf{C}\}$. Ganguly *et al.*, 1991 assume that rules with aggregation in the body have a single body atom. That is, these rules have the form

$$\mathsf{P}(\mathsf{X}_1, \ldots, \mathsf{X}_m, \mathsf{C}) \text{ :- } \mathsf{min}(\mathsf{C}, \mathsf{S}, \mathsf{Q}(\mathsf{Y}_1, \ldots, \mathsf{Y}_n))$$

This is no loss of generality as we can always bring rules with min aggregates into this form by introducing auxiliary predicates.

The semantics of programs with "special atoms" is defined via rewriting to a program with negation. That is, we replace "$\mathsf{min}(\mathsf{C}, \mathsf{S}, \mathsf{Q})$" by "$\mathsf{Q} \wedge \mathsf{d}(\mathsf{C}), \wedge \neg(\mathsf{Q}' \wedge \mathsf{d}(\mathsf{C}') \wedge \mathsf{C}' <$ where (1) Q' is obtained from Q by replacing every occurrence of a variable $W \notin S$ by a new variable W' and (2) the predicates d and $<_d$ are pre-interpreted in the cost domain. Then M is a model of program P iff it is a model of the rewritten program $foe(P)$ (= first-order extension of P).

In case of the APSP program, the last rule looks as follows in the syntax of Ganguly *et al.*, 1991:

$$\mathsf{spath}(\mathsf{X}, \mathsf{Y}, \mathsf{C}) \text{ :- } \mathsf{min}(\mathsf{C}, (\mathsf{X}, \mathsf{Y}), \mathsf{path}(\mathsf{X}, \mathsf{Y}, \mathsf{C}))$$

The special atom $\mathsf{min}(\mathsf{C}, (\mathsf{X}, \mathsf{Y}), \mathsf{path}(\mathsf{X}, \mathsf{Y}, \mathsf{C}))$ in this rule actually requires a slightly bigger rewriting than the one shown above when we illustrated the basic idea of the approach by Ganguly *et al.*, 1991. The body of the above rule is thus transformed as follows:

$$\mathsf{path}(\mathsf{X}, \mathsf{Y}, \mathsf{C}) \wedge \mathsf{d}(\mathsf{C}) \wedge \neg(\mathsf{path}(\mathsf{X}, \mathsf{Y}, \mathsf{C}') \wedge \mathsf{d}(\mathsf{C}') \wedge (\mathsf{C}' < \mathsf{C}))$$

To obtain a sufficient criterion for the well-founded model to be 2-valued, Ganguly *et al.*, 1991 introduce the notion of *cost monotonicity*. To this end, they first modify the notions of rule instantiations and of the dependency graph so as to take the intended meaning of the cost domain into account. An *interpreted instantiation* of a rule r containing a special atom is an instantiation in which each subgoal corresponding to an interpreted predicate (i.e., $\mathtt{d}$ and $<_d$) is `true`. The *interpreted dependency graph* of a program P is a directed graph whose nodes are ground atoms of $foe(P)$. Moreover, there is an arc from a node A to another node B if there is an interpreted instantiation r' of some rule r in P such that A appears in the body and B in the head of r'. Then cost monotonicity is defined as follows:

Definition 3.12. A program P containing min-aggregates with interpreted dependency graph G_P is said to be *cost monotone* if for every pair (A, B) of ground atoms with a cost argument and sharing the same predicate, a path from A to B may only exist if $cost(B) \geq cost(A)$ holds. Moreover, if the path contains a negated arc, then $cost(B) > cost(A)$ holds. Here $cost(\cdot)$ denotes the value of the cost argument.

That is, cost monotonicity requires that if a cost atom A is used in the derivation of a cost atom B with the same predicate, then the derived atom B must have greater or at least the same cost value as A. The intuition of this requirement is best seen when looking at the second rule in the APSP program: the cost argument C in the head atom $\mathsf{path(X, Y, C)}$ is guaranteed to be $\geq$ the cost argument Cxz in the body atom $\mathsf{path(X, Z, Cxz)}$ with the same predicate, provided that all arcs in the graph have non-negative length.

Cost monotonicity has the following desirable consequence:

Theorem 3.13. Every cost monotone program with min aggregates is weakly stratified and, hence, has a 2-valued well-founded model.

The proof idea is to show that if a program P is cost monotone, then its first-order extension $foe(P)$ is weakly stratified, where the stratification is directly derived from the interpreted dependency graph: if there is a path from A to B, then $A \preceq B$ must hold in the stratification; this is strengthened to $A \prec B$, if the path from A to B goes through negation.

In case of the APSP program, cost monotonicity is easily verified. However, in general, cost monotonicity is an undecidable property. Hence, Ganguly *et al.*, 1991 give a simple sufficient criterion for cost monotonicity. To this end, they define the cost graph CG_P as follows: the nodes in CG_P are pairs Q/k, where Q is a predicate symbol and k is the position of the cost argument (we set $k = 0$ if Q contains no cost argument). The directed graph CG_P has an arc from Q/k to R/ℓ if P contains an interpreted instantiation of a rule r with head predicate R and a body atom with predicate Q. The arc is marked with

"$\geq$" if r has no aggregate in the body and the cost argument of Q is $\geq$ the cost argument of R in every interpreted instance of r. If the arc is derived from a rule with a min-predicate in the body, then it is marked with "min". Then a strongly connected component of the cost graph CG_P is called *uniformly monotone* if all its node have a cost argument and all its arcs are marked with "$\geq$" or "min". A program P is said to be *uniformly monotone* if all SCCs in CG_P that contain min predicates are uniformly monotone. Ganguly *et al.*, 1991 show that every uniformly monotone program P is indeed cost monotone.

We now verify that an optimized version of the APSP program P indeed is uniformly monotone and, hence, cost monotone. Therefore, its rewritten version $foe(P)$ is weakly stratified and, hence, has a 2-valued well-founded model.

Example 3.18. Consider the following version of the APSP program, where the minimization has been pushed inside the recursion.

$$\begin{aligned} \mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{C}) &\text{ :- } \mathsf{arc}(\mathsf{X},\mathsf{Y},\mathsf{C}) \\ \mathsf{path}(\mathsf{X},\mathsf{Z},\mathsf{Y},\mathsf{C}) &\text{ :- } \mathsf{spath}(\mathsf{X},\mathsf{Z},\mathsf{C1}), \mathsf{arc}(\mathsf{Z},\mathsf{Y},\mathsf{C2}), \mathsf{C} = \mathsf{C1} + \mathsf{C2} \\ \mathsf{spath}(\mathsf{X},\mathsf{Y},\mathsf{C}) &\text{ :- } \mathsf{min}(\mathsf{C},(\mathsf{X},\mathsf{Y}),\mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{C})) \end{aligned}$$

The only relevant SCC of the cost graph has two vertices path/3 and spath/3. Moreover, assuming non-negative costs, this SCC has an arc from path/3 to spath/3 for the rule $r3$ marked with "min" and an arc spath/3 to path/3 for the rule $r2$ marked with "$\geq$". Hence, this SCC and, therefore, the entire program P is uniformly monotone. Thus, when rewriting this program to a Datalog$^\neg$ program, we get a weakly stratified program by Theorem 3.13.

3.2.3 Aggregation with Fitting's 3-Valued Semantics

Denecker *et al.*, 2000 proposed a framework for approximating the major semantics of logic programs and various forms of non-monotonic reasoning. The study of this framework was later deepened by the same authors in (Denecker *et al.*, 2004). Pelov *et al.*, 2007 showed how this framework can be further extended so as to accommodate also aggregates. In particular, they provided an extension of Fitting's 3-valued semantics as well as the well-founded and the stable model semantics to logic programs with aggregates. In this section, we first recall the basic ideas of the approximation framework developed by Denecker *et al.*, 2000; Denecker *et al.*, 2004 and then show how it can be used to extend Fitting's 3-valued semantics. We will revisit this framework in Sections 3.2.4 and 3.2.5, when we also look at the extension of well-founded semantics and stable model semantics, respectively, to Datalog^{agg}.

Approximations and approximating operators

Denecker *et al.*, 2000 develop a very general approximation theory where elements of a complete lattice $(L, \leq)$ are approximated by pairs in the bilattice L^2 equipped with 2 orderings $\leq$ and $\leq_i$ defined as follows:

$$(x, y) \leq (x', y') \text{ if } x \leq x' \text{ and } y \leq y'$$
$$(x, y) \leq_i (x', y') \text{ if } x \leq x' \text{ and } y' \leq y$$

Denecker *et al.*, 2000 call $\leq$ the *lattice ordering* and $\leq_i$ the *information ordering*. The latter is referred to as *precision* ordering in (Denecker *et al.*, 2004). Pairs $(x, y) \in L^2$ with $x \leq y$ can be considered as *approximations* of all elements $z \in L$ with $x \leq z \leq y$. In other words, rather than taking the precise value z, we "approximate" it by an interval $[x, y]$. Clearly, the pair $(x, x) \in L^2$ exactly specifies an element $x \in L$. Let $\bot$ and $\top$ denote the least and greatest element of the complete lattice $(L, \leq)$. Then $(\bot, \top)$ is the least element w.r.t. $\leq_i$ in L^2; it "approximates" all of L.

This idea of approximations is extended to operators $O\colon L \to L$. That is, an operator $A\colon L^2 \to L^2$ is called an approximation of O, if (1) A is $\leq_i$-monotone and (2) $A(x, x) = (O(x), O(x))$. By condition (2), whenever we are given an approximating operator A, it is clear what the approximated operator O looks like. The monotonicity of condition (1) is crucial for ensuring – by the Knaster-Tarski Theorem – that A has a least fixpoint (and also a greatest fixpoint) in L^2. Moreover, the least fixpoint can be obtained by iterating over the least element $(\bot, \top)$ of L^2 w.r.t. $\leq_i$. The least fixpoint of an approximating operator A is referred to as the Kripke-Kleene fixpoint of A, denoted as $k(A)$. Clearly, if $x \in L$ is a fixpoint of O, then $(x, x) \in L^2$ is a fixpoint of A. Moreover, $k(A) \leq_i (x, x)$ holds for every fixpoint x of O, since $k(A)$ is the least fixpoint of A. In other words, the Kripke-Kleene fixpoint $k(A)$ approximates every fixpoint of O.

Note As mentioned above, only pairs $(x, y) \in L^2$ with $x \leq y$ constitute approximations of elements in L. Hence, also for the approximation A of an operator O, only its behavior on pairs $(x, y) \in L^2$ with $x \leq y$ is relevant and, of course, also the result of applying A should be an approximation, i.e., $A(x, y)_1 \leq A(x, y)_2$ should hold. Denecker *et al.*, 2000 therefore impose the restriction that approximation operators must be *symmetric*, i.e., $A(x, y) = A(y, x)$ for all $x, y \in L$. A different approach is taken by Denecker *et al.*, 2004 and continued by Pelov *et al.*, 2007, who restrict their attention to the subset $L^c \subseteq L^2$ of *consistent* pairs (x, y), i.e., pairs satisfying $x \leq y$. The problem of the latter approach is, that L^c is not a sublattice of L^2: indeed, each element of the form (x, x) is maximal w.r.t. $\leq_i$ and the join of any two such maximal

elements is not in L^c. Hence, the existence of certain fixpoints can no longer be guaranteed by the Knaster-Tarski Theorem. However, by exploiting the fact that L^c is *chain-complete* (i.e., every totally ordered subset of L^c has a least upper bound), Denecker *et al.*, 2004 show that all relevant fixpoint results for L^2 in (Denecker *et al.*, 2000) also hold for L^c. In the sequel, we are only interested in the behavior of approximations on consistent pairs (x, y) and we will only be dealing with approximating operators that map consistent pairs to consistent pairs. So it is inessential, which of the two restrictions imposed by Denecker *et al.*, 2000 or by Denecker *et al.*, 2004 are actually applied.

The idea of an approximating operator A is to apply it to an approximation (a, b) in order to get a "better" approximation $A(a, b)$, that is $(a, b) \leq_i A(a, b)$. Of course, if we start with $(a_0, b_0) = (\bot, \top)$ and iterate through applications of A, then it is guaranteed by the monotonicity of A w.r.t. $\leq_i$ that A constantly produces "better" approximations until the least fixpoint $k(A)$ is reached, i.e., $(a_k, b_k) \leq_i (a_{k+1}, b_{k+1})$ with $(a_{k+1}, b_{k+1}) = A(a_k, b_k)$ holds for every $k \geq 0$. However, for arbitrarily chosen pair (a, b), the desired "improvement" when applying A is not guaranteed. Denecker *et al.*, 2004 call a pair (a, b) "*A-reliable*" if $(a, b) \leq_i A(a, b)$ holds.

To apply the approximation theory to Datalog or, more generally, to logic programming, Denecker et al. consider the complete lattice of Herbrand interpretations K (represented as the set of ground atoms whose truth value is `true` in K) for a given program P with the $\subseteq$-ordering. In this lattice, the minimal and maximal elements are $\bot = \emptyset$ and $\top = HB(P)$ (it is convenient to consider the edb as part of the program, namely rules with empty body). Approximations are then given in the form (I, J) with $I \subseteq J$, representing all interpretations K with $I \subseteq K \subseteq J$. An approximation (I, J) can also be considered as 3-valued interpretation with the understanding that (1) all atoms in I are `true`, (2) atoms in $J \setminus I$ are `undef`, and (3) all atoms in $HB(P) \setminus J$ are `false`. In this case, the *information* ordering $\leq_i$ is precisely the *knowledge* ordering $\leq_k$ used in the definition of Fitting's semantics of Datalog$^\neg$ discussed in Section 3.1.3.

The $\leq_i$-minimal element among the approximations of Herbrand interpretations is $(\bot, \top)$, i.e., $(\emptyset, HB(P))$. In other words, all ground atoms have truth value `undef`. 2-valued interpretations correspond to pairs (K, K). The operator $O\colon L \to L$ of interest is the ICO T_P for a given program $P \in$ Datalog$^\neg$. It is easy to verify that the 3-valued immediate consequence operator Φ_P in the definition of Fitting's semantics is an approximation of the operator T_P. In particular, it is $\leq_i$-monotone. Therefore, by the Knaster-Tarski Theorem, the least fixpoint of Φ_P exists and it can be obtained as $\Phi_P^\omega((\emptyset, HB(P)))$. Denecker et al. refer to it as the *Kripke-Kleene fixpoint*. Recall from Section 3.1.3 that also Fitting referred to the lfp of Φ_P as the "Kripke-Kleene semantics".

Application of the approximation theory to Datalog^{agg}

Pelov *et al.*, 2007 extend the idea of approximations and, in particular, of the Kripke-Kleene fixpoint to programs with aggregates. Syntactically, programs may now also have *aggregate atoms* in the rule body. These consist of 3 components: an aggregate symbol, a set expression (over which the aggregate has to be computed) and a value. For instance, we write $\mathsf{CARD}_{\geq}(\{\mathsf{X} \mid \mathsf{p}(\mathsf{X})\}, 2)$ to denote an aggregate atom that is `true` if there are at least 2 possible values for X to make $p(X)$ `true`. Clearly, aggregate atoms according to Pelov *et al.*, 2007 are just a different syntax for comparison atoms considered by Liu and Stoller, 2022 (see Section 3.2.1).

Pelov *et al.*, 2007 assume a fixed signature Σ with pre-defined symbols (including standard function symbols such as $+, *, -, \ldots$ and standard predicate symbols such as $=, \neq, \leq, <, \ldots$) over standard domains (such as $\mathbb{N}, \mathbb{R}, \ldots$). Moreover, they also assume a collection of aggregate symbols, CARD, SUM, MIN, MAX, etc., subscripted by a comparison operator (which can be omitted in case of "="). In addition, a program can have *defined* predicates from a signature Π. Then a program with aggregates consists of rules of the form A :- ϕ, where A is an atom of a (defined) predicate from Π and the body ϕ may possibly contain aggregate atoms. For instance, the Party Invitation program from Example 3.14 could be realized in the syntax proposed by Pelov *et al.*, 2007 with the following single rule:

$$\mathsf{coming(X)} \text{ :- } \mathsf{requires(X,N)} \wedge \mathsf{CARD}_{\geq}(\{\mathsf{Y} \mid \mathsf{knows(X,Y)} \wedge \mathsf{coming(Y)}\}, \mathsf{N})$$

In order to define a 3-valued semantics of aggregate programs, Pelov *et al.*, 2007 first of all have to consider 3-valued sets, i.e., the elements of a set are annotated with truth values `true` and `undef`. There is no need for an annotation with `false`, since such elements would simply not be listed as part of the set. For instance, $\tilde{S} = \{1^t, 2^u, 3^t, 5^u\}$ is the 3-valued set that contains the elements $1, 2$ for sure, that may possibly contain the elements $2, 5$, and that definitely does not contain any other elements. Alternatively, and possibly more in the spirit of approximations, we can denote this 3-valued set by the pair (S_1, S_2) with $S_1 = \{1, 3\}$ and $S_2 = \{1, 2, 3, 5\}$. That is, the elements in S_1 are definitely in $\tilde{S}$, the elements in $S_2 \setminus S_1$ are possibly in $\tilde{S}$, and the elements in $\mathbb{N} \setminus S_2$ are definitely not in $\tilde{S}$.

Then the aggregate atom $\mathsf{CARD}_{\geq}(\tilde{\mathsf{S}}, \mathsf{N})$ has the truth value

- `true` for $\mathsf{N} \in \{0, 1, 2\}$,
- `undef` for $\mathsf{N} \in \{3, 4\}$,
- `false` for all other natural numbers N.

Again, approximations of the "exact" interpretation of an aggregate atom are obtained by extending the space of `undef` values. For instance, for the above aggregate atom $\mathsf{CARD}_{\geq}(\tilde{\mathsf{S}}, \mathsf{N})$, an approximation could assign the following truth values:

- `true` for $\mathsf{N} \in \{0\}$
- `undef` for $\mathsf{N} \in \{1, 2, 3, 4, 5, 6\}$
- `false` for all other natural numbers N

Clearly, the latter interpretation is smaller w.r.t. $\leq_i$ than the exact interpretation of $\mathsf{CARD}_{\geq}(\tilde{\mathsf{S}}, \mathsf{N})$.

Aggregate atoms are interpreted by 3-valued *aggregate relations* $\mathcal{R}$. Pelov *et al.*, 2007 take here a very liberal position in that they allow $\mathcal{R}$ to be any function that maps every combination $(\tilde{S}, d)$ of 3-valued set expression $\tilde{S}$ and value d from the co-domain of the aggregate to one of the truth values $\{\texttt{false}, \texttt{undef}, \texttt{true}\}$. They only impose the following restrictions on $\mathcal{R}$:

1. $\mathcal{R}$ must be $\leq_i$-monotone, i.e., for any two combinations $(\tilde{S}_1, d)$ and $(\tilde{S}_2, d)$ with $\tilde{S}_1 \leq_i \tilde{S}_2$, and any value d, we have $\mathcal{R}(\tilde{S}_1, d) \leq_i \mathcal{R}(\tilde{S}_2, d)$ and
2. if $\tilde{S}$ is two-valued (i.e., $\tilde{S} = (S, S)$ for a 2-valued set S), then, for every value d, we have $\mathcal{R}(\tilde{S}, d) \in \{\texttt{false}, \texttt{true}\}$.

However, it might be more intuitive to restrict aggregate relations to the interpretation of comparison atoms $aggS \odot k$ as proposed by Liu and Stoller, 2022, see Section 3.2.1. Then an aggregate atom of the form $\mathsf{agg}(\mathsf{expr}, \mathsf{t})$ with set expression $\mathsf{expr} = \{(\mathsf{X}_1, \ldots, \mathsf{X}_n) \mid \mathsf{p}(\mathsf{X}_1, \ldots, \mathsf{X}_n)\}$ is evaluated in a 3-valued interpretation (I, J) as follows:

- We first of all evaluate the set expression expr to a 3-valued set. That is, every n-tuple $(\mathsf{d}_1, \ldots, \mathsf{d}_n)$ gets assigned the truth value that the ground atom $\mathsf{p}(\mathsf{d}_1, \ldots, \mathsf{d}_n)$ has in (I, J).
- Now suppose that t is either the constant d or a variable that is instantiated to d. Then $\mathsf{agg}(\mathsf{expr}, \mathsf{t})$ gets one of the truth values $\{\texttt{undef}, \texttt{false}, \texttt{true}\}$ according to the aggregate relation for interpreting comparison atoms in Section 3.2.1.

Then the operator Φ_P on 3-valued interpretations can be defined on Datalog^{agg} programs analogously to Datalog$^{\neg}$ programs in Section 3.1.3, namely:

- A ground atom A is assigned `true` in $\Phi_P((I, J))$ if there exists a ground instance A :- $\mathsf{B}_1, \ldots, \mathsf{B}_n$ of a rule in P, such that all literals B_i are `true` in (I, J).

- A ground atom A is assigned `false` in $\Phi_P((I,J))$ if for every ground instance of rules in P with head atom A, the body of r evaluates to `false` in (I,J) (this includes the case that no rule in P has a ground instance with head atom A).
- otherwise, $\Phi_P((I,J))$ assigns `undef` to A.

Then, analogously to Datalog$^\neg$, Fitting's 3-valued semantics (also called the Kripke-Kleene semantics) of a Datalog^{agg} program P is the least fixpoint $\Phi_P^\omega((\emptyset, HB(P)))$ of the operator Φ_P. We illustrate the extension of Fitting's 3-valued semantics to Datalog^{agg} presented in Pelov *et al.*, 2007 by slightly extending the win-move game.

Example 3.19. Recall the win-move game from Example 3.4. We again consider the graph shown in Figure 3.2. But now we want to answer the question if there are at least 2 winning positions. Then the Datalog^{agg} program looks as follows:

$$\begin{aligned} \mathsf{Win(X)} &\text{ :- } \mathsf{Edge(X,Y)}, \neg\, \mathsf{Win(Y)} \\ \mathsf{ans} &\text{ :- } \mathsf{Count}_{\geq}(\{\mathsf{X} \mid \mathsf{Win(X)}\}, 2) \end{aligned}$$

Iterating the ICO Φ_P computes the following sequence of IDB instances. It is convenient to abbreviate Win to W and the truth values `undef`, `false`, and `true`. to `u`, `f`, `t`, respectively.

	W(a)	W(b)	W(c)	W(d)	W(e)	W(f)	ans	
$\Phi_P^0 =$	u	u	u	u	u	u	u	
$\Phi_P^1 =$	u	u	u	u	u	f	u	
$\Phi_P^2 =$	u	u	u	u	t	f	u	
$\Phi_P^3 =$	u	u	u	f	t	f	u	
$\Phi_P^4 =$	u	u	t	f	t	f	u	
$\Phi_P^5 =$	u	u	t	f	t	f	t	$= \Phi_P^6$

That is, the ans predicate gets the expected value `true`. Note however what happens if we ask if there are at least 3 winning positions by changing the second rule to

$$\mathsf{ans} \text{ :- } \mathsf{Count}_{\geq}(\{\mathsf{X} \mid \mathsf{Win(X)}\}, 3)$$

Then we would end up with truth value `undef` for the predicate ans, which is slightly unintuitive since, in this game, the truth value `undef` of Win(X) means that position X is a draw. Hence, for this particular application, we would rather expect ans to get truth value `false`.

Ultimate approximations and ultimate semantics of aggregates

In (Denecker *et al.*, 2001), (Denecker *et al.*, 2004), and (Pelov *et al.*, 2007), the authors asked the question if, among all possible approximations of an operator O, a most informative (also referred to as "most precise") approximation exists. As the authors point out, such a most informative approximation, which they refer to as *ultimate approximation* would lead to the most informative Kripke-Kleene and well-founded fixpoint and also to the largest set of exact stable models (for the latter two, details will be provided in Section 3.2.5). Given an operator O on a complete lattice. Denecker et al. define the *ultimate approximation operator* U_O of O as follows:

$$U_O(x,y) = \big(\bigwedge_{z \in [x,y]} O(z), \bigvee_{z \in [x,y]} O(z) \big)$$

Moreover, it is shown by Denecker *et al.*, 2004 that U_O is indeed the most informative approximation of O: First, it is easy to see that U_O is an approximation of O, i.e.: $U_O(x,x) = \big(\bigwedge_{z\in[x,x]} O(z), \bigvee_{z\in[x,x]} O(z)\big) = (O(x), O(x))$ and also monotonicity w.r.t. $\leq_i$ is easily verified. It remains to show that U_O is maximal w.r.t. $\leq_i$. Consider another approximation C of O and consider an arbitrary pair (x,y) with $x \leq y$. We have to show that $C(x,y)_1 \leq U_O(x,y)_1$ and $U_O(x,y)_2 \leq C(x,y)_2$. Since C is an approximating operator of O, for every $z \in [x,y]$, we have $C(x,y) \leq_i (O(z), O(z))$. Hence, also $C(x,y)_1 \leq \bigwedge_{z\in[x,y]} O(z) = U_O(x,y)_1$ and $U_O(x,y)_2 = \bigvee_{z\in[x,y]} O(z) \leq C(x,y)_2$ hold.

In case of the ICO T_P of a logic program P with negation and/or aggregates, the ultimate approximation operator, denoted as U_P, is defined as

$$U_P(I_1, I_2) = \big(\bigcap_{I \in [I_1, I_2]} T_P(I), \bigcup_{I \in [I_1, I_2]} T_P(I) \big)$$

By the above considerations, it is the most precise approximation of T_P, i.e., we have $\Phi_P \leq_i U_P$ but the converse, in general, does not hold. This property clearly carries over to Datalog^{agg} programs, i.e., in particular, we have, $\Phi_P^{aggr} \leq_i U_P^{aggr}$ but not vice versa. In principle, this seems a desirable property. However, as we will see in Section 3.2.5, the downside of U_P is that the ultimate well-founded fixpoint and the ultimate exact stable models of a Datalog$^{\neg}$ program may deviate from the usual definition of the well-founded and stable models discussed in Sections 3.1.4 and 3.1.2.

As mentioned above, the approximation framework of Denecker et al. is very liberal for the approximation of aggregates. That is, an approximation of aggregate atoms only needs to be monotone w.r.t. $\leq_i$ and two-valued for two-valued (multi-)sets. Similarly to the ultimate approximation of an operator O, Pelov *et al.*, 2004 define an *ultimate approximating aggregate* as the most precise approximation of an aggregate relation. Recall that an

aggregate atom of the form $agg_{op}(S, d)$ is interpreted by an aggregate relation $R \subseteq \mathcal{M}(D) \times D$, where D is the domain and $\mathcal{M}(D)$ denotes the set of finite multisets over D. Moreover, let $\mathcal{M}(D)^c$ denote the set of consistent pairs (M_1, M_2) of multisets from $\mathcal{M}(D)$, i.e., $M_1 \subseteq M_2$. Then Pelov *et al.*, 2004 define the *ultimate approximating aggregate* U_R of R as the following mapping $U_R \colon \mathcal{M}(D)^c \times D \to \{\texttt{undef}, \texttt{false}, \texttt{true}\}$ as follows:

$$((M_1, M_2), d) \in U_R^1 \text{ if and only if } \forall M \in [M_1, M_2] \colon (M, d) \in R \text{ and}$$
$$((M_1, M_2), d) \in U_R^2 \text{ if and only if } \exists M \colon [M_1, M_2] \colon (M, d) \in R.$$

For instance, suppose that we are given an aggregate atom $\mathsf{SUM}(\mathsf{X} \colon \mathsf{p}(\mathsf{X}) > 3)$ (using the syntax of Liu and Stoller, 2022 introduced in Section 3.2.1). To determine its truth value in U_R for a specific 3-valued interpretation I of the p-predicate, we would first determine the 3-valued (multi-)set $\tilde{S} = (S_1, S_2)$ resulting from I and then check the truth value of $((S_1, S_2), 3)$ according to (U_R^1, U_R^2) as defined above. That is, the aggregate atom evaluates to

- $\texttt{true}$, if $((S_1, S_2), 3) \in U_R^1$, i.e., if every $S \subseteq S_1$ has the property that applying the aggregate SUM to S yields a value > 3;
- $\texttt{undef}$, if $((S_1, S_2), 3) \in U_R^2 \setminus U_R^1$, i.e., if there exists $S \subseteq S_2$ such that applying the aggregate SUM to S yields a value > 3 but there also exists $S' \subseteq S_2$ such that applying the aggregate SUM to S' yields a value ≤ 3;
- $\texttt{false}$ otherwise.

We illustrate this reasoning by inspecting two example interpretations I.

- Let $I = \{(\{\mathsf{p}(1)\}, \{\mathsf{p}(1), \mathsf{p}(3)\})$, i.e., we have $I(\mathsf{p}(1)) = \texttt{true}$, $I(\mathsf{p}(3)) = \texttt{undef}$, and $I(\mathsf{p}(2)) = \texttt{false}$. We would expect the above aggregate atom to evaluate to $\texttt{undef}$, since, depending on whether the undefined truth value $I(\mathsf{p}(3))$ ultimately becomes $\texttt{false}$ or $\texttt{true}$, also the aggregate atom would ultimately become $\texttt{false}$ or $\texttt{true}$. We verify that $\texttt{undef}$ is also the truth value assigned to the aggregate atom by the ultimate approximate aggregate: By applying I to p, we obtain the 3-valued set $\{1^t, 3^u\}$, represented as $\tilde{S} = (S_1, S_2)$ with $S_1 = \{1\}$ and $S_2 = \{1, 3\}$. By inspecting the above definition of U_R, we have to check that we indeed have $((S_1, S_2), 3) \notin U_R^1$ and $((S_1, S_2), 3) \in U_R^2$. We note that $((S_1, S_2), 3) \notin U_R^1$ holds, because there exists an $S \in [S_1, S_2]$ with $(S, 3) \notin R$. That is, we have $S_1 = \{1\} \in [S_1, S_2]$ and, clearly, $\mathsf{SUM}(\{1\} > 3$ is $\texttt{false}$. This falsifies the condition that $\forall S \in [S_1, S_2] : (S, 3) \in R$ must hold. On the other hand, we have $S_2 = \{1, 3\} \in [S_1, S_2]$ and, clearly, $\mathsf{SUM}(\{1, 3\} > 3$ is $\texttt{true}$. Hence, by the definition of U_R^2, we have $((S_1, S_2), 3) \in U_R^2$.

- Now let $I = \{(\{\mathsf{p}(1), \mathsf{p}(3)\}, \{\mathsf{p}(1), \mathsf{p}(2), \mathsf{p}(3)\})$, i.e., we have $I(\mathsf{p}(1)) = \texttt{true} = I(\mathsf{p}(3)) = \texttt{undef}$, and $I(\mathsf{p}(2)) = \texttt{undef}$. Now we expect the above aggregate atom to evaluate to `true`, since, no matter how the undefined truth value $I(\mathsf{p}(3))$ is ultimately replaced by `false` or `true`, the aggregate atom always becomes `true`. We verify that `true` is also the truth value assigned to the aggregate atom by the ultimate approximate aggregate: By applying I to p, we obtain the 3-valued set $\{1^t, 2^u, 3^t\}$, represented as $\tilde{S} = (S_1, S_2)$ with $S_1 = \{1, 3\}$ and $S_2 = \{1, 2, 3\}$. By inspecting the above definition of U_R, it is easy to verify that we now have $((S_1, S_2), 3) \in U_R^1$: Indeed, the only 2-valued sets in $[S_1, S_2]$ are $S = \{1, 3\}$ and $S' = \{1, 2, 3\}$. We clearly have $(S, 3) \in R$ and $(S', 3) \in R$ since, applying the SUM aggregate to either of the two sets yields a value > 3.

We note that the ultimate approximate aggregate is closely related to the way how the satisfaction relation for aggregate atoms is defined by Liu and Stoller, 2022. But of course, there is a crucial difference in that Liu and Stoller, 2022 only give an explicit definition of when an aggregate atom evaluates to `true` in a 3-valued interpretation.

3.2.4 Aggregation with Well-Founded Semantics

Kemp and Stuckey, 1991 proposed a straightforward extension of the well-founded semantics from Datalog$^\neg$ to Datalog^{agg}. The main step in their approach is to define the evaluation of aggregate subgoals under a 3-valued interpretation. With this, it is straightforward to extend all other ingredients needed for the definition of the well-founded semantics to programs with aggregation, i.e., the 3-valued ICO of a Datalog^{agg} program P, unfounded sets of P relative to some 3-valued interpretation and, finally, the well-founded model of P. A short-coming of the approach of Kemp and Stuckey, 1991 is that in their well-founded model, possibly an unintuitively large set of atoms is assigned the truth value `undef`. Gelder, 1992 therefore proposes an improvement of this approach by carrying over from Datalog$^\neg$ to Datalog^{agg} the definition of the well-founded model via the alternating fixpoint construction (see Section 3.1.4 for the two definitions of the well-founded model of Datalog$^\neg$ programs: via unfounded sets and via alternating fixpoints). By applying this approach to several classical examples (APSP, Company Control), he illustrates the advantage of his approach over the one by Kemp and Stuckey, 1991 but also over the stratification-based approach by Mumick *et al.*, 1990. Finally, as mentioned in Section 3.2.3, Pelov *et al.*, 2007 not only applied the approximation framework from (Denecker *et al.*, 2000; Denecker *et al.*, 2004) to extend Fitting's semantics to Datalog^{agg} but also the well-founded semantics (and, as we will see in Section 3.2.5, also the stable model semantics). Below,

we summarize these 3 approaches.

Well-founded model via unfounded sets

Kemp and Stuckey, 1991 essentially took over the syntax of groupby subgoals (now referred to as "aggregate subgoals") introduced by Mumick *et al.*, 1990 to use aggregates in Datalog or, more generally, in logic programs. As is noted by Kemp and Stuckey, 1991, the main weakness of the approach by Mumick *et al.*, 1990 is that it imposes restrictions on the Datalog^{agg} programs in the form of some stratification or monotonicity-requirement. In contrast, analogously to Datalog$^{\neg}$, the well-founded semantics is applicable to Datalog^{agg} with unrestricted use of aggregate subgoals. The first step towards the definition of a well-founded semantics of Datalog^{agg} is to define the evaluation of aggregate subgoals in a 3-valued Herbrand interpretation.

Definition 3.13. Consider a 3-valued Herbrand interpretation I and the ground instance A of an aggregate subgoal of the form

$$\mathsf{group\_by}(\mathsf{p}(\mathsf{x}_0, \mathsf{z}), [\mathsf{x}_0], \mathsf{y}_0 = \mathsf{agg}(\mathsf{Expr}(\mathsf{x}_0, \mathsf{z})),$$

where x_0 is an instantiation of the grouping variables, z are the local variables, y_0 is a constant, agg is an aggregate function, and $\mathsf{Expr}(\mathsf{x}_0, \mathsf{z})$ is an expression with free variables z. Then $I \models A$ holds, if

1. for all z, $\mathsf{p}(\mathsf{x}_0, \mathsf{z})$ has one of the values `true` or `false` in I,
2. $\mathsf{agg}(\mathsf{S})$ is defined for the multiset $\mathsf{S} = \{\mathsf{Expr}(\mathsf{x}_0, \mathsf{z}) \mid (\mathsf{p}(\mathsf{x}_0, \mathsf{z})$ is `true` in $I\}$, and
3. $\mathsf{y}_0 = \mathsf{agg}(\mathsf{S})$.

Analogously, $I \models \neg A$ holds, if the third condition in the above definition is changed to "$\mathsf{y}_0 \neq \mathsf{agg}(\mathsf{S})$"

The rationale (and, as Gelder, 1992 points out, a weakness) of this definition is that, according to Kemp and Stuckey, 1991, it only makes sense to take the aggregate over a multiset S if S is fully defined. This means that, if for some instantiation z_0 of z, the atom $\mathsf{p}(\mathsf{x}_0, \mathsf{z}_0)$ has truth value `undef` in I, then also the aggregate atom has truth value `undef` in I.

With the evaluation of aggregate subgoals in place, the definition of the operators T_P, U_P, W_P and of the well-founded semantics W_P^* of Datalog^{agg} programs can be taken over almost literally from the corresponding Definitions 3.4 and 3.5 for Datalog$^{\neg}$ programs:

Definition 3.14. Consider a Datalog^{agg} program P and a partial interpretation I of P. A set $U \subseteq HB(P, D)$ of ground atoms is an *unfounded set* of P w.r.t.

I, if, for every $A \in U$ it holds that every ground instance of a rule r of P with head atom A satisfies (at least) one of the following conditions:

1. some body literal (i.e., positive literal, negative literal or *aggregate subgoal*) of r is `false` in I;

2. some *positive* atom B in the body of r occurs in U.

Definition 3.15. Consider a Datalog^{agg} program P and a partial interpretation I of P. The operators T_P, U_P, and W_P are defined as follows.

- A ground atom A is in $T_P(I)$, if there exists a ground instance of a rule r in P, such that A is the head of r and every literal in the body of r (i.e., positive literal, negative literal, or aggregate subgoal) is `true` in I;

- $U_P(I)$ is the greatest unfounded set of P w.r.t. I;

- $W_P(I) = T_P(I) \cup \neg U_P(I)$, where $\neg U_P(I) = \{\neg B \mid B \in U_P(I)\}$.

The *well-founded semantics* W_P^* of program P is defined as the least fixpoint (w.r.t. $\leq_k$) of W_P.

After having defined the well-founded semantics of Datalog^{agg}, Kemp and Stuckey, 1991 pay considerably more attention to the stable model semantics of Datalog^{agg}. We will come back to this in Section 3.2.5. The following example is given by Kemp and Stuckey, 1991 solely to discuss the stable-model semantics. However, we already introduce this example here to illustrate the weakness of the well-founded semantics as defined by Kemp and Stuckey, 1991, namely: too many atoms are assigned the truth value `undef`.

Example 3.20. In the Profit Sharing problem, a predicate $\mathsf{share}(\mathsf{X}, \mathsf{Y}, \mathsf{S})$ indicates that company X owns a fraction S of company Y. A predicate $\mathsf{profit}(\mathsf{X}, \mathsf{K})$ indicates that the profit of X is K. The dividend company X receives from its share S of company Y with $\mathsf{profit}(\mathsf{Y}, \mathsf{K})$ is $\mathsf{K} * \mathsf{S}$. The profit of a holding company is defined as the sum over all the dividends it receives. For companies not owning shares of other companies, the profit is given as input. The following program describes a setting with 3 companies $\mathsf{a}, \mathsf{b}, \mathsf{c}$, where a, b are holding companies owning shares of c and of one another. Company c does not own shares of other companies and, therefore, its profit is given as part of the input:

$$\begin{aligned}
\mathsf{profit}(\mathsf{X},\mathsf{N}) &\text{ :- } \mathsf{group_by}(\mathsf{dividend}(\mathsf{X},\mathsf{Y},\mathsf{M}),[\mathsf{X}],\mathsf{N}=\mathsf{SUM}(\mathsf{M}))\\
\mathsf{dividend}(\mathsf{X},\mathsf{Y},\mathsf{M}) &\text{ :- } \mathsf{share}(\mathsf{X},\mathsf{Y},\mathsf{S}),\mathsf{profit}(\mathsf{Y},\mathsf{K}),\mathsf{M}=\mathsf{S}*\mathsf{K}\\
\mathsf{profit}(\mathsf{c},100) &\text{ :-}\\
\mathsf{share}(\mathsf{a},\mathsf{b},0.60) &\text{ :-}\\
\mathsf{share}(\mathsf{a},\mathsf{c},0.52) &\text{ :-}\\
\mathsf{share}(\mathsf{b},\mathsf{a},0.20) &\text{ :-}\\
\mathsf{share}(\mathsf{b},\mathsf{c},0.16) &\text{ :-}
\end{aligned}$$

We observe that $\mathsf{profit}(\mathsf{b},\_)$ is initially `undef`. Hence, T_P^3 does not allow us to derive a `true` fact $\mathsf{dividend}(\mathsf{a},\mathsf{b},\_)$ by the second rule. On the other hand, $\mathsf{share}(\mathsf{a},\mathsf{b},0.60)$ is part of the input and will, therefore, never have truth value `false`. Hence, $\mathsf{dividend}(\mathsf{a},\mathsf{b},\_)$ will not end up in an unfounded set. This means, that $\mathsf{dividend}(\mathsf{a},\mathsf{b},\_)$ will retain the truth value `undef` throughout the evaluation of this program. But then, in the first rule, the atom $\mathsf{dividend}(\mathsf{a},\mathsf{b},\_)$ violates the condition in Definition 3.13, that $\mathsf{dividend}(\mathsf{a},\mathsf{z},\_)$ must be defined for all z. Hence, $\mathsf{profit}(\mathsf{a},\_)$ ends up with truth value `undef` in the well-founded model as defined by Kemp and Stuckey, 1991. By the analogous considerations, we conclude that also $\mathsf{profit}(\mathsf{b},\_)$ ends up with truth value `undef`. This is unsatisfactory since the problem has a natural solution via the system of two equations $p_a = 52 + 0.6 * p_b$ and $p_b = 16 + 0.2 * p_a$, where p_a, p_b stand for the profit of a and b, respectively. This system of equations has the solution $p_a = 70$ and $p_b = 30$. We will see in Section 3.2.5, that the stable model semantics defined by Kemp and Stuckey, 1991 indeed produces a unique stable model with $\mathsf{profit}(\mathsf{a},70)$ and $\mathsf{profit}(\mathsf{b},30)$ set to `true`.

Despite this apparent weakness of the way how the well-founded semantics of arbitrary Datalog^{agg} programs is defined by Kemp and Stuckey, 1991, the semantics also enjoys favorable properties. According to Kemp and Stuckey, 1991, the following properties hold:

- For arbitrary Datalog^{agg} programs P, the well-founded model of P is indeed a *model* of P.
- For aggregate stratified and group stratified programs P (we have recalled these definitions of Mumick *et al.*, 1990 in Section 3.2.2), the well-founded model is the perfect model of P.

Well-founded model via alternating fixpoint

In his approach to defining the well-founded semantics of Datalog^{agg} programs, Gelder, 1992 distinguishes between aggregates that can be expressed by first-

order formulas (such as min and max) and those which are defined via some kind of induction principle (such as count and sum). For the former, he proposes a rewriting of the aggregate similar to the approach of Ganguly *et al.*, 1991.

For the min aggregate, the rewriting looks as follows: Suppose that we are given a predicate $p(\boldsymbol{X}, C)$ where $\boldsymbol{X}$ is a vector of variables and C is the cost argument. Moreover, let $\boldsymbol{X}_G \subseteq \boldsymbol{X}$ denote the grouping variables. Then, an aggregate atom $\mathsf{group\_by}(\mathsf{p}(\mathsf{X}, \mathsf{C}), [\mathsf{X_G}], \mathsf{Y} = \mathsf{min}(\mathsf{C}))$ in the syntax used by Kemp and Stuckey, 1991 would be expressed in (Gelder, 1992) as $\mathsf{min_{p,G}}(\mathsf{X}, \mathsf{C})$, whose definition is given by the following Datalog$^\neg$ program:

$$\begin{aligned} \mathsf{min_{p,G}}(\mathsf{X}, \mathsf{C}) &\text{ :- } \mathsf{p}(\mathsf{X}, \mathsf{C}), \neg\, \mathsf{bp}(\mathsf{X_G}, \mathsf{C}) \\ \mathsf{bp}(\mathsf{Y_G}, \mathsf{C}) &\text{ :- } \mathsf{p}(\mathsf{Y}, \mathsf{D}), \mathsf{D} < \mathsf{C} \end{aligned}$$

Intuitively, $\mathsf{bp}(\mathsf{X_G}, \mathsf{C})$ means that predicate p admits a "better" value than C for the group defined by $\mathsf{X_G}$. Then the well-founded model of a Datalog^{agg} program P involving min and max aggregates is defined as the well-founded model of the rewritten Datalog$^\neg$ program. This well-founded model of a Datalog$^\neg$ program is characterized by Gelder, 1992 in terms of the alternating fixpoint iteration A_P.

Recall from Section 3.1.4 that the operator A_P alternatingly computes underestimates and overestimates of the (negated) predicates in P until it reaches the alternating fixpoints. The well-founded model is then obtained by taking the meet w.r.t. $\leq_k$ of the two alternating fixpoints (which are complete interpretations). Gelder, 1992 puts this approach to work by inspecting 3 versions of the APSP program.

Example 3.21. Similarly to Example 3.13, consider predicates $\mathsf{arc}(\mathsf{X}, \mathsf{Y}, \mathsf{C})$ $\mathsf{path}(\mathsf{X}, \mathsf{Y}, \mathsf{I}, \mathsf{D})$ and $\mathsf{spath}(\mathsf{X}, \mathsf{Y}, \mathsf{I}, \mathsf{D})$. However, Gelder, 1992 chooses a slightly different meaning of the two idb predicates. Here, $\mathsf{path}(\mathsf{X}, \mathsf{Y}, \mathsf{I}, \mathsf{D})$ means that there is a path from X to Y of length D with I being the *first* node after X and $\mathsf{spath}(\mathsf{X}, \mathsf{Y}, \mathsf{I}, \mathsf{D})$ means that the shortest path from X to Y over any *first* intermediate node I has length D.[4] In the notation of Gelder, 1992, this minimality condition is expressed by the following rule:

$$\mathsf{spath}(\mathsf{X}, \mathsf{Y}, \mathsf{I}, \mathsf{D}) \text{ :- } \mathsf{min_{path,1,2}}(\mathsf{X}, \mathsf{Y}, \mathsf{I}, \mathsf{D})$$

That is, $\mathsf{spath}(\mathsf{X}, \mathsf{Y}, \mathsf{I}, \mathsf{D})$ minimizes the cost argument D in $\mathsf{path}(\mathsf{X}, \mathsf{Y}, \mathsf{I}, \mathsf{D})$, grouping over the arguments at positions 1 and 2. Then the rewriting presented above looks as follows:

[4] Recall that Ross and Sagiv, 1992 chose I to be the *last* intermediate node. This difference is totally inessential here and the following considerations equally apply to both versions of the idb predicates.

$$\begin{aligned} \mathsf{spath}(\mathsf{X},\mathsf{Y},\mathsf{I},\mathsf{D}) &\text{ :- } \mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{I},\mathsf{D}), \neg\, \mathsf{bpath}(\mathsf{X},\mathsf{Y},\mathsf{D}) \\ \mathsf{bpath}(\mathsf{X},\mathsf{Y},\mathsf{D}) &\text{ :- } \mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{J},\mathsf{C}), \mathsf{C} < \mathsf{D} \end{aligned}$$

The $\mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{I},\mathsf{D})$ predicate is defined by the usual base case plus one of the recursive rules (1) – (3) given below.

$$\begin{aligned} &\mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{Y},\mathsf{D}) \text{ :- } \mathsf{arc}(\mathsf{X},\mathsf{Y},\mathsf{D}) \\ (1)\quad &\mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{I},\mathsf{D}) \text{ :- } \mathsf{path}(\mathsf{X},\mathsf{Z},\mathsf{I},\mathsf{D1}), \mathsf{arc}(\mathsf{Z},\mathsf{Y},\mathsf{D2}), \mathsf{D} = \mathsf{D1} + \mathsf{D2} \\ (2)\quad &\mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{I},\mathsf{D}) \text{ :- } \mathsf{spath}(\mathsf{X},\mathsf{Z},\mathsf{I},\mathsf{D1}), \mathsf{arc}(\mathsf{Z},\mathsf{Y},\mathsf{D2}), \mathsf{D} = \mathsf{D1} + \mathsf{D2} \\ (3)\quad &\mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{I},\mathsf{D}) \text{ :- } \mathsf{spath}(\mathsf{X},\mathsf{Z},\mathsf{I},\mathsf{D1}), \mathsf{spath}(\mathsf{Z},\mathsf{Y},\mathsf{J},\mathsf{D2}), \mathsf{D} = \mathsf{D1} + \mathsf{D2} \end{aligned}$$

That is, variant (1) considers in $\mathsf{path}(\mathsf{X},\mathsf{Y},\mathsf{Y},\mathsf{D})$ *all*, not necessarily simple, paths while variants (2) and (3) are based on the intuitions underlying Dijkstra's and the Floyd-Warshall algorithm, respectively. That is, every shortest path of length ≥ 2 can be obtained (2) as the extension of a shortest path by one more edge, or (3) as the composition of two shortest paths, respectively. Only the first variant is aggregate stratified according to Mumick *et al.*, 1990. However, the well-founded semantics yields the same result for all 3 variants. This is verified by inspecting the alternating fixpoint computation for the variants (2) and (3):

We start with the underestimate of bpath by setting all bpath atoms to `false`. Then all $\neg\,\mathsf{bpath}$ literals are `true` and $\mathsf{cpath}(\mathsf{X},\mathsf{Y},\mathsf{I},\mathsf{D})$ yields all possible paths from X to Y starting with I also in the variants (2) and (3). Clearly, if the graph contains a cycle of non-zero length, then this set is infinite.

Since $\mathsf{cpath}(\mathsf{X},\mathsf{Y},\mathsf{I},\mathsf{D})$ contains *all* paths, it in particular, contains *all shortest* paths. Hence, at the end of the first iteration of the alternating fixpoint operator A_P, the "overestimate" of bpath is actually the correct relation. Therefore, in the second iteration of A_P, $\neg\,\mathsf{bpath}(\mathsf{X},\mathsf{Y},\mathsf{D})$ correctly expresses that there does not exist a path from X to Y that is shorter than D. Hence, if $\mathsf{cpath}(\mathsf{X},\mathsf{Y},\mathsf{I},\mathsf{D})$ exists for this combination of $(\mathsf{X},\mathsf{Y},\mathsf{D})$, then this is indeed a shortest path.

Gelder, 1992 notes that all 3 variants of the program also handle cycles of negative length correctly by setting all atoms $\mathsf{spath}(\mathsf{X},\mathsf{Y},\mathsf{I},\mathsf{D})$ to `undef`, if (X,Y) is part of a cycle of negative length. This is due to the above observation that, after the first iteration of A_P, $\neg\,\mathsf{bpath}(\mathsf{X},\mathsf{Y},\mathsf{D})$ correctly expresses that there does not exist a path from X to Y that is shorter than D. Clearly, if (X,Y) is part of a negative-length cycle, then $\neg\,\mathsf{bpath}(\mathsf{X},\mathsf{Y},\mathsf{D})$ will be `false` for every D. Hence, we will never derive an atom $\mathsf{spath}(\mathsf{X},\mathsf{Y},\_,\_\mathsf{D})$.

We note that Gelder, 1992 is primarily interested in the problem of assigning an intuitive semantics to Datalog^{agg} programs rather than the (efficient) evaluation of such programs. This is, for instance, witnessed by the fact that the potentially infinite models arising in all three variants of the APSP program in Example 3.21 are not further discussed. Moreover, Gelder, 1992 already identified a special case of pre-mappability introduced by Zaniolo *et al.*, 2017 (see Section 3.2.2) but with the opposite motivation. Indeed, rather than aiming to push aggregation into the recursion to achieve a more efficient evaluation, Gelder, 1992 aims at pulling min/max aggregates out of the recursion in order to achieve stratification. Essentially, he proves that if a program is monotone w.r.t. some order $\sqsubseteq$ on the cost argument, then the least upper bound w.r.t. $\sqsubseteq$ can be pulled out of the recursion. This step is motivated by the fact that it "makes the program easier to analyze", as Gelder, 1992 notes.

For aggregates defined via some kind of induction principle (such as count and sum) Gelder, 1992 observes that handling *subsets* of tuples satisfying a certain property is at the heart of defining a monontonic predicate. For instance, in the Company Control problem introduced in Example 3.8, a company A controls a company C if a *set* of companies B_i controlled by A own a combined portion of the shares of C that, together with A's direct ownership of C, exceeds 0.50. In the approach of Kemp and Stuckey, 1991, this intuition would be directly translated into the following goal:

$$\mathsf{group\_by}(\mathsf{controlsStk}(\mathsf{C1},\mathsf{C2},\mathsf{C3},\mathsf{P}),[\mathsf{C1},\mathsf{C3}],\mathsf{Y}=\mathsf{sum}(\mathsf{P})),\mathsf{Y}>0.5$$

However, Gelder, 1992 observes that in this case, the sum aggregate behaves monotonically in that it suffices to find a *subset* of companies B_i controlled by A so that their shares of C together with A's share of C exceed 0.50. If later a further company B controlled by A is detected that owns a share of C, this will never invalidate the previous derivation of a $\mathsf{controls}(\mathsf{A},\mathsf{C})$ atom. Gelder, 1992 identifies aggregating over the *entire set* of atoms (i.e., all possible $\mathsf{controlsStk}(\mathsf{A},\mathsf{B_i},\mathsf{C}.\_)$ atoms in case of the Company Control problem) rather than aggregating over a sufficiently big *subset* as the main weakness in the semantics definition by Kemp and Stuckey, 1991. More specifically, Kemp and Stuckey, 1991 would leave $\mathsf{controls}(\mathsf{A},\mathsf{C})$ `undef` as long as there is some relevant $\mathsf{controlsStk}(\mathsf{A},\mathsf{B_i},\mathsf{C},\_)$ atom `undef`. In contrast, Gelder, 1992 sets $\mathsf{controls}(\mathsf{A},\mathsf{C})$ to `true` as soon as a sufficiently big *subset* of $\mathsf{controlsStk}(\mathsf{A},\mathsf{B_i},\mathsf{C},\_)$ atoms with truth value `true` has been derived.

Well-founded semantics via approximation theory

In Section 3.2.3, we have already seen how Pelov *et al.*, 2007 use an approximation-theoretic framework to define Fitting's semantics of Datalog^{agg} programs.

In the same work, the authors also define the well-founded semantics and (as we will see in Section 3.2.5) the stable model semantics of Datalog^{agg} programs. Analogously to Section 3.2.3, we first recall the ideas of defining stable revision operators and well-founded sets in the general framework of Denecker *et al.*, 2000 and Denecker *et al.*, 2004, before we look at the application of this framework to Datalog^{agg} programs by Pelov *et al.*, 2007.

Recall from Section 3.2.3 that the approximation theory studied by Denecker et al. is centered around a complete lattice $(L, \leq)$ whose elements are approximated by pairs in the bilattice L^2 equipped with the lattice ordering $\leq$ and the information ordering $\leq_i$. Elements $z \in L$ are "approximated" by pairs $(x, y) \in L^2$ with $x \leq z \leq y$. An operator $A\colon L^2 \to L^2$ is an approximation of an operator $O\colon L \to L$ if A is $\leq_i$-monotone and $A(x, x) = (O(x), O(x))$ for every x. Moreover, recall from Section 3.2.3 two important properties of approximations: an approximation (a, b) is *consistent* if $a \leq b$ and it is *A-reliable* for given approximating operator A, if $(a, b) \leq_i A(a, b)$. Intuitively, the latter property means that revising an approximation by an application of A yields a "better" (i.e., more informative, more precise) approximation.

From now on, we are only interested in consistent and A-reliable approximations. For such approximations (a, b) and approximating operators A, the following important property follows from the $\leq_i$-monotonicity of A:

Lemma 3.14. Let L be a complete lattice, let A be an approximating operator on L^2, and let $(a, b) \in L^2$ be consistent and A-reliable. Then the following properties hold:

(1) If $x \in [\bot, b]$, then $A(x, b)_1 \in [\bot, b]$.
(2) If $x \in [a, \top]$, then $A(x, b)_2 \in [a, \top]$.

Proof. The first property is shown by the following chain of equalities and inequalities:

$$A(x,b)_1 \overset{(1)}{\leq} A(b,b)_1 \overset{(2)}{=} A(b,b)_2 \overset{(3)}{\leq} A(a,b)_2 \overset{(4)}{\leq} b$$

Here, (1) holds by the assumption $x \in [\bot, b]$, by the definition of $\leq_i$, and by the monotonicity of A w.r.t. $\leq_i$. That is, from $x \leq b$ it follows that $(x, b) \leq_i (b, b)$ holds and, therefore, also $A(x, b) \leq_i A(b, b)$ holds; hence, the $\leq_i$-smaller pair $A(x, b)$ is smaller in the first component than $A(b, b)$. (2) applies the definition of approximating operators which have to satisfy $A(y, y)_1 = A(y, y)_2$ for every $y \in L$. (3) holds by the monotonicity of A w.r.t. $\leq_i$ and the definition of $\leq_i$. That is, $A(a, b) \leq_i A(b, b)$ holds and the $\leq_i$-smaller pair $A(a, b)$ is greater in the second component than $A(b, b)$. Finally, (4) makes use of the reliability of (a, b), i.e., $(a, b) \leq_i A(a, b)$. Again, this means that the second component b of the smaller pair (a, b) is greater than the second component of $A(a, b)$. The second property stated in the lemma is shown analogously. □

Consider an A-reliable pair (a, b). We can now define an operator $A(\cdot, b)_1$ on $[\bot, b]$ and an operator $A(a, \cdot)_2$ on $[a, \top]$. That is, for the first operator, we fix the second component and only consider the change in the first component when A is applied. Likewise, for the second operator, we fix the first component and only consider the change in the second component when A is applied. The above lemma guarantees that $A(x, b)_1$ is again in $[\bot, b]$ for every $x \in [\bot, b]$, and $A(a, x)_2$ is again in $[a, \top]$ for every $x \in [a, \top]$. Moreover, $A(\cdot, b)_1$ and $A(a, \cdot, b)_2$ have the following monotonicity properties: $A(\cdot, b)_1$ is monotone w.r.t. $\leq$ on $[\bot, b]$ and $A(a, \cdot)_2$ is monotone w.r.t. $\leq$ on $[a, \top]$. Hence, $A(\cdot, b)_1$ and $A(a, \cdot)_2$ have least fixed points in the respective lattice $[\bot, b]$ and $[a, \top]$. This leads to the following definition.

Definition 3.16. Let L be a complete lattice, let A be an approximating operator on L^2, and let $(a, b) \in L^2$ be consistent and A-reliable. We define $b^{A\downarrow}$ as the least fixpoint of $A(\cdot, b)_1$ in $[\bot, b]$ and $a^{A\uparrow}$ as the least fixpoint of $A(a, \cdot)_2$ in $[a, \top]$. Moreover, we call the mapping $\mathcal{S}t_A \colon L^2 \to L^2$ which maps (a, b) to $(b^{A\downarrow}, a^{A\uparrow})$ the *stable revision operator* for A.

That is, the stable revision operator revises a given A-reliable approximation (a, b) in two separate ways by either fixing the upper bound b and computing a new lower bound $b^{A\downarrow}$ or by fixing the lower bound a and computing a new upper bound $a^{A\uparrow}$. Denecker *et al.*, 2004 show the following properties of these two new bounds:

Lemma 3.15. Let $b^{A\downarrow}$ and $a^{A\uparrow}$ as defined above. Then the following properties hold:

- $b^{A\downarrow}$ yields a lower bound on all fixpoints x of O with $x \leq b$;
- $[a, a^{A\uparrow}]$ is closed under the operator O, i.e., if $x \in [a, a^{A\uparrow}]$, then also $O(x) \in [a, a^{A\uparrow}]$.

Note that, in general, it is not guaranteed that $(a, b) \leq_i (b^{A\downarrow}, a^{A\uparrow})$ holds. Denecker *et al.*, 2004 therefore introduce a further restriction on pairs (a, b) of interest: an A-reliable approximation (a, b) is called A-*prudent* if $a \leq b^{A\downarrow}$ holds. With this property, they can prove the following crucial result:

Theorem 3.16. Let L be a complete lattice and A an approximating operator on $L^c \subseteq L^2$ (i.e., the subset of consistent pairs in L^2). Then the set of A-prudent elements of L^c is a chain complete poset w.r.t. $\leq_i$ that contains $(\bot, \top)$ as the minimal element. Moreover, the stable revision operator $\mathcal{S}t_A$ is well-defined, increasing, and monotone on this poset.

This means that the stable revision operator $\mathcal{S}t_A$ has fixpoints and, in particular, a least fixpoint. This least fixpoint is called the *well-founded fixpoint* of A, denoted $\mathsf{wf}(A)$. Moreover, every fixpoint of $\mathcal{S}t_A$ in L^c is called a *stable*

fixpoint of A. It is easy to verify that every stable fixpoint of A is also a fixpoint of A: indeed, for (x, y) to be a fixpoint of $\mathcal{S}t_A$ means that $x = \mathsf{lfp}(A(\cdot, y)_1)$ and $y = \mathsf{lfp}(A(x, \cdot)_2)$ holds. This means, in particular, that $x = A(x, y)_1$ and $y = A(x, y)_2$ holds. That is, $(x, y) = A(x, y)$.

As already recalled in Section 3.2.3, Denecker et al. apply these notions and results from approximation theory to Datalog or, more generally, to logic programming, by considering the complete lattice of Herbrand interpretations K for a given program P with the $\subseteq$-ordering. We have also recalled in Section 3.2.3 that the 3-valued ICO Φ_P in the definition of Fitting's semantics is an approximation of the ICO T_P of P. It turns out that $\mathsf{wf}(\Phi_P)$ is the well-founded model of program P. This follows from results in (Fitting, 1993), which we presented in Section 3.1.5. Recall that Fitting, 1993 proceeds in several steps when studying the relationship between Φ_P and a generalization of the Gelfond-Lifschitz transformation. As was mentioned in Section 3.1.5, Fitting considers 4-valued interpretations allowing also inconsistent truth assignments of both `false` and `true` to a ground atom. However, this is inessential for our considerations since the well-founded model of a Datalog$^\neg$ problem is 3-valued and, hence, consistent.

1. Fitting first extends the evaluation of literals from single (4-valued) interpretations to pairs of interpretations (I_1, I_2) (there referred to as *valuations* (v_1, v_2)) with the idea of interpreting positive literals in I_1 and negative literals in I_2. The combination of two interpretations I_1, I_2 is denoted as $I_1 \Delta I_2$.

2. Fitting, 1993 then defines the *extended immediate consequence operator* Ψ_P, which carries this idea of separating the interpretations of positive and negative parts over to Datalog$^\neg$ programs. W.l.o.g., we may assume that all grounded rules with the same head atom are combined to a single rule. Then, for a ground atom A and interpretations I_1, I_2, we have that $\Psi_P(I_1, I_2)(A)$ is (1) `false` if no rule with head A exists and (2) $\Psi_P(I_1, I_2)(A)$ is set to $(I_1 \Delta I_2)(Body)$ if $A \leftarrow Body$ is the rule with head atom A in the grounded program.

3. It is easy to verify that $\Psi_P(I, I) = \Phi_P(I)$ for every interpretation I. Likewise, the following monotonicity-properties of Ψ_P are easy to verify: Ψ_P is monotone w.r.t. the knowledge ordering $\leq_k$ (which corresponds to the information ordering $\leq_i$ in the terminology of Denecker et al.) in both arguments, it is monotone w.r.t. the truth ordering $\leq_t$ (= the lattice ordering $\leq$) in the first argument and anti-monotone w.r.t. $\leq_t$ in the second argument.

4. Fitting, 1993 then defines the *stability operator* Ψ'_P, which maps a single interpretation J to the least fixpoint w.r.t. $\leq_t$ of the mapping

$(\lambda I)\Psi_P(I, J)$). By the $\leq$-monotonicity of Ψ_P in the first component, this lfp exists.

5. Fitting, 1993 calls a (4-valued) interpretation *stable* if it is a fixpoint of Ψ'_P. He also proves that Ψ'_P is monotone w.r.t. the knowledge ordering $\leq_k$ and, therefore, the least fixpoint of Ψ'_P w.r.t. $\leq_k$ exists, and it is the least (w.r.t. $\leq_k$) stable model of program P. Moreover, as discussed in Section 3.1.5, even though Ψ'_P is defined on 4-valued interpretations, the lfp of Ψ'_P is 3-valued. In fact, this is the well-founded model of P obtained as the smallest (w.r.t. $\leq_k$) stable model as described in Section 3.1.4.

The crux for establishing the relationship with the approximation theory of Denecker et al. is to show that the equality $\Psi'_P = \mathsf{wf}(\Phi_P)$ holds. This equality follows from the above considerations on Ψ' and it is stated in (Denecker *et al.*, 2000) without proof.

The extension of the well-founded semantics by Pelov *et al.*, 2007 to Datalog^{agg} is then straightforward. We have already discussed in Section 3.2.3 how to extend the 3-valued ICO Φ_P in the definition of Fitting's semantics from Datalog$^{\neg}$ to Datalog^{agg}. Then the well-founded semantics of Datalog^{agg} is simply obtained as $\mathsf{wf}(\Phi_P)$, i.e., the well-founded fixpoint as defined by Denecker *et al.*, 2000 and Denecker *et al.*, 2004 applied to the approximating operator Φ_P.

Do we need an example here? E.g., the company control example? I understand that the approximation-based approach by Denecker et al. is a bit hard to digest. On the other hand, the main message here is probably just that Denecker et al. provide yet another approach. But we have already seen 2 ones. So maybe no more details are needed here? –Reinhard.

3.2.5 Aggregation with Stable Model Semantics

In Section 3.2.2 we have seen several approaches of defining a semantics of Datalog^{agg} programs where the main difference was the scope of the various approaches. That is, for which class of Datalog^{agg} programs each approach could provide a semantics definition. However, on classes of programs in the scope of several approaches, the resulting semantics definitions essentially yielded the same result. For stable model semantics, the situation changes completely: Here, all approaches aim at defining an intuitive semantics to arbitrary Datalog^{agg} programs. So the scope is essentially the same (apart from some approaches that even cover more general programs that contain Datalog^{agg} as a special case). However, the resulting definitions of stable model

semantics for Datalog^{agg} may yield significantly different results. Below, we summarize several of these approaches: starting with the pioneering approach by Kemp and Stuckey, 1991 and ending with more general approaches for which Datalog^{agg} programs constitute a special case.

The pioneering approach of Kemp and Stuckey, 1991

Recall from Section 3.2.4 that Kemp and Stuckey, 1991 consider *aggregate subgoals* of the form

$$\mathsf{group\_by}(\mathsf{p}(\mathsf{x}_0, \mathsf{z}), [\mathsf{x}_0], \mathsf{y}_0 = \mathsf{agg}(\mathsf{Expr}(\mathsf{x}_0, \mathsf{z}))$$

We have also recalled in Section 3.2.4 how Kemp and Stuckey, 1991 define the evaluation of a ground instance A of an aggregate subgoal in a 3-valued interpretation. In particular, $I \models A$ or $I \models \neg A$ requires that $\mathsf{p}(\mathsf{x}_0, \mathsf{z})$ has one of the values `true` or `false` in I *for all* z.

In order to define stable models of programs with aggregate subgoals, we only need to consider 2-valued Herbrand interpretations. Kemp and Stuckey, 1991 define the Gelfond-Lifschitz reduct P^M of a Datalog^{agg} program P w.r.t. a subset M of the Herbrand base by treating aggregate subgoals analogously to negative body literals. More specifically, for each ground instance r of a rule in P, do the following:

- if each negative literal *and each aggregate subgoal* in r is satisfied by M, then remove the negative literals and aggregate subgoals from the body of r and add the resulting rule to P^M;
- otherwise, r does not contribute to P^M.

Then, a 2-valued Herbrand interpretation M is a *stable model* of a Datalog^{agg} program P, if M is the least Herbrand model of P^M. Kemp and Stuckey, 1991 observe that, as expected, every *stable model* of a Datalog^{agg} program P is, in particular, also a *model* of P. However, they also observe that stable models of Datalog$^{\neg}$ programs are not necessarily minimal models. And, moreover, stable models may no longer be incomparable w.r.t. $\subseteq$. This is illustrated by the following example:

Example 3.22. Consider the following Datalog^{agg} program from Gelfond and Zhang, 2014 (in the syntax used above):

$$\mathsf{p}(\mathsf{a}) \text{ :- } \mathsf{group\_by}(\mathsf{p}(\mathsf{X}), [\,], 1 \leq \mathsf{COUNT}(\mathsf{X}))$$

First look at the interpretation $M = \{\mathsf{p}(\mathsf{a})\}$. We observe that the aggregate atom evaluates to `true` in M. Hence, according to Kemp and Stuckey, 1991,

in the reduct P^M, we may delete this atom from the body of the rule and we are left with the single-atom rule $\mathsf{p(a)}$:- , which has precisely $M = \{\mathsf{p(a)}\}$ as minimal model. Hence, M is a stable model of P.

However, it is easily verified that also the empty model $\emptyset$ is a stable model of P. Hence, the stable model $\{\mathsf{p(a)}\}$ is not a minimal model of P and the two stable models of P are not incomparable w.r.t. $\subseteq$.

Kemp and Stuckey, 1991 observe that, in the relationship with the well-founded model, stable models behave as expected:

Theorem 3.17. Let P be a Datalog^{agg} program. Then the following properties hold:

- Every stable model M of P is an extension of the well-founded model W_P^* of P, i.e., M is obtained from W_P^* by possibly switching some of the `undef` values to `false` or `true`.
- If P has a 2-valued well-founded model, then it is the unique stable model of P.

These properties can be shown analogously (but now taking aggregate subgoals into account) to the definition of the well-founded model of a Datalog$^\neg$ program as the $\leq_k$-minimal 3-valued stable model recalled in Section 3.1.4.

We conclude our discussion of the stable model semantics according to Kemp and Stuckey, 1991 by revisiting the Profit Sharing problem from Example 3.20. There, we observed that setting the profit of company a to 70 and the profit of company b to 30 would be the intuitive solution obtained by solving the system of linear equations. Nevertheless, the well-founded semantics according to Kemp and Stuckey, 1991 assigns the value `undef` to any atom $\mathsf{profit(a, \_)}$ and $\mathsf{profit(b, \_)}$.

In contrast, the expected solution $M = \{\mathsf{share(a, b, 0.60)}, \mathsf{share(a, c, 0.52)}, \mathsf{share(b, a, 0.20)}, \mathsf{share(b, c, 0.16)}, \mathsf{dividend(a, b, 18)}, \mathsf{dividend(a, c, 52)}, \mathsf{dividend(b, a, 14)}, \mathsf{dividend(b, c, 16)}, \mathsf{profit(a, 70)}, \mathsf{profit(b, 30)}, \mathsf{profit(c, 100)}\}$ is a stable model. To verify this, we inspect the Gelfond-Lifschitz reduct of the only rule involving an aggregate subgoal:

$$\mathsf{profit(X, N)} \text{ :- } \mathsf{group\_by(dividend(X, Y, M), [X], N = SUM(M))}$$

In the ground instance obtained by instantiating X to a and N to 70, the body evaluates to `true`, since summing up the last component of the atoms $\mathsf{dividend(a, b, 18)}$ and $\mathsf{dividend(a, c, 52)}$ indeed yields $\mathsf{N = 70}$. Likewise, the ground instance obtained by instantiating X to b and N to 30, the body evaluates to `true`, Hence, the Gelfond-Lifschitz reduct w.r.t. M contains the two rules "$\mathsf{profit(a, 70)}$:- " and "$\mathsf{profit(b, 30)}$:- " and the minimal model of P^M is M. Therefore, M is indeed a stable model of program P. In fact, it is the unique stable model of P.

The approach of Faber *et al.*, 2011

Faber *et al.*, 2011 propose the integration of aggregates into Datalog[5] by treating aggregate and non-aggregate atoms in a uniform way. Moreover, they present a slightly simplified version of the original Gelfond-Lifschitz reduct, which nevertheless leads to an equivalent definition of stable models when restricted to Datalog$^{\neg}$.

The syntax of aggregate atoms used by Faber *et al.*, 2011 is essentially the same as in (Liu and Stoller, 2022) discussed in Section 3.2.1, i.e., $aggS \odot k$, where agg is an aggregate operator, S is a set expression of the form *odot* for variables X_i and a conjunction B of standard atoms containing all the X_i variables, $\odot$ is a comparison operator and k is a variable or constant. For instance, $\max\{\mathsf{Z}\colon \mathsf{r}(\mathsf{Z}), \mathsf{q}(\mathsf{Z},\mathsf{V})\} > \mathsf{Y}$ is an aggregate atom. A ground instance of it could look like this: $\max\{\langle 2\colon r(2), q(2,a)\rangle, \langle 2\colon r(2), q(2,a)\rangle\} > 1$.

Also the semantics definition of aggregate atoms is very similar to the approach by Liu and Stoller, 2022. The only major difference now is that, for the stable model semantics, only 2-valued interpretations need to be considered. The following example (a shortened and slightly modified version of Example 2.8 in (Faber *et al.*, 2011)) illustrates the syntax and semantics of aggregate atoms in this approach:

Example 3.23. Consider the interpretation $I = \{g(1,2), g(1,3), g(1,4), g(2,4), h(1)\}$. Then we have:

- $\text{count}\{X\colon g(X,Y) \wedge h(X)\} > 2$ is `false` in I, because count is evaluated over the set $\{1,2\}$.
- $\text{count}\{X,Y\colon g(X,Y) \wedge h(X)\} > 2$ is `true` in I, because count is evaluated over the set $\{(1,2),(1,3),(1,4)\}$. Note that, strictly speaking, Faber *et al.*, 2011 first transform this set into the multiset consisting of the first component only and then apply the aggregate function. In this case, we would thus get the multiset $\{\{1,1,1\}\}$. Clearly, for the count aggregate, this transformation into a multiset is irrelevant. In contrast, for aggregates such as $\min, \max, \text{sum}$, it clearly is relevant, as we shall see next.
- $\max\{X,Y\colon g(X,Y) \wedge h(X)\} = 1$ is `true` in I, because the pairs obtained by instantiation are projected to the first component, which is 1 for all pairs in this example.
- $\text{sum}\{X,Y\colon g(X,Y) \wedge h(X)\} \leq 3$ is `true` in I.

[5] Strictly speaking, they even allow disjunctive Datalog, where the rule heads may actually be disjunctions of atoms. But this is outside the scope of our work.

The crucial step in the approach by Faber *et al.*, 2011 is how they define the reduct of a program w.r.t. some interpretation I. Here, the crux is that they delete rules whose body evaluates to false in this interpretation, but they leave the bodies of the remaining rules unchanged. This is in stark contrast to the approach by Kemp and Stuckey, 1991, who treat aggregate atoms in the same way as negative literals. In particular, Kemp and Stuckey, 1991 delete the aggregate atoms which are `true` in I from rule bodies while Faber *et al.*, 2011 leave such atoms untouched. Formally, given a ground program P and an interpretation I, Faber *et al.*, 2011 define the *reduct* of P w.r.t. I (denoted $ground(P)^I$) as $P^I = \{r \mid r \in P$ and all body literals of r evaluate to `true` in $I\}$. Moreover, an interpretation I is a *stable model* (referred to as "answer set" by Faber *et al.*, 2011) of a Datalog^{agg} program P, if it is a subset-minimal model of $ground(P)^I$.

We revisit the program from the previous section where the approach of Kemp and Stuckey, 1991 yielded 2 stable models $\emptyset$ and $\{p(a)\}$. It will turn out that we now only get 1 stable model. In the syntax of Faber *et al.*, 2011, the program looks as follows.

$$\mathsf{p(a)} \text{ :- } \text{count}\{\mathsf{X}\text{: } \mathsf{p(X)}\} \geq 1$$

Clearly, $\emptyset$ is again a stable model. In contrast, $I = \{p(a)\}$ is not. This is due to the fact that the reduct w.r.t. I consists of the only ground rule

$$\mathsf{p(a)} \text{ :- } \text{count}\{\langle \mathsf{a}\text{: } \mathsf{p(a)}\rangle\} \geq 1$$

The minimal model of this program is $\emptyset$. As pointed out by Alviano *et al.*, 2023, the shortcoming of the approach by Kemp and Stuckey, 1991 is that aggregated atoms are treated like negative literals even though they may possibly behave like positive ones. Indeed, in the above example, the aggregate atom $\text{count}\{\langle a\text{: } p(a)\rangle\} \geq 1$ behaves more like $p(X)$ rather than $\neg p(X)$.

Faber *et al.*, 2011 prove the following desirable property of their semantics:

Theorem 3.18. For every Datalog^{agg} program P, in the approach of Faber *et al.*, 2011, every stable model of P is a minimal model of P. Consequently, the stable models of a Datalog^{agg} program P are incomparable.

Proof. Let I be a stable model of P. Then I is a model of all rules in $ground(P)^I$. Moreover, I is also a model of the rules in $ground(P)\backslash ground(P)^I$, since the bodies of these rules evaluate to `false` in I by the definition of the reduct. Hence, I is indeed a model of I.

To prove the minimality, suppose to the contrary that there exists a model J of P with $J \subset I$. By the definition of stable models, I is a subset-minimal model of $ground(P)^I$. Hence, J is not a model of $ground(P)^I$, i.e., some rule $r \in ground(P)^I$ is `false` in J. But r is also a rule in $ground(P)$. Hence, J is not a model of P – a contradiction. □

The approach of Gelfond and Zhang, 2014 via the Vicious Circle Principle

Gelfond and Zhang, 2014 present the knowledge representation language $\mathcal{A}log$, which extends Answer Set Programming with aggregates. The authors aim at providing an intuitive semantics of Datalog^{agg} by referring to the *Vicious Circle Principle* (VCP). They mention that the VCP was already formulated in the early twentieth century by Poincaré to mitigate paradoxes of set theory. In its original form, the VCP states that "no object or property can be introduced by the definition referring to the totality of objects satisfying this property." Clearly, approaches based on stratification satisfy this condition. However, Gelfond and Zhang, 2014 aim at an intuitive semantics of arbitrary Datalog^{agg} programs. They therefore relax the VCP as follows: "$p(a)$ cannot be introduced by the definition referring to a set of objects satisfying p if this set can contain a.

The syntax of aggregated atoms proposed by Gelfond and Zhang, 2014 is quite similar to the one proposed by Faber *et al.*, 2011. The most important difference is that Gelfond and Zhang, 2014 consider the variables *vars* occurring in a set expression $vars : cond$ always as local. In contrast, Faber *et al.*, 2011 consider only the variables not occurring outside the aggregate atom as local, whereas the other variables constitute the grouping variables. Of course, this has a significant effect on grounding a rule with aggregate atoms in the body. A further difference concerning the syntax of the two approaches is that Gelfond and Zhang, 2014 allow only a single occurrence of $vars: cond$ in the set expression whereas Faber *et al.*, 2011 allow multiple such expressions.

The stable models (again referred to a "answer sets" as by Faber *et al.*, 2011) are defined via 2 steps which differ from the approach of Faber *et al.*, 2011: first the *aggregate reduct* of a program P w.r.t. an interpretation I is constructed, which only operates on *aggregate* atoms in P and, as a result, eliminates all aggregate atoms from the program. Hence, the resulting program P' is a Datalog$^{\neg}$ program. It is then possible to define the stable models of the original Datalog^{agg} program P simply as the stable models of the Datalog$^{\neg}$ program P'.

Formally, the *aggregate reduct* of a ground program P w.r.t. a set S of (regular, i.e., not aggregate) ground atoms is obtained from P by (1) deleting from P all rules that contain an *aggregate atom* `false` in S and (2) replacing every remaining *aggregate atom* $agg\{X : p(X)\}$ by the set $\{p(t) : p(t) \in S\}$. The second part of the definition realizes the vicious circle principle in that such a rule can only fire after all ground atoms $p(t)$ with $p(t) \in S$ have been derived.

It is shown in Alviano *et al.*, 2023 that, for all Datalog^{agg} programs, every stable model according to Faber *et al.*, 2011 is also a stable model according to Gelfond and Zhang, 2014 but not vice versa. Hence, the minimality property shown in Theorem 3.18 for stable models according to Faber *et al.*, 2011 also

holds for stable models according to Gelfond and Zhang, 2014. Gelfond and Zhang, 2014 give the following example, which has a stable model according to Faber *et al.*, 2011 but no stable model in the approach of Gelfond and Zhang, 2014.

Example 3.24. Consider the following Datalog^{agg} program P:

$$\begin{aligned}&\mathsf{p(a)} \text{ :- } \mathsf{p(b)}\\&\mathsf{p(b)} \text{ :- } \mathsf{p(a)}\\&\mathsf{p(a)} \text{ :- } \mathrm{count}\{\mathsf{X}\colon \mathsf{p(X)}\} \neq 1\end{aligned}$$

In the approach of Faber *et al.*, 2011, $I = \{p(a), p(b)\}$ is a stable model. That is, the reduct P^I is the same as P and I is indeed a minimal model of P^I. In contrast, P has no stable model at all according to Gelfond and Zhang, 2014. This is due to the fact that the aggregate reduct w.r.t. I replaces the body of the third rule by $\{p(a), p(b)\}$. Since this set contains the head atom, this rule will never fire. Hence, the only stable model of the aggregate reduct of P w.r.t. I is the empty set. Since $I \neq \emptyset$, this means that I cannot be a stable model.

The approximation framework of Pelov *et al.*, 2007

In Sections 3.2.3 and 3.2.4, we have already seen how Pelov *et al.*, 2007 used the approximation-theoretic framework of Denecker *et al.*, 2000 and Denecker *et al.*, 2004 to extend Fitting's semantics and the well-founded semantics to Datalog^{agg}. The extension of the stable model semantics to Datalog^{agg} is analogous. Recall from Section 3.2.3 how pairs in L^2 can be used to *approximate* elements in a complete lattice L and how an operator $A\colon L^2 \to L^2$ can approximate an operator $O\colon L \to L$. In Section 3.2.4, the definition of the *stable revision operator* $\mathcal{S}t_A$ of an approximating operator A has been recalled. Fixpoints of $\mathcal{S}t_A$ are referred to as *stable fixpoints* and the $\leq_i$-least fixpoint of $\mathcal{S}t_A$ is called the *well-founded fixpoint* of A.

Denecker et al. also study the special case of *exact* stable fixpoints, i.e., fixpoints of the form (x, x) of $\mathcal{S}t_A$. They are also referred to as *A-stable fixpoints of O*. When considering the approximation Φ_P of the ICO T_P of a Datalog$^\neg$ program, the Φ_P-stable fixpoints of T_P are precisely the stable models of P recalled in Section 3.1.2. In Section 3.2.4, we have already recalled that the relationship between $\mathsf{wf}(\Phi_P)$ (the well-founded fixpoint of the approximating operator Φ_P according to Denecker et al.) and the well-founded model of a Datalog$^\neg$ program (as discussed in Section 3.1.4) follows from the generalization of the Gelfond-Lifschitz transformation studied by Fitting, 1993. The same holds true for the exact stable models of Φ_P and the stable models of a Datalog$^\neg$ program P (as discussed in Section 3.1.2). In fact, Fitting, 1993 mentions that the *stability operator* Ψ' (which we recalled in Section 3.2.4) generalizes

the Gelfond-Lifschitz transformation. In the terminology and notation used here, this generalization comes down to the following property: let a 2-valued interpretation I be given as a subset of the Herbrand base, let $P' = P^I$ denote the Gelfond-Lifschitz reduct of P w.r.t. I, and let I' be the least fixpoint of $T_{P'}$. Then $\Psi'(I) = I'$ holds. Hence, if $I' = I$ (i.e., I is a stable model of P), then I is a fixpoint of Ψ'.

We have already discussed the extension of Φ_P to Datalog^{agg} programs in Section 3.2.3. Fitting's semantics of a Datalog^{agg} program P is thus obtained as the $\leq_i$-minimal fixpoint of Φ_P and the well-founded model of a Datalog^{agg} program P is obtained as $\mathsf{wf}(\Phi_P)$. Analogously, Pelov *et al.*, 2007 get the set of stable models of a Datalog^{agg} program P as the set of exact stable models of the approximating operator $\mathcal{S}t_A$.

Recall from Section 3.2.3 that the approximation framework of Pelov *et al.*, 2007 leaves a lot of flexibility for the approximation of aggregates. However, as was mentioned in our discussion there, the *ultimate aggregate approximation* proposed by Pelov *et al.*, 2004 as the most precise (aka most informative) approximation of aggregate relations seems to be the preferred one. Actually, it turns out that, for Datalog^{agg} programs, the exact stable models of the approximating operator $\mathcal{S}t_A$ when using *ultimate aggregate approximations* coincide with the stable model semantics of Datalog^{agg} programs proposed by Son and Pontelli, 2007, even though the latter approach, on the surface, seems to take a completely different path: in a nutshell, Son and Pontelli, 2007 define the stable models of a Datalog^{agg} program as follows:

- They define the reduct P^M of a Datalog^{agg} program P w.r.t. a subset M of the Herbrand base in such a way that it only eliminates negative literals and lets the positive literals and aggregate literals untouched.
- To define an ICO for the reduct, the authors first define the notion of "conditional satisfaction", i.e., when checking if an interpretation $I \subseteq M$ satisfies an *aggregate* atom, we also have to take M into account. For regular (i.e., non-aggregate) atoms, the conditional satisfaction is simply defined as the usual satisfaction, i.e., $I \models p$ iff $p \in I$.
- We then get the ICO in the usual way, i.e., add the head of those rules of the reduct to I, whose body is conditionally satisfied by I taking M into account.
- A set M of atoms is indeed an answer set if the above defined ICO of the reduct w.r.t. M yields as lfp again the set M.

As detailed in (Son and Pontelli, 2007), the stable models (referred to as answer sets there) obtained by this process are precisely the exact stable models of the approximating operator $\mathcal{S}t_A$ when using ultimate aggregate approximations.

We conclude our discussion of the approximation framework of Pelov *et al.*, 2007 by briefly looking at the ultimate approximating operator U_P of a program P with negation and/or aggregates. We have already mentioned in Section 3.2.3 that, on the positive side, U_P constitutes the most precise approximation of T_P but, on the negative side, the exact stable models (and also the well-founded model) obtained via U_P and the "usual" stable models of a Datalog$^\neg$ program may differ. Denecker *et al.*, 2004 explain this by the following property of U_P, which is an immediate consequence of its definition, namely: consider two Datalog$^\neg$ programs P, P' such that $T_P = T_{P'}$ holds. Then also $U_P = U_{P'}$ holds, and, therefore, the two programs P and P' give rise to the same well-founded fixpoint and the same exact stable models. However, as is pointed out by Denecker *et al.*, 2004, this property does not hold for the standard notion of well-founded model and stable models of Datalog$^\neg$ programs as presented in Sections 3.1.4 and 3.1.2. This can be seen by inspecting the following 2 programs $P = \{p \leftarrow \top\}$ and $P' = \{p \leftarrow p, p \leftarrow \neg p\}$. Clearly, the ICOs T_P and $T_{P'}$ of the two programs coincide. Howoever, as far as the standard definition of stable models of Datalog$^\neg$ programs is concerned, the program P has the stable model $\{p\}$, while P' has no stable model. In particular, the reduct of P' w.r.t. $\{p\}$ is $p \leftarrow p$, whose minimal model is $\emptyset$. In contrast, both U_P and $U_{P'}$ lead to the exact stable model $\{p\}$. Note that we also have this discrepancy for the well-founded model: The ultimate approximating operator $U_{P'}$ yields the well-founded fixpoint $\{p\}$, whereas the standard definition of the well-founded model would leave p `undef`.

Datalog^{agg} as special case of more general programs

We mention 3 general approaches, which contain Datalog^{agg} as a special case: The approach of Ferraris, 2011, which considers programs with *nested implications* and the approaches of Marek and Remmel, 2004 and Liu *et al.*, 2010, which both consider programs with *set constraints*.

Ferraris, 2011 presents a very general proposal for defining stable models of *nested programs*, i.e., programs where, in addition to aggregates, arbitrary nesting of implications is allowed. Formally, nested programs are constructed from atoms (regular or aggregate atoms), the logical constant $\bot$ (encoding `false`) and the connectives $\wedge, \vee, \rightarrow$. These connectives can be nested arbitrarily. Note that this syntax also captures negation, since a negated formula $\neg F$ can be represented as $F \rightarrow \bot$.

Intuitively, the reduct F^X of a formula F w.r.t. X is defined by replacing each maximal subformula `false` in X by $\bot$. Formally, the reduct F^X is constructed by recursively applying the following rules: (1) $\bot^X = \bot$, (2) for every atom a, we set $a^X = a$ if $X \models a$ and $a^X = \bot$ otherwise, and (3) for a compound formula $\Phi = F \circ G$ with $\circ \in \{\wedge, \vee \rightarrow\}$, we set $\Phi^X = F^X \circ G^X$ if

$X \models \Phi$ and $\Phi^X = \bot$ otherwise. Then the reduct P^X of a program P is defined as the set of reducts of its rules w.r.t. X.

An interpretation I is a stable model of a program P, if it is a minimal model of P^X. Ferraris, 2011 show that, if programs are restricted to Datalog with aggregates but without negation, then the stable models according to the two approaches of Ferraris, 2011 and Faber *et al.*, 2011 coincide. Actually, the discrepancy between the two approaches does not come from negation in general but from negation applied to aggregate atoms. In the semantics according to Faber *et al.*, 2011, we can always get rid of negated aggregate atoms by inverting the comparison operator, e.g., replacing $=$ by $\neq$, replacing $<$ by $\geq$, etc. In the approach of Ferraris, 2011, such a transformation may change the semantics of a program. This is illustrated by the following example from Ferraris, 2011:

Example 3.25. Consider the program P (using the syntax of Faber *et al.*, 2011) consisting of a single rule:

$$\mathsf{p} \text{ :- } \neg(\text{sum}\{1 : \mathsf{p}\} \leq 0)$$

For Faber *et al.*, 2011, this program is equivalent to the program P' consisting of the following rule:

$$\mathsf{p} \text{ :- } (\text{sum}\{1 : \mathsf{p}\} > 0)$$

It is easy to verify that the only stable model of P' is the empty set. In case of P', this is the same for both approaches of Faber *et al.*, 2011 and Ferraris, 2011. In contrast, it is shown in Ferraris, 2011, that according to that semantics, the original program P has two stable models: $\emptyset$ and $\{p\}$. In other words, this also shows that, if negation of aggregate atoms is allowed, then stable models according to Ferraris, 2011 are not necessarily minimal.

Marek and Remmel, 2004 propose a general framework for logic programs with set constraints. This is later further extended by Liu *et al.*, 2010.

The syntax of *set constraint logic programs* (SC-logic programs, for short) is defined by Marek and Remmel, 2004 (and similarly by Liu *et al.*, 2010) as follows: Let X be a set of atoms and let $\mathcal{P}(X)$ denote the power set of X. A *set constraint atom* for the set X is a pair $\langle X, \mathcal{F}\rangle$ with $\mathcal{F} \subseteq \mathcal{P}$. A *set constraint clause* (SC-clause, for short) is an expression of the form $s \leftarrow s_1, \ldots, s_k$, where s and each s_i are set constraint atoms. The body of such a clause is the set of set constraint atoms $\{s_1, \ldots, s_k\}$. A set constraint logic program (SC-logic program, for short) P is a collection of set constraint clauses.

The semantics of SC-logic programs is defined as follows: An interpretation given by a set M of atoms satisfies the set constraint atom $\langle X, \mathcal{F}\rangle$ (written $M \models \langle X, \mathcal{F}\rangle$) if $M \cap X \subseteq \mathcal{F}$. Moreover, M satisfies an SC-clause $s \leftarrow s_1, \ldots, s_k$,

if the fact that M satisfies all set constraint atoms $s_1, \ldots, s_k$ in the body of the clause implies that M also satisfies the set constraint atom s in the head. Finally, M is a model of an SC-logic program P if M satisfies all SC-clauses of P.

Marek and Remmel, 2004 explain that SC-logic programs generalize logic programs with negation and aggregates: For instance, for an atom p, we have $M \models p$ iff $M \models \langle\{p\}, \{p\}\rangle$ and $M \models \neg p$ iff $M \models \neg\langle\{p\}, \{\}\rangle$. Likewise, aggregate atoms using the usual aggregates can be represented as set constraints. For instance, let X be a finite set of (ground) atoms and let $\mu\colon X \to \mathbb{R}$ so that, in case of sum, min, or max, we actually want to compute the sum, min, or max of $\mu(x)$ over all $x \in X$. Then, for real numbers $a \leq b$, aggregate atoms of the form $a \leq \mathit{count}(X) \leq b$, $a \leq \mathit{sum}(X) \leq b$, $a \leq \mathit{min}(X) \leq b$, and $a \leq \mathit{max}(X) \leq b$ can be captured by a set constraint atom $\langle X, \mathcal{F}\rangle$ for appropriately chosen set $\mathcal{F}$. That is, we choose $\mathcal{F}$ as $\{Y \mid a \leq |Y| \leq b\}$, $\{Y \mid a \leq \sum_{y\in Y} \mu(y) \leq b\}$, $\{Y \mid a \leq \min_{y\in Y} \mu(y) \leq b\}$, and $\{Y \mid a \leq \max_{y\in Y} \mu(y) \leq b\}$, respectively.

Also Liu *et al.*, 2010 study logic programs with set constraints. However, their focus is on the process of actually computing a stable model (referred to as answer set). To this end, Liu *et al.*, 2010 identify several principles that apply to the computation of the answer sets of Datalog$^\neg$ programs and define a semantics of their general logic programming language with set constraints based on these principles.

Our interest here is restricted to Datalog^{agg} programs. It is therefore convenient to follow the treatment of the approaches of Marek and Remmel, 2004 and Liu *et al.*, 2010 in the survey article by Alviano *et al.*, 2023.

Recall that the computation of a stable model (aka answer set) of a program P can be divided in two steps: first we guess a candidate answer set X and then we have to validate this candidate. That is, we check if the reduct of P w.r.t. X allows us to derive precisely the ground atoms in X.

Of course, instead of explicitly defining the reduct, one can directly specify conditions under which a rule of the original program P fires, given a candidate answer set X plus a set Y of ground atoms that have already been derived in the process of validating this candidate. When checking if a rule in program P fires, we have to check if all body literals are satisfied by Y (taking into account also X). For regular atoms and their negation, this is clear. However, for an aggregate atom A, the two approaches of Marek and Remmel, 2004 and Liu *et al.*, 2010 differ in the way how they take the candidate answer set X into account.

Following the notation in (Alviano *et al.*, 2023), let $\models_{MR}$ and $\models_{LPST}$ denote the satisfaction relation (of aggregate atoms) according to the approaches of Marek and Remmel, 2004 and Liu *et al.*, 2010, respectively. These satisfaction relations are defined as follows:

- $(Y, X) \models_{MR} A$ if $X \models A$ and there exists an interpretation Z with

$Y \subseteq Z \subseteq X$ such that $Z \models A$.

- $(Y, X) \models_{LPST} A$ if $Z \models A$ holds for every interpretation Z with $Y \subseteq Z \subseteq X$.

The ICO $\mathcal{T}^{MR}_{P,X}$ and $\mathcal{T}^{LPST}_{P,X}$, respectively for validating a candidate answer set X of ground program P is then defined as

$$\mathcal{T}^{\Delta}_{P,X}(Y) = \bigcup_{r \in P, (Y,X) \models_{\Delta} B(r)} H(r),$$

for $\Delta \in \{MR, LPST\}$, where $B(r)$ and $H(r)$ denote the body and head of a rule r, respectively. In (Alviano *et al.*, 2023), the relationships between stable models in the various approaches are compared. In particular, they establish that $GZ \subseteq LPST \subseteq FFLP \subseteq MR$, that is: all stable models according to Gelfond and Zhang, 2014 are stable models according to Liu *et al.*, 2010, which are, in turn, stable models according to Ferraris, 2011 and also according to Faber *et al.*, 2011, which are, in turn, stable models according to Marek and Remmel, 2004. It is also shown by Alviano *et al.*, 2023 that the opposite inclusions to not hold.

3.3 Adding Non-Determinism

We can keep this section short, since it is not the main thread of our survey. –Dan.

Non-determinism is a bit more general than "choice", which is one way of realizing non-determinism. –Reinhard.

Plan

- Giannotti *et al.*, 2001: [from the abstract] The paper studies the semantics and expressive power of various non-deterministic constructs, including the choice operator and the witness operator. The paper establishes an expressiveness hierarchy that correlates the various operators with each other and with other constructs such as negation and fixpoint. The paper thus gives a good overview of previous approaches listed below.
- Krishnamurthy and Naqvi, 1988: possibly the earliest proposal of a choice operator in Datalog
- Abiteboul and Vianu, 1991: [from the abstract:] The focus is on two closely related families of non-deterministic languages. The first consists of extensions of Datalog with negations in bodies and/or heads of rules,

with non-deterministic fixpoint semantics. The second consists of non-deterministic extensions of first-order logic and fixpoint logics, using the w tness operator.

- Giannotti *et al.*, 1997: claims that the importance of non-determinism has been recognized only recently; → give an overview of recent approaches; relates non-deterministic constructs to stable-model semantics
- Giannotti and Pedreschi, 1998: [from the abstract] We prove that Datalog augmented with the dynamic choice expresses exactly the non-deterministic time-polynomial queries. We thus obtain a complete characterization of the expressiveness of the dynamic choice, and conversely achieve a characterization of the class of non-deterministic time-polynomial queries (NDB-PTIME) by means of a simple, declarative, and efficiently implementable language.
- Greco *et al.*, 1995: more on complexity and the choice operator.
- Vianu, 2021: in his PODS 2021 invited tutorial, Victor Vianu also devotes one section to Non-Determinism in Datalog.
- Further (early) works by Zaniolo et al.: Saccà and Zaniolo, 1990 and Greco *et al.*, 1992; the latter seems to have been significantly extended/improved in Greco and Zaniolo, 1998.Ã¤

3.4 Allowing Existentials in the Rule Heads

Should we call it "Adding Existentials in the Rule Heads"? We can also keep this section short. My sense is that the intricacies of datalog$^{\pm}$ are too complex and too recent to describe here in detail. I suggest that we just illustrate how existentials lead to non-termination, maybe describe some base-line solutions (weak acyclicity?), then have a short, non-technical discussion of the literature. –Dan.

Another natural way to extend Datalog is to allow existentials in the head. We will call this extension *existential Datalog*, or Datalog$^{\exists}$. A rule in Datalog$^{\exists}$ is of the form $\exists \mathbf{x}.\mathsf{R}(\mathbf{x},\mathbf{y})$:- $\phi(\mathbf{y},\mathbf{z})$, where $\phi(\mathbf{y},\mathbf{z})$ is a conjunction of atoms over the variables $\mathbf{y}$ and $\mathbf{z}$. A major application of Datalog$^{\exists}$ is reasoning about incomplete data. For example, given the tables in Figure 3.4, the rule $\exists \mathsf{s}.\mathsf{Exam}(\mathsf{id},\mathsf{s})$:- $\mathsf{Student}(\mathsf{id},\mathsf{n})$ says that there must be an exam score for every student. In other words, the student ID is a foreign key between the tables. The tables in Figure 3.4 do not satisfy this rule, because there is no exam score for Chris. In other words, the database is *incomplete*. Evaluating the Datalog$^{\exists}$ rule *completes* the database by marking the missing exam score as

unknown, and the database administrators can choose to act on unknown values in different ways. For example, they may look up the score by other means; replace the score with a default value; or remove the student from the database.

The domain of a Datalog$^{\exists}$ program contains the set of data values as well as a set of *marked nulls*, and every tuple ranges over both values and nulls. The input to a Datalog$^{\exists}$ program is a database of relations over such tuples, and the program computes another set of such relations. When we interpret Datalog$^{\exists}$ rules as logical constraints, there can be many output databases satisfying the constraints. For the exam score example, assigning any score to Chris would satisfy the foriegn key constraint. However, making up a score from thin air would likely upset Chris or their classmates. In general, we should avoid introducing new information when completing a database. The ideal completion should contain exactly the information from the original database as well as the constraints, but nothing more.

A seminal paper by identified *homomorphisms* as a way to compare the information content of two databases. Given two database instances I_1 and I_2 over domain D[6] , a homomorphism from I_1 to I_2 is a mapping $h : D \to D$ satisfying the following properties:

1. $h(c) = c$ for every constant c.
2. $(h(t_1), \ldots, h(t_k)) \in I_2$ for every tuple $(t_1, \ldots, t_k) \in I_1$.

Intuitively, h may replace marked nulls in I_1 with constants or other marked nulls, so that the resulting database is contained in I_2. I_1 contains less information than I_2 because giving values to unknowns in I_1 recovers I_2's tuples.

Using the above notion of homomorphisms, defined the *universal* solution of a set of Datalog$^{\exists}$ rules over an input instance I as follows:

Definition 3.17. Given a set of Datalog$^{\exists}$ rules P and an input instance I, the universal solution $P(I)$ satisfies the following properties:

1. $P(I)$ contains all tuples in I.
2. $P(I)$ satisfies all rules in P interpreted as constraints.
3. For any other instance I' satisfying the above two properties, there exists a homomorphism h from $P(I)$ to I'.

Universal solutions of a Datalog$^{\exists}$ program may not be unique, but they are unique up to homomorphism.

We can compute the universal solution in a very similar way to the naïve evaluation of Datalog programs: simply apply each rule until reaching a fixpoint. When a rule does not contain any existential variables, we apply it the same

[6]Recall the domain contains both data values and marked nulls.

way we do with a Datalog rule. Otherwise, given an existential rule of the form $\exists \mathbf{x}.R(\mathbf{x}, \mathbf{y})$:- $\phi(\mathbf{y}, \mathbf{z})$, we first check if $\phi(\mathbf{y}, \mathbf{z})$ holds in the current instance, and bind the variables $\mathbf{y}$ if so. Then, we check if there is already a tuple $R(\mathbf{x}, \mathbf{y})$ in the current instance. If so, we move to the next rule; otherwise we introduce a fresh marked null, null_i, for each $x_i \in \mathbf{x}$, and add $\mathsf{R}(\mathsf{null}_1, \ldots, \mathsf{null}_\mathsf{k}, \mathbf{y})$ to the instance.

```
changed = true ;
while changed do
    changed = false ;
    foreach ∃x.R(x, y) :- φ(y, z) ∈ P do
        if I ⊨ φ(b, c) for some b, c then
            if R(a, b) ∈ I for some a then
                continue
            end
            else
                nulls = null_1, ..., null_arity(a);   // fresh nulls
                I = I ∪ R(nulls, b) ;
                changed = true ;
            end
        end
    end
end
```

Algorithm 3: The Chase algorithm.

In general, a $\mathsf{Datalog}^\exists$ program may not terminate. Consider the rule $\exists \mathsf{z}.\mathsf{R}(\mathsf{y}, \mathsf{z})$:- $\mathsf{R}(\mathsf{x}, \mathsf{y})$. Starting with $\mathsf{R} = \{(1, 2)\}$, applying the rule once produces the tuple $(2, \mathrm{null}_1)$; applying again (on the new tuple) produces $(\mathrm{null}_1, \mathrm{null}_2)$, and the next application produces $(\mathrm{null}_2, \mathrm{null}_3)$, etc. It is also undecidable to check if a $\mathsf{Datalog}^\exists$ program terminates on a given input instance. However, if a $\mathsf{Datalog}^\exists$ program terminates, it always returns a universal solution.

Other variants of the chase have been developed to improve the efficiency of the algorithm and the quality of the output. For example, the *unrestricted Chase* eliminates the check on line **??**, at the cost of returning a non-universal solution. The Skolem Chase improves upon this by constructing new null values with Skolem functions. For example, an existential rule $\exists \mathbf{x}.\mathsf{R}(\mathbf{x}, \mathbf{y})$:- $\phi(\mathbf{y}, \mathbf{z})$ becomes $\mathsf{R}(\mathsf{f}_\mathsf{R}(\mathbf{y}), \mathbf{y})$:- $\phi(\mathbf{y}, \mathbf{z})$, where f_R is a Skolem function. The Skolem Chase always returns a universal solution when it terminates, but there are programs that terminate under the standard Chase but not under the Skolem

Chase. Finally, another variant, called the *core Chase*, is less efficient but computes a more compact solution called the *core*. The core is the smallest universal solution, and is unique up to isomorphism (recall that universal solutions are only unique up to homomorphism). The algorithm for the core Chase is the same as the standard Chase, except that after each iteration, the core Chase replaces the current instance with its core. Another advantage of the core Chase is that it is the only Chase variant that can find a finite universal model when one exists:

Theorem 3.19. A $\mathsf{Datalog}^{\exists}$ program P over input I has a finite universal model if and only if the core Chase terminates on P and I.

Nevertheless, computing the core is NP-complete, and it remains active research to find new Chase algorithms that are both efficient and return compact results.

Because Chase termination is undecidable in general, researchers have proposed various syntactic restrictions to ensure termination. One such well-known condition is *weak acyclicity*: Moving beyond termination, researchers have also studied the complexity of evaluating $\mathsf{Datalog}^{\exists}$ programs. The $\mathsf{Datalog}^{\pm}$ family of languages have proposed many fragments of $\mathsf{Datalog}^{\exists}$ that are decidable and have low data complexity.

3.5 Allowing Equality in the Rule Heads

Datalog systems that support existentials in the rule heads often also support equality in the rule heads. An *equality rule* has the form $\mathsf{x} = \mathsf{y}$:- $\phi(\mathsf{x}, \mathsf{y})$, and has the intuitive meaning that constrains x and y to be the same, if $\phi(x, y)$ holds. Equality rules can be handled along side the existential rules in the Chase algorithm, as shown by the pseudocode below. For each equality rule that matches, each head variable may be bound to either a constant or a marked null. If both variables are mapped to the same constant, the algorithm continues to the next rule. If the variables are mapped to different constants, the algorithm returns **failure**, because the equality constaint is violated. Otherwise at least one variable will be bound to a marked null, in which case we replace it with the value bound to the other variable everywhere in the current instance. In other words, we *unify* the variables with each other.

Adding equality constraints to the Chase does not weaken the guarantees of the algorithm: the standard Chase still computes a universal model, and the core Chase still computes the core. However, additional equality constraints may change the termination behavior of the program, as demonstrated by the following examples.

Plan

- Overview of $\mathsf{Datalog}^{\pm}$ Languages: see Figure 3.5.

foreach $\mathsf{x} = \mathsf{y} \text{ :- } \phi(\mathsf{x}, \mathsf{y}, \mathbf{z}) \in P$ **do**
  if $I \models \phi(a, b, \mathbf{c})$ *for some* $a, b, \mathbf{c}$ **then**
    if *a and b are constants* **then**
      if $I \models a = b$ **then**
        continue
      end
      else
        return **failure**
      end
    end
    else if *a is a marked null* **then**
      $I = I[a \mapsto b]$
    end
    else if *b is a marked null* **then**
      $I = I[b \mapsto a]$
    end
    changed = **true** ;
  end
end

Algorithm 4: The Chase algorithm.

ID	Name
1	Alice
2	Bob
3	Chris

(a) Student names

ID	Score
1	100
2	90

(b) Exam scores

Figure 3.4: Data about students and exams.

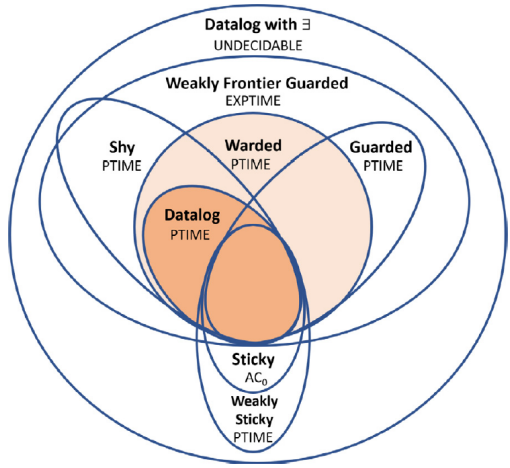


Figure 3.5: Overview of Datalog$^{\pm}$ languages and their data complexity Bellomarini *et al.*, 2022

- References for the various sublanguages:
  - Calì *et al.*, 2013: guarded fragment; proof of undecidability as soon as a single non-guarded rule is there.
  - Calì *et al.*, 2012a: not sure if useful for our purposes. Mainly advertises the use of Datalog for the Semantic Web community by showing that various Description Logics can thus be captured.
  - Calì *et al.*, 2010: looks like a useful overview paper (invited talk?): also mentions Weakly-guarded and linear Datalog$^{\pm}$; moreover the paper gives an alternative, possibly more intuitive definition of sticky Datalog$^{\pm}$.
  - Gottlob and Pieris, 2015: warded Datalog$^{\pm}$ as the basis of the Vadalog system: Bellomarini *et al.*, 2018
  - Calì *et al.*, 2012b: introduce sticky Datalog$^{\pm}$ and also sticky join Datalog$^{\pm}$ and weakly sticky Datalog$^{\pm}$.
  - Baget *et al.*, 2011a: introduce a new framework for identifying decidable classes of existential rules; for our purposes most relevant is probably the introduction of weakly frontier-guarded Datalog$^{\pm}$ as an extension of frontier-guarded Datalog$^{\pm}$.
  - Baget *et al.*, 2011b: study the complexity of several fragments of Datalog$^{\pm}$, namely frontier-one, frontier-guarded and weakly

frontier-guarded rules, with respect to com- bined complexity (with bounded and unbounded predicate arity) and data complexity.

– Leone *et al.*, 2012: shy fragment of $\mathsf{Datalog}^{\pm}$

3.6 Equality and Equivalence

Remy: can you add some historical context here. Briefly discuss eqsat, its history, then explain how Zhang *et al.*, 2023 unified eqsat with datalog. Also, mention briefly EGDs and the chase, and give references. –Dan.

Many applications of Datalog require reasoning with equality and equivalence. For example, a primary key constraint in a database states that a pair of tuples must be equal if they have the same primary key. For example, the following rule asserts that the first attribute of R is a key:

$$(\mathsf{X} = \mathsf{Y}) \text{ :- } \mathsf{R}(\mathsf{Key}_1, \mathsf{X}) \wedge \mathsf{R}(\mathsf{Key}_2, \mathsf{Y}) \wedge (\mathsf{Key}_1 = \mathsf{Key}_2)$$

We will prefer to use a named predicate $\mathsf{Eq}(\mathsf{X}, \mathsf{Y})$ to indicate equality, and the rule above becomes:

We should use $\wedge$ instead of comma. –Dan.

$$\mathsf{Eq}(\mathsf{X}, \mathsf{Y}) \text{ :- } \mathsf{R}(\mathsf{Key}_1, \mathsf{X}) \wedge \mathsf{R}(\mathsf{Key}_2, \mathsf{Y}) \wedge \mathsf{Eq}(\mathsf{Key}_1, \mathsf{Key}_2) \tag{3.2}$$

Such constraints are called *equality-generating dependencies* (EGDs) in the literature. For another example, in pointer analysis, our goal is to determine which variables may point to the same memory location. Such variables are called *aliases*[7]. When performing pointer analysis in Datalog, we may store aliases in an *equivalence relation*. For example, Nappa *et al.*, 2019 contains the following rule:

$$\mathsf{Alias}(\mathsf{y}, \mathsf{p}) \text{ :- } \mathsf{Store}(\mathsf{x}, \mathsf{f}, \mathsf{y}) \wedge \mathsf{Load}(\mathsf{p}, \mathsf{q}, \mathsf{f}) \wedge \mathsf{Alias}(\mathsf{x}, \mathsf{q}) \tag{3.3}$$

The rule encodes the following reasoning: given the progam statements `x.f = y;` (represented by $\mathsf{Store}(\mathsf{x}, \mathsf{f}, \mathsf{y})$) and `p = q.f;` (by $\mathsf{Load}(\mathsf{p}, \mathsf{q}, \mathsf{f})$), if `x` and `q` are aliases, then so are `y` and `p`.

[7] We consider "alias analysis" a synonym of "pointer analysis" as is common in the literature.

The naïve way to represent an equivalence relation in Datalog is to declare the relation to be reflexive, symmetric, and transitive using the following rules:

$$\mathsf{Eq}(\mathsf{X},\mathsf{X}). \tag{3.4}$$

$$\mathsf{Eq}(\mathsf{X},\mathsf{Y}) \text{ :- } \mathsf{Eq}(\mathsf{Y},\mathsf{X}). \tag{3.5}$$

$$\mathsf{Eq}(\mathsf{X},\mathsf{Z}) \text{ :- } \mathsf{Eq}(\mathsf{X},\mathsf{Y}) \wedge \mathsf{Eq}(\mathsf{Y},\mathsf{Z}). \tag{3.6}$$

Doing so requires every Datalog rule to "join across" the equivalence relation. For example, in (3.2), we need to join with $\mathsf{Eq}(\mathsf{Key}_1, \mathsf{Key}_2)$ even though $\mathsf{Key}_1, \mathsf{Key}_2$ are considered equal. This complicates the encoding and leads to inefficiency. Another performance issue is this naïve encoding of the equivalence relation has quadratic size in the number of equivalent values, when there are linear space data structures for equivalence relations such as the union-find data structure (Tarjan, 1975).

The first step to address these issues is to replace the naïve encoding of equivalence relations with a specialized relation using efficient data structures. Nappa *et al.*, 2019 proposes to extend the Soufflé Datalog engine (Jordan *et al.*, 2016) with a parallel equivalence relation using the union-find data structure. Instead of explicitly writing down the rules (3.4)–(3.6), the user annotates a relation with `eqrel` to declare it an equivalence relation. The Datalog engine then automatically maintains the equivalence relation in the union-find data structure which is joined with other relations during evaluation.

When we can treat equivalent values as the same, we may further simplify and speed up the Datalog program. For example, the rule (3.2) becomes:

$$\mathsf{X} \equiv \mathsf{Y} \text{ :- } \mathsf{R}(\mathsf{Key},\mathsf{X}) \wedge \mathsf{R}(\mathsf{Key},\mathsf{Y}). \tag{3.7}$$

where the head $\mathsf{X} \equiv \mathsf{Y}$ means to *unify* values bound to the variables X and Y. With this approach, explicit equivalence relations in the body are no longer needed, leading to cleaner encoding and faster execution. This is the approach taken by systems implementing the classic Chase algorithm (Abiteboul *et al.*, 1995; Aho *et al.*, 1979; Maier *et al.*, 1979; Maier *et al.*, 1981; Fagin, 1982; Beeri and Vardi, 1984). We will discuss the Chase in more detail later, and focus on supporting equivalence in Datalog for the moment.

There are different ways to unify equivalent values. One may simply replace one value with the other in the entire database, which can be inefficient when we need to unify frequently. Zhang *et al.*, 2023 proposes to leverage the union-find data structure to speed up unification. During each iteration of evaluating the Datalog program, each pair of values to be unified are added to the same equivalence class in the union-find data structure. At the end of the iteration when all rules have been applied, every value in an equivalence class is then replaced with the representative of that class. At this point, all equivalent values have been unified, and the next iteration starts. This approach is more

efficient, because it amortizes the cost of unification. We will now present a simplified version of this approach which we call Datalog$^{\equiv}$.

3.6.1 Datalog$^{\equiv}$

This section uses too frequently the pronoun "we". Instead of "we do", "we propose", "we define", we should have citations: cite(...) describes the algorithm blah. –Dan.

This is actually new content that hasn't been published before. Datalog$^{\equiv}$ shares the spirit of Zhang *et al.*, 2023 but the semantics are different. –Remy.

In this section we present Datalog$^{\equiv}$, a language extending Datalog with equivalence. Datalog$^{\equiv}$ adds a single new syntactic construct to Datalog: the head of a rule can be of the form $\mathsf{X} \equiv \mathsf{Y}$ (but such an atom cannot appear in the body). The semantics of a Datalog$^{\equiv}$ program can be defined in terms of another Datalog program. To "de-sugar" Datalog$^{\equiv}$ to Datalog, we first define a new relation $\mathsf{Eq}(\mathsf{X}, \mathsf{Y})$ and add the following rules to the program:

$$\mathsf{Eq}(\mathsf{X}, \mathsf{X}). \tag{3.8}$$
$$\mathsf{Eq}(\mathsf{X}, \mathsf{Y}) \text{ :- } \mathsf{Eq}(\mathsf{Y}, \mathsf{X}). \tag{3.9}$$
$$\mathsf{Eq}(\mathsf{X}, \mathsf{Z}) \text{ :- } \mathsf{Eq}(\mathsf{X}, \mathsf{Y}) \wedge \mathsf{Eq}(\mathsf{Y}, \mathsf{Z}). \tag{3.10}$$

The rules declare Eq to be reflexive, symmetric, and transitive, and thus an equivalence relation. Note the first rule (3.8) is not grounded – we interpret X to range over all values in the other relations, i.e., the active domain. Alternatively, we introduce a set of rules for each relation R:

$$\begin{aligned}
&\mathsf{Eq}(\mathsf{X}, \mathsf{X}) \text{ :- } \mathsf{R}(\mathsf{X}, _, _, \ldots).\\
&\mathsf{Eq}(\mathsf{Y}, \mathsf{Y}) \text{ :- } \mathsf{R}(_, \mathsf{Y}, _, \ldots).\\
&\mathsf{Eq}(\mathsf{Z}, \mathsf{Z}) \text{ :- } \mathsf{R}(_, _, \mathsf{Z}, \ldots).\\
&\ldots
\end{aligned}$$

Next, for every relation R we introduce a rule: $\mathsf{R}(\mathsf{X}', \mathsf{Y}', \mathsf{Z}', \ldots) \text{ :- } \mathsf{R}(\mathsf{X}, \mathsf{Y}, \mathsf{Z}, \ldots) \wedge \mathsf{Eq}(\mathsf{X}, \mathsf{X}') \wedge$ Because we consider equivalent values as the same, the above rule "closes" every relation with equivalence. Then, we replace each head of the form $\mathsf{X} \equiv \mathsf{Y}$ with $\mathsf{Eq}(\mathsf{X}, \mathsf{Y})$. Finally, for every variable X in the body of a rule, we label each occurrence of X with a unique identifier X_i, and add the following atoms to the body:

$$\mathsf{Eq}(\mathsf{X}_1, \mathsf{X}_2) \wedge \mathsf{Eq}(\mathsf{X}_2, \mathsf{X}_3) \wedge \cdots \wedge \mathsf{Eq}(\mathsf{X}_{n-1}, \mathsf{X}_n)$$

In other words, the relations in the body now join *across* Eq.

The Datalog$^{\equiv}$ program returns a relation $\equiv$ that is the same as Eq(X, Y). Every IDB relation R computed by Datalog$^{\equiv}$, for which we denote R$^{\equiv}$, relates to the corresponding relation R in the Datalog program, which we denote R^D, as follows:

$$R^{\equiv}(X, Y, \ldots) \implies R^D(X, Y, \ldots)$$
$$R^D(X, Y, \ldots) \implies \exists X' \equiv X, Y' \equiv Y, \ldots : R^{\equiv}(X', Y', \ldots)$$

This is not very clear. Are all relations in Datalog$^{\equiv}$ superscripted with $\equiv$? How exactly is Datalog$^{\equiv}$ constructed from the original Datalog program? –Dan.

$R^{\equiv}$ refers to "the R relation computed by the Datalog$^{\equiv}$ program", which is related to the R relation computed by the Datalog program as shown above. (I also added a superscript to R to make it clearer.) Datalog$^{\equiv}$ is not constructed from the original Datalog program. It's the other way – we construct a Datalog program from a Datalog$^{\equiv}$ program to explain the semantics of Datalog$^{\equiv}$. –Remy.

In other words, R$^{\equiv}$ "factors out" the equivalence relation from each relation to store R in a compressed form. Joining R$^{\equiv}$ with $\equiv$ on each attribute recovers the relation R.

3.6.2 Naïve Evaluation for Datalog$^{\equiv}$

To evaluate Datalog$^{\equiv}$ program efficiently, we implement the $\equiv$ relation using the union-find data structure, and keep only one value per equivalence class in each IDB relation. Algorithm **??** shows the naïve evaluation of a Datalog$^{\equiv}$ program.

The program is hard to read at this point. Suggestion: (1) Define earlier the reflexive, symmetric, transitive closure of a relation $\equiv$ as $(\equiv)^*$, then line 3 becomes $\equiv_{i+1} = (\equiv_i \cup T_{\equiv}(I_i))^*$; don't use + instead of $\cup$. (2) Define earlier $T_{\equiv}$, T_P, when you introduce the construction of Datalog$^{\equiv}$ from the Datalog$^{\equiv}$ program. Then, the only undefined operator is $\mathcal{C}$. –Dan.

Will fix this. –Remy.

We start by initializing the equivalence relation $\equiv$ and each IDB relation in I to be empty. Then, we repeatedly apply the rules to update the relations

until they stop changing. Within each iteration, we first update the equivalence relation $\equiv$ by running every rule with a head of the form $\mathsf{X} \equiv \mathsf{Y}$ (line **??**).

Does Datalog$^{\equiv}$ still have rules with head $\equiv$? I thought they were replaced with Eq. –Dan.

Yes. I only use Eq to explain the semantics. –Remy.

We will call such a rule an $\equiv$-rule, and all other rules non-$\equiv$ rules. $T_{\equiv}$ is the immediate consequence operator of the $\equiv$-rules, defined in the usual way. Note that we combine the result with the previous equivalence relation $\equiv_i$ with "+", which means to union the relations and take the equivalence closure. This entire operation can be implemented by calling `union` on the head values whenever an $\equiv$-rule fires. Next (line **??**), we run all other rules to compute the update to the IDB relations, where T_P is the ICO of the non-$\equiv$ rules. Note that line **??** and line **??** are independent – we might as well interleave an $\equiv$-rule with a non-$\equiv$ rule, or run all rules in parallel. Finally (line **??**), we *canonize* the updates using the new equivalence relation before adding them to the IDB relations. This is done by calling $\mathcal{C}(\equiv_{i+1}, I')$, which replaces every value with the representative of its equivalence class. For efficiency, the representative is usually chosen to be leader of each class in the union-find; but for simplicity we assume the value domain to be totally ordered, and use the smallest value in each equivalence class as the representative. Formally:

$$\begin{aligned}
\mathcal{C}(\equiv, v) &= \min\{v' \mid v' \equiv v\} \\
\mathcal{C}(\equiv, (v_1, v_2, \ldots)) &= (\mathcal{C}(\equiv, v_1), \mathcal{C}(\equiv, v_2), \ldots) \\
\mathcal{C}(\equiv, R) &= \{\mathcal{C}(\equiv, t) \mid t \in R\} \\
\mathcal{C}(\equiv, I) &= \{\mathcal{C}(\equiv, R) \mid R \in I\}
\end{aligned}$$

where v is a value, R is a relation, and I is a set of relations. We will write $\mathcal{C}_i(I)$ for $\mathcal{C}(\equiv_i, I)$, or drop the subscript when it is clear from the context. We now present some useful properties of $\mathcal{C}$.

Proposition 3.6.1. The following properties hold for $\mathcal{C}$:

$$\mathcal{C}(\equiv, I) = \mathcal{C}(\equiv, \mathcal{C}(\equiv', I)) \text{ if } \equiv' \subseteq \equiv \tag{3.11}$$

$$\mathcal{C}(\equiv, T_P(I)) \subseteq \mathcal{C}(\equiv, T_P(\mathcal{C}(\equiv, I))) \tag{3.12}$$

Proof. (3.11) follows from the fact that min is idempotent and is monotone over $\subseteq$. To see why (3.12) holds, consider tuples $t_1, t_2, \ldots, t_k$ in I as well as a gound rule t :- $t_1, t_2, \ldots, t_k$. If the rule fires, then $\mathcal{C}(t) = \mathcal{C}(t_1), \mathcal{C}(t_2), \ldots, \mathcal{C}(t_k)$ must also fire, because $\mathcal{C}$ preserves equality between values. Therefore, whenever t is in $T_P(I)$ (and thus $\mathcal{C}(t)$ in $\mathcal{C}(\equiv, T_P(I))$), $\mathcal{C}(t)$ must also be in $\mathcal{C}(\equiv, T_P(\mathcal{C}(\equiv, I)))$. □

The naïve evaluation for classic Datalog is *inflationary*, i.e., the IDB relations monotonically grows with each iteration. This is no longer true for Datalog$^{\equiv}$, as values are replaced with their representatives after each iteration. Nevertheless, we can show that Datalog$^{\equiv}$ is inflationary "modulo $\equiv$", which means that for every tuple t at iteration i, there is an equivalent tuple (namely, $\mathcal{C}_{i+1}(t)$) at iteration $i+1$. Formally:

Theorem 3.20 (Datalog$^{\equiv}$ is inflationary modulo $\equiv$)**.** $\forall i : \mathcal{C}_{i+1}(I_i) \subseteq I_{i+1}$.

Proof. We prove by induction on i. The theorem holds at $i = 0$ because $\mathcal{C}_1(I_0) = I_0 = \emptyset$. For $i \geq 1$:

$$\begin{aligned}
\mathcal{C}_{i+1}(I_i) &= \mathcal{C}_{i+1}(\mathcal{C}_i(T_P(I_{i-1}))) && \text{by definition of } I_i \\
&\subseteq \mathcal{C}_{i+1}(\mathcal{C}_i(T_P(\mathcal{C}_i(I_{i-1})))) && \text{by 3.12} \\
&= \mathcal{C}_{i+1}(T_P(\mathcal{C}_i(I_{i-1}))) && \text{by 3.11} \\
&\subseteq \mathcal{C}_{i+1}(T_P(I_i)) && \text{by IH \& monotonicity of } T_P, \mathcal{C} \\
&= I_{i+1} && \text{by definition of } I_{i+1}
\end{aligned}$$

□

Does the theorem imply that Datalog$^{\equiv}$ terminates? In PTIME? –Dan.

Yes. –Remy.

3.6.3 Semi-naïve Evaluation for Datalog$^{\equiv}$

We should refer to Algorithm 2 in Chapt. 2 when we mention the differences. –Dan.

We now present a semi-naïve evaluation algorithm for Datalog$^{\equiv}$ where we incrementally maintain the IDB relations. Algorithm **??** shows the pseudocode for the algorithm.

We start by initializing the $\equiv$ relation, the IDB relations, and the delta relations to be empty. Note that there is no delta relation for $\equiv$, because the difference of two equivalence relations is not necessarily an equivalence relation, and so we cannot store it in the union-find data structure. In each iteration, we first compute the updates to $\equiv$ by running the delta $\equiv$-rules $\delta T_{\equiv}$, defined in the same way as for the semi-naïve evaluation of Datalog. Next, we use the delta non-$\equiv$ rules δT_P to compute the updates to the IDB relations. Unlike in classic Datalog, here we compute I_{i+1} before ΔI_{i+1}. The reason is that $\delta T_P(I_i, \Delta I_i)$ is not the only source of new tuples – canonizing both $\delta T_P(I_i, \Delta I_i)$ and I_i may also introduce new tuples. We therefore first canonize the union of I_i and $\delta T_P(I_i, \Delta I_i)$ to compute I_{i+1}, before subtracting

the old relation I_i from it to compute ΔI_{i+1}. An efficient implementation may compute ΔI_{i+1} on-the-fly while computing $\delta T_P(I_i, \Delta I_i)$ and calling $\mathcal{C}$.

We now show the correctness of the semi-naïve algorithm, by showing that it computes the same result as the naïve algorithm. We will denote results computed by the naïve algorithm as $\equiv^N$ and I^N, and those by the semi-naïve algorithm as $\equiv^S$ and I^S.

Theorem 3.21 (Semi-naïve evaluation is correct). $\forall i.(\equiv_i^S, I_i^S) = (\equiv_i^N, I_i^N)$.

Proof. We prove by induction on i. It's easy to show the theorem holds for $i = 0, 1$. Assuming it holds for i, we prove it holds for $i + 1$. First we present some facts about the relations before iteration i. Because $\Delta I_i \stackrel{\text{def}}{=} I_i - I_{i-1}$, we have:

$$I_{i-1} \supseteq I_i - \Delta I_i \tag{3.13}$$

$$T_P(I_{i-1}) \supseteq T_P(I_i - \Delta I_i) \text{ and same for } T_{\equiv} \tag{3.14}$$

$$T_P(I_{i-1}) \cup \delta T_P(I_i, \Delta I_i) \supseteq T_P(I_i - \Delta I_i) \cup \delta T_P(I_i, \Delta I_i) \tag{3.15}$$

$$\supseteq T_P(I_i) \text{ and same for } T_{\equiv} \tag{3.16}$$

3.14 follows from the monotonicity of T_P; 3.16 holds for the same reason that semi-naïve evaluation is correct for classic Datalog.

We now show the semi-naïve algorithm computes the same $\equiv$ relation for every interation. Recall that $+$ unions two equivalence relations and takes the equivalence closure. We use $(R)^{\equiv}$ to denote the equivalence closure of a relation R.

$$\begin{aligned}
\equiv_{i+1}^S &\stackrel{\text{def}}{=} \equiv_i^S + \delta T_{\equiv}(I_i^S, \Delta I_i^S) \\
&\quad \text{by inductive hypothesis and definition of } \equiv_i^N \\
&= \left(\equiv_{i-1}^N + T_{\equiv}(I_{i-1}^N)\right) + \delta T_{\equiv}(I_i^S, \Delta I_i^S) \\
&\quad \text{by properties of } + \text{ and } (\cdot)^{\equiv} \\
&= \left(\equiv_{i-1}^N \cup T_{\equiv}(I_{i-1}^N) \cup \delta T_{\equiv}(I_i^S, \Delta I_i^S)\right)^{\equiv} \\
&\quad \text{by 3.16} \\
&= \left(\equiv_{i-1}^N \cup T_{\equiv}(I_{i-1}^N) \cup \delta T_{\equiv}(I_i^S, \Delta I_i^S) \cup T_{\equiv}(I_i^N)\right)^{\equiv} \\
&\quad \text{because } \delta T_{\equiv}(I_i^S, \Delta I_i^S) \subseteq T_{\equiv}(I_i^N) \\
&= \left(\equiv_{i-1}^N \cup T_{\equiv}(I_{i-1}^N) \cup T_{\equiv}(I_i^N)\right)^{\equiv} \\
&\quad \text{by definition of } \equiv_i \text{ and properties of } (\cdot)^{\equiv} \\
&= \equiv_i^N + T_{\equiv}(I_i^N) = \equiv_{i+1}^N
\end{aligned}$$

The proof for $I^S_{i+1} = I^N_{i+1}$ essentially follows the same steps:

$$
\begin{aligned}
I^S_{i+1} \overset{\text{def}}{=} & \mathcal{C}_{i+1}\left(I^S_i \cup \delta T_P(I^S_i, \Delta I^S_i)\right) \\
& \text{by inductive hypothesis and definition of } I^N_i \\
= & \mathcal{C}_{i+1}\left(\mathcal{C}_i(T_P(I^N_{i-1})) \cup \delta T_P(I^S_i, \Delta I^S_i)\right) \\
& \text{because } \mathcal{C} \text{ distributes over } \cup \text{ and by 3.11} \\
= & \mathcal{C}_{i+1}\left(T_P(I^N_{i-1}) \cup \delta T_P(I^S_i, \Delta I^S_i)\right) \\
& \text{by 3.16} \\
= & \mathcal{C}_{i+1}\left(T_P(I^N_{i-1}) \cup \delta T_P(I^S_i, \Delta I^S_i) \cup T_P(I^N_i)\right) \\
& \text{because } \delta T_P(I^S_i, \Delta I^S_i) \subseteq T_P(I^N_i) \\
= & \mathcal{C}_{i+1}\left(T_P(I^N_{i-1}) \cup T_P(I^N_i)\right) \\
& \text{by 3.11} \\
= & \mathcal{C}_{i+1}\left(\mathcal{C}_i(T_P(I^N_{i-1})) \cup T_P(I^N_i)\right) \\
& \text{by 3.20} \\
= & \mathcal{C}_{i+1}\left(T_P(I^N_i)\right) = I^N_{i+1}
\end{aligned}
$$

□

3.7 XY-Stratification

As we have seen in the previous sections, using aggregations and negation is unavoidable in Datalog program. Section 3.1.1 introduced stratified Datalog in order to handle Datalog programs with certain restrictions on the usage of aggregation and negation. The basic idea is to construct the predicate dependance graph of Datalog program and make sure that there is no cycle caused by a negation/aggregation rule.

We have also seen other classes of Datalog programs that are not stratified. However, they can be stratified in the runtime based on the values of one of their attributes. These programs are referred to as locally stratified programs.

Example 3.26. Consider the following program that represents the Batch Gradient Decent (BGD) for a linear regression model:

$$\mathsf{Model}(\mathsf{T},\mathsf{C},\mathsf{P}) \text{ :- } \mathsf{T}=0, \mathsf{Model0}(\mathsf{T},\mathsf{C},\mathsf{P}) \tag{3.17}$$

$$\mathsf{Predict}(\mathsf{T},\mathsf{Id},\mathsf{Sum}(\mathsf{Y})) \text{ :- } \mathsf{Model}(\mathsf{T},\mathsf{C},\mathsf{P}), \mathsf{Xtrain}(\mathsf{Id},\mathsf{C},\mathsf{V}), \mathsf{Y}=\mathsf{V}*\mathsf{P} \tag{3.18}$$

$$\begin{aligned}\mathsf{Gradient}(\mathsf{T},\mathsf{C},\mathsf{Sum}(\mathsf{G})) \text{ :- } & \mathsf{Predict}(\mathsf{T},\mathsf{Id},\mathsf{Y}'), \mathsf{Ytrain}(\mathsf{Id},\mathsf{Y}) \\ & ,\mathsf{Xtrain}(\mathsf{Id},\mathsf{C},\mathsf{V}), \mathsf{G}=2*(\mathsf{Y}-\mathsf{Y}')*\mathsf{V}\end{aligned} \tag{3.19}$$

$$\mathsf{Model}(\mathsf{T}+1,\mathsf{C},\mathsf{P}') \text{ :- } \mathsf{Model}(\mathsf{T},\mathsf{C},\mathsf{P}), \mathsf{Gradient}(\mathsf{T},\mathsf{C},\mathsf{G}), \mathsf{P}'=\mathsf{P}-(\mathsf{A}*\mathsf{G}/\mathsf{N}) \tag{3.20}$$

This program is not stratified, as in its PDG there is a cycle between the predicates Predict, Gradient, Model with two aggregate edges. However, this program is locally stratified because one can stratify the program in runtime by unrolling the values of the attribute T.

In general, statically detecting whether a Datalog program is locally stratified or not is undecidable (Cholak and Blair, 1994; Palopoli, 1992). Thus, here we introduce a class of locally stratifiable Datalog program the detection of which is decidable.

Definition 3.18. A Datalog program is XY-stratified whenever for all the mutually recursive rules, the following holds:

- Every recursive predicate has a distinguished temporal argument.
- Each rule is either X-rule or Y-rule, where
  - A rule is X-rule if the temporal argument in its every recursive predicate is the same (e.g., T).
  - A rule is Y-rule if (i) the temporal argument of the head is $\mathsf{T}+1$, (ii) at least one predicate in the body has the temporal argument of T, and (iii) the rest of predicates have the temporal argument of T or $\mathsf{T}+1$.
- The program after the priming transformation is stratified.

The first two conditions are syntactic and can be checked by statically analyzing the input program. The last condition requires a transformation which is presented next.

Definition 3.19. The priming process for a set of mutually recursive rules is as follows:

- Rename all recursive predicates in an X-rule with the postfix of prime (e.g., Pred to Pred′).

- Rename all recursive predicates in an Y-rule with the same temporal argument as the head with the postfix of prime.
- Remove the temporal argument from all the recursive predicates.

Example 3.27. In the BGD example, the last three rules are mutually recursive and all recursive predicates have a distinguished temporal argument. The first two of them are X-rules, because all recursive predicates share the same temporal argument. The last one is a Y-rule, because the head has $\mathsf{T}+1$ and at least one of the predicates in the body has T.

The priming process produces the following program:

$$\mathsf{Predict}'(\mathsf{Id},\mathsf{Sum}(\mathsf{Y})) \text{ :- } \mathsf{Model}'(\mathsf{C},\mathsf{P}),\mathsf{Xtrain}(\mathsf{Id},\mathsf{C},\mathsf{V}),\mathsf{Y}=\mathsf{V}*\mathsf{P} \tag{3.21}$$

$$\mathsf{Gradient}'(\mathsf{C},\mathsf{Sum}(\mathsf{G})) \text{ :- } \mathsf{Predict}'(\mathsf{Id},\mathsf{Y}'),\mathsf{Ytrain}(\mathsf{Id},\mathsf{Y}),\ \mathsf{Xtrain}(\mathsf{Id},\mathsf{C},\mathsf{V}),\mathsf{G}=2*(\mathsf{Y}-\mathsf{Y}')*\mathsf{V} \tag{3.22}$$

$$\mathsf{Model}'(\mathsf{C},\mathsf{P}') \text{ :- } \mathsf{Model}(\mathsf{C},\mathsf{P}),\mathsf{Gradient}(\mathsf{C},\mathsf{G}),\mathsf{P}'=\mathsf{P}-(\mathsf{A}*\mathsf{G}/\mathsf{N}) \tag{3.23}$$

The program is obviously stratified, as it has no longer any recursive predicate.

Thus, the initial Datalog program is XY-stratified.

3.8 Further Readings

4

The Sum-Product Abstraction

By the late 1980s and early 1990s, it was recognized that there is a very expressive and powerful abstraction that can be used to capture a massive number of problems in constraint satisfaction, relational databases, logic, probabilistic inference, graph theory, information theory, signal processing, digital communications, and discrete optimization (Shenoy and Shafer, 1988; Shafer and Shenoy, 1991). The abstraction was powerful due to two main reasons: the algebraic formulation of the problem, and the common algorithmic technique used to solve the problem.

There were many different names given to this problem, including *local computation* (Lauritzen and Spiegelhalter, 1988), *marginalization the product function* (Aji and McEliece, 2000), and *sum-product computation* (Dechter, 1997), and another slightly more general *valuation algebra* (Kohlas and Shenoy, 2000). We shall use SumProd to refer to this problem. As shall be evident shortly, a SumProd query is a very natural generalization of a conjunctive query. Moreover, as we shall see in Chapter 5, this problem shares an algebraic commonality with Datalog.

Since SumProd is defined over an algebraic structure called *semiring*, we will start this chapter with a brief introduction to semirings in Section 4.1. We will then define the SumProd problem in Section 4.2 and present a host of example problems from many different domains that can be modeled as a SumProd problem.

In Section 4.3 and 4.4, we present two complementary "meta" algorithms for SumProd, one based on backtracking search, the other on dynamic pro-

gramming. The backtracking search algorithm can be shown to be "worst-case optimal" in some precise sense, and it is tightly connected with the notion of "worst-case optimal joins" in relational database query processing. The dynamic programming technique is a powerful algorithmic technique called *variable elimination*.

Finally, Section 4.5 explains how *tree-decomposition* and *message-passing* algorithms are related to variable elimination, and how the back-tracking search algorithm can be used as a building block for answering general SumProd queries.

4.1 Semirings

If we look with a magnifying glass at how a standard relational query operates we notice that it combines tuples using two basic primitives: it *joins* tuples during the computation of joins or cartesian products, and it *unions* tuples during duplicate elimination or in order to express a union operator. An algebraic structure that supports these two basic operations is called a semiring.

Formally, a *semiring* is a tuple $\boldsymbol{S} = (D, \oplus, \otimes, \mathbf{0}, \mathbf{1})$, where:

- D is a set.
- $(D, \oplus, \mathbf{0})$ is a commutative monoid, meaning that $\oplus$ is associative, commutative, and has identity $\mathbf{0}$,
- $(D, \otimes, \mathbf{1})$ is a (not necessarily commutative) monoid,
- $\otimes$ distributes over $\oplus$: $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$ and similarly $(y \oplus z) \otimes x = (y \otimes x) \oplus (z \otimes x)$, and,
- $\mathbf{0}$ is absorptive, meaning $\mathbf{0} \otimes x = x \otimes \mathbf{0} = \mathbf{0}$

Readers familiar with rings will recognize that a semiring is like a ring but misses the additive inversion. When the absorption rule is dropped, then the semiring is called a *pre-semiring*. When the multiplicative operator $\otimes$ is commutative, then $\boldsymbol{S}$ is called a *commutative (pre-)semiring*. If $\boldsymbol{S}$ and $\boldsymbol{S}'$ are two pre-semirings, then a *homomorphism* is a function $h : D \to D'$ that commutes with the semiring operations, $h(x \oplus y) = h(x) \oplus' h(y)$ and $h(x \otimes y) = h(x) \otimes' h(y)$, and also preserves the identities, meaning $h(\mathbf{0}) = \mathbf{0}'$ and $h(\mathbf{1}) = \mathbf{1}'$.

The following are classic examples of semirings:

- The Boolean semiring: $\boldsymbol{B} = (\{0, 1\}, \vee, \wedge, 0, 1)$.
- The natural numbers with the standard operations, $(\mathbb{N}, +, \times, 0, 1)$ and, similarly, the real numbers $\mathbb{R}$.
- The semiring of non-negative real numbers $(\mathbb{R}_+, +, \times, 0, 1)$.

- The tropical semiring: $\mathsf{Trop} = (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$. Here "addition" is $x \oplus y = \min(x, y)$ and "multiplication" is $x \otimes y = x + y$. We note that some texts define the tropical semiring over $\mathbb{N} \cup \{\infty\}$ (Gunawardena, 1998) or over $\mathbb{R} \cup \{\infty\}$ (Gondran and Minoux, 2008). Some texts use the nation Trop^+ to emphasize that the semiring is over non-negative numbers. We will use the simpler notation Trop in this chapter.
- If $\boldsymbol{S}$ is a semiring, then for any $n \in \mathbb{N}$, the set of $n \times n$ $\boldsymbol{S}$-matrices forms a non-commutative semiring, denoted $\boldsymbol{S}^{n \times n}$, where addition and multiplication are the usual matrix operations, and the identities are $\mathbf{0}_n$ and I_n respectively (the zero-matrix, and the identity matrix).

  Notice that, if $\boldsymbol{S}$ is a pre-semiring instead of a semiring, then $\boldsymbol{S}^{n \times n}$ is neither a pre-semiring nor a semiring, because it does not have an identity for multiplication: $AI_n \neq A$, since $\mathbf{0} \cdot a_{ij} \neq \mathbf{0}$, and therefore $\boldsymbol{S}^{n \times n}$. Instead, the following weaker identity holds: $A(I_n + B) = A + AB$, see (Lehmann, 1977).

The semiring of *polynomials* $\mathbb{N}[\boldsymbol{x}]$ plays a special role in our discussion in the next chapter, and we introduce it here in some detail. Fix a set of N variables $\boldsymbol{x} = \{x_1, \ldots, x_N\}$. Then $(\mathbb{N}[\boldsymbol{x}], +, \cdot, 0, 1)$ is the semiring of *formal polynomials* with coefficients in $\mathbb{N}$, where each element $f \in \mathbb{N}[\boldsymbol{x}]$ is a formal polynomial (e.g. $f = 3x_1^3 x_3 x_4 + x_1 x_2^4$), and where addition and multiplication are defined in the standard way. When we discuss polynomials we denote variables by lower case, to distinguish them from the logical variables in datalog, which are denoted by upper case.

We say that $\mathbb{N}[\boldsymbol{x}]$ is the *freely generated commutative semiring* by the set $\boldsymbol{x}$ because it enjoys the following property. For any commutative semiring $\boldsymbol{S}$ and any function $h : \boldsymbol{x} \to \boldsymbol{S}$, there exists a homomorphism of semirings $\bar{h} : \mathbb{N}[\boldsymbol{x}] \to \boldsymbol{S}$ that extends h: $h(x) = \bar{h}(x)$ for all $x \in \boldsymbol{x}$. The function $\bar{h}$ is simply the evaluation function, which evaluates any polynomial on the values $h(x_1), \ldots, h(x_N)$. For a simple example, consider the polynomial:

$$f(x_1, x_2, x_3) = 2x_1^3 x_2 + x_2^2 x_3^5$$

Fix a commutative semiring $\boldsymbol{S}$, and let $h : \boldsymbol{x} \to S$ map the variables x_1, x_2, x_3 to the values $a_1, a_2, a_3 \in S$. Then, the value of the polynomial is:

$$f(a_1, a_2, a_3) = (a_1^3 \otimes a_2) \oplus (a_1^3 \otimes a_2) \oplus (a_2^2 \otimes a_3^5)$$

where we used the common convention $a_1^3 \stackrel{\text{def}}{=} a_1 \otimes a_1 \otimes a_1$.

Of course, $\mathbb{N}[\boldsymbol{x}]$ is a commutative semiring. The freely generated *non-commutative* semiring also exists: it consists of formal polynomials whose monomials are strings in $\boldsymbol{x}^*$. For example, $f = 3x_1 x_3 x_1 x_1 + 5x_1 x_1 x_2$. We do not need the freely non-commutative semiring in this chapter, and will not consider it any further.

4.2 The Sum-Product Problem

We are now ready to define the SumProd problem (or query). Let $\mathcal{H} = (V, \mathcal{E})$ denote a hypergraph, where V is a set of variables. The domain of a variable $v \in V$ is denoted by $\mathsf{Dom}(v)$; abusing notations, for $S \subseteq V$, we define

$$\mathsf{Dom}(S) := \prod_{v \in S} \mathsf{Dom}(v).$$

To each hyperedge $S \in \mathcal{E}$, there corresonds a "factor" function

$$\psi_S : \prod_{v \in S} \mathsf{Dom}(v) \to D,$$

where D denotes a common domain of these functions (such as $D = \mathbb{R}$, or $D = \{\texttt{true}, \texttt{false}\}$). For each subset S of variables, we use $\boldsymbol{x}_S \in \prod_{v \in S} \mathsf{Dom}(v)$ to denote a tuple of values over the domains of variables in S.

Definition 4.2.1 (The SumProd problem)**.** The SumProd *problem*, also called SumProd *query* interchangeably, is to evaluate the following query (i.e. to compute the output factor $\varphi(\boldsymbol{x}_F)$), given the input factors ψ_S, $S \in \mathcal{E}$:

$$\varphi(\boldsymbol{x}_F) := \bigoplus_{\boldsymbol{x}_{V \setminus F} \in \mathsf{Dom}(V \setminus F)} \bigotimes_{S \in \mathcal{E}} \psi_S(\boldsymbol{x}_S) \tag{4.1}$$

where the sum $\oplus$ and $\otimes$ operators are such that $(D, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ is a semiring.

In words, the query aggregates away[1] all variables except those in F. The variables in F are called *free* variables, and the variables in $V \setminus F$ are called *bound* variables.

Example 4.2.2 (Conjunctive query)**.** When the semiring is a Boolean semiring, the SumProd query is precisely a *conjunctive* query (i.e. a Datalog rule), where the summation corresponds to existential quantification, the product is a conjunction, and the factors are input relations.

From this observation, the connection to relational databases, logic programming and constraint satisfaction is straighforward. Essentially, each factor ψ_S represents a *constraint* over the variables S, the product $\otimes$ becomes a conjunction of those constraints, and the summation $\oplus$ is an existential quantification which asks whether the constraints can be satisfied together.

We next present a few examples illustrating the power of this abstraction in expressing problems in a variety of domains. *Many* more examples can be found in (Pearl, 1989; Aji and McEliece, 2000; Kohlas and Wilson, 2008; Wainwright and Jordan, 2008; Khamis *et al.*, 2016) and references thereoff.

[1] Or "marginalize away", in the language of probabilistic graphical models.

Example 4.2.3 (Matrix Chain Multiplication)**.** Given a series of matrices $\boldsymbol{A}_1, \ldots, \boldsymbol{A}_n$ over some field $\mathbb{F}$, where the dimension of $\boldsymbol{A}_i$ is $p_i \times p_{i+1}$, $i \in [n]$, we wish to compute the product $\boldsymbol{A} = \boldsymbol{A}_1 \cdots \boldsymbol{A}_n$. This problem is a SumProd problem, set up as as follows. There are $n+1$ variables $X_1, \ldots, X_{n+1}$ with domains $\mathsf{Dom}(X_i) = [p_i]$, for $i \in [n+1]$. For each $i \in [n]$, the matrix $\boldsymbol{A}_i$ is a function of two variables

$$\psi_{i,i+1} : \mathsf{Dom}(X_i) \times \mathsf{Dom}(X_{i+1}) \to \mathbb{F},$$

where $\psi_{i,i+1}(x, y) = (\boldsymbol{A}_i)_{xy}$. The problem is to compute the output function

$$\varphi(x_1, x_{n+1}) = \sum_{x_2 \in \mathsf{Dom}(X_2)} \cdots \sum_{x_n \in \mathsf{Dom}(X_n)} \prod_{i=1}^{n} \psi_{i,i+1}(x_i, x_{i+1}).$$

Example 4.2.4 (SAT and #SAT, CSP and #CSP)**.** Let φ be a CNF formula over n Boolean variables $V = \{X_1, \ldots, X_n\}$. Let $\mathcal{H} = (V, \mathcal{E})$ be the hypergraph of φ. Then, each clause of φ is a factor ψ_S, and the question of whether φ is satisfiable is the same as evaluating the constant function

$$\varphi = \bigvee_{\boldsymbol{x}} \bigwedge_{S \in \mathcal{E}} \psi_S(\boldsymbol{x}_S).$$

To solve #SAT, we change the input factors ψ_S to map to $\{0, 1\}$, and the semiring to be sum-product:

$$\varphi = \sum_{\boldsymbol{x}} \prod_{S \in \mathcal{E}} \psi_S(\boldsymbol{x}_S).$$

The more general problems CSP and #CSP are encoded similarly.

Example 4.2.5 (Probabilistic graphical models)**.** We consider *probabilistic graphical models* (PGMs) on discrete finite domains. Without loss of generality, consider only *undirected graphical models* (i.e. *Markov random fields*, *factor graphs*). Directed PGMs can be turned into this undirected form by the process of *moralizing* (Wainwright and Jordan, 2008).

The model can be represented by a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where there are n discrete random variables $X_1, \ldots, X_n$ on finite domains $\mathsf{Dom}(X_1), \ldots, \mathsf{Dom}(X_n)$ respectively, and m *factors* (also called *potential functions*):

$$\psi_S : \prod_{i \in S} \mathsf{Dom}(X_i) \to \mathbb{R}_+.$$

Typically, we want to *learn* the model and perform *inference* from the model. For example, we might want to (1) Compute the marginal distribution of some set of variables, (2) Compute the conditional distribution $p(\boldsymbol{x}_A \mid \boldsymbol{x}_B)$ of some

set of variables A given specific values to another set of variables $\boldsymbol{x}_B$, or (3) Compute $\arg\max_{\boldsymbol{x}_A} p(\boldsymbol{x}_A \mid \boldsymbol{x}_B)$ (for MAP queries, for example).

When we condition on some variables, we can restrict the factors to only those entries that match the conditioned variables. It is obvious that the first two questions above are special cases of the SumProd problem on the sum-product semiring. The third question is on the max-product semiring (Wainwright and Jordan, 2008).

Example 4.2.6 (Permanent)**.** #SAT is #**P**-complete. Another canonical #**P**-complete problem is Permanent, which is the problem of evaluating the *permanent* $\mathsf{perm}(\boldsymbol{A})$ of a given *binary* square matrix $\boldsymbol{A}$. Let S_n denote the symmetric group on $[n]$, then the permanent of $\boldsymbol{A} = (a_{ij})_{i,j=1}^n$ is defined as follows.

$$\mathsf{perm}(\boldsymbol{A}) := \sum_{\pi \in S_n} \prod_{i=1}^{n} a_{i\pi(i)}.$$

The Permanent problem can be written in the sum-product form as follows. Say the input matrix is $\boldsymbol{A} = (a_{ij})$, there is a singleton factor ψ_i for each vertex i, where $\psi_i(j) = a_{ij}$. Then, there are $\binom{n}{2}$ factors ψ_{jk} for $j \neq k \in [n]$, where $\psi_{jk}(x, y) = 1$ if $x \neq y$ and 0 if $x = y$. The problem is to evaluate the constant function

$$\varphi = \sum_{\boldsymbol{x}} \prod_{i \in [n]} \psi_i(x_i) \prod_{(j \neq k)} \psi_{jk}(x_j, x_k).$$

Example 4.2.7 (Discrete Fourier Transform)**.** Recall that the discrete Fourier transform (DFT) is the matrix vector multiplication where $A_{xy} = e^{i2\pi \frac{x \cdot y}{n}}$. For this example, we will consider the case when $n = p^m$ for some prime p and integer $m \geq 1$. Recall that the DFT is defined as follows:

$$\varphi(x) = \sum_{y=0}^{n-1} b_y \cdot e^{i2\pi \frac{x \cdot y}{n}}.$$

Write $x = \sum_{i=0}^{m-1} x_i \cdot p^i$ and $y = \sum_{j=0}^{m-1} y_j \cdot p^j$ in their base-p form. Then we can re-write the transform above as follows:

$$\varphi(x_0, x_1, \ldots, x_{m-1}) = \sum_{(y_0, \ldots, y_{m-1}) \in \mathbb{F}_p^m} b_y \cdot e^{i2\pi \frac{\sum_{0 \leq j+k \leq 2m-2} x_j \cdot y_k \cdot p^{j+k}}{n}}.$$

Recalling that $n = p^m$, note that for any $j + k \geq m$, we have $e^{i2\pi \frac{x_j \cdot y_k \cdot p^{j+k}}{n}} = 1$. Thus, the above is equivalent to

$$\varphi(x_0, x_1, \ldots, x_{m-1}) = \sum_{(y_0, \ldots, y_{m-1}) \in \mathbb{F}_p^m} b_y \cdot \prod_{0 \leq j+k < m} e^{i2\pi \frac{x_j \cdot y_k}{p^{m-j-k}}}.$$

The above immediately suggests the following reduction to SumProd. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V} = \{X_0, X_1, \ldots, X_{m-1}, Y_0, Y_1, \ldots, Y_{m-1}\}$ and $\mathcal{E}$ has an edge (X_j, Y_k) for every $j, k \in \{0, 1, \ldots, m-1\}$ such that $j + k < m$. Further, there is another edge $(Y_0, Y_1, \ldots, Y_{m-1})$. The variable domains are $\mathsf{Dom}(X_i) = \mathsf{Dom}(Y_i) = \{0, 1, \ldots, p-1\}$. For every $j, k \in \mathbb{F}_p$ such that $j+k < m$, the corresponding factor

$$\psi_{X_j, Y_k} : \mathbb{F}_p^2 \to \mathbf{D}$$

is defined as

$$\psi_{X_j, Y_k}(x, y) = e^{i2\pi \frac{x \cdot y}{p^{m-j-k}}}.$$

Finally, the factor $\psi_{\boldsymbol{Y}} : \mathbb{F}_p^m \to \mathbf{D}$ is defined as

$$\psi_{\boldsymbol{Y}}(y_0, y_1, \ldots, y_{m-1}) = b_{(y_0, y_1, \ldots, y_{m-1})}.$$

Then the output is

$$\varphi(x_0, x_1, \ldots, x_{m-1}) = \sum_{(y_0, \ldots, y_{m-1}) \in \mathbb{F}_p^m} \psi_Y(y_0, y_1, \ldots, y_{m-1}) \cdot \prod_{0 \le j+k < m} \psi_{X_j, Y_k}(x_j, y_k).$$

The SumProd problem was shown to be NP-hard in (Cooper, 1990), which should not be surprising because we knew that answering conjunctive queries was NP-hard in query complexity (Chandra and Merlin, 1977). Beyond conjunctive queries, (Khamis *et al.*, 2016) generalized SumProd into Functional Aggregate Query (or FAQ), to capture even more problems where multiple semirings are required to model them. The same algorithmic framework presented in the next section can also be applied to FAQs.

4.3 Backtracking Search and Worst-Case Optimal Joins

In order to discuss the computational complexity of solving the SumProd problem (4.1), we need to be specific on how the input factors ψ_S are represented.

Listing Representation We shall assume the *listing* representation, where every input factor ψ_S is represented as an arity-$(|S|+1)$ relation: $|S|$ attributes to store the key $\boldsymbol{x}_S$, and the extra column storying the *value* $\psi_S(\boldsymbol{x}_S)$. Furthermore, when the value $\psi_S(\boldsymbol{x}_S)$ is $\mathbf{0}$ (the zero element of the semiring), we will omit the corresponding tuple from the relation. The listing representation is the natural generalization of the sparse representation of a matrix.

For example, every input factor in Example 4.2.3 are represented as a table of triples (i, j, a_{ij}). When the semiring is a Boolean semiring, the column storing the value is not needed; this case corresponds precisely to the *conjunctive* queries in relational databases.

We use $\|\psi\|$ to denote the number of non-zero-valued tuples of the given function ψ. In data-complexity, $\|\psi\|$ is the *size* of the factor ψ.

Join and backtracking search A simple way to solve the SumProd problem (4.1) is to enumerate all tuples $\boldsymbol{x}_V$ for which $\psi_S(\boldsymbol{x}_S) \neq \mathbf{0}$, for all $S \in \mathcal{E}$. Then, the output factor $\varphi(\boldsymbol{x}_F)$ can be computed by aggregating over all such tuples with the $\oplus$ operator, grouping by the free variables $\boldsymbol{x}_F$. The strategy is outline in Algorithm ??.

; `// Iterating over` $\boldsymbol{x}_V$ `via backtracking search`
for $\boldsymbol{x}_V \in \mathit{Dom}(\boldsymbol{x}_V)$ *such that* $\psi_S(\boldsymbol{x}_S) \neq \mathbf{0}$ *for all* $S \in \mathcal{E}$ **do**
  if $\varphi(\boldsymbol{x}_F)$ *is not initialized* **then**
    $\varphi(\boldsymbol{x}_F) \leftarrow \mathbf{0}$;
  end
  $\varphi(\boldsymbol{x}_F) \leftarrow \varphi(\boldsymbol{x}_F) \oplus \prod_{S \in \mathcal{E}} \psi_S(\boldsymbol{x}_S)$;
end
return φ

Algorithm 5: Backtracking search for SumProd query (4.1)

We think of $\psi_S(\boldsymbol{x}_S)$ as encoding constraints where $\boldsymbol{x}_S$ satisfies the constraint if $\psi_S(\boldsymbol{x}_S) \neq \mathbf{0}$, then one can identify all satisfying tuples $\boldsymbol{x}_V$ with a *backtracking search* algorithm. We will assume that the search is done via a fixed ordering of the variables in V. Naturally, thinking of $\psi_S(\boldsymbol{x}_S)$ as a database relation, then enumerating the satisfying tuples is exactly a (natural) join query evaluation problem.

Backtracking search is one way to solve the join query evaluation problem. While join is a more general primitive, we have chosen to stick to the backtracking search angle in this section because it contrasts very well with the dynamic programming approach presented in the next section; furthermore, efficient join computation is a *much* deeper database topic, both in theory and in practice than what can be reasonably discussed in this section.

Backtracking search is a generic algorithmic technique to solve the SumProd problem (4.1), but it may not be the most efficient. However, there are some non-trivial properties of this algorithm that were only discovered recently (Ngo *et al.*, 2012; Veldhuizen, 2014). In particular, *how* backtracking search is implemented can have a significant impact on the runtime. We briefly present these properties here as they are needed to understand the notions of "width" in the next section.

Worst-case optimal join and fractional edge cover The backtracking search algorithm to solve (4.1) can be analyzed in terms of a combinatorial property of the input hypergraph $\mathcal{H} = (V, \mathcal{E})$. In database terminology, this is the *query hypergraph*. The *fractional edge cover* of $\mathcal{H}$ is an assignment of a non-negative "weight" λ_S for each $S \in \mathcal{E}$ such that for every $v \in V$, the total weight of edges containing v is at least 1; and the *fractional edge cover number* $\rho^*(\mathcal{H})$ is the minimum total weight over all fractional edge covers. In particular, $\rho^*(\mathcal{H})$ is the objective value of the following linear program:

$$\min \sum_{S \in \mathcal{E}} \lambda_S \tag{4.2}$$

$$\text{s.t.} \sum_{S \in \mathcal{E}: v \in S} \lambda_S \geq 1, \quad \forall v \in V, \tag{4.3}$$

$$\lambda_S \geq 0, \quad \forall S \in \mathcal{E}. \tag{4.4}$$

Example 4.3.1 (Loomis-Whitney hypergraphs)**.** The *Loomis-Whitney hypergraph* of order n, named after (Loomis and Whitney, 1949), is the hypergraph $\mathcal{H}$ on $V = [n]$ and $\mathcal{E} = \binom{[n]}{n-1}$, i.e. the hypergraph where every edge contains all but one vertex. The fractional edge cover number of the Loomis-Whitney hypergraph is $\rho^*(\mathcal{H}) = n/(n-1)$, where $\lambda_S = 1/(n-1)$ for all $S \in \mathcal{E}$. The triangle (non-hyper) graph is the Loomis-Whitney hypergraph of order 3, with $\rho^* = 3/2$.

The main property of the backtracking search algorithm that is of our interest is the following, due to (Ngo *et al.*, 2012; Veldhuizen, 2014).

Theorem 4.3.2. Let N denote the maximum input factor size to (4.1), i.e. $N = \max_{S \in \mathcal{E}} \|\psi_S\|$. There is a way to implement backtracking search to solve (4.1) so that it runs in time $O(|\mathcal{E}| \cdot |V|^2 \cdot \log N \cdot N^{\rho^*(\mathcal{H})})$.

Should we include a proof of this theorem? Or is it too technical? – Hung.

In some database applications, it is often convenient to subsume factors that are dependent on the size of the query hypergraph and $\log N$; in that case, we say that the backtracking search algorithm runs in time $\tilde{O}(N^{\rho^*(\mathcal{H})})$.

In the context of database query evaluation, such an algorithm is called a *worst-case optimal*, because in the worst case the output size is $N^{\rho^*(\mathcal{H})}$. The subject of worst-case optimal join algorithms is much more involved than what was presented here. In particular, it has an interesting history with deep connection to information theory. See Section 4.6 for further discussions and references.

Example 4.3.3 (Loomis-Whitney queries)**.** Loomis-Whitney queries are SumProd queries whose hypergraphs are Loomis-Whitney. For example, order-3 Loomis-Whiney query is the triangle query, such as

$$Q(A, B, C) \coloneq R(A, B) \wedge S(B, C) \wedge T(C, A).$$

for some relations R,S,T. If R,S, and T are the same edge relation of a graph, then the query asks for listing all triangles in a graph. Worst-case optimal join returns all the triangles in time $\tilde{O}(N^{3/2})$, where N is the number of edges in the graph. This example also illustrates a key strength of worst-case optimal joins: there are input instances for which *any* join-project query plan will necessarily runs in $\Omega(N^2)$-time (Ngo *et al.*, 2012).

An order-4 Loomis-Whitney query is of the form

$$Q(A, B, C, D) \coloneq R(A, B, C) \wedge S(A, B, D) \wedge T(A, C, D) \wedge U(B, C, D).$$

This query can be answered in $O(N^{4/3})$-time.

It is worth noting that, if R, S, T, U are dense 3D-tensors, over domains of size M for each of the variables A, B, C, D, then $N = M^3$ (the number of non-zero entries in each tensor), and the bound is $N^{4/3} = M^4$, which is the same as the naïve bound. In particular, the fractional edge covering bound is good when the input factors (or tensors) are sparse.

4.4 Dynamic Programming with Variable elimination

Backtracking search has a well-known weakness: it may repeat the same computation many times. The other side of the coin is dynamic programming, which "caches" the results of subproblems and reuses them when needed. Given how expressive the SumProd abstraction is, it was equally amazing that there is a unified *meta* dynamic programming algorithm to solve (4.1): it is called the *variable elimination* algorithm, or equivalently the *message passing* or *belief propagation* algorithm (Pearl, 1982; Shafer and Shenoy, 1990; Zhang and Poole, 1994; Dechter, 1999; Aji and McEliece, 2000).

As surveyed and nicely presented in (Aji and McEliece, 2000; Kohlas and Wilson, 2008), this meta-algorithm has as special cases a massive number of well-known (even best-known) algorithms for problems that can be cast as SumProd, such as the fast Hadamard transform, the turbo decoding algorithm, the FFT on any finite Abelian group, many best known algorithms for answering inference queries in graphical models (Wainwright and Jordan, 2008) (such as Viterbi's algorithm, Pearl's "belief propagation" algorithm, the Shafer-Shenoy probability propagation algorithm, the Baum-Welch "forward-backward" algorithm, and discrete-state Kalman filtering).

From the database perspective, variable elimination is a very general algorithmic abstraction for designing logical query optimizers, which has been

adopted in a couple of commercial relational DBMSs and shown to be very robust in practice (Nguyen *et al.*, 2015). In fact, when formulated correctly, Yannakakis's classic algorithm for computing join queries (Yannakakis, 1981) is an earlier incarnation of this algorithm[2].

The variable elimination (meta) algorithm is very simple; it can be explained with a couple of examples. Suppose our problem is to count the number of tuples (x, y, z, u) satisfying the conjunction $R(x, y) \wedge S(y, z) \wedge T(z, u)$, where R, S, and T are input relations. This problem can be represented algebraically as follows. Overloading notations, we use R to denote the indicator function that $(x, y) \in R$, i.e. $R(x, y) = \mathbf{1}_{(x,y)\in R}$. Similarly, S and T are indicator functions. In particular, we "lifted" the relations from the Boolean domain to be real functions for the purpose of counting. Then, the counting query is of the form (4.1):

$$Q() = \sum_a \sum_b \sum_c \sum_d R(a, b) \cdot S(b, c) \cdot T(c, d). \tag{4.5}$$

Note the freedom we have with the range of a, b, c, d: since R, S, and T are indicator functions, we are now free to sum over the entire ranges of a, b, c, d as values that do not belong to the input relations are zeroed out. The problem can be solved by applying the distributive law, eliminating one variable at a time:

$$\begin{aligned} Q() &= \sum_a \sum_b \sum_c \sum_d R(a, b) \cdot S(b, c) \cdot T(c, d) \\ &= \sum_a \sum_b \sum_c R(a, b) \cdot S(b, c) \cdot \sum_d T(c, d) \\ &= \sum_a \sum_b \sum_c R(a, b) \cdot S(b, c) \cdot W(c) \\ &= \sum_a \sum_b R(a, b) \cdot \sum_c S(b, c) \cdot W(c) \\ &= \sum_a \sum_b R(a, b) \cdot V(b) \end{aligned} \tag{4.6}$$

The algebraic reasoning leads to a query plan, where we first compute an intermediate result $W(c)$, then another intermediate result $V(b)$, and finally Q is computed. It is easy to see how this plan can be translated to relational query plans and the meta algorithm takes $\tilde{O}(N)$-time.

If the query was not a count query, but rather a proper conjunctive query

$$Q() = \bigvee_a \bigvee_b \bigvee_c \bigvee_d R(a, b) \wedge S(b, c) \wedge T(c, d).$$

[2]See (Khamis *et al.*, 2016) for an explanation

Then, precisely the same query plan works. All we have to do is to replace $\sum$ with $\vee$ and $\cdot$ with $\wedge$. That is the power of algebraic abstraction. The above ideas are often expressed in the database textbooks as *pushing aggregates through joins* and *pushing existential quantifications through joins* (Ramakrishnan and Gehrke, 2003). They are all instance of the variable elimination framework.

In a more general setting, the variable elimination algorithm can be described as follows. Without loss of generality, suppose $F = \{1, \cdots, f\}$, so that the SumProd query is

$$\varphi(x_1, \ldots, x_f) = \sum_{x_{f+1}, \ldots, x_n} \prod_{S \in \mathcal{E}} \psi_S(\boldsymbol{x}_S).$$

Then, we inductively "eliminate" the nth variable x_n by writing:

$$\begin{aligned} \varphi(x_1, \ldots, x_f) &= \sum_{x_{f+1}, \ldots, x_n} \prod_{S \in \mathcal{E}} \psi_S(\boldsymbol{x}_S) \\ &= \sum_{x_{f+1}, \ldots, x_{n-1}} \prod_{\substack{S \in \mathcal{E} \\ n \notin S}} \psi_S(\boldsymbol{x}_S) \cdot \underbrace{\sum_{x_n} \prod_{\substack{S \in \mathcal{E} \\ n \in S}} \psi_S(\boldsymbol{x}_S)}_{\psi_T(\boldsymbol{x}_T)}. \end{aligned}$$

Here, $\psi_T(\boldsymbol{x}_T)$ is the *intermediate result* that is computed by the variable elimination algorithm. The algorithm is then applied recursively on the newly created problem with factors $\psi_S(\boldsymbol{x}_S)$, $n \notin S$, and $\psi_T(\boldsymbol{x}_T)$.

While one can use variable orderings as representations of query plans based on variable elimination, in practice it is often more robust to use an equivalent representation called tree-decompositions, as we shall see in the next section, where we will also discuss the optimization problem of how to select the best variable elimination ordering.

4.5 Tree-Decomposition and Message-Passing

As is well-known in the graphical model literature since the 1980s, an equivalent way to express the query plan from variable elimination is the notion of *tree decomposition* (or *hypertree* decomposition). Section 4.6 provides a more detailed background on tree-decompositions and their applications in database query processing and graphical models. We explain the main intuition through an example.

The variable elimination strategy shown in (4.6) can be captured with a combinatorial object where we create a "bag" (a set of variables) for each sub-problem in the query plan. The first bag is $\{c, d\}$ because the sub-problem producing $W(c)$ involves variables $\{c, d\}$. This bag produces an intermediate result $W(c)$ which is called a "message" passed to the second bag $\{b, c\}$.

Finally, the bag $\{b, c\}$ passes the message $V(b)$ to the bag $\{a, b\}$ in the same manner. The combinatorial structure connecting these bags, where there is an edge between two bags if one passes a message to the other, is called a *tree decomposition* of the input query (4.5). We shall define this notion formally below.

The variable elimination algorithm when expressed via tree decompositions is called the *message passing* algorithm. See Figure 4.1 for an illustration of the message passing algorithm for the query (4.5).

Figure 4.1: A Tree Decomposition for problem (4.5)

The key optimization problem one has to solve in order to apply the variable elimination framework is to decide the variable ordering under which to gradually push the summations in (to eliminate variables). To decide if a variable order is "better" than another, we then need a cost function to tell them apart, in the same way query optimizers have cost functions for selecting query plans.

One sensible cost function is the worst-case cardinalities over all intermediate result. (Minimizing the intermediate result sizes is a main goal of DBMS query optimizers.) Each intermediate result is computed over a bag of the corresponding tree decomposition. Thus, minimizing the maximum cardinality of the intermediate results is the same as minimizing the maximum number of satisfying tuples over each bag of the tree decomposition. In the solution we just described for Problem (4.5), the worst-case cardinalities of all intermeidate results is $O(N)$, i.e. linear in the input size.

For a more interesting example, consider the query (in SumProd form):

$$Q = \sum_{a,b,c,d,e,f} R(a,b)S(a,c)T(b,c,d,e)W(e,f)V(d,f) \tag{4.7}$$

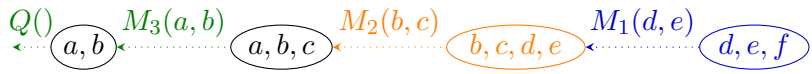


Figure 4.2: A Tree Decomposition for query (4.7)

One way to answer this query is the following variable elimination strategy:

$$
\begin{aligned}
Q &= \sum_{a,b,c,d,e,f} R(a,b)S(a,c)T(b,c,d,e)W(e,f)V(d,f) \\
&= \sum_{a,b,c,d,e} R(a,b)S(a,c)T(b,c,d,e) \sum_{f} W(e,f)V(d,f) \\
&= \sum_{a,b,c,d,e} R(a,b)S(a,c)T(b,c,d,e)M_1(e,d) \\
&= \sum_{a,b,c} R(a,b)S(a,c) \sum_{e,d} T(b,c,d,e)M_1(e,d) \\
&= \sum_{a,b,c} R(a,b)S(a,c)M_2(b,c) \\
&= \sum_{a,b} R(a,b) \sum_{c} S(a,c)M_2(b,c) \\
&= \sum_{a,b} R(a,b)M_3(a,b)
\end{aligned}
$$

The corresonding tree decomposition (TD) is shown in Figure 4.2. This TD has a worst-case cost of N^2, because the bag $\{d, e, f\}$ is the join of two input relations $W(e, f)$ and $V(d, f)$. The size of the join is $O(N^2)$ in the worst-case. Similarly, one can reason straightforwardly that bag $\{b, c, d, e\}$ has $O(N)$-size at worst, bag $\{a, b, c\}$ has $O(N^2)$-size at worst, and bag $\{a, b\}$ has $O(N)$-size at worst. Overall, this query plan has cost $O(N^2)$. The exponent 2 is called the (generalized) *hypertree width* of the tree decomposition (Gottlob *et al.*, 2016).

There is, however, a better query plan that yields a cost less than quadratic. This query plan is based on an idea introduced in (Khamis *et al.*, 2016) called the "indicator projection" idea that was designed to exploit worst-case optimal join algorithms (WCOJ) presented in Section 4.3, and the notion of fractional edge covering number of a (sub-)query.

We illustrate these concepts with the following variable elimination strategy for the input query (4.7), where an indicator projection of the factor $T(b, c, d, e)$ is used to *reduce* the size of the intermediate result. In particular, with some

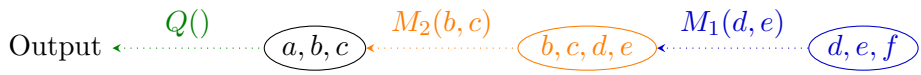


Figure 4.3: Second Tree Decomposition for query (4.7)

notational abuse, define the function $\pi_{d,e}T$ by

$$\pi_{d,e}T(d,e) = \begin{cases} \mathbf{1} & \text{if } \exists b, c \text{ s.t. } T(d,e,b,c) \neq \mathbf{0} \\ \mathbf{0} & \text{otherwise} \end{cases}$$

Then, $\pi_{d,e}T$ is called the *indicator projection* of the factor T onto the variables d, e. With this notation, the query plan is as follows:

$$\begin{aligned}
Q &= \sum_{a,b,c,d,e,f} R(a,b)S(a,c)T(b,c,d,e)W(e,f)V(d,f) \\
&= \sum_{a,b,c,d,e} R(a,b)S(a,c)T(b,c,d,e) \underbrace{\sum_{f} \pi_{de}T(d,e)W(e,f)V(d,f)}_{\tilde{O}(N^{3/2}),\ \text{WCOJ}} \\
&= \sum_{a,b,c,d,e} R(a,b)S(a,c)T(b,c,d,e)M_1(d,e) \\
&= \sum_{a,b,c} R(a,b)S(a,c) \sum_{d,e} T(b,c,d,e)M_1(d,e) \\
&= \sum_{a,b,c} R(a,b)S(a,c)M_2(b,c) \\
&= \underbrace{\sum_{a,b,c} R(a,b)S(a,c)M_2(b,c)}_{\tilde{O}(N^{3/2}),\ \text{WCOJ}}
\end{aligned}$$

It should be obvious that introducing this indicator projection does not change the semantic of the query. The idea is that, we do not need to compute entries $M_1(d,e)$ that do not have a corresponding projection from T. The bag $\{e, f, d\}$ that is used to create the message M_1 is of size $O(N^{1.5})$. This number 1.5 is the fractional edge covering number of the triangle graph consisting of edges de, ef and df. Similarly, the bag $\{a, b, c\}$ is also bounded by $N^{1.5}$. Thus, overall, this query plan with the tree-decomposition shown in Figure 4.3 has cost $O(N^{1.5})$, and it is said to have a *fractional hypertree width* of 1.5. This type of query plan and the notation of fractional hypertree width was introduced for constraint satisfation problems in (Grohe and Marx, 2014).

The variable ordering could have been different, but with the same cost,

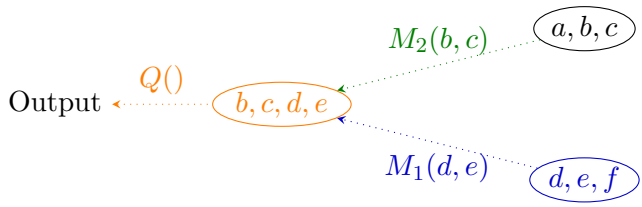


Figure 4.4: Third Tree Decomposition for query (4.7)

as shown in the following strategy:

$$
\begin{aligned}
Q &= \sum_{a,b,c,d,e,f} R(a,b)S(a,c)T(b,c,d,e)W(e,f)V(d,f) \\
&= \sum_{a,b,c,d,e} R(a,b)S(a,c)T(b,c,d,e) \underbrace{\sum_{f} \pi_{de} T(d,e)W(e,f)V(d,f)}_{M_1(d,e) \text{ in } \tilde{O}(N^{3/2}),\ \text{WCOJ}} \\
&= \sum_{b,c,d,e} T(b,c,d,e)M_1(d,e) \underbrace{\sum_{a} \pi_{bc} T(b,c)R(a,b)S(a,c)}_{M_2(b,c) \text{ in } O(N^{3/2}),\ \text{WCOJ}} \\
&= \sum_{b,c,d,e} M_1(d,e)T(b,c,d,e)M_2(b,c)
\end{aligned}
$$

This results in the TD shown in Figure 4.4

Finally, we formalize the notion of tree-decompositions and how they can be used to represent a query plan for SumProd queries. We also formalize the notion of fractional hypertree width of a tree decomposition (i.e. a query plan) and the fractional hypertree width of a query, which represents a good asymptotic measure of the optimal cost of a query plan.

Definition 4.5.1 (Tree-decomposition)**.** Consider a SumProd query in the form (4.1), defined via a hypergraph $\mathcal{H} = (V, \mathcal{E})$. A *tree decomposition* of the query is a pair (T, χ), where $T = (V(T), E(T))$ is a tree and $\chi : V(T) \to 2^V$ is a map from the nodes of T to subsets of V that satisfies the following properties:

- for all hyperedge $S \in \mathcal{E}$, there is a node $t \in V(T)$ such that $S \subseteq \chi(t)$;
- and, for any vertex $v \in V$, the set $\{t \mid v \in \chi(t)\}$ forms a connected sub-tree of T.

Each set $\chi(t)$ is called a *bag* of the tree-decomposition, and we will assume w.l.o.g. that the bags are distinct, i.e. $\chi(t) \neq \chi(t')$ when $t \neq t'$ are nodes in T.

For an acyclic (conjunctive) query, the well-known database concept *join-tree* of the query is a tree decomposition (see (Garcia-Molina *et al.*, 2009)).

Let $S \subseteq V$ be a set of variables. The edge covering number of S with respect to a hypergraph $\mathcal{H} = (V, \mathcal{E})$ is defined to be $\rho^*_{\mathcal{H}}(S) := \rho^*(\mathcal{H}[S])$, where $\mathcal{H}[S]$ is the sub-hypergraph of $\mathcal{H}$ induced by the variables in S. The *fractional hypertree width* of a tree decomposition (T, χ), denoted by $\mathsf{fhtw}(T, \chi)$, is the maximum fractional edge covering number over all bags of the tree decomposition:

$$\mathsf{fhtw}(T, \chi) = \max_{t \in V(T)} \rho^*_{\mathcal{H}}(\chi(t)). \tag{4.8}$$

Given a SumProd query $\varphi(\boldsymbol{x}_F)$ as defined in (4.1) a *free-connex* tree decomposition for the query is a tree decomposition for φ with the additional property that there is a connected sub-tree T' of T for which $F = \bigcup_{t \in V(T')} \chi(t)$. Free-connex tree decompositions are meant to capture the variable elimination strategy where all bound variables are eliminated before the free variables. This requirement is necessary to ensure that the query plan is correct, i.e. the output factor is computed correctly.

As valid variable elimination strategies are captured by free-connex tree decompositions of the query, and the "cost" of each tree-decomposition is model by the fractional hypertree width, we can define the fractional hypertree width of a query as follows:

Definition 4.5.2 (Fractional hypertree width of a SumProd query)**.** Given a SumProd query $\varphi(\boldsymbol{x}_F)$ with hypergraph $\mathcal{H} = (V, \mathcal{E})$, the *fractional hypertree width* of the query, denoted by $\mathsf{fhtw}(\varphi)$, is the minimum fractional hypertree width over all free-connex tree decompositions of the query.

From the examples described above, it is not hard to prove the following result:

Theorem 4.5.3 (Khamis *et al.*, 2016)**.** Given a free-connex tree decomposition (T, χ) of a SumProd query $\varphi(\boldsymbol{x}_F)$; let N denote the maximum input factor size; then, we can compute the output factor $\varphi(\boldsymbol{x}_F)$ in time

$$\tilde{O}\left(N^{\mathsf{fhtw}(T,\chi)} + \|\varphi\|\right) \tag{4.9}$$

In particular, in data-complexity, the query can be answered in time

$$\tilde{O}\left(N^{\mathsf{fhtw}(\varphi)} + \|\varphi\|\right) \tag{4.10}$$

4.6 Further Readings

The SumProd problem (with a variety of different names) was studied extensively starting in the late 1980s (Shenoy and Shafer, 1988; Shafer and Shenoy,

1991; Lauritzen and Spiegelhalter, 1988). Variable elimination and equivalently message passing as a general technique for solving SumProd queries were also studied extensively in the probabilistic graphical model literature (Dechter, 1997; Kohlas and Shenoy, 2000; Aji and McEliece, 2000) The SumProd problem was shown to be NP-hard in (Cooper, 1990),

More detailed background knowledge on semirings can be found in (Gondran and Minoux, 2008). Graphical models and their applications are in Wainwright and Jordan, 2008; Koller and Friedman, 2009. (Gottlob *et al.*, 2016) provides a comprehensive survey of tree-decompositions and their use in database query processing.

The fractional hypertree width runtime for SumProd queries can also be achieved using top-down (memoized) dynamic programming (Olteanu and Závodný, 2015). This approach, along with a novel compressed data representation was proposed in (Olteanu and Závodný, 2015), which has many practical applications (Olteanu and Schleich, 2016; Olteanu, 2020).

The notion of worst-case optimal join algorithms was developed through a series of works. (Grohe and Marx, 2014; Atserias *et al.*, 2008) derived worst-case output size bounds for a full conjunctive queries, and demonstrated that a join-project query plan can attain a runtime of $\tilde{O}(N^{1+\rho^*(\mathcal{H})})$, while a join-only plan cannot and in fact can be arbitrarily worse. There are inputs for which the output size is $\Omega(N^{\rho^*(\mathcal{H})})$ (Atserias *et al.*, 2008), and thus the runtime of a join-project plan is off from optimality by a linear factor. Then, (Ngo *et al.*, 2012) and (Veldhuizen, 2014) showed that the runtime of $\tilde{O}(N^{\rho^*(\mathcal{H})})$ can be achieved by a more sophisticated class of join algorithms; these are worst-case optimal. High-level summary presentations on this topic can be found in (Ngo, 2018; Ngo *et al.*, 2013). These concepts and algorithms were extended to deal with more general knowledge about the input databases in (Gottlob *et al.*, 2009; Abo Khamis *et al.*, 2017; Abo Khamis *et al.*, 2016).

The notion of free-connex tree-decomposition was developed in (Bagan *et al.*, 2007) to answer conjunctive queries. Naturally, free-connex tree-decompositions can be applied to SumProd queries as well, via the variable elimination algorithm as shown in (Khamis *et al.*, 2016)

(Khamis *et al.*, 2016) generalized the SumProd problem to capture computations over mutiple semirings, and explained how variable elimination can be used to solve these queries. The more general problem is called a *funcitonal aggregate query*. As explained in (Khamis *et al.*, 2016), the classic Yannakakis algorithm (Yannakakis, 1981) is a variable elimination algorithm. The problem of counting the number of solutions to a conjunctive query can be formulated as a functional aggregate query; its complexity and algorithms were studied in an earlier work (Pichler and Skritek, 2013).

5

Extensions Based on Algebra

This chapter is almost complete.

- The beginning needs to be cleaned up
- There is a better convergence result, should we include it?
- The further reading section needs to be updated

We have seen in Chap. 3 several extensions of Datalog based on logic, which we introduced in order to add support for negation, choice, object invention, and aggregates. Among them, aggregates are the most important feature for modern data management tasks, and some of the logic-based approaches that we have discussed required specialized semantics.

An alternative extension of datalog has been proposed in the literature, which is based on algebra, and was motivated mainly by recent applications of datalog to program analysis and machine learning (Madsen *et al.*, 2016; Aref *et al.*, 2015; Khamis *et al.*, 2022b). In this chapter we discuss algebraic approaches that extend datalog with numerical computations, which include operators like $+, \times, \min, \max$, and possibly others. Aggregates become natural operations in such an extension of datalog. These algebraic approaches are based on the concept of *semirings*, and their extensions to *Kleene algebras*, which have been studied in the 60s and 70s as a unifying theory for graph algorithms, automata theory, and formal languages. The critical bridge between semirings and relational query languages is the concept of *K-relations*, introduced by Green et al. (Green *et al.*, 2007).

5.1 Automata Theory, Languages, and Algorithms

Solving a (recursive) Datalog program is a special case of solving fixpoint equations over semirings, which was studied in the automata theory (Kuich, 1997), program analysis (Cousot and Cousot, 1977; Nielson *et al.*, 1999), and graph algorithms (Carré, 1979; Lipton and Tarjan, 1980; Lipton *et al.*, 1979) communities since the 1970s. (See (Rote, 1990; Hopkins and Kozen, 1999; Lehmann, 1977; Gondran and Minoux, 2008; Zimmermann, 1981) and references thereof). The problem took slighly different forms in these domains, but at its core, it is to find a solution to the equation $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x})$, where $\boldsymbol{x} \in \boldsymbol{S}^n$ is a vector over the domain $\boldsymbol{S}$ of a semiring, and $\boldsymbol{f} : \boldsymbol{S}^n \to \boldsymbol{S}^n$ has multivariate polynomial component functions.

In the theory context-free languages, the celebrated Parikh's theorem (Parikh, 1966) plays a special role in our theoretical development. The theorem states that, if the symbol connatenation operator is commutative, then context-free languages and regular sets are indistinguishable. There have been many proofs and simplifications of this result produced over the years (Pilling, 1973; Hopkins and Kozen, 1999; Esparza *et al.*, 2011; Leeuwen, 1974; Goldstine, 1977; Aceto *et al.*, 2002). Some proofs are combinatorial, some are algebraic, and we can learn a lot from them in analyzing the convergence rate of the naïve algorithm to compute the fixpoint of $\boldsymbol{f}$.

When $\boldsymbol{f}$ is affine (what we called *linear* in this paper), researchers realized that many problems in different domains are instances of the same problem, with the same underlying algebraic structure: transitive closure (Warshall, 1962), shortest paths (Floyd, 1962), Kleene's theorem on finite automata and regular languages (Kleene, 1956), continuous dataflow (Cousot and Cousot, 1977; Kam and Ullman, 1976), etc. Furthermore, these problems share the same characteristic as the problem of computing matrix inverse (Backhouse and Carré, 1975; Tarjan, 1976; Gondran, 1975). The problem is called the *algebraic path problem* (Rote, 1990), among other names, and the main task is to solve the matrix fixpoint equation $X = AX + I$ over a semiring.

There are several classes of solutions to the algebraic path problem, which have pros and cons depending on what we can assume about the underlying semiring (whether or not there is a closure operator, idempotency, natural orderability, etc.). We refer the reader to (Gondran and Minoux, 2008; Rote, 1990) for more detailed discussions. Here, we briefly summarize the main approaches.

The first approach is to keep iterate until a fixpoint is reached; in different contexts, this has different names: the naïve algorithm, Jacobi iteration, Gauss-Seidel iteration, or Kleene iteration. The main advantage of this approach is that it assumes less about the underlying algebraic structure: we do not need both left and right distributive law, and do not need to assume a closure operator.

The second approach is based on Gaussian elimination (also, Gauss-Jordan) elimination, which, assuming we have oracle access to the solution x^* of the 1D problem $x = 1 + ax$, can solve the algebraic path problem in $O(n^3)$-time (Rote, 1990; Lehmann, 1977).

The third approach is based on specifying the solutions based on the free semiring generated when viewing $\boldsymbol{A}$ as the adjanceny matrix of a graph (Tarjan, 1981a). The underlying graph structure (such as planarity) may sometimes be exploited for very efficient algorithm (Lipton and Tarjan, 1980; Lipton *et al.*, 1979).

todo Robert Tarjan's work on the "Path queries" (Tarjan, 1981a; Tarjan, 1981b); Algebraic path problems (Master, 2021; Rote, 1990; Lehmann, 1977)

Beyond the affine case, since the 1960s researchers in formal languages have been studying the structure of the fixpoint solution to $x = f(x)$ when f's component functions are multivariate polynomials over Kleene algebra (Pilling, 1973; Parikh, 1966; Kuich, 1987). It is known, for example, that Kleene iteration does not always converge (in a finite number of steps), and thus methods based on Galois connection or on widening/narrowing approaches (Cousot and Cousot, 1992) were studied. These approaches are (discrete) lattice-theoretic. More recently, a completely different approach drawing inspiration from Newton's method for solving a system of (real) equations was proposed (Hopkins and Kozen, 1999; Esparza *et al.*, 2010).

todo Newton method, 2nd order method, cite the following works (Luttenberger and Schlund, 2013; Reps *et al.*, 2016; Esparza *et al.*, 2010) Algebraic program analysis `https://link.springer.com/chapter/10.1007/978-3-030-81685-8_3` (talk: `https://ucl-pplv.github.io/CAV21/poster_P_k2/`) In computer algebra, on computation with formal power series, they have already used Newton's method, to compute the inverse (Chan *et al.*, 2021)

Recall that Newton's method for solving a system of equations $g(x) = 0$ over reals is to start from some point x_0, and at time t we take the first order approximation $g(x) \approx g_t(x) := g(x_t) + g'(x_t)(x - x_t)$, and set x_{t+1} to be the solution of $g_t(x) = 0$, i.e. $x_{t+1} = x_t - [g'(x_t)]^{-1} g(x_t)$. Note that in the multivariate case g' is the Jacobian, and $[g'(x_t)]^{-1}$ is to compute matrix inverse. In *beautiful* papers, Esparza et al. (Esparza *et al.*, 2010) and Hopkins and Kozen (Hopkins and Kozen, 1999) were able to generalize this idea to the case when $g(x) = f(x) - x$ is defined over ω-continuous semirings. They were able to define an appropriate *minus* operator, derivatives of power series over semirings, matrix inverse, and prove that the method converges at least as fast as Kleene iteration, and there are examples where Kleene iteration does not converge, while Newton method does. Furthermore, if the semiring is commutative and idempotent (in addition to being ω-continuous), then Newton method always converges in n Newton steps.

todo Connection to Automatic Differentiation (Rote, 1990)

todo Combinatorics (Pivoteau *et al.*, 2012)
todo Recursive Markov chain (Etessami and Yannakakis, 2009)
todo Virginia Williams has lots work on path queries, matrix mult, (they are special classes of datalogo queries). Lots of lower-bounds (Chan *et al.*, 2022; Gu *et al.*, 2021; Chan *et al.*, 2021)

5.2 Background

5.2.1 K-Relations

Throughout this section we assume a *commutative* semiring $\boldsymbol{S}$. Fix a schema $\boldsymbol{X} = \{X_1, \ldots, X_n\}$, let Dom be some infinite domain, and let Dom^n be the set of n-tuples. A *K-relation* is a function $R : \mathsf{Dom}^n \to S$ with *finite support*, where the support is defined as $\mathsf{supp}(R) \stackrel{\text{def}}{=} \{\boldsymbol{x} \in \mathsf{Dom}^n \mid R(\boldsymbol{x}) \neq \boldsymbol{0}\}$. The letter K in a K-relation originates from the fact that (Green *et al.*, 2007) denoted semirings by the letter K, and the term *K-relation* is now standard. When we want to mention the specific semiring $\boldsymbol{S}$ we will call the K-relation an $\boldsymbol{S}$*-relation*; thus, we will talk about $\mathbb{B}$-relations, $\mathbb{N}$-relations, Trop-relations, etc. A $\mathbb{B}$-relation is equivalent to a standard set of tuples, because $R(\boldsymbol{x}) = 1$ can be interpreted as saying that $\boldsymbol{x}$ is in the set R, while $R(\boldsymbol{x}) = 0$ means that $\boldsymbol{x}$ is not in the set R. An $\mathbb{N}$-relation is equivalent to a bag of tuples, because we interpret the number $R(\boldsymbol{x})$ as the multiplicity of the tuple $\boldsymbol{x}$ in the bag R. We will discuss more examples below.

Suppose we have a database schema consisting of m relations $R_1, \ldots, R_m$, and a relational query Q over this schema; we assume Q is positive, i.e. it consists of selections, projections, joins, and unions, and does not contain set difference. Under the standard semantics of the relational query, if $R_1, \ldots, R_m$ are standard sets of tuples, then the answer to Q is also a *set* of tuples. Assume now that the m relations are $\boldsymbol{S}$-relations, over the same semiring $\boldsymbol{S}$. Then the output of the query Q is also an $\boldsymbol{S}$-relation, in other words it is a function $Q : \mathsf{Dom}^a \to S$, defined inductively on the structure of the expression Q:

- If Q is a selection $\sigma_p(Q')$, then its output is:

$$Q(\boldsymbol{x}) \stackrel{\text{def}}{=} Q'(\boldsymbol{x}) \otimes \mathbb{1}_{p(\boldsymbol{x})}$$

  where $\mathbb{1}_{p(\boldsymbol{x})}$ is the indicator function:

$$\mathbb{1}_{p(\boldsymbol{x})} \stackrel{\text{def}}{=} \begin{cases} \boldsymbol{1} & \text{if } p(\boldsymbol{x}) \text{ is true} \\ \boldsymbol{0} & \text{otherwise} \end{cases}$$

- If $Q = Q' \cup Q''$ then its output is defined as:

$$Q(\boldsymbol{x}) \stackrel{\text{def}}{=} Q'(\boldsymbol{x}) \oplus Q''(\boldsymbol{x})$$

- If $Q = \Pi_{\boldsymbol{X}}(Q')$ where Q, Q' have schemas $\boldsymbol{X} \subseteq \boldsymbol{X}'$ respectively, then:

$$Q(\boldsymbol{x}) \stackrel{\text{def}}{=} \bigoplus_{\boldsymbol{x}' \in \mathsf{supp}(Q') : \Pi_{\boldsymbol{X}}(\boldsymbol{x}') = \boldsymbol{x}} Q'(\boldsymbol{x}')$$

- If $Q = Q' \bowtie Q''$, then denote by $\boldsymbol{X}', \boldsymbol{X}''$ the schemas of Q', Q'', and assume w.l.o.g. that the join is a natural join. The output of Q is defined as:

$$Q(\boldsymbol{x}) \stackrel{\text{def}}{=} Q'(\Pi_{\boldsymbol{X}'}(\boldsymbol{x})) \otimes Q''(\Pi_{\boldsymbol{X}''}(\boldsymbol{x}))$$

  In other words, for each tuple $\boldsymbol{x}$ the value $Q(\boldsymbol{x}) \in \boldsymbol{S}$ is obtained by multiplying $Q'(\Pi_{\boldsymbol{X}'}(\boldsymbol{x}))$ with $Q''(\Pi_{\boldsymbol{X}''}(\boldsymbol{x}))$, where $\Pi_{X'}(\boldsymbol{x}), \Pi_{\boldsymbol{X}''}(\boldsymbol{x})$ are the projections of $\boldsymbol{x}$ on $\boldsymbol{X}'$ and $\boldsymbol{X}''$ respectively.

- If Q is base relation R_i, then the output is $Q(\boldsymbol{x}) \stackrel{\text{def}}{=} R_i(\boldsymbol{x})$.

In other words, the relational algebra query is interpreted by using $\otimes$ to "join" tuples, and using $\oplus$ to "union" tuples.

In standard relational query languages, every positive relational algebra expression is equivalent to a Union of Conjunctive Queries (UCQ). The same equivalence holds for K-relations, where the analogue of CQ is a *sum-product expression* and the analogue of a UCQ is a *sum-sum-product expression*, which we define next.

A *sum-product* expression is an expression of the form:

$$P(\boldsymbol{X}_0) = \bigoplus_{\boldsymbol{Y}} R_1(\boldsymbol{X}_1) \otimes \cdots \otimes R_m(\boldsymbol{X}_m) \tag{5.1}$$

Each expression $R_i(\boldsymbol{X}_i)$ is called an *atom*, the expression $P(\boldsymbol{X}_0)$ is called the *head*, $\boldsymbol{X}_0$ are called the *head variables*, while $\boldsymbol{Y}$ are sometimes called *existential variables*. Every head variable must occur in some atom.

A *sum-sum-product expression* is an expression of the form:

$$Q(\boldsymbol{X}_0) = P_1(\boldsymbol{X}_0) \oplus \cdots \oplus P_k(\boldsymbol{X}_0) \tag{5.2}$$

where each $P_i(\boldsymbol{X}_0)$ is a sum-product expression as in (5.1).

Notice that, when the intended semiring is that of Booleans, then $\oplus, \otimes$ becomes $\vee, \wedge$, and (5.1) is a standard Conjunctive Query, while (5.2) is standard Union of Conjunctive Queries.

We define now the semantics of (5.1) and (5.2). Assume that each relation R_i is an $\boldsymbol{S}$-relation, for a fixed semiring $\boldsymbol{S}$. We define the semantics of (5.1) as:

$$P(\boldsymbol{x}_0) = \bigoplus_{\boldsymbol{x} \in \mathsf{Dom}^{|\boldsymbol{X}|} : \Pi_{\boldsymbol{X}_0}(\boldsymbol{x}) = \boldsymbol{x}_0} R_1(\Pi_{\boldsymbol{X}_1}(\boldsymbol{x})) \otimes \cdots \otimes R_m(\Pi_{\boldsymbol{X}_n}(\boldsymbol{x})) \tag{5.3}$$

The summation above is well defined because, although $\boldsymbol{x}$ ranges over an infinite set, only a finite number of terms are $\neq \mathbf{0}$, because every $\boldsymbol{S}$-relations R_i has finite support, and $\mathbf{0} \cdot a = 0$ for all $a \in S$. After dropping all the terms that are $= \mathbf{0}$ we are left with finitely many terms, and the semantics of $P(\boldsymbol{x}_0)$ is defined as their sum.

The semantics of a sum-sum-product (5.2) is defined as the sum of its sum-products.

Example 5.2.1. Consider the following sum-product expression:

$$Q(X, Z) = \bigoplus_{Y} (A(X, Y) \otimes B(Y, Z))$$

For the sake of the discussion, consider the following two K-relations, over some semiring $\boldsymbol{S}$ specified below:

X	Y	$A(X,Y)$
x	y_1	a_1
x	y_2	a_2

Y	Z	$B(Y,Z)$
y_1	z_1	b_1
y_1	z_2	b_2
y_2	z_2	b_3

(All tuples not shown above are mapped to $\mathbf{0}$.) Depending on the semiring $\boldsymbol{S}$, the query Q has several interpretations:

- If $\boldsymbol{S} = \mathbb{B}$ (the Booleans) then Q is a conjunctive query:

  $$Q(X, Z) \stackrel{\text{def}}{=} \exists Y (A(X, Y) \wedge B(Y, Z))$$

  All semiring values in A, B are $a_1 = a_2 = \cdots = b_3 = 1$, and the query's output is a $\mathbb{B}$-relation representing a standard set:

X	Z	$Q(X,Z)$
x	z_1	1
x	z_2	1

- If $\boldsymbol{S} = \mathbb{N}$, then Q is a join-project query under bag semantics. For any tuple $(x, z) \in \mathsf{Dom}^2$, $Q(x, z)$ is a number representing the multiplicity of (x, z) in the output. In our example, the values $a_1, a_2, \ldots, b_3 \in \mathbb{N}$ represent tuple multiplicities, and the query's answer is:

X	Z	$Q(X,Z)$
x	z_1	$a_1 b_1$
x	z_2	$a_1 b_2 + a_2 b_3$

  To make this concrete, if all tuples in A, B have multiplicity 1, i.e. $a_1 = \cdots = b_3 = 1$, then $Q(x, z_1) = 1$, $Q(x, z_2) = 2$.

- If $\boldsymbol{S} = \mathbb{R}$, then we view A, B as representing sparse matrices $A, B \in \mathbb{R}^{n \times n}$. For example, the sparse matrix on the left can be be represented as the $\mathbb{R}$-relation on the right:

$$\begin{pmatrix} 0 & 0 & 15 \\ 10 & 0 & 20 \end{pmatrix} \qquad \begin{array}{|c|c|c|} \hline X & Y & A(X,Y) \\ \hline 1 & 2 & 15 \\ 2 & 1 & 10 \\ 2 & 3 & 20 \\ \hline \end{array}$$

  The query's output is the matrix-matrix product: $Q(X, Z) = \sum_Y A_{XY} \cdot B_{YZ}$.

- If $\boldsymbol{S} = \mathsf{Trop}$ (the tropical semiring), then Q computes shortest paths of lengths 2 between certain nodes in a graph. Let $G = (X, Y, Z; E, E')$ be a tripartite graph, where $E \subseteq X \times Y$ and $E' \subseteq Y \times Z$. Assume that each edge has a cost, denoted by $A(x, y) \in \mathbb{R}_+ \cup \{\infty\}$ or $B(y, z) \in \mathbb{R}_+ \cup \{\infty\}$ respectively, with the convention that the cost is ∞ if the edge is not present. Then the query can be written as $Q(X, Z) = \min_Z(A(X, Z) + B(Z, Y))$, and each output $Q(x, z)$ represents the length of the shortest path from x to z. In our example, the query's output is:

X	Z	$Q(X,Z)$
x	z_1	$a_1 + b_1$
x	z_2	$\min(a_1 + b_2, a_2 + b_3)$

The take away of the previous example is that every monotone query has a natural semantics over $\boldsymbol{S}$-relations, once we fix the commutative semiring $\boldsymbol{S}$. For now, our discussion is restricted to non-recursive queries, but we will consider recursive queries shortly.

(Green *et al.*, 2007) proved that all identities that are satisfied by standard relational algebra expressions with bag semantics, continue to hold when the queries are interpreted over $\boldsymbol{S}$-relations, for any commutative semiring $\boldsymbol{S}$. For example, the identity $R \bowtie (\sigma_p(S) \bowtie T) = T \bowtie \sigma_p(R \bowtie S)$ holds when both expressions are interpreted using bag semantics. Therefore these expressions continue to be equal if we interpret them over the reals, or over the tropical semiring, or over any other commutative semiring. (Green *et al.*, 2007) also proved a form of converse: the only set of identities for the structure $(S, \otimes, \oplus, \mathbf{0}, \mathbf{1})$ that have this property are the identities of a commutative semiring. This observation has major implications for data management, because it means that query optimization and processing techniques developed for relational databases can carry over, with little changes, to other applications, for example to optimize programs over sparse tensors.

So far we have assumed that the K-relations are defined over a commutative *semiring*. It is possible to generalize this definition to a commutative *pre-semiring*, however, we need to be careful how we deal with the fact that

$x \otimes \mathbf{0} \neq \mathbf{0}$ and, in that case, the summation (5.3) may be infinite. To prevent that, (Khamis *et al.*, 2021) require the sum-product expressions (5.1) to explicitly specify the range of $\boldsymbol{Y}$. We will follow a similar convention in this paper.

5.2.2 Provenance Polynomials

An important concept related to K-relations is that of *provenance polynomials*. Fix a standard relational database instance $DB = (R_1, \ldots, R_m)$, and associate a unique variable $x_1, \ldots, x_N$ to each tuple in the database; one should think of x_i as the ID of the i'th tuple in the database. (Green *et al.*, 2007) call this an "abstract tagging" of the database DB. Given this tagging, we view each relation R_i as an $\mathbb{N}[\boldsymbol{x}]$-relation, where $\mathbb{N}[\boldsymbol{x}]$ are the formal polynomials with variables $\boldsymbol{x} = \{x_1, \ldots, x_N\}$ and coefficients in $\mathbb{N}$, such that $R_j(\boldsymbol{t}) = x_i$, where x_i is the variable associated to the tuple $\boldsymbol{t}$ in R_j; here x_i is viewed as a polynomial in $\mathbb{N}[\boldsymbol{x}]$. Then, the result of a query Q is an $\mathbb{N}[x]$ relation: each output tuple $\boldsymbol{z}$ is annotated with a polynomial $Q(\boldsymbol{z}) \in \mathbb{N}[\boldsymbol{x}]$ called the *provenance polynomial* of the output $\boldsymbol{z}$. The significance of the provenance polynomial is that it can be used to interpret Q over any other semiring, by simply evaluating the polynomial.

For a simple illustration, given the query and the K-relations in Example 5.2.1, assuming that the set of variables is $\boldsymbol{v} = \{a_1, a_2, b_1, b_2, b_3\}$, thus, both relations A, B in the example are $\mathbb{N}[\boldsymbol{v}]$-relations. Then the query's output has two provenance polynomials, one for (x, z_1) and one for (x, z_2):

X	Z	$Q(X, Z)$
x	z_1	$f_1 \stackrel{\text{def}}{=} a_1 b_1$
x	z_2	$f_2 \stackrel{\text{def}}{=} a_1 b_2 + a_2 b_3$

Here $Q(x, z_1) = a_1 b_1$ is the polynomial consisting of a single monomial, $a_1 b_1$, and $Q(x, z_2) = a_1 b_2 + a_2 b_3$ is a polynomial that is the sum of two monomials. Suppose that we wanted to compute Q over some $\boldsymbol{S}$-relations A, B, where $\boldsymbol{S}$ is some other pre-semiring. Then we can simply evaluate the two polynomials f_1, f_2 above by substituting the variables $a_1, \ldots, b_3$ with the values of of the $\boldsymbol{S}$-relations A, B; thus, a_1 will be substituted with $A(x, y_1) \in \boldsymbol{S}$, etc.

5.2.3 Partially Ordered Sets

So far our discussion of queries over K-relations has been limited to non-recursive queries. In order to interpret recursive queries over K-relations, we need to extend the semiring with a partial order, and define the query as a least fixpoint, similarly to basic datalog. We include in this section a brief review of partially ordered sets.

A *partially order set*, or *poset*, is a pair $(P, \sqsubseteq)$, where $\sqsubseteq$ is a reflexive, antisymmetric, and transitive relation. If $A \subseteq P$ is any subset, then we denote by $\bigvee A$ or $\sup A$ the least upper bound of A, and by $\bigwedge A$ or $\inf A$ the greatest lower bound of A, when they exist. In this paper we will always assume that P has a smallest element, and denote it with $\bot$; notice that $\bot = \bigwedge P = \bigvee \emptyset$.

A function $f : (P, \sqsubseteq) \to (P', \sqsubseteq')$ is *monotone* if $x \sqsubseteq y$ implies $f(x) \sqsubseteq' f(y)$. The Cartesian product of two posets $\boldsymbol{P}_1 = (P_1, \sqsubseteq_1)$ and $\boldsymbol{P}_2 = (P_2, \sqsubseteq_2)$, is ordered component-wise: $\boldsymbol{P}_1 \times \boldsymbol{P}_2 \stackrel{\text{def}}{=} (P_1 \times P_2, \sqsubseteq)$, where $(x_1, x_2) \sqsubseteq (y_1, y_2)$ if $x_1 \sqsubseteq_1 y_1$ and $x_2 \sqsubseteq_2 y_2$. An *ω-chain* in a poset $\boldsymbol{P}$ is a sequence $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$, or, equivalently, it is a monotone function $\mathbb{N} \to \boldsymbol{P}$. We say that the chain is finite if there exists n_0 such that $x_{n_0} = x_{n_0+1} = x_{n_0+2} = \cdots$, or, equivalently, if $x_{n_0} = \bigvee_n x_n$. A poset $(P, \sqsubseteq)$ is *ω-complete* if every ω-chain has a least upper bound $\bigvee_n x_n$. A function f is *ω-continuous* if $f(\bigvee_n x_n) = \bigvee_n f(x_n)$, for any ω-chain x_n, $n \geq 0$.

Given a monotone function $f \colon \boldsymbol{P} \to \boldsymbol{P}$, a *fixpoint* is an element x such that $f(x) = x$. We denote by $\mathsf{lfp}_{\boldsymbol{P}}(f)$ the *least fixpoint* of f, when it exists, and drop the subscript $\boldsymbol{P}$ when it is clear from the context. Consider the following ω-sequence:

$$f^{(0)}(\bot) \stackrel{\text{def}}{=} \bot \qquad\qquad f^{(n+1)}(\bot) \stackrel{\text{def}}{=} f(f^{(n)}(\bot)) \tag{5.4}$$

One can check by induction on n:

Proposition 5.2.2. If x is any fixpoint of f, then $f^{(n)}(\bot) \sqsubseteq x$.

If $\boldsymbol{P}$ is ω-complete and $f : \boldsymbol{P} \to \boldsymbol{P}$ is ω-continuous, then the least upper bound of f exists, and is given by $\mathsf{lfp}(f) = \bigvee_n f^{(n)}(\bot)$; this is called Kleene's theorem, see (Davey and Priestley, 1990).

The following technique is often used to prove the existence of a least fixpoint of a monotone function $f : \boldsymbol{P} \to \boldsymbol{P}$. Call an element $x \in \boldsymbol{P}$ a *pre-fixpoint* if $f(x) \sqsubseteq x$. Then:

Proposition 5.2.3. If the least pre-fixpoint of f exists, then the least fixpoint of f exists too, and they are equal. However, if the least fixpoint exists, this does not imply, in general, that the least pre-fixpoint exists.

Proof. Let x be the least pre-fixpoint of f. In particular, $f(x) \sqsubseteq x$. Since f is monotone, it holds that $f(f(x)) \sqsubseteq f(x)$, which means that $f(x)$ is also a pre-fixpoint. By assumption, x is the *least* pre-fixpoint, which implies that $x \sqsubseteq f(x)$, which means that x is a fixpoint. Let z be any other fixpoint $f(z) = z$. Then z is also a pre-fixpoint, and, by assumption, $x \sqsubseteq z$. This proves that x is the least fixpoint.

For the second part, consider the poset $\mathbb{Z} \cup \{\infty\}$, and define $f(x) \stackrel{\text{def}}{=} x - 1$, where $f(\infty) = \infty - 1 \stackrel{\text{def}}{=} \infty$. Then ∞ is the least fixpoint of f. However, every

$x \in \mathbb{Z}$ is a pre-fixpoint, hence there is no least pre-fixpoint. In this example the poset does not have a minimal element, $\bot$. One can extend it to an example that does have $\bot$ as follows. Take the disjoint union of $\mathbb{N}$ and $\mathbb{Z} \cup \{\infty\}$, and define every element in $\mathbb{N}$ to be $<$ every element in $\mathbb{Z} \cup \{\infty\}$. Now $0 \in \mathbb{N}$ is the least element of the poset. Extend the function f above by defining it on $\mathbb{N}$ as $f(n) = n + 1$. Here, too, ∞ is the least (and only) fixpoint, and every $x \in \mathbb{Z}$ is a pre-fixpoint, hence there is no least one. □

A *lattice* is a poset $\boldsymbol{L} = (L, \sqsubseteq)$ where any two elements have a least upper bound and a greatest lower bound, i.e. $x \vee y$ and $x \wedge y$ exists for all $x, y \in L$. Both $\vee$ and $\wedge$ are associative, commutative, and idempotent (meaning $x \vee x = x \wedge x = x$), but $\wedge$ does not necessarily distribute over $\vee$: when the distributivity law holds, $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$, then $\boldsymbol{L}$ is called a *distributive lattice*. In a distributive lattice, $\vee$ also distributes over $\wedge$: $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$. Any distribute lattice that has both a smallest element $\bot$ and a largest element $\top$ forms a semiring $(L, \vee, \wedge, \bot, \top)$.

A common trick to construct a lattice on a finite set L is to define only one of the two operations and its identity element, say $\vee$ and $\bot$. If $\vee$ is associative, commutative, and idempotent, then one can define the order relation as $x \sqsubseteq y$ iff $x \vee y = y$, and the greatest lower bound always exists because $x \wedge y \stackrel{\text{def}}{=} \bigvee \{z \in L \mid z \sqsubseteq x, z \sqsubseteq y\}$, where the $\bigvee$ is taken over a finite set, hence it exists by assumption.[1]

5.2.4 Partially Ordered Pre-Semirings

We present here the concept that combines (pre-)semirings and partial orders, following (Khamis *et al.*, 2022a).

A *Partially Ordered Pre-Semiring*, or *POPS*, is a structure $\boldsymbol{P} = (P, \oplus, \otimes, \mathbf{0}, \mathbf{1}, \sqsubseteq)$, where $(P, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ is a commutative pre-semiring, $(P, \sqsubseteq)$ is a partial order, and both operations $\oplus, \otimes$ are monotone w.r.t. $\sqsubseteq$. In other words, a POPS is both a pre-semiring and a partial order. We will usually denote a POPS by $\boldsymbol{P}$, and use the letter $\boldsymbol{S}$ to denote a semirings or a pre-semirings. As we shall see, the POPS is the algebraic structure needed to generalize datalog from sets, to K-relations.

A POPS satisfies the identities $\bot \oplus \bot = \bot$ and $\bot \otimes \bot = \bot$, because, by monotonicity, we have $\bot \oplus \bot \sqsubseteq \bot \oplus \mathbf{0} = \bot$, and $\bot \otimes \bot \sqsubseteq \bot \otimes \mathbf{1} = \bot$. We say that the multiplicative operator $\otimes$ is *strict* if the identity $x \otimes \bot = \bot$ holds for every $x \in P$.

A special case of POPS that have been widely studied in the literature are *naturally ordered pre-semirings*. Fix a pre-semiring $\boldsymbol{S} = (S, \oplus, \otimes, \mathbf{0}, \mathbf{1})$, and

[1] We need to require the presence of $\bot$ because, when the set is empty, then $\bigvee \emptyset = \bot$.

define the relation $x \preceq y$ as: $\exists z, x \oplus z = y$. Then $\preceq$ is reflexive and transitive, but it is not always antisymmetric; for example, in the semiring of reals, $\mathbb{R}$, the relation $x \preceq y$ holds for all $x, y \in \mathbb{R}$ (take $z = y - x$).

Definition 5.2.4. A pre-semiring $\boldsymbol{S}$ is called *naturally ordered* if the relation $x \preceq y$ if $\exists z, x \oplus z = y$ is antisymmetric. In that case, $\preceq$ is called the *natural order* on $\boldsymbol{S}$.

Every naturally ordered pre-semiring is also a POPS, and, moreover, every naturally ordered semiring is a strict POPS. However, there exists POPS that are not naturally ordered. An important such example is the POPS of *lifted reals*: $\mathbb{R}_\perp = (\mathbb{R} \cup \{\perp\}, +, \cdot, 0, 1, \sqsubseteq)$, obtained by extending the set of reals with one new element, $\perp$, and defining $x + \perp = \perp + x = \perp$, and similarly $x \cdot \perp = \perp \cdot x = \perp$. The order relation is defined as $x \sqsubseteq y$ if $x = \perp$ or $x = y$. Notice that $\mathbb{R}_\perp$ is pre-semiring, and not a semiring, because $0 \cdot \perp = \perp$.

In this paper we will denote by $\preceq$ the natural order on the pre-semiring, if it exists, and denote by $\sqsubseteq$ some arbitrary partial order.

5.3 Extending Datalog to Partially Ordered Pre-Semirings

We have seen that every relational query Q can be interpreted over K-relations, by simply replacing $\wedge, \vee, \bigvee$ with $\otimes, \oplus, \bigoplus$. We show here that a basic datalog program P can similarly be interpreted over any POPS, by defining its semantics as the least fixpoint of the immediate consequence operator, when it exists. This extension of datalog from sets to POPS was called $\mathsf{Datalog}^\circ$ in (Khamis *et al.*, 2022a; Wang *et al.*, 2022a), a term that we will also adopt here.

A $\mathsf{Datalog}^\circ$ program Q has EDBs $E_1, \ldots, E_m$ and IDBs $R_1, \ldots, R_n$, which are interpreted over the same POPS $\boldsymbol{P}$. The program consists of one rule for each IDB predicate:

$$\begin{aligned} R_1(\boldsymbol{X}_1) &\text{ :- } F_1 \\ &\ldots \\ R_n(\boldsymbol{X}_n) &\text{ :- } F_n \end{aligned} \tag{5.5}$$

where each F_i is a sum-sum-product expression, as defined in (5.2), over the predicates $E_1, \ldots, E_m, R_1, \ldots, R_m$. Denote by $\boldsymbol{E} = (E_1, \ldots, E_m)$ the tuple of EDBs, and by $\boldsymbol{R} = (R_1, \ldots, R_n)$ the tuple of IDBs. The *immediate consequence operator* of the program Q, denoted by $T_Q(\boldsymbol{E}, \boldsymbol{R})$, takes as input EDB and IDB instances, and returns a new state of the IDB relations, by computing each rule on the inputs $\boldsymbol{E}, \boldsymbol{R}$. The *semantics* of a $\mathsf{Datalog}^\circ$ program is the least fixpoint of its immediate consequence operator, $\boldsymbol{R} \mapsto T_Q(\boldsymbol{E}, \boldsymbol{R})$, when it exists, and undefined otherwise. We denote the least fixpoint as $\mathsf{lfp}(T_Q)$, when it exists. The naive evaluation procedure in Algorithm 1 can be extended directly to

Algorithm 6 for Datalog°. It is easy to see that the intermediate values of the IDB predicates form an ω-sequence: $\boldsymbol{R}^{(0)} \sqsubseteq \boldsymbol{R}^{(1)} \sqsubseteq \boldsymbol{R}^{(2)} \sqsubseteq \cdots$ Moreover, if $\boldsymbol{R}^{(k)} = \boldsymbol{R}^{(k+1)}$ for any k, then the algorithm terminates and returns $\mathsf{lfp}(T_Q)$.

$\boldsymbol{E}$ is the input EDBs;
$\boldsymbol{R} \leftarrow \bot$;
for $k \leftarrow 1$ **to** ∞ **do**
  $\boldsymbol{R}^{(k)} \leftarrow T_Q(\boldsymbol{E}, \boldsymbol{R}^{(k-1)})$;
  if $\boldsymbol{R}^{(k)} = \boldsymbol{R}^{(k-1)}$ **then**
    $\boldsymbol{R}^{*} \leftarrow \boldsymbol{R}^{(k)}$;
    Break
  end
end
return $\boldsymbol{R}^{*}$

Algorithm 6: Naïve evaluation of a Datalog° program Q.

Example 5.3.1 (All Sources Shortest Path)**.** Consider a graph $G = (V, E : V \times V \to \mathbb{R}_+ \cup \{\infty\})$, where $E(x, y) \in \mathbb{R}_+ \cup \{\infty\}$ represents the cost of the edge (x, y), or $E(x, y) = \infty$ if no such edge exists. We want to compute, for every pair of nodes $x, y \in V$, the length $R(x, y)$ of the shortest path from x to y. For that, we view E as a Trop-relation, and define the following Datalog° program:

$$R(x, y) \text{ :- } E(x, y) \oplus \bigoplus_z R(x, z) \otimes E(z, y)$$

When we replace $\oplus, \otimes$ with their concrete definitions $\min, +$ in Trop, then the program becomes:

$$R(x, y) \text{ :- } \min(E(x, y), \min_z(R(x, z) + E(z, y)))$$

The sequence $R^{(k)}$ computed by the Naive Algorithm 6 has a simple interpretation: $R^{(k)}(x, y)$ is the length of the shortest k-hop path from x to y, or ∞ if no such path exists.

Several extensions of datalog to various POPS have been proposed in the literature. During the 80s several authors studied the extension of datalog to bag semantics, which is equivalent as Datalog° over the semiring $\mathbb{N} \cup \{\infty\}$, see e.g. (Mumick and Shmueli, 1993). The first extension to abstract semirings was introduced by (Green *et al.*, 2007), who showed that datalog can be extended to naturally ordered semirings, and that a fixpoint always exists when the

semiring is ω-continuous (a concept that we will define shortly). Their main motivation was to define provenance for datalog queries, and they described *provenance power series*, which are natural extensions of the provenance polynomials introduced by the same authors; while a power series is an infinite object, (Green *et al.*, 2007) showed that the coefficient of any fixed monomial of this series is computable. (Madsen *et al.*, 2016) introduced Flix, an extension of datalog to a distributive lattice. Recall that a distributive lattice is, in particular, a semiring. (Khamis *et al.*, 2021) showed that datalog can be extended to arbitrary POPS, which generalize naturally ordered semirings, and studied the convergence of Datalog$^\circ$.

A parallel development happened in the programming language community, which developed algebraic program analysis methods, based on solving abstract dataflow equations, which are equivalent to polynomial equations in some semiring. This approach was pioneered by (Sharir and Pnueli, 1978), who considered equations over lattices, extended to ω-continuous semirings by (Esparza *et al.*, 2010), who also introduced Newton's method for computing fixpoints, which was later called *Newtonian Program Analysis*, NPA, by (Reps *et al.*, 2016). We will discuss polynomial equations in Sec. 5.4, and Newton's method in Sec. 5.6; we will not discuss program analysis, and refer the reader to the survey by (Kincaid *et al.*, 2021).

Add the example of the win-move program on the FOUR lattice – Dan.

5.4 Convergence of the Naive Algorithm

When does the least fixpoint $\mathsf{lfp}(T_Q)$ of a Datalog$^\circ$ program Q exists? When does the Naive Evaluation Algorithm 6 converge? The answer to these questions depends, in general, on algebraic properties of the semiring. We first discuss ω-continuous semirings, where the least fixpoint is guaranteed to exist, but where the Naive Algorithm does not necessarily converge. Then we introduce stable semirings, where convergence is guaranteed.

Before we start our discussion, we show that the fixpoint of a Datalog$^\circ$ program is equivalent to the fixpoint of polynomials over the same pre-semiring. Polynomial equations have been studied in the literature in several contexts, like context-free languages, program analysis, graph algorithms, and they also offer the natural framework for studying the convergence of Datalog$^\circ$ programs.

5.4.1 Polynomial Equations

Fix a set of variables $\boldsymbol{x} = \{x_1, \ldots, x_N\}$, and a commutative pre-semiring $\boldsymbol{S}$. A *monomial* is a formal expression of the form:

$$m = x_1^{k_1} \otimes \cdots \otimes x_N^{k_N} \tag{5.6}$$

where $k_1, \ldots, k_N \in \mathbb{N}$, and x^k is a shorthand for $x \otimes \cdots \otimes x$ k-times. A *polynomial* over $\boldsymbol{S}$ is a sum of distinct monomials multiplied with coefficients in $\boldsymbol{S}$.

$$f = (a_1 \otimes m_1) \oplus (a_2 \otimes m_2) \oplus \cdots \oplus (a_r \otimes m_r) \tag{5.7}$$

We denote by $\boldsymbol{S}[\boldsymbol{x}]$ the set of polynomials over $\boldsymbol{S}$. Notice that, when $\boldsymbol{S}$ is a pre-semiring, but not a semiring, then we cannot remove a monomial from (5.7) by setting its coefficient $a := \mathbf{0}$, because, in general, $\mathbf{0} \otimes x \neq \mathbf{0}$; for example, if we have a single variable x, then it would be wrong to define a polynomial as a formal expression $f = \bigoplus_{k=0,d} a_k \otimes x^k$, since we cannot remove unwanted monomials $a_k \otimes x^k$. Instead, (5.7) simply specifies the desired set of monomial explicitly.

Let $\boldsymbol{P}$ be a POPS, and fix a tuple of polynomials $f_1, \ldots, f_N \in \boldsymbol{P}[\boldsymbol{x}]$. A *set of polynomial equations* is a system of the form:

$$\begin{aligned} x_1 &= f_1(x_1, \ldots, x_N) \\ &\cdots \\ x_N &= f_N(x_1, \ldots, x_N) \end{aligned}$$

If $\boldsymbol{x}$ and $\boldsymbol{f}$ represent the tuples $(x_1, \ldots, x_N)$ and $(f_1, \ldots, f_N)$ respectively, then the set of polynomial equations can be written concisely as

$$\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x}) \tag{5.8}$$

We are interested in the least solution $\boldsymbol{x}$ of this polynomial equation, in other words, we want to compute $\mathsf{lfp}(\boldsymbol{f})$.

The *Kleene sequence* of a tuple of polynomials $\boldsymbol{f}$ is the following:

$$\boldsymbol{x}^{(0)} \stackrel{\text{def}}{=} \bot \qquad\qquad \boldsymbol{x}^{(k+1)} \stackrel{\text{def}}{=} \boldsymbol{f}(\boldsymbol{x}^{(k)}) \tag{5.9}$$

The reader may observe the analogy between the Kleene sequence and the Naive Evaluation Algorithm 6; we will make this precise below. As with the Naive Algorithm, we observe that the Kleene sequence is an ω-chain, $\boldsymbol{x}^{(0)} \sqsubseteq \boldsymbol{x}^{(1)} \sqsubseteq \boldsymbol{x}^{(2)} \sqsubseteq \cdots$ If $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)}$ for some $k \geq 0$, then $\mathsf{lfp}(\boldsymbol{f}) = \boldsymbol{x}^{(k)}$.

The least fixpoint of a $\mathsf{Datalog}^\circ$ program Q defined in (5.5) can be defined equivalently as the least fixpoint of a set of polynomial equations, as follows.

Fix the POPS $\boldsymbol{P}$, and fix a set of $\boldsymbol{P}$-EDB instances $E_1, \ldots, E_m$. We construct a standard instance $E_1^0, \ldots, E_m^0, R_1^0, \ldots, R_n^0$ of both EDBs and IDBs as follows. $E_j^0 \stackrel{\text{def}}{=} \mathsf{supp}(E_j)$, i.e. E_j^0 is the support of E_j (i.e. a finite set of tuples), and $R_i^0 \stackrel{\text{def}}{=} \mathsf{ADom}^{|\text{arity}(R_i)|}$, where ADom be the active domain of the relations $E_1^0, \ldots, E_m^0$, i.e. R_i^0 consists of all tuples formed with constants in ADom; this is the same as the Herbrand Universe, HB, that we used in Chapt. 3. Let M and N be the total number of tuples in $E_1^0, \ldots, E_m^0$ and $R_1^0, \ldots, R_n^0$ respectively. Consider the *abstract tagging* of these relations, where we tag EDB relations with variables $\boldsymbol{a} = \{a_1, \ldots, a_M\}$ and the IDB relations with variables $\boldsymbol{x} = \{x_1, \ldots, x_N\}$ (see Sec. 5.2.2). Each rule of the $\mathsf{Datalog}^\circ$ program (5.5) defines a provenance polynomial $f_i(\boldsymbol{a}, \boldsymbol{x}) \in \mathbb{N}[\boldsymbol{a}, \boldsymbol{x}]$, one for each tuple in $R_1^0, \ldots, R_n^0$. We substitute the variables $\boldsymbol{a}$ with their corresponding values in $\boldsymbol{P}$ given by the EDBs $E_1, \ldots, E_m$, and the polynomial becomes $f_i(\boldsymbol{x}) \in \boldsymbol{P}[\boldsymbol{x}]$. We define the *grounded* $\mathsf{Datalog}^\circ$ *program* to be the following set of N rules:

$$\begin{aligned} x_1 &= f_1(x_1, \ldots, x_N) \\ &\ldots \\ x_N &= f_N(x_1, \ldots, x_N) \end{aligned} \tag{5.10}$$

In other words, each variable x_i associated to an IDB tuple is defined by some polynomial $f_i \in \boldsymbol{P}[\boldsymbol{x}]$. It is not hard to check that the least fixpoint of (5.10), if it exists, is precisely the least fixpoint of the $\mathsf{Datalog}^\circ$ program (5.5). We notice that the grounded program (5.10) is quite similar to the grounded datalog program defined in Chapter 2. The minor difference is that we do not apply the idempotence rule: if the same monomial m occurs twice, then the grounded program will have a monomial $2m$ (which is equivalent to $m \oplus m$), while in the grounding of basic datalog we had $m \vee m = m$, hence only one occurrence of m suffices.

Example 5.4.1. Consider the following $\mathsf{Datalog}^\circ$ program:

$$\begin{aligned} R(X, Y) &\text{ :- } E(X, Y) \\ R(X, Y) &\text{ :- } \bigoplus_Z R(X, Z) \otimes R(Z, Y) \end{aligned}$$

When interpreted over the Booleans, this is the standard non-linear transitive closure datalog program. Consider the following dataset:

X	Y	$E(X,Y)$
m	n	a
n	m	b
n	n	c

The active domain is $\mathsf{ADom} = \{m, n\}$, thus the IDB R has four tuples, and its abstract tagging is:

X	Y	$R(X,Y)$
m	m	x
m	n	y
n	m	z
n	n	u

where x, y, z, u are fresh variables. The grounded $\mathsf{Datalog}^\circ$ program is the following system of polynomial equations, where we replace the bulky $\oplus, \otimes$ with $+, \cdot$, and even write yz instead of $y \cdot z$:

$$\begin{aligned} x &= yz \\ y &= a + yu \\ z &= b + uz \\ u &= c + u^2 + zy \end{aligned}$$

Thus, from now on we will consider a system of polynomial equations (5.8). The Kleene sequence (5.9) of the grounded program is precisely the sequence of IDB relations of the Naive Evaluation Algorithm 6. The questions asked at the beginning of this section become: when does a set of polynomial equations (5.8) have a least fixpoint? And when does the Kleene sequence converge, in other words, when is the Kleene sequence finite?

5.4.2 ω-Continuous Semirings

Most of the literature on polynomial equations over semirings has required the semiring to be ω-continuous, which guarantees that every polynomial equation has a least fixpoint. We present here the main ideas, following mostly (Green *et al.*, 2007).

Call a semiring $\boldsymbol{S} = (S, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ *ω-continuous* if it is naturally ordered (Def. 5.2.4), the poset $(S, \preceq)$ is ω-complete, and both $\oplus$, $\otimes$ are ω-continuous. While the notion of ω-continuity could also be extended to an arbitrary POPS, where the order $\sqsubseteq$ may be different from the natural order $\preceq$, ω-continuous semirings are defined in the literature only for the natural order, and we will adopt the same definition here.

Let $\boldsymbol{S}$ be a commutative semiring, and let $f \in \boldsymbol{S}[\boldsymbol{x}]$ be a polynomial in N variables. The polynomial defines a function $\boldsymbol{S}^N \to \boldsymbol{S}$, in an obvious way. With some abuse, we denote the function with the same symbol, f, and call it a *polynomial function*.

If $\boldsymbol{S}$ is ω-continuous, then every polynomial function is also ω continuous. In other words, given an ω-chain $\boldsymbol{x}^{(0)} \sqsubseteq \boldsymbol{x}^{(1)} \sqsubseteq \cdots$ in $\boldsymbol{S}^N$, we have $f(\bigvee_k \boldsymbol{x}^{(k)}) =$

$\bigvee_k f(\boldsymbol{x}^{(k)})$. Consider now a tuple of N polynomials, $\boldsymbol{f} = (f_1, \ldots, f_N)$, and let $\boldsymbol{x}^{(k)}$ be the Kleene sequence defined in (5.9). On one hand, $\bigvee_k \boldsymbol{x}^{(k)}$ is a fixpoint of $\boldsymbol{f}$, because $\boldsymbol{f}(\bigvee_k \boldsymbol{x}^{(k)}) = \bigvee_k f(\boldsymbol{x}^{(k)}) = \bigvee_k \boldsymbol{x}^{(k+1)} = \bigvee_k \boldsymbol{x}^{(k)}$, and, on the other hand, by Prop. 5.2.2, $\bigvee_k \boldsymbol{x}^{(k)}$ is below any fixpoint of $\boldsymbol{f}$. We have proven:

Theorem 5.4.2. If $\boldsymbol{S}$ is an ω-continuous semiring, then every polynomial equation $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x})$ has a least fixpoint. Moreover, $\mathsf{lfp}(\boldsymbol{f}) = \bigvee_k \boldsymbol{x}^{(k)}$, where $\boldsymbol{x}^{(k)}$ is the Kleene sequence defined in (5.9). As a consequence, every $\mathsf{Datalog}^\circ$ program also has a least fixpoint.

The following are some examples of ω-continuous semirings:

- The natural numbers extended with infinity, $\mathbb{N}^\infty \stackrel{\text{def}}{=} (\mathbb{N} \cup \{\infty\}, +, \times, 0, 1)$. We need to add ∞ in order to define the least upper bound of $0 \leq 1 \leq 2 \leq \cdots$
- The tropical semiring $\mathsf{Trop} = (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$. Here the natural order is the reverse of the standard order of real numbers: $x \preceq y$ iff $x \geq y$.
- The semiring $([0, 1], \max, \min, 0, 1)$, which is related to fuzzy sets.
- More generally, call a lattice $\boldsymbol{L} = (L, \sqsubseteq)$ *complete* if $\bigvee A$ exists for any set $A \subseteq L$; it follows that $\boldsymbol{L}$ has a smallest element $\hat{0}$ (namely $\bigvee \emptyset$), a largest element $\hat{1}$ (namely $\bigvee L$), and that $\bigwedge A$ exists for any set A, because $\bigwedge A = \bigvee\{x \in L \mid \forall y \in A, x \sqsubseteq y\}$. The lattice is called a *complete, distributive lattice* if the identity $(\bigvee A) \wedge x = \bigvee\{a \wedge x \mid a \in A\}$ holds. Every complete, distributive lattice is an ω-continuous semiring, $(L, \vee, \wedge, \hat{0}, \hat{1})$. For example if Ω is any set, then $(\mathcal{P}(\Omega), \subseteq)$ is a complete, distributive lattice, and its associated ω-continuous semiring is $(\mathcal{P}, \cup, \cap, \emptyset, \Omega)$.

An important example of an ω-continuous semiring is the set of formal power series, which was used by (Green *et al.*, 2007) to compute the provenance of a datalog program. Fix a set of N variables, $\boldsymbol{x} = \{x_1, \ldots, x_N\}$. Recall the definition of a monomial in Eq. 5.6, and let M be the set of all monomials. A *formal power series* is an expression:

$$F \stackrel{\text{def}}{=} \bigoplus_{m \in M} a_m \otimes m \tag{5.11}$$

where $a_m \in S$ for all $m \in M$. Thus, a power series generalizes a formal polynomial (5.7) from a finite sum, to an infinite sum. The set of formal power series is denoted $\boldsymbol{S}[[\boldsymbol{x}]]$, and forms a semiring with the standard operations

of addition and multiplication of power series. When the semiring $\boldsymbol{S}$ is ω-continuous, then $\boldsymbol{S}[[\boldsymbol{x}]]$ is also ω-continuous.

(Green *et al.*, 2007) show that power series provide a natural extension of provenance polynomials (Sec. 5.2.2) from non-recursive queries to datalog queries. Let Q be a basic datalog program and let $E_1, \ldots, E_m$ be input EDB relation instances. Here each EDB is a standard relation, i.e. a set of tuples. Let M be the total number of EDB tuples, and let $\boldsymbol{a} = \{a_1, \ldots, a_M\}$ be a set of variables. Convert the input EDBs from standard relations to $\mathbb{N}^\infty[[\boldsymbol{a}]]$-relations, by annotating each tuple with a distinct variable a_i; this is the *abstract tagging* from Sec. 5.2.2. Our datalog program now becomes a $\mathsf{Datalog}^\circ$ program, over $\mathbb{N}^\infty[[\boldsymbol{a}]]$-relations. Consider its *grounded program* $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x})$, as defined in Eq. (5.10), where the variables $\boldsymbol{x} = \{x_1, \ldots, x_N\}$ are associated to the IDB tuples, and each f_i is a polynomial, with coefficients in $\mathbb{N}^\infty[[\boldsymbol{a}]]$ (i.e. $f_i \in (\mathbb{N}^\infty[[\boldsymbol{a}]])[\boldsymbol{x}]$). Since the semiring $\mathbb{N}^\infty[[\boldsymbol{a}]]$ is ω-continuous, the polynomial equation $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x})$ has a least fixpoint, $\mathsf{lfp}(\boldsymbol{f})$. Each component i of this least fixpoint is a power series called the *provenance series* of the output tuple associated to the variable x_i.

For a simple example, assume that the grounded datalog program consists of a single polynomial equation:

$$x = b + ax^2$$

Here a, b are variables associated to EDB tuples, and x is the variable associated to the unique IDB tuple. Thus, the equation is $x = f(x)$, where $f = b + ax^2$ is a polynomial in x, with coefficients in $\mathbb{N}^\infty[[a, b]]$. Then one can show that the least fixpoint is the following power series (see (Green *et al.*, 2007) or (Wang *et al.*, 2022b)):

$$\mathsf{lfp}(f) = \sum_{n=0}^{\infty} \frac{1}{n+1} \binom{2n}{n} a^n b^{n+1}$$

While the provenance series is an infinite object, and thus cannot be computed explicitly, (Green *et al.*, 2007) show that all finitely representable information can be computed. For example they show that for any monomial m, the coefficient a_m is computable, and that it is decidable whether the power series is finite, i.e. if it is in $\mathbb{N}[x]$.

to check out this paper: (Deutch *et al.*, 2014) –Dan.

5.4.3 Stable Semirings

We have seen that, if the semiring $\boldsymbol{S}$ is ω-continuous, then every $\mathsf{Datalog}^\circ$ program has a least fixpoint. Our next question is whether this least fixpoint can be computed by the Naive Algorithm 6. In other words, we ask whether

the Naive Algorithm always terminates. When the Naive Algorithm terminates for some Datalog° program Q, then we simply say that Q terminates, or converges. It is not hard to see (and we will prove shortly) that every Datalog° program terminates if the semiring is the set of Booleans, or if it is Trop, or $\mathbb{R}_\perp$. However, Datalog° programs may diverge if the semiring is $\mathbb{N}^\infty$ even though this semiring is ω-continuous.

A simple sufficient convergence condition often used in the literature is the Ascending Chain Condition (ACC), see e.g. (Nielson *et al.*, 1999; Esparza *et al.*, 2010; Madsen *et al.*, 2016). A partially ordered set $(P, \sqsubseteq)$ satisfies the Ascending Chain Condition if there are no infinite ω-chain, in other words any strictly increasing chain $x_0 \sqsubset x_1 \sqsubset \cdots$ is finite. However, ACC is only a sufficient, but not necessary condition. For example, every Datalog° converges on Trop, yet the natural order on Trop does not satisfy ACC.

We show in this section that convergence is fully characterized by a different property of the semiring, called *stability*, and, moreover, this characterization extends from semirings to POPS. Throughout this section we will use the symbols $+, \cdot$ instead of the abstract $\oplus, \otimes$, to reduce clutter.

The notion of stability is justified by the following observation. Consider a simple polynomial equation in a semiring $\boldsymbol{S}$:

$$x = \mathbf{1} + a \cdot x \tag{5.12}$$

For any $p \geq 0$, denote by $a^{(p)}$ the following expression:

$$a^{(p)} = 1 + a + a^2 + \cdots + a^p \tag{5.13}$$

It is not hard to see that the Kleene sequence of (5.12) is precisely $(a^{(k)})_{k \geq 0}$. It follows that the Kleene sequence is finite iff there exists $p \geq 0$ such that $a^{(p)} = a^{(p+1)}$. This justifies the following definition:

Definition 5.4.3. Fix a semiring $\boldsymbol{S}$.

- An element $a \in S$ is called *p-stable* if $a^{(p)} = a^{(p+1)}$, where $p \geq 0$. An element $a \in S$ is called *stable*, if there exists $p \geq 0$ such that a is p-stable.
- The semiring $\boldsymbol{S}$ is called *p-stable* if every $a \in S$ is p-stable, for some fixed $p \geq 0$; in that case we also say that $\boldsymbol{S}$ is *uniformly stable*. The semiring $\boldsymbol{S}$ is called *stable* if every $a \in S$ is stable; we also say that $\boldsymbol{S}$ is *non-uniformly stable*.

The difference between a uniformly stable semiring and a (non-uniformly) stable semiring is that in the former all elements $a \in S$ are p-stable for the same p, while in the latter p may depend on a. The following elegant property was shown in (Gondran and Minoux, 2008):

Proposition 5.4.4. If $\mathbf{1}$ is p-stable then $\boldsymbol{S}$ is naturally ordered. In particular, every stable semiring is naturally ordered.

Proof. If $\mathbf{1}$ is p-stable, then $\mathbf{1}^{(p+1)} = \mathbf{1}^{(p)}$, which means that $\mathbf{1} + \mathbf{1} + \cdots + \mathbf{1}(p+1 \text{ times}) = \mathbf{1} + \cdots + \mathbf{1}(p \text{ times})$, or, in short, $p + \mathbf{1} = p$. We prove that, if $a \preceq b$ and $b \preceq a$ hold, then $a = b$. By definition of $\preceq$, there exist x, y such that $a + x = b$ and $b + y = a$. On one hand we have $a = b + y = a + x + y$, which implies $a = a + k(x + y)$ for every $k \geq 0$, in particular $a = a + p(x + y)$. On the other hand we have $b = a + x = a + k(x + y) + x = a + (k + 1)x + ky$. We set $k = p$ and obtain $b = a + (p + 1)x + py = a + px + py = a + p(x + y)$ and the latter we have seen is $= a$, proving that $a = b$. □

Many important semirings are uniformly stable. For example, $\mathbb{B}$, Trop, and any semiring defined by a lattice $(L, \vee, \wedge, \hat{0}, \hat{1})$ is 0-stable. For example in the Boolean semiring we have $a^{(0)} = 1$ and $a^{(1)} = 1 \vee a = 1$, while in Trop we have $a^{(0)} = 0$ and $a^{(1)} = \min(0, a) = 0$. In fact the case of 0-stable semiring has been most extensively studied. Such semirings are called *simple* by (Lehmann, 1977), are called *c-semirings* by (Kohlas and Wilson, 2008), and *absorptive* by (Dannert *et al.*, 2021). In all cases, the authors require $1 + a = 1$ for all a (or, equivalently, $b + ab = b$ for all a, b (Dannert *et al.*, 2021)), which is equivalent to stating that a is 0-stable, and also equivalent to stating that $(S, +)$ is a join-semilattice with maximal element 1. The next two examples illustrate two stable semirings which are not 0-stable; they were first introduced by (Gondran and Minoux, 2008) and described in more detail in (Khamis *et al.*, 2021).

Example 5.4.5. Fix a number $p \geq 0$. The semiring Trop_p is defined as follows. It consists of bags of $p+1$ elements in $\mathbb{R}_+ \cup \{\infty\}$. The operation $a \oplus_p b$ takes the union of the bags a, b and returns the smallest $p + 1$ elements. The operation $a \otimes_p b$ computes the bag $\{\!\{x + y \mid x \in a, y \in b\}\!\}$, then returns the smallest $p + 1$ elements. The identities are $\mathbf{0}_p = \{\!\{\infty, \ldots, \infty\}\!\}$ ($p + 1$ copies of ∞) $\mathbf{1}_p = \{\!\{0, \infty, \ldots, \infty\}\!\}$ ($p + 1$ elements in total). For a simple example, if $p = 2$ then $\{\!\{3, 7, 9\}\!\} \oplus_2 \{\!\{3, 7, 7\}\!\} = \{\!\{3, 3, 7\}\!\}$ and $\{\!\{3, 7, 9\}\!\} \otimes_2 \{\!\{3, 7, 7\}\!\} = \{\!\{6, 10, 10\}\!\}$. Then Trop_p is a p-stable semiring, and, when $p \geq 1$, Trop_p is not $p - 1$-stable. When $p = 0$ then Trop_0 is the standard tropical semiring Trop.

The Trop_p semiring is very useful for some graph computations. For example, we have seen in Example 5.3.1 a $\mathsf{Datalog}^\circ$ program that, when interpreted over Trop, computes the All Sources Shortest Path in a graph E. If we interpret this program over Trop_p, then it computes the lengths of the shortest $p + 1$ paths between any two nodes.

Example 5.4.6. Fix a real number $\eta \geq 0$. We define the semiring $\mathsf{Trop}_{\leq \eta}$ as follows. Its elements are finite, nonempty sets $a = \{x_1, \ldots, x_n\}$ where $x_i \in \mathbb{R}_+ \cup \{\infty\}$, such that $\max(a) - \min(a) \leq \eta$. The operation $a \oplus_{\leq \eta} b$ computes the union $c = a \cup b$, then returns the set of elements in c that are

$\leq \min(c) + \eta$. The operation $a \otimes_{\leq \eta} b$ computes $c = \{x + y \mid x \in a, y \in b\}$ and returns the set of elements in c that are $\leq \min(c) + \eta$. For example, if $\eta = 6.5$ then: $\{3, 7\} \oplus_{\leq \eta} \{5, 9, 10\} = \{3, 5, 7, 9\}$ and $\{1, 6\} \otimes_{\leq \eta} \{1, 2, 3\} = \{2, 3, 4, 7, 8\}$. The identities are $\mathbf{0}_{\leq \eta} = \{\infty\}$ and $\mathbf{1}_{\leq \eta} = \{0\}$. Then $\mathsf{Trop}_{\leq \eta}$ is a non-uniformly stable semiring. Here, too, Trop is a special case of $\mathsf{Trop}_{\leq \eta}$, namely when $\eta = 0$.

If we interpret the $\mathsf{Datalog}^\circ$ program in Example 5.3.1 over $\mathsf{Trop}_{\leq \eta}$, then it computes, for any two pairs of nodes, the lengths of all paths that differ by no more than η from the shortest path.

The following was proven by (Khamis *et al.*, 2022a) (see also the extended version by (Khamis *et al.*, 2021)):

Theorem 5.4.7. Fix a commutative semiring $\boldsymbol{S}$.

- If $\boldsymbol{S}$ is stable, then the Kleene sequence of a polynomial equation (5.9) converges.
- If $\boldsymbol{S}$ is p-stable, for some $p \geq 0$, then the Kleene sequence converges in at most $\sum_{i=1,N}(p+2)^i$ steps, where N denotes the number of possible IDB tuples.
- If $\boldsymbol{S}$ is 0-stable, then the Kleene sequence converges in at most N steps.

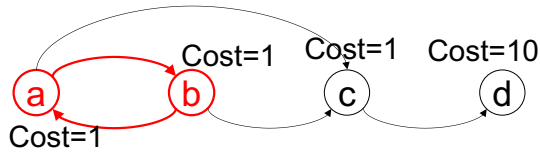
Thus, stability is a necessary and sufficient condition for the Kleene sequence of any set of polynomial equations to converge. Necessity follows from the fact that the Kleene sequence of the polynomial equation (5.12) is $a^{(p)}$, while the theorem above proves sufficiency. The second bullet of the theorem gives us an upper bound on the runtime of the naive algorithm, while the third sharpens this bound in the special case $p = 0$.

Using Theorem 5.4.7, (Khamis *et al.*, 2022a) described a full characterization of the POPS for which every $\mathsf{Datalog}^\circ$ program converges. To present that characterization, we need a definition.

Fix a POPS $\boldsymbol{P} = (P, \oplus, \otimes, \mathbf{0}, \mathbf{1}, \sqsubseteq)$, and recall that the POPS requires the pre-semiring to be commutative. Assume that $\boldsymbol{P}$ is strict, meaning that $x \otimes \bot = \bot$ for all $x \in P$. Define the *core* of $\boldsymbol{P}$ as the set $P \oplus \bot \stackrel{\text{def}}{=} \{x \oplus \bot \mid x \in P\}$. It is immediate to check that the core of a POPS $\boldsymbol{P}$ is a commutative semiring, with the same operations $\oplus, \otimes$ as $\boldsymbol{P}$, and with identities $\mathbf{0} \oplus \bot$ and $\mathbf{1} \oplus \bot$.

For a simple illustration, if the POPS is the tropical semiring Trop ordered with the natural order, then the core is itself, because $\bot = \infty$ and the set $\{\min(x, \infty) \mid x \in \mathbb{R}_+ \cup \{\infty\}\}$ is equal to $R_+ \cup \{\infty\}$. On the other hand, if the POPS is $\mathbb{R}_\bot$, then the core is the set $\{\bot\}$, which is a trivial semiring.

(Khamis *et al.*, 2022a) proved the following (see their extended version (Khamis *et al.*, 2021)):



	$T(a)$	$T(b)$	$T(c)$	$T(d)$
$T^{(0)}$	0	0	0	0
$T^{(1)}$	1	1	1	10
$T^{(2)}$	3	3	11	10
$T^{(3)}$	15	15	11	10
$T^{(4)}$	27	27	11	10
$T^{(5)}$	39	39	11	10
$T^{(6)}$	...	...	...	...

	$T(a)$	$T(b)$	$T(c)$	$T(d)$
$T^{(1)}$	$\bot$	$\bot$	$\bot$	$\bot$
$T^{(2)}$	$\bot$	$\bot$	$\bot$	10
$T^{(3)}$	$\bot$	$\bot$	11	10
$T^{(4)}$	$\bot$	$\bot$	11	10

Figure 5.1: A graph with cycles on which we attempt to compute the bill-of-material query (5.14). The Naive Algorithm diverges if interpreted over the semiring $\mathbb{N}^\infty$, but it converges if interpreted over the POPS $\boldsymbol{R}_\bot$.

Theorem 5.4.8. Let $\boldsymbol{P}$ be a strict POPS. Assume that the operations $\oplus, \otimes$ can be computed in constant time. Then the following hold:

- Every Datalog$^\circ$ program converges iff the semiring $\boldsymbol{P} \oplus \bot$ is stable.
- Every Datalog$^\circ$ program converges in a number of steps that depends only on the size of the support of the input EDBs iff the semiring $\boldsymbol{P} \oplus \bot$ is uniformly stable.
- If $\boldsymbol{P} \oplus \bot$ is 0-stable, then every Datalog$^\circ$ program converges in a number of steps polynomial in the size of the support of the input EDBs.

It follows immediately that every Datalog$^\circ$ program converges on the semirings (or POPS): $\mathbb{B}$, Trop_p, $\mathsf{Trop}_{\leq \eta}$, $\mathbb{R}_\bot$, and any lattice $(L, \vee, \wedge, \hat{0}, \hat{1})$, since in each cases the core is a stable semiring. On the other hand, there are programs that diverge when interpreted over $\mathbb{N}^\infty$ (because the core is itself, and is not stable).

We illustrate with an example.

Example 5.4.9. Consider the following *bill of materials*, adapted from (Ross and Sagiv, 1992). We have a hierarchy of parts, described an edge relation $E(X, Y)$ saying that "X has subpart Y", and each part X has a cost $C(X) \in \mathbb{N}$. The task is to compute, for each part X, the total sum of its own cost and the cost of all its direct or indirect subparts, denoted $T(X) \in \mathbb{N}$. We model both

$E(X,Y)$ and $C(X)$ as $\mathbb{N}^\infty$-relations, where $E(x,y) = 0$ or 1, while $C(x)$ is the cost of the node x, and compute $T(X)$ using the following Datalog$^\circ$ program:

$$T(X) = C(X) + \sum_Y E(X,Y) \cdot T(Y) \tag{5.14}$$

We used here directly the standard notations $+, \cdot$ in $\mathbb{N}^\infty$ instead of abstract $\oplus, \otimes$. If $E(X,Y)$ represents a DAG, then Datalog$^\circ$ computes the bill-of-material correctly, and the Naive Algorithm converges in the number of steps equal to the longest path in the DAG.

However, what happens if E has cycles? Fig. 5.1 shows a simple graph with cycles, together with the intermediate steps of the Naive Evaluation Algorithm. Obviously, the cycle between the nodes a and b cause the values of $T(a), T(b)$ to grow indefinitely.

The solution is to interpret the program over the POPS $\mathbb{R}_\perp$. In this case all values of $R(X)$ start at $\perp$. Since addition and multiplication are strict, $z + \perp = z \cdot \perp = \perp$, the values of $T(a), T(b)$ can never escape $\perp$, since they depend on each other. It follows that the Naive Algorithm terminates in 4 steps.

However, the careful reader may have noticed that we encounter a problem if we interpret the Datalog$^\circ$ program (5.14) using $\mathbb{R}_\perp$. As we defined the semantics of a sum-product expression in Equation (5.3), none of the values $T(a), T(b), T(c), T(d)$ can escape $\perp$. For example, the grounded rule for $T(d)$ is:

$$T(d) = C(d) + E(d,a) \cdot T(a) + E(d,b) \cdot T(b) + E(d,c) \cdot T(c) + E(d,d) \cdot T(d)$$

While $E(d,d) = 0$, when $T(d) = \perp$ (which holds at the beginning of the Naive Algorithm) then $E(d,d) \cdot T(d) = 0 \cdot \perp = \perp$. Furthermore, $x + \perp = \perp$ implies that $T(d)$ can never escape the value $\perp$. A simple work-around to this, proposed by (Khamis *et al.*, 2022a), is to extend the syntax of a sum-product expression (5.1) to specify explicitly the range of the summation variable. Using this syntax, we can rewrite the Datalog$^\circ$ program (5.14) as:

$$T(X) = C(X) + \sum_Y \{T(Y) \mid E(X,Y) = 1\}$$

The grounded program will only expand the summation as specified by the relation E, and become:

$$\begin{aligned} T(a) &= C(a) + T(b) \\ T(b) &= C(b) + T(a) + T(c) \\ T(c) &= C(c) + T(d) \\ T(d) &= C(d) \end{aligned}$$

The naive algorithm converges on $\mathbb{R}_\perp$, as shown in Fig. 5.1.

5.5 The Closure Operator

The Datalog$^\circ$ evaluation problem, or solving a set of polynomials equations $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x})$, bears a striking similarity to Kleene's theorem for regular languages. The theorem states that, if a language is accepted by an automaton, then it is a regular language. Let's take a closer look at this classic result. If the language accepted by an automaton, then it can be described by a set of linear equations $\boldsymbol{x} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}$ over the non-commutative semiring of regular sets $\mathbf{Reg}_\Sigma$, where $+, \cdot$ are set union and concatenation, $\boldsymbol{A}$ is the transition matrix of the automaton, and $\boldsymbol{b}$ is a vector of $\{\varepsilon\}$ and $\emptyset$ values on positions of final and non-final states respectively. In short, a language is recognized by an automaton iff it is the *fixpoint of a linear system*. This is quite similar to our polynomial equations. However, we solved polynomial equations using Kleene's sequence (5.9), which is an infinite process, while Kleene's theorem says that we can express the regular language using a finite expression, consisting of the operators $+, \cdot$, and the closure a^*. Thus, in Kleene's theorem there is no need for an infinite sequence, instead the solution is given explicitly, by a finite expression. This is surprising, and leads to a natural question. Could we compute the output of a Datalog$^\circ$ program explicitly, by using a closure operator, and bypassing the Naive Evaluation Algorithm? This venue has not yet been explored for either datalog or Datalog$^\circ$, however, it has been explored in other, related settings, in the context of closed semirings and Kleene Algebras. For that reason, this section will be limited to reviewing the background material of closed semirings and Kleene Algebras, and include only a high level discussion of their potential applicability to datalog.

The key starting observation is the fact that, in order to be able to evaluate Datalog$^\circ$ programs, we must at least be able to compute the least fixpoint of a linear equation, $x = \mathbf{1} + a \cdot x$. Denote by a^* the least fixpoint of this equation, and call it the *closure* of a. In this section we will assume that, in addition to the semiring operators $\oplus, \otimes$, we are also given the *-operator. The question is whether we can use only these three operators to construct an explicit expression for the fixpoint of a Datalog$^\circ$ program, or, equivalently, of a polynomial equation.

5.5.1 Closed Pre-Semirings

(Lehmann, 1977) observed that several algorithms discovered in different communities share a common structure, and a common correctness proof. This includes Warshall's algorithm for computing the transitive closure of a Boolean matrix, Floyd's algorithms for the minimum-cost paths, and Gauss-Jordan's method for inverting real matrices. (Lehmann, 1977) presented these quite disparate algorithms in a simple, unified framework by using the notion of a closed pre-semiring.

Definition 5.5.1. A *closed pre-semiring* is a structure $\boldsymbol{S} = (S, +, \cdot, ^*, \mathbf{0}, \mathbf{1})$, where $(S, +, \cdot, \mathbf{0}, \mathbf{1})$ is a (not necessarily commutative) pre-semiring, and the unary operator * called *closure* satisfies: $a^* = \mathbf{1} + a \cdot a^* = \mathbf{1} + a^* \cdot a$.

(Lehmann, 1977) calls the structure simply a closed semiring, but we prefer to call it a closed pre-semiring, since this is today standard terminology when multiplication is not absorptive. We also note that the term closed semiring has various other definitions in the literature, which are more restrictive than Def. 5.5.1.

What is remarkable about this definition is what is *missing*: the absorption rule $a \cdot \mathbf{0} = \mathbf{0} \cdot a = 0$ is missing (hence *pre*-semiring), multiplication need not be commutative $a \cdot b \neq b \cdot a$, and various other identities of the closure operator that have been considered in the literature are missing too, like:

$$(a + b)^* = (a^* \cdot b)^* \cdot b \qquad (a \cdot b)^* = \mathbf{1} + a \cdot (b \cdot a)^* \cdot b \tag{5.15}$$

There exist closed pre-semirings where these identities fail.

The key technical result proven by (Lehmann, 1977) is that the set of matrices $\boldsymbol{S}^{N \times N}$ also forms a closed pre-semiring, except for the fact that it does not have an identity for multiplication, i.e. in general $I_n \cdot \boldsymbol{A} \neq \boldsymbol{A}$, and $\boldsymbol{A} \cdot I_n \neq \boldsymbol{A}$ (see Sec. **??**). More precisely, (Lehmann, 1977) defines the following operation $\boldsymbol{A}^*$ on matrices. If the matrix is a 1×1 matrix, then $[a]^* \stackrel{\text{def}}{=} [a^*]$. Otherwise, write $\boldsymbol{A}$ as a block matrix:

$$\boldsymbol{A} = \begin{pmatrix} B & C \\ D & E \end{pmatrix}$$

where B, E are square matrices (C, D need not be square), then define:

$$\boldsymbol{A}^* \stackrel{\text{def}}{=} \begin{pmatrix} B^* + B^* C \Delta^* D B^* & B^* C \Delta^* \\ \Delta^* D B^* & \Delta^* \end{pmatrix} \quad \text{where } \Delta = E + D B^* C \tag{5.16}$$

(Lehmann, 1977) proves that (a) the definition of $\boldsymbol{A}^*$ does not depend on the block representation of $\boldsymbol{A}$, and (b) the operator $\boldsymbol{A}^*$ satisfies the axioms of the closed semiring: $\boldsymbol{A}^* = I_n + \boldsymbol{A} \cdot \boldsymbol{A}^* = I_n + \boldsymbol{A}^* \cdot \boldsymbol{A}$. The consequence of interest to us is the following: every linear equation $\boldsymbol{x} = \boldsymbol{A} \cdot \boldsymbol{x} + \boldsymbol{b}$ has the solution $\boldsymbol{x} = \boldsymbol{A}^* \cdot \boldsymbol{b}$, where the matrix $\boldsymbol{A}^*$ has an explicit, closed-form expression, given by the recursive definition (5.16). This gives us the desired closed form solution for a linear polynomial equation.

The definition of $\boldsymbol{A}^*$ above is a little more complex than similar definitions in the literature, because the notion of a closed pre-semiring is very general. (Lehmann, 1977) explains that, if one requires the closed semiring to include the absorption rule plus the two identities in Eq. (5.15), then $B^* + B^* C \Delta^* D B^*$ can be replaced by $(B + C E^* D)^*$, thus simplifying the definition of $\boldsymbol{A}^*$; (Lehmann,

1977) conjectures that this is possible even without the absorption rule. Next, (Lehmann, 1977) presents several algorithms over closed pre-semirings, which, when instantiated to particular closed pre-semirings specialize to classical algorithms such as Warshall's algorithm, Floyd's algorithm, Gauss' method, and Dijkstra's shortest path algorithm. We will not include them here and refer the reader to (Lehmann, 1977).

What are the applications to Datalog°? Fix a POPS $\boldsymbol{P}$, which, recall, is commutative, and assume that it is ω-continuous (recall: any ω sequence has a least upper bound, and multiplication distributes over the lub of ω-sequences). Then, for every $a \in P$, the equation $x = a \cdot x + \mathbf{1}$ has a least fixpoint given by $a^* \stackrel{\text{def}}{=} \bigvee_n a^n$. It is not hard to check that the POPS extended with the closure operator * forms a closed pre-semiring.

Consider now a *linear* Datalog° program Q over the POPS $\boldsymbol{P}$, where "linear" means that every sum-product expression in any rule of the Datalog° program (5.5) has at most one IDB predicate. Its grounding consists of a set of polynomial equations:

$$\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x})$$

see (5.10), where each polynomial f_i has degree ≤ 1, meaning that every monomial occurring in its definition is either a single variable, or $\mathbf{1}$, see (5.7). Assume that we can write the function $\boldsymbol{f}(\boldsymbol{x})$ as

$$\boldsymbol{f}(\boldsymbol{x}) \stackrel{\text{def}}{=} \boldsymbol{A} \cdot \boldsymbol{x} + \boldsymbol{b}$$

where $\boldsymbol{A}, \boldsymbol{b}$ are a matrix and a vector respectively. (This is always possible when $\boldsymbol{P}$ is a semiring, but may not be possible when $\boldsymbol{P}$ is a pre-semiring.) In other words, we assume that the grounding of the Datalog° program is the linear equation:

$$\boldsymbol{x} = \boldsymbol{A} \cdot \boldsymbol{x} + \boldsymbol{b}$$

Instead of computing the fixpoint using the Kleene sequence (5.9), we could compute it explicitly, as:

$$\boldsymbol{x} \stackrel{\text{def}}{=} \boldsymbol{A}^* \cdot \boldsymbol{b}$$

Thus, in theory, it is possible to bypass the Naive Evaluation Algorithm for linear Datalog° programs and replace it with some efficient closure computation algorithm. Such algorithms exists, see for example the WFK algorithm and the Gaussian elimination algorithm in (Lehmann, 1977), and some have been optimized to exploit the sparsity of $\boldsymbol{A}$.

5.5.2 Kleene Algebras

In the previous section we have suggested how to replace the Naive Algorithm when the Datalog$^\circ$ query is linear, by using the closure operator a^*. In this section we discuss how to extend this to non-linear Datalog$^\circ$. The closed pre-semiring in Def. 5.5.1 is too general for this purpose, and, instead, we will consider a more restrictive algebraic structure: a *Kleene algebra*.

"The closed pre-semiring is too general" is confusing. To fix. –DAN.

In a nutshell, Kleene algebras are special kinds of a closed semirings. They have a long history. (Kozen, 1990) gives a good overview, and points out that there have been several non-equivalent definitions in the literature. The original motivation of Kleene algebras was to provide an axiomatization of the closed semiring of regular sets, $(\mathbf{Reg}_\Sigma, +, \cdot, \emptyset, \{\epsilon\}, ^*)$, but Redko[2] proved in 1965 that no complete, finite axiomatization exists. Therefore, the goal became to provide a complete *deductive system*, meaning a set of possibly conditional identities (see for example Eq. (5.19) below). In 1966 (Salomaa, 1966) presented two such systems, but they used some conditional identities that hold only for $\mathbf{Reg}_\Sigma$ and do not hold in other useful examples. Instead, (Kozen, 1994) was the first to present a complete deductive system for $\mathbf{Reg}_\Sigma$, and proposed this definition as *the* Kleene Algebra, a definition mostly adopted today: this is Definition 5.5.2 below. As we shall see, Kleene Algebras are special cases of closed semirings.

One restriction of Kleene algebras is that the additive operator must be idempotent, and we start our discussion with a brief overview of idempotent semirings. A semiring $\boldsymbol{S} = (S, +, \cdot, \mathbf{0}, \mathbf{1})$ is called a *diod* if $\mathbf{1} + \mathbf{1} = \mathbf{1}$ (Gondran and Minoux, 2008; Gunawardena, 1998). It immediately follows that $+$ is idempotent, i.e. $a + a = a$. If $\boldsymbol{S}$ is a diod, then the relation $a \preceq b$ if $a + b = b$ is a partial order, and coincides with the natural order on $\boldsymbol{S}$, which we introduced in Def. 5.2.4. In other words, a diod is naturally ordered, and the $+$ operator is the same as the least upper bound, hence we may well write $a \vee b$ for $a + b$. Equivalently, a diod is an ordered monoid $(S, \cdot, \mathbf{1}, \preceq)$ where $(S, \preceq)$ is a join-semilattice with a smallest element $\mathbf{0}$, and where $\cdot$ distributes over $\vee$.

Definition 5.5.2 (Kleene Algebra)**.** A *Kleene Algebra* is a structure $\boldsymbol{K} = (K, +, \cdot, ^*, \mathbf{0}, \mathbf{1})$, where $(K, +, \cdot, \mathbf{0}, \mathbf{1})$ is a dioid and the following identities and conditional identities hold:

$$\mathbf{1} + a \cdot a^* \preceq a^* \tag{5.17}$$

$$\mathbf{1} + a^* \cdot a \preceq a^* \tag{5.18}$$

$$\text{if } b + a \cdot x \preceq x \quad \text{then } a^* \cdot b \preceq x \tag{5.19}$$

$$\text{if } b + x \cdot a \preceq x \quad \text{then } b \cdot a^* \preceq x \tag{5.20}$$

[2]The paper is in Ukrainian; see reference [25] in (Kozen, 1994).

It can be shown that in a Kleene algebra the inequalities (5.17) and (5.18) are actually equalities, hence $\boldsymbol{K}$ is a closed semiring. (Kozen, 1994) proves that the axioms of the Kleene algebra form a complete axiomatization of $\mathbf{Reg}_\Sigma$; equivalently, $\mathbf{Reg}_\Sigma$ is the free Kleene algebra generated by the set Σ. Although the number of axioms in Def. 5.5.2 is finite, this does not contradict Redko's result, because not all axioms are identities: (5.19) and (5.20) are conditional identities, also known as Horn clauses, or deductive rules. Surprisingly, while Kleene algebra axioms are complete for the *identities* of $\mathbf{Reg}_\Sigma$, they are not complete for the *conditional identities* of $\mathbf{Reg}_\Sigma$: (Kozen, 1990) shows that the conditional identity "If $a^2 = \mathbf{1}$ then $a = \mathbf{1}$" holds in $\mathbf{Reg}_\Sigma$ (this is easy to check), but fails in another Kleene algebra.

Our goal is to use the operators $+, \cdot, ^*$ to compute explicit expressions for the fixpoint of a $\mathsf{Datalog}^\circ$ program over some POPS $\boldsymbol{P}$. A Kleene algebra $\boldsymbol{P}$ is still too general for our purpose, however (Hopkins and Kozen, 1999) proves that this becomes possible if $\boldsymbol{P}$ is a *commutative* Kleene Algebra; as usual, "commutative" refers to the multiplicative operator. This restriction is quite reasonable for our purpose, because all POPS we consider for $\mathsf{Datalog}^\circ$ are already commutative. To motivate the surprising result for commutative Kleene algebras below, it will be useful to discuss its context: Parikh's theorem.

In general, a context-free language L over the alphabet Σ is not a regular language. However, (Parikh, 1966) proved that, if one makes concatenation commutative, $a \cdot b = b \cdot a$, then every context-free language is equivalent to a regular language. For example, the grammar $S \to \varepsilon$, $S \to aSb$, generates the context-free language $L = \{a^n b^n \mid n \geq 0\}$; if we make $\cdot$ commutative, then L is equivalent to $(a \cdot b)^*$. More formally, let $k = |\Sigma|$ and define the *Parikh's image* of a word $w \in \Sigma^*$ to be $(v_1, \ldots, v_k) \in \mathbb{N}^k$, where v_i is the number of occurrences of the symbol a_i in w (Esparza *et al.*, 2011). (The Parikh image is called the *commutative image* by (Hopkins and Kozen, 1999).) The Parikh image captures the intuition behind making $\cdot$ commutative: we only retain the number of occurrences of each letter, not their order.

Theorem 5.5.3 (Parikh's Theorem)**.** For any context-free language L, there exists a regular language L' such that L and L' have the same Parikh image.

Equivalently, Parik's theorem states the following. Consider the semiring $(\mathbf{Reg}(\mathbb{N}^\Sigma), +, \cdot, \emptyset, \{(0, \ldots, 0)\})$, where $\mathbb{N}^\Sigma$ is the set of all Parikh sequences $(v_1, \ldots, v_k)$, $\mathbf{Reg}(\mathbb{N}^\Sigma)$ are Parikh images of regular sets $\mathbf{Reg}_\Sigma$, $+$ is set union, and $\cdot$ is "string concatenation": $a \cdot b = \{(u_1 + v_1, \ldots, u_n + v_n) \mid (u_1, \ldots, u_n) \in a, (v_1, \ldots, v_n) \in b\}$. Then, the Parikh image of a context-free language can be described as the least fixpoint of a set of polynomial equations in this semiring, and Parikh's theorem says that its least fixpoint can be expressed entirely in terms of $+, \cdot, ^*$.

In a beautiful paper (Hopkins and Kozen, 1999) extended Parikh's theorem from the specific semiring described above, to arbitrary commutative Kleene

algebras. Fix a commutative Kleene algebra $\boldsymbol{K}$, and let $\boldsymbol{z} = \{z_1, \ldots, z_M\}$ be a set of variables. An *extended polynomial* g is a formal expression that is either (a) $g = z_i$ for some variable z_i, (b) $g = g_1 + g_2$ or $g = g_1 \cdot g_2$ for two extended polynomials g_1, g_2, or (c) $g = (g_1)^*$ for some extended polynomial g_1.

Theorem 5.5.4. Let $\boldsymbol{K}$ be a commutative Kleene algebra, and let[3] $\boldsymbol{f}(\boldsymbol{x})$ be a tuple of n polynomials in n variables. Then $\boldsymbol{f}$ has a unique pre-fixpoint $\boldsymbol{x}$, i.e. $f(\boldsymbol{x}) \preceq \boldsymbol{x}$. Moreover, $\boldsymbol{x}$ can be expressed using extended polynomials in the coefficients of the polynomials in $\boldsymbol{f}$.

If we apply $\boldsymbol{f}$ to both sides of $\boldsymbol{f}(\boldsymbol{x}) \preceq \boldsymbol{x}$ we obtain $\boldsymbol{f}(\boldsymbol{f}(\boldsymbol{x})) \preceq \boldsymbol{f}(\boldsymbol{x})$, which implies that $f(\boldsymbol{x})$ is also a pre-fixpoint. By uniqueness, $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{x}$, meaning that $\boldsymbol{x}$ is also the unique fixpoint of $\boldsymbol{f}$; see also Prop. 5.2.3. In other words, the theorem also says that the equation $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x})$ has a unique fixpoint (which, of course, is also the least fixpoint), which can be expressed in terms of the coefficients of the polynomials $\boldsymbol{f}$ and the operators $+, \cdot, ^*$.

We illustrate the theorem with a simple example adapted from (Hopkins and Kozen, 1999).

Example 5.5.5. Consider the alphabet $\{a, b\}$ and the grammar $S \to aSb|SS|\varepsilon$, which generates the context-free language of balanced strings of parentheses, where a, b stand for (and) respectively. This language can be described as the fixpoint of:

$$x = axb + xx + \mathbf{1}$$

If we apply commutativity and reorder the terms, the expression above becomes:

$$x = \mathbf{1} + abx + x^2$$

The theorem states that the unique solution is $x = (ab)^*$, which is an extended polynomial in the coefficients a, b.

Could Theorem 5.5.4 be applied to improve the evaluation of Datalog°? As in the previous section, the suggestion would be to replace the Naive Algorithm with an explicit computation of the extended polynomials that define the fixpoint. However, while efficient algorithms for computing the closure of a matrix $\boldsymbol{A}^*$ are known, no efficient algorithm is known yet for computing the extended polynomials in Theorem 5.5.4. Still, in a surprising twist, (Esparza *et al.*, 2010) show that the key idea in Theorem 5.5.4 can be used to devise an alternative algorithm to the Naive Algorithm, which is based on Newton's method for solving equations. We will discuss it in Sec. 5.6.3.

[3] The result in (Hopkins and Kozen, 1999) allows $\boldsymbol{f}$ to be extended polynomials.

5.6 Optimizations

After the previous chapters are stable, we should update the notations in this section to match those in the rest of the paper. Also, some of the text is from prior papers, we may want to make small updates. – DAN.

As we saw in Chapter 2, basic Datalog admits a few very powerful optimization techniques such as the semi-naïve evaluation and magic-set rewriting. We describe here how these optimizations extend and generalize to Datalog°.

5.6.1 Semi-Naive Evaluation Algorithm

In this section we show that the semi-naïve algorithm can be generalized to Datalog°, for certain restricted POPS. The Naïve Algorithm 6 repeatedly applies the immediate consequence operator F and computes $J^{(0)}, J^{(1)}, J^{(2)}, \ldots$, where $J^{(t+1)} \stackrel{\text{def}}{=} F(J^{(t)})$. This strategy is inefficient because all facts discovered at iteration t will be re-discovered at iterations $t+1, t+2, \ldots$ The *semi-naïve* optimization consists of a modified program that computes $J^{(t+1)}$ by first computing only the "novel" facts $\delta^{(t)} = F(J^{(t)}) - J^{(t)}$, which are then added to $J^{(t)}$ to form $J^{(t+1)}$. Furthermore, the difference $F(J^{(t)}) - J^{(t)}$ can be computed efficiently, *without* fully evaluating $F(J^{(t)})$, by using techniques from incremental view maintenance.

This section generalizes the semi-naïve algorithm to Datalog°. The main problem we encounter is that, while the difference operator is well defined in the Boolean semiring, as $x - y \stackrel{\text{def}}{=} x \wedge \neg y$, no generic difference operator exists in an arbitrary POPS. In order to define a difference operator, we will restrict the POPS to be a complete, distributive dioid.

Recall that a *dioid* is a semiring $\boldsymbol{S} = (S, \oplus, \otimes, 0, 1)$ where $\oplus$ is idempotent. We have seen that in a diod the relation $a \preceq b$ if $a \oplus b = b$ is a partial order, which coincides with the natural order on $\boldsymbol{S}$.

A diod $\boldsymbol{S} = (S, \oplus, \otimes, 0, 1, \sqsubseteq)$ is called a *complete, distributive dioid* if the ordered set $(S, \sqsubseteq)$ is a *complete, distributive lattice*, which means that every set $A \subseteq S$ has a greatest lower bound $\bigwedge A$, and $x \vee \bigwedge A = \bigwedge\{x \vee a \mid a \in A\}$. In a complete, distributive dioid, the *difference* operator is defined by

$$b \ominus a \stackrel{\text{def}}{=} \bigwedge\{c \mid a \oplus c \sqsupseteq b\} \tag{5.21}$$

In order to extend the semi-naïve algorithm to Datalog°, we require the POPS to be a complete, distributive dioid. There are many examples of complete, distributive dioids: $(2^{\boldsymbol{U}}, \cup, \cap, \emptyset, \boldsymbol{U}, \subseteq)$ is a complete, distributive dioid, whose difference operator is exactly set-difference $b - a = \bigcap\{c \mid b \subseteq a \cup c\} =$

$J^{(0)} \leftarrow \mathbf{0}$;
for $t \leftarrow 0$ **to** ∞ **do**
  $\delta^{(t)} \leftarrow F(J^{(t)}) \ominus J^{(t)}$; `// incremental computation, see text`
  $J^{(t+1)} \leftarrow J^{(t)} \oplus \delta^{(t)}$;
  if $\delta^{(t)} = \mathbf{0}$ **then**
    Break
  end
end
return $J^{(t)}$

Algorithm 7: Semi-naïve evaluation for Datalog°

$b \setminus a$; $\mathsf{Trop}^+ = (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0, \geq)$ is also a complete, distributive dioid, whose difference operator is defined by:

$$v \ominus u = \begin{cases} v & \text{if } v < u \\ \infty & \text{if } v \geq u \end{cases} \tag{5.22}$$

Also, $\mathbb{N} \cup \{\infty\}$ is a complete, distributive dioid. On the other hand, Trop_p^+, $\mathsf{Trop}_{\leq \eta}^+$, $\mathbb{R}_\perp$ are not dioids.

We describe now the semi-naïve algorithm for a program over a complete, distributive dioid. We write $F(J)$ for the immediate consequence operator of the program, where J is an instance of the IDB predicates. The semi-naïve algorithm is shown in Algorithm 7. It proceeds almost identically to the Naïve Algorithm 6, but splits the computation $J \leftarrow F(J)$ into two steps: first compute the difference $F(J) \ominus J$, then add it to J. The following was proven by (Khamis *et al.*, 2022a) (see more details in (Khamis *et al.*, 2021)):

Theorem 5.6.1. Consider a Datalog° program over a complete, distributive dioid. Then the Semi-naïve Algorithm 7 returns the same answer as the Naïve Algorithm 6.

As described, the semi-naïve algorithm is no more efficient than the naïve algorithm. Its advantage comes from the fact that we can compute the difference

$$\delta^{(t)} \leftarrow F(J^{(t)}) \ominus J^{(t)} \tag{5.23}$$

incrementally, without computing the ICO F. Recall that the Datalog° program consists of n rules T_i :- $F_i(T_1, \ldots, T_n)$, one for each IDB $T_1, \ldots, T_n$, where F_i is a sum-sum-product expression (5.2), and the ICO is the vector of the

sum-sum-product expressions, $F = (F_1, \ldots, F_n)$. The difference (5.23) consists of computing the following differences, for $i = 1, n$:

$$\delta_i^{(t)} \leftarrow F_i(T_1^{(t)}, \ldots, T_n^{(t)}) \ominus T_i^{(t)} \tag{5.24}$$

For the purpose of incremental evaluation, we will assume w.l.o.g. that each T_j occurs at most once in any sum-product term of F_i; otherwise, we give each occurrence of T_j in (5.24) a unique name; we illustrate with an example below. Therefore, F_i is linear [4] in each argument T_j. Notice that, when $t \geq 1$, then $T_j^{(t)} = T_j^{(t-1)} \oplus \delta_j^{(t-1)}$ for $j = 1, n$, and $T_i^{(t)} = F_i(T_1^{(t-1)}, \ldots, T_n^{(t-1)})$.

Assuming that the sum-sum-product expression $F_i(T_1, \ldots, T_n)$ is linear in each argument T_j, then, when $t \geq 1$, the difference (5.24) can be computed as follows:

$$\delta_i^{(t)} \leftarrow \left(\bigoplus_{j=1,n} F_i(T_1^{(t)}, \ldots, T_{j-1}^{(t)}, \delta_j^{(t-1)}, T_{j+1}^{(t-1)}, \ldots, T_n^{(t-1)}) \right) \ominus T_i^{(t)} \tag{5.25}$$

Furthermore, if the sum-sum-product F_i can be written as $F_i(T_1, \ldots, T_n) = E_i \oplus G_i(T_1, \ldots, T_n)$, where E_i is independent of $T_1, \ldots, T_n$ (i.e. it depends only on the EDBs), then Eq. (5.25) can be further simplified to:

$$\delta_i^{(t)} \leftarrow \left(\bigoplus_{j=1,n} G_i(T_1^{(t)}, \ldots, T_{j-1}^{(t)}, \delta_j^{(t-1)}, T_{j+1}^{(t-1)}, \ldots, T_n^{(t-1)}) \right) \ominus T_i^{(t)} \tag{5.26}$$

In particular, we can replace the expensive computation $F_i(T_1^{(t)}, \ldots, T_n^{(t)})$ in (5.24) with several computations $F_i(\cdots)$ (or $G_i(\cdots)$), where one argument is δ_j instead of T_j. This is usually more efficient, because δ_j is much smaller than T_j.

We end this section with a short example.

Example 5.6.2 (Quadratic transitive closure)**.** Consider the following non-linear datalog program computing the transitive closure:

$$T(x, y) \text{ :- } E(x, y) \vee (\exists z : T(x, z) \wedge T(z, y))$$

The semi-naïve algorithm with the differential rule (5.26) proceeds as follows. It initializes $T^{(0)} = \emptyset$ and $\delta^{(0)} = E$, then repeats the following instructions for $t = 1, 2, \ldots$

$$\begin{aligned} \delta^{(t)}(x, y) \leftarrow & \left(\exists z : \delta^{(t-1)}(x, z) \wedge T^{(t-1)}(z, y)\right) \vee \\ & \left(\exists z : T^{(t)}(x, z) \wedge \delta^{(t-1)}(z, y)\right) \setminus T^{(t)}(x, y) \\ T^{(t+1)}(x, y) \leftarrow & T^{(t)}(x, y) \vee \delta^{(t)}(x, y) \end{aligned}$$

[4] In the datalog context, "linear" is often used instead of "affine". We follow the convention and use the term "linear".

The semi-naïve algorithm immediately extends to a stratified variant of Datalog°, by applying the algorithm to each stratum.

The differential rule (5.25) or (5.26) is not the only possibility for incremental evaluation. It has the advantage that it requires only n computations of the sum-sum-product expression F_i, where n is the maximum number of IDBs that occur in any sum-product of F_i. But these expressions use both the previous IDBs $T_j^{(t-1)}$ and current IDBs $T_j^{(t)}$, which, one can argue, is not ideal, because the newer IDB instances are slightly larger than the previous ones. An alternative is to use the $2^n - 1$ discrete differentials of F_i, which use only the previous IDBs $T_j^{(t-1)}$, at the cost of increasing the number of expressions. Yet a more sophisticated possibility is to use higher order differentials, as pioneered by the DBToaster project (Koch *et al.*, 2014).

5.6.2 The FGH-Rule

(Wang *et al.*, 2022a) describe a simple optimization rule for Datalog°, called the FGH-rule, which subsumes several optimizations for datalog and extensions of datalog, such as magic set optimization and the PreM optimization (Zaniolo *et al.*, 2017).

Consider an iterative program that repeatedly applies a function F until some termination condition is satisfied, then applies a function G that returns the final answer Y:

$$\begin{aligned} &X \leftarrow X_0 \\ &\texttt{loop } X \leftarrow F(X) \texttt{ end loop} \\ &Y \leftarrow G(X) \end{aligned} \tag{5.27}$$

We call this an FG-program. The FGH-rule provides a sufficient condition to compute the final answer Y by another program, called the GH-program:

$$\begin{aligned} &Y \leftarrow G(X_0) \\ &\texttt{loop } Y \leftarrow H(Y) \texttt{ end loop} \end{aligned} \tag{5.28}$$

The FGH-Rule states that, if the following identity holds:

$$G(F(X)) = H(G(X)) \tag{5.29}$$

then the FG-program (5.27) and the GH-program (5.28) are equivalent. We supply here a "proof by picture" of the claim:

$$\begin{array}{ccccccc} X_0 & \xrightarrow{F} & X_1 & \xrightarrow{F} & X_2 & \cdots \xrightarrow{F} & X_n \\ \downarrow{\scriptstyle G} & & \downarrow{\scriptstyle G} & & \downarrow{\scriptstyle G} & & \downarrow{\scriptstyle G} \\ Y_0 & \xrightarrow{H} & Y_1 & \xrightarrow{H} & Y_2 & \cdots \xrightarrow{H} & Y_n \end{array}$$

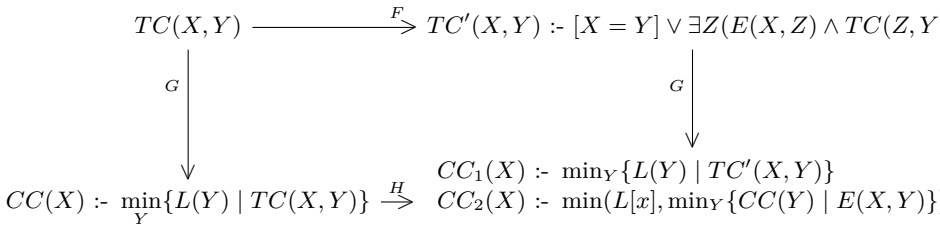


Figure 5.2: Visualization of the FGH-rule used in Example 5.6.3.

The FGH-rule immediately applies to Datalog° programs, as follows. Consider two Datalog° programs Q_1 and Q_2 given below:

$$Q_1: \quad \begin{array}{l} X \text{ :- } F(X) \\ Y \text{ :- } G(X) \end{array} \qquad Q_2: \quad Y \text{ :- } H(Y)$$

Here X and Y are tuples of IDBs, and F, G, H represent sum-sum-product expressions over these IDBs. In both cases, only the IDBs Y are returned. Then, if the FGH-rule (5.29) holds, and, moreover, $G(\bot) = \bot$, then Q_1 is equivalent to Q_2. We notice that, under these conditions, if Q_1 terminates, then Q_2 terminates as well.

The FGH-rule is an extension of the *pre-mappability*, or PreM-rule by (Zaniolo *et al.*, 2017), which requires the $G(F(X)) = G(F(G(X)))$ holds: in this case one can define H as $H(X) = G(F(X))$, and the FGH-rule holds automatically. The PreM rule is more restricted than the FGH-rule.

We illustrate the FGH rule with several examples, then briefly discuss how it was applied to Datalog° by (Wang *et al.*, 2022a).

Example 5.6.3 (Connected Components)**.** We are given an undirected graph, with edge relation $E(x, y)$, where each node x has a unique numerical label $L(X)$. The task is to compute for each node X, the minimum label $CC(X)$ in its connected component. This program is a well-known target of query optimization in the literature (Zaniolo *et al.*, 2017). A naïve approach is to first compute the reflexive and transitive closure of E, then apply a min-aggregate:

$$\begin{array}{l} TC(X,Y) \text{ :- } [X=Y] \vee \exists Z(E(X,Z) \wedge TC(Z,Y)) \\ CC(X) \text{ :- } \min_Y\{L(Y) \mid TC(X,Y)\} \end{array}$$

An optimized program interleaves aggregation and recursion:

$$CC(X) \text{ :- } \min(L(X), \min_Y\{CC(Y) \mid E(X,Y)\})$$

We prove their equivalence, by checking the FGH-rule which is shown in

$$
\begin{aligned}
CC_1[x] &\stackrel{\text{def}}{=} \min_y\{L[y] \mid TC'(x,y)\} \\
&= \min_y\{L[y] \mid [x=y] \vee \exists z(E(x,z) \wedge TC(z,y))\} \\
&= \min(L[x], \min_y\{L[y] \mid \exists z(E(x,z) \wedge TC(z,y))\}) \\
&= \min(L[x], \min_{y,z}\{L[y] \mid E(x,z) \wedge TC(z,y)\}) \\
CC_2[x] &\stackrel{\text{def}}{=} \min(L[x], \min_y\{CC[y] \mid E(x,y)\}) \\
&= \min(L[x], \min_y\{\min_{y'}\{L[y'] \mid TC(y,y')\} \mid E(x,y)\}) \\
&= \min(L[x], \min_{y',y}\{L[y'] \mid E(x,y) \wedge TC(y,y')\})
\end{aligned}
$$

Figure 5.3: Computing CC_1 and CC_2 from Ex. 5.6.3.

Fig. 5.2. More precisely, we need to check that $CC_1 \stackrel{\text{def}}{=} G(F(TC)) = G(TC')$ is equivalent to $CC_2 \stackrel{\text{def}}{=} H(G(TC)) = H(CC)$, which is shown in Fig. 5.3.

Example 5.6.4 (Simple Magic)**.** The simplest application of magic-set optimization (Bancilhon *et al.*, 1986; Beeri and Ramakrishnan, 1991) converts *transitive closure* to *reachability*, by rewriting this program:

$$
Q_1: \quad \begin{aligned} TC(x,y) &\text{ :- } [x=y] \vee \exists z(TC(x,z) \wedge E(z,y)) \\ Q(y) &\text{ :- } TC(a,y) \end{aligned} \tag{5.30}
$$

where a is some constant, into this program:

$$
Q_2: \quad Q(y) \text{ :- } [y=a] \vee \exists z(Q(z) \wedge E(z,y)) \tag{5.31}
$$

This is a powerful optimization, because it reduces the run time from $O(n^2)$ to $O(n)$. Several Datalog systems support some form of magic-set optimizations. We check that (5.30) is equivalent to (5.31) by verifying the FGH-rule. The functions F, G, H are shown in Fig. 5.4. One can verify that $G(F(TC)) = H(G(TC))$, for any relation TC. Indeed, after converting both expressions to normal form (i.e. in sum-sum-product form), we obtain $G(F(TC)) = H(G(TC)) = P$, where:

$$
P(y) \stackrel{\text{def}}{=} [a=y] \vee \exists z(TC(a,z) \wedge E(z,y))
$$

Replacing $TC(a,\_)$ by $Q(\_)$ now yields precisely program Q_2 in (5.31). It is shown by (Wang *et al.*, 2022b) that, given a sideways information passing strategy (called SIPS, see (Beeri and Ramakrishnan, 1991; Bancilhon *et al.*, 1986)) every magic-set optimization over a Datalog program can be proven correct by a sequence of FGH-rule applications.

$$\begin{array}{ccc} TC(x,y) & \xrightarrow{F} & [x=y] \vee \exists z(TC(x,z) \wedge E(z,y)) \\ \Big\downarrow{\scriptstyle G} & & \Big\downarrow{\scriptstyle G} \\ TC(\,\mathrm{a}\,,y) & \xrightarrow{H} & [\,\mathrm{a}\,=y] \vee \exists z(TC(\,\mathrm{a}\,,z) \wedge E(z,y)) \end{array}$$

Figure 5.4: Expressions F, G, H in Ex. 5.6.4.

Loop Invariants Some applications of the FGH-rule require us to first infer a loop invariant. The general principle is the following. Let $\phi(X)$ be any predicate satisfying the following three conditions:

$$\begin{aligned} &\phi(X_0) \\ &\phi(X) \Rightarrow \phi(F(X)) \\ &\phi(X) \Rightarrow (G(F(X)) = H(G(X))) \end{aligned} \tag{5.32}$$

then the FG-program (5.27) and the GH-program (5.28) are equivalent. This *conditional* FGH-rule is very powerful; we briefly illustrate it with an example.

Example 5.6.5 (Beyond Magic)**.** Consider the following program:

$$Q_1: \quad \begin{array}{l} TC(x,y) \text{ :- } [x=y] \vee \exists z(E(x,z) \wedge TC(z,y)) \\ Q(y) \text{ :- } TC(a,y) \end{array} \tag{5.33}$$

which we want to optimize to:

$$Q_2: \quad Q(y) \text{ :- } [y=a] \vee \exists z(Q(z) \wedge E(z,y)) \tag{5.34}$$

Unlike the simple magic program in Ex. 5.6.4, here rule (5.33) is right-recursive. As shown in (Beeri and Ramakrishnan, 1991), the magic-set optimization using the standard sideways information passing optimization (Abiteboul *et al.*, 1995) yields a program that is more complicated than our program (5.34). Indeed, consider a graph that is simply a directed path $a_0 \to a_1 \to \cdots \to a_n$ with $a = a_0$. Then, even with magic-set optimization, the right-recursive rule (5.33) needs to derive *quadratically many* facts of the form $T(a_i, a_j)$ for $i \leq j$, whereas the optimized program (5.34) can be evaluated in linear time. Note also that the FGH-rule cannot be applied directly to prove that the program (5.33) is equivalent to (5.34). To see this, denote by $P_1 \stackrel{\text{def}}{=} G(F(TC))$ and $P_2 \stackrel{\text{def}}{=} H(G(TC))$, and observe that P_1, P_2 are defined as:

$$\begin{aligned} P_1(y) &\stackrel{\text{def}}{=} [y=a] \vee \exists z(E(a,z) \wedge TC(z,y)) \\ P_2(y) &\stackrel{\text{def}}{=} [y=a] \vee \exists z(TC(a,z) \wedge E(z,y)) \end{aligned}$$

In general, $P_1 \neq P_2$. The problem is that the FGH-rule requires that $G(F(TC)) = H(G(TC))$ for *every* input TC, not just the transitive closure of E. However, the FGH-rule *does* hold if we restrict TC to relations that satisfy the following loop-invariant $\phi(TC)$:

$$\exists z_1(E(x, z_1) \wedge TC(z_1, y)) \Leftrightarrow \exists z_2(TC(x, z_2) \wedge E(z_2, y)) \tag{5.35}$$

If TC satisfies this predicate, then it follows immediately that $P_1 = P_2$, allowing us to optimize program (5.33) to (5.34). It remains to prove that ϕ is indeed an invariant for the function F. The base case (5.32) holds because both sides of (5.35) are empty when $TC = \emptyset$. It remains to check $\phi(TC) \Rightarrow \phi(F(TC))$. Denote $TC' \stackrel{\text{def}}{=} F(TC)$, then we need to check that, if (5.35) holds, then the predicate $\Psi_1(x, y) \stackrel{\text{def}}{=} \exists z_1(E(x, z_1) \wedge TC'(z_1, y))$ is equivalent to the predicate $\Psi_2(x, y) \stackrel{\text{def}}{=} \exists z_2(TC'(x, z_2) \wedge E(z_2, y))$. Using (5.35) we can prove the equivalence of the predicates Ψ_1 and Ψ_2.

Semantic optimization Finally, we illustrate how the FGH-rule takes advantage of database constraints (Ramakrishnan and Srivastava, 1994). In general, a priori knowledge of database constraints can lead to more powerful optimizations. For instance, in (Bancilhon *et al.*, 1986), the counting and reverse counting methods are presented to further optimize the same-generation program if it is known that the underlying graph is acyclic. We present a principled way of exploiting such a priori knowledge. As we show here, recursive queries have the potential to use *global* constraints on the data during semantic optimization; for example, the query optimizer may exploit the fact that the graph is a tree, or the graph is connected. We will denote by Γ the set of constraints on the EDBs. Then, the FGH-rule (5.29) needs to be be checked only for EDBs that satisfy Γ, as we illustrate in this example:

Example 5.6.6. Consider again the bill-of-material problem in Ex. 5.4.9, where $E(X, Y)$ indicates that Y is a subpart of X, and $C(X) \in \mathbb{N}$ represents the cost of the part X. Since we interpret the program over the semiring $\mathbb{N}$, to avoid non-termination on a cyclic graph, we first compute the transitive closure, and then use it to sum up all costs:

$$\begin{aligned} Q_1: \qquad R(X, Y) &\text{ :- } [X = Y] \vee \exists Z\,(R(X, Z) \wedge E(Z, Y)) \\ Q(X) &\text{ :- } \sum_Y \{C(Y) \mid T(X, Y)\} \end{aligned} \tag{5.36}$$

Consider now the case when $E(X, Y)$ forms a tree. Then, we can compute the total cost much more efficiently by using the program in Eq. (5.14), repeated here for convenience:

$$Q_2: \qquad R(X) = C(X) + \sum_Y E(X,Y) \cdot R(Y) \tag{5.37}$$

Optimizing the program (5.36) to (5.37) is an instance of *semantic optimization*, since this only holds if the database instance is a tree. We do this in three steps. First, we define the constraint Γ stating that the data is a tree. Second, using Γ we infer a loop-invariant Φ of the program Q_1. Finally, using Γ and Φ we prove the FGH-rule, concluding that Q_1 and Q_2 are equivalent. The constraint Γ is the conjunction of the following statements:

$$\begin{aligned}
&\forall X_1, X_2, Y. E(x_1, y) \wedge E(x_2, y) \Rightarrow x_1 = x_2 \\
&\forall X, Y. E(X,Y) \Rightarrow T(X,Y) \\
&\forall X, Y, Z. T(X,Z) \wedge E(Z,Y) \Rightarrow T(X,Y) && (5.38) \\
&\forall X, Y. T(X,Y) \Rightarrow X \neq Y && (5.39)
\end{aligned}$$

The first asserts that Y is a key in $E(X,Y)$. The last three are an Existential Second Order Logic statement: they assert that there exists some relation $T(X,Y)$ that contains E, is transitively closed, and irreflexive. Next, we infer the following loop-invariant of the program Q_1:

$$\Phi: R(X,Y) \Rightarrow [X = Y] \vee T(X,Y) \tag{5.40}$$

Finally, we check the FGH-rule, under the assumptions Γ, Φ. Denote by $P_1 \stackrel{\text{def}}{=} G(F(R))$ and $P_2 \stackrel{\text{def}}{=} H(G(R))$. To prove $P_1 = P_2$ we simplify P_1 using the assumptions Γ, Φ, as shown in Fig. 5.5. We explain each step. Line 2-3 are inclusion/exclusion. Line 4 uses the fact that the term on line 3 is $= 0$, because the loop invariant implies:

$$\begin{aligned}
& R(X,Z) \wedge E(Z,Y) && \\
\Rightarrow & ([X = Z] \vee T(X,Z)) \wedge E(Z,Y) && \text{by (5.40)} \\
\equiv & E(X,Y) \vee (T(X,Z) \wedge E(Z,Y)) && \\
\Rightarrow & T(X,Y) \vee T(X,Y) \equiv T(X,Y) && \text{by (5.38)} \\
\Rightarrow & X \neq Y && \text{by (5.39)}
\end{aligned}$$

The last line follows from the fact that Y is a key in $E(Z,Y)$. A direct calculation of $P_2 = H(G(R))$ results in the same expression as line 5 of Fig. 5.5, proving that $P_1 = P_2$.

We end this section with a brief discussion of how to infer the rules H from a Datalog$^\circ$ program where we are given only F and G. Recall that our

$$
\begin{aligned}
P_1(X) = &\sum_Y \{C(Y) \mid [X = Y] \vee \exists Z\,(R(X,Z) \wedge E(Z,Y))\} \\
= &C(X) + \sum_Y \{C(Y) \mid \exists Z\,(R(X,Z) \wedge E(Z,Y))\} \\
&- \sum_Y \{C(Y) \mid [X = Y] \wedge \exists Z\,(R(X,Z) \wedge E(Z,Y))\} \\
= &C(X) + \sum_Y \{C(Y) \mid \exists Z\,(R(X,Z) \wedge E(Z,Y))\} \\
= &C(X) + \sum_Y \sum_Z \{C(Y) \mid (R(X,Z) \wedge E(Z,Y))\}
\end{aligned}
$$

Figure 5.5: Transformation of $P_1 \stackrel{\text{def}}{=} G(F(R))$ in Ex. 5.6.6.

task is to ensure that $G(F(X)) = H(G(X))$. Denote $G(F(X))$ and $H(G(X))$ by P_1, P_2 respectively. There are two ways to find H: using rewriting, or using program synthesis with an SMT solver.

Rule-based Synthesis In *query rewriting using views* we are given a query Q and a view V, and want to find another query Q' that answers Q by using only the view V instead of the base tables X; in other words, $Q(X) = Q'(V(X))$ (Halevy, 2001; Goldstein and Larson, 2001). The problem is usually solved by applying rewrite rules to Q, until it only uses the available views. The problem of finding H is an instance of query rewriting using views, and one possibility is to approach it using rewrite rules; for this purpose we used the rule engine egg (Willsey *et al.*, 2021), a state-of-the-art *equality saturation* system; see details in (Wang *et al.*, 2022a).

Counterexample-based Synthesis Rule-based synthesis explores only correct rewritings P_2, but its space is limited by the hand-written axioms. The alternative approach, pioneered in the programming language community (Solar-Lezama *et al.*, 2006), is to generate candidate programs P_2 from a much larger space, then using an SMT solver to verify correctness. This technique, called Counterexample-Guided Inductive Synthesis, or CEGIS, can find rewritings P_2 even in the presence of interpreted functions, because it exploits the semantics of the underlying domain.

5.6.3 Newton's Evaluation Method

(Esparza *et al.*, 2010) introduced a new approach for computing fixpoint solutions to polynomial equations over semirings, $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x})$, inspired by Newton's method for solving an equation over the reals.

Fix a naturally ordered, commutative semiring[5] $\boldsymbol{S}$, where $\preceq$ is the natural order. We will use $+, \cdot$ to denote its operations, instead of $\oplus, \otimes$, but will keep in mind that these are abstract operations in the semiring. A *polynomial* (in one variable) is a formal expression

$$f(x) = a_0 + a_1 x + \cdots + a_n x^n$$

where all coefficients a_i are in the semiring $\boldsymbol{S}$. One can easily check that the set of polynomials, $\boldsymbol{S}[x]$, is also naturally ordered: $f \preceq g$ if there exists a polynomial h such that $f + h = g$.

The *derivative* of f is defined formally as

$$\frac{df}{dx} \stackrel{\text{def}}{=} \sum_{k=1,n} k a_k x^{k-1}$$

where "k" means $1 + 1 + \cdots + 1$ k times. We notice that the derivative is a pure syntactic construct, unlike the standard derivative in calculus which has a semantic definition. However, the following semantic property holds for the derivative:

$$f(x) + \frac{df}{dx}\Delta \preceq f(x + \Delta), \quad \forall x, \Delta \in S \tag{5.41}$$

In fact, it is not hard to construct explicitly a polynomial $g(x, \Delta)$ such that $f(x) + \frac{df}{dx}\Delta + g(x, \Delta) = f(x + \Delta)$. For that, it suffices to define g when $f(x)$ is a single monomial, $a_k x^k$. In that case $f(x + \Delta) = a_k \sum_{i=0,k} \binom{k}{i} x^{k-i}\Delta^i = a_k x^k + k a_k x^{k-1}\Delta + a_k \sum_{i=2,k} \binom{k}{i} x^{k-i}\Delta^i$, and we set $g(x, \Delta) \stackrel{\text{def}}{=} a_k \sum_{i=2,k} \binom{k}{i} x^{k-i}\Delta^i$.

Consider a Datalog$^\circ$ program Q over a semring $\boldsymbol{S}$, and fix some EDB instance I. The grounding of Q is the fixpoint equation of a tuple of N polynomials over N variables:

$$\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x}) \tag{5.42}$$

Recall that *Kleene's method* computes the fixpoint as the least upper bound $\bigvee_t \boldsymbol{k}^{(t)}$ of the following sequence:

$$\boldsymbol{k}^{(0)} \stackrel{\text{def}}{=} 0 \qquad\qquad \boldsymbol{k}^{(t+1)} \stackrel{\text{def}}{=} \boldsymbol{f}(\boldsymbol{k}^{(t)}) \tag{5.43}$$

The Naïve Algorithm 6 is, essentially, Kleene's method.

(Esparza *et al.*, 2010) introduced an alternative method for computing the least fixpoint of (5.42), inspired by Newton's method for solving an equation $g(x) = 0$ over the reals. Netwon's method starts at some $x^{(0)} \in \mathbb{R}$, then defines

[5] (Esparza *et al.*, 2010) defines Newton's method for non-commutative semirings. For simplicity, we restrict our discussion to commutative semirings.

$x^{(t+1)} = x^{(t)} + \Delta^{(t)}$, where $\Delta^{(t)}$ is chosen such that $g(x^{(t)}) + g'(x^{(t)})\Delta^{(t)} = 0$, which leads to the familiar recurrence $x^{(t+1)} = x^{(t)} - \frac{g(x^{(t)})}{g'(x^{(t)})}$. If we apply Newton's method to compute a fixpoint $x = f(x)$ (still over the reals), then we can define $g(x) \stackrel{\text{def}}{=} f(x) - x$, and repeat the argument, namely:

$$x^{(t+1)} = x^{(t)} + \Delta^{(t)} \tag{5.44}$$

where $\Delta^{(t)}$ is the solution to

$$\Delta^{(t)} = f'(x^{(t)})\Delta^{(t)} + (f(x^{(t)}) - x^{(t)}) \tag{5.45}$$

(Esparza *et al.*, 2010) made the important observation that $\Delta^{(t)}$ itself is the result of a *linear* fixpoint computation (5.45), then generalized it to a method to compute the fixpoint (5.42) in a semiring, as follows. Let $D\boldsymbol{f}$ be the Jacobian of the vector of polynomials $\boldsymbol{f}$, i.e. the matrix: $D\boldsymbol{f} \stackrel{\text{def}}{=} (\partial f_i/\partial x_j)_{ij}$. A *Newton sequence* for $\boldsymbol{f}$ is any sequence $\boldsymbol{x}^{(t)}$ such that $\boldsymbol{x}^{(0)} = 0$ and, for all $t \geq 0$:

$$\boldsymbol{x}^{(t+1)} = \boldsymbol{x}^{(t)} + \boldsymbol{\Delta}^{(t)} \tag{5.46}$$

where $\boldsymbol{\Delta}^{(t)}$ is the least fixpoint of

$$\boldsymbol{\Delta}^{(t)} = (D\boldsymbol{f}(\boldsymbol{x}^{(t)})) \cdot \boldsymbol{\Delta}^{(t)} + \boldsymbol{\delta}^{(t)} \tag{5.47}$$

and $\boldsymbol{\delta}^{(t)}$ is any value s.t.

$$\boldsymbol{f}(\boldsymbol{x}^{(t)}) = \boldsymbol{x}^{(t)} + \boldsymbol{\delta}^{(t)} \tag{5.48}$$

The expression $D\boldsymbol{f} \cdot \boldsymbol{\Delta}$ represents a matrix-vector product. This is a very elegant definition that generalizes the earlier fixpoint over reals.[6] (Esparza *et al.*, 2010) proved that the Newton sequence $(\boldsymbol{x}^{(t)})_{n \geq 0}$ is unique (i.e. doesn't depend on the choice of $\boldsymbol{\delta}^{(t)}$), and, if Kleene's sequence $\boldsymbol{k}^{(t)}$ converges to the least fixpoint, then so does $\boldsymbol{x}^{(t)}$, and it converges "faster", meaning $\boldsymbol{k}^{(t)} \preceq \boldsymbol{x}^{(t)}$ for all n.

The significance of Newton's method is that it converts the original, possibly non-linear Datalog$^\circ$ program, into a sequence of linear programs (5.47). The problem is that this does not necessarily lead to performance gains. On one hand, it does nothing to improve a linear Datalog$^\circ$ program, because in that case the first super step of Newton's method degenerates into Kleene's method.[7]. On the other hand, even when the programs are non-linear, one experimental evaluation by (Reps *et al.*, 2016) conducted for program flow analysis has concluded that Newton's method is slower than Kleene's.

[6] Equations (5.46) and (5.47) correspond directly to (5.44) and (5.45), while Eq, (5.48) simply says that $\boldsymbol{\delta}^{(t)}$ stands for $\boldsymbol{f}(\boldsymbol{x}^{(t)}) - \boldsymbol{x}^{(t)}$.

[7] If the Datalog$^\circ$ program is linear, then $\boldsymbol{f}(\boldsymbol{x})$ is an affine function, $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{A} \cdot \boldsymbol{x} + \boldsymbol{b}$ for some matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$. When $t = 0$ we have $\boldsymbol{\delta}^{(0)} = \boldsymbol{b}$, and $\Delta^{(0)}$ is the least fixpoint of $\boldsymbol{\Delta}^{(0)} = \boldsymbol{A} \cdot \Delta^{(0)} + \boldsymbol{b}$, which is the same as $\boldsymbol{\Delta}^{(0)} = \boldsymbol{f}(\boldsymbol{\Delta}^{(0)})$. Solving for $\boldsymbol{\Delta}^{(0)}$ is the same as Kleene's method for $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x})$.

$\boldsymbol{x}^{(0)} \leftarrow 0$;
for $n \leftarrow 0$ **to** ∞ **do**
  `let` $\boldsymbol{\delta}^{(t)}$ `be s.t.` $\quad \boldsymbol{f}(\boldsymbol{x}^{(t)}) = \boldsymbol{x}^{(t)} + \delta^{(t)}$;
  `let` $\boldsymbol{\Delta}^{(t)}$ `be the lfp of` $\quad \boldsymbol{\Delta}^{(t)} = \boldsymbol{m}(\boldsymbol{x}^{(t)}) \cdot \boldsymbol{\Delta}^{(t)} + \boldsymbol{\delta}^{(t)}$;
  $\boldsymbol{x}^{(t+1)} \leftarrow \boldsymbol{x}^{(t)} + \boldsymbol{\Delta}^{(t)}$;
end
return $\bigvee_n \boldsymbol{x}^{(t)}$

Algorithm 8: Matrix Algorithm.

We can unify Newton's method and the seminaïve Algorithm 7 in a single algorithm, call it the *Matrix Algorithm* 8. To see this, notice that if we replace the Jacobian $D\boldsymbol{f}(\boldsymbol{x}^{(t)})$ in Eq. (5.47) with 0, then $\boldsymbol{\Delta}^{(t)} = \boldsymbol{\delta}^{(t)}$ and *Newton's method becomes the Semi Naïve Algorithm 7*. In fact, one can show something stronger. Let $\boldsymbol{m}(\boldsymbol{x})$ be any $N \times N$ matrix of polynomials satisfying the following:

$$\forall \boldsymbol{x}, \boldsymbol{\Delta} \in \boldsymbol{S}^N : \qquad \boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{m}(\boldsymbol{x}) \cdot \boldsymbol{\Delta} \preceq \boldsymbol{f}(\boldsymbol{x} + \boldsymbol{\Delta}) \tag{5.49}$$

Notice that this inequality holds when $\boldsymbol{m} \stackrel{\text{def}}{=} 0$, or when $\boldsymbol{m} \stackrel{\text{def}}{=} D\boldsymbol{f}$, or when $\boldsymbol{m}$ is anything in between, $0 \preceq \boldsymbol{m} \preceq D\boldsymbol{f}$. Then, if we replace the Jacobian $D\boldsymbol{f}(\boldsymbol{x}^{(t)})$ with $\boldsymbol{m}(\boldsymbol{x}^{(t)})$ in Eq. (5.47), the resulting sequence $\boldsymbol{x}^{(t)}$ has the same properties as Newton's sequence: it converges to the least fixpoint of $\boldsymbol{f}$ (assuming Kleene's sequence does), converges at least as fast as Kleene's sequence, and is unique.

5.7 Further Readings

6

Applications

This application chapter is **far** from complete. We need to discuss what to do here

6.1 Databases

Dan O.

Datalog is an expressive and declarative logic programming language that found its primary application as a database query language. Since late 70s, there has been interest for Datalog in academic circles, primarily in the database, logic and AI research communities, with four workshops dedicated to Datalog taking place over the previous decade (Moor *et al.*, 2011; Alviano and Pieris, 2019; Barceló and Pichler, 2012; Alviano and Pieris, 2022) and several research monographs and tutorials, e.g., (Huang *et al.*, 2011; Green *et al.*, 2013; Ketsman and Koutris, 2022).

Commercial database systems that support Datalog-based query languages include: Blazegraph[1], Datomic[2], EVE[3], GraphDB[4], LogicBlox[5] (Aref *et al.*, 2015; Green *et al.*, 2015), MarkLogic[6], Oracle's RDF-enabled database system

[1] http://www.blazegraph.com
[2] http://www.datomic.com
[3] https://incidentalcomplexity.com
[4] http://ontotext.com
[5] http://logicblox.com
[6] http://www.marklogic.com

extension[7] (Wu *et al.*, 2008), RDFox[8] (Motik *et al.*, 2014), Vadalog[9] (Bellomarini *et al.*, 2018), and Google's Yedalog (Chin *et al.*, 2015).

Motivating the choice for Datalog as their core query language, these systems noted the usability, expressive power, succinctness of application logic specification, safety, and performance. Yet to support real-world applications, their query languages require constructs beyond the pure Datalog discussed in Chapter 2. We next highlight a few extensions of pure Datalog with examples previously used to introduce the LogicBlox Datalog-based query language (Aref *et al.*, 2015). LogicBlox is a platform for developing sophisticated analytical applications that typically require the capabilities commonly found in programming languages, spreadsheets, OLAP and OLTP systems, statistical systems, and mathematical optimization systems. The glue that unifies such heterogeneous workloads is LogiQL (Halpin and Rugaber, 2014; Green, 2015), a Datalog extension that is expressive enough to allow coding entire applications including database queries, stored procedures, reactive rules and triggers, and statistical and mathematical modeling.

Application logic can be encoded more faithfully by allowing types and integrity constraints to be expressed in the language. For instance, a value can be of a primitive type, like integer, float, decimal, date, or string, and even a user-defined entity type. Integrity constraints, such as inclusion dependencies and functional dependencies, can encode knowledge about the domain.

Example 6.1. Consider an EDB predicate Stock that maps each product (p) to the quantity (x) available on stock. The following integrity constraint states that this predicate maps products, expressed using the entity type Product, to quantities on stock, expressed using the float primitive type:

$$\text{Stock}[p] = x \rightarrow \text{Product}(p), \text{float}(x).$$

The bracket notation $\text{Stock}[p] = x$ encodes the functional dependency $p \rightarrow x$, which means that the product is the key in Stock. To distinguish between derivation rules, as introduced in pure Datalog in Chapter 2, and integrity constraints, such as the one above, LogiQL uses the left arrow ($\leftarrow$) for the former and the right arrow ($\rightarrow$) for the latter.

The following inclusion dependency expresses that all products are available on stock (the underscore stands for a variable whose values are not relevant in the expression):

$$\text{Product}[p] \rightarrow \text{Stock}[p] = \_.$$

[7] http://docs.oracle.com/database/122/RDFRM/

[8] https://www.oxfordsemantic.tech

[9] https://www.meltwater.com/en/about/press-releases/meltwater-acquires-deepreason-ai

Arithmetic expressions and aggregation are essential for real-world applications. LogiQL has support for aggregation via a higher order predicate-to-predicate rule.

Example 6.2. The following rule computes the total profit for all products on stock:

$$\text{totalProfit}[\,] = u \leftarrow \textbf{agg} \ll u = \text{sum}(z) \gg \text{Stock}[p] = x, \text{profitPerProduct}[p] = y, z = x * y.$$

The aggregate construct expresses the sum over all values of z, which are defined for each product by the multiplication of the product quantity x and the profit y per product. The resulting IDB predicate totalProfit maps the empty key to the result u of the aggregation.

Rules with aggregation can be expressed in all aforementioned systems, albeit their syntax departs from pure Datalog more than LogiQL. The above rule can be encoeed as follows in EVE:

search
  Stock = [#p x], profitPerProduct = [#p y]
  profit = **gather/sum** [**for:** Stock, profitPerProduct,
    value: x * y]
bind [#**ui/text** *result* : "${{*profit*}}"]

Datalog can also be extended with more complex analytical tasks such as prescriptive mathematical optimization and predictive analytics. Modelling this mathematical optimization problem in a Datalog-like language can be achieved using two ingredients: We need to allow predicates as (1) free second-order variables that a mathematical solver can populate, and (2) objective functions to minimize or maximize.

Example 6.3. Let us continue our example and assume we would like to compute the stock that maximizes the total profit. For this, we model the predicate Stock as a free variable and expect the solver to find a stock quantity for each known product:

lang:solve:variable('Stock).

The modelling also needs to specify the IDB predicate totalProfit as the objective function to maximize:

lang:solve:max('totalProfit).

Furthermore, the optimization takes into account any further derivation rules and integrity constraints specified in the Datalog program.

Predictive analytics requires constructs to train and evaluate machine learning models that blend in Datalog rules.

Example 6.4. Assume we have the EDB predicates Sales, which gives the amount of sales per product p, store s and week w, and Feature, which gives the value f of a feature for each triple of product p, store s and feature name n. The following rule defines a linear regression model m to predict the sales v for each pair of product p and store s and that is trained using the available features f:

$$\mathrm{M}[p,s] = m \leftarrow \textbf{predict} \ll m = \mathrm{linreg}(v|f) \gg \\ \mathrm{Sales}[p,s,w] = v, \mathrm{Feature}[p,s,n] = f.$$

Once computed, the model can be used to predict sales for vectors of features represented as tuples in the predicate Test:

$$\mathrm{PredictedSales}[p,s] = v \leftarrow \textbf{predict} \ll v = eval(m|f) \gg \\ \mathrm{M}[p,s] = m, \mathrm{Test}[p,s,n] = f.$$

Besides analytical workloads, transactions are vital for concurrent access to databases. A natural extension of pure Datalog is by reactive rules to make and detect changes to the database state. A reactive rule is like a derivation rule in pure Datalog, yet with an indication whether it inserts, deletes, or upserts.

I plan to add an example here, much in the spirit of the one in the LB paper. –DanO.

LogicBlox also supports "live programming" in the database, where user's iterative data exploration triggers changes to installed application code written in LogiQL and its output in real time. Static analysis, such as enforcing logical invariants of LogiQL programs, optimization, and maintenance of application code written in LogiQL is done via another Datalog-like declarative language called MetaLogiQL (Green *et al.*, 2015). The LogicBlox meta-engine uses MetaLogiQL programs to incrementally maintains metadata representing application code, guide its compilation into an internal representation in the database kernel, and orchestrate maintenance of materialized views based on those changes. This declarative approach has key advantages over a typical imperative specification of the meta-engine, including less code, less error prone code, correctness guarantees while providing efficient maintenance, and easy extensibility via new MetaLogiQL rules.

the section reads well, and the length is right. But its dedicated entirely to LogicBlox; perhaps mention some other systems, e.g. Naiad? Or even recursion in SQL? Some of Thorsten Grust's recent work? – Dan.

6.2 Programming Languages

Amir, Hung

Program analysis has been considered a "killer app" for Datalog (Bravenboer and Smaragdakis, 2009; Hajiyev *et al.*, 2006; Huang *et al.*, 2011). The canonical program analysis task is the points-to analysis of object-oriented programs. Consider a Java program where we first allocate a new object on heap by `Object a = new Object();` and then we assign the reference to another object by `Object b = a;`. The points-to analysis asserts that both variables `a` and `b` point to the same heap location. This can be enabled by the following Datalog expression:

$$\mathsf{VarPointsTo}(\mathsf{H},\mathsf{V}) \text{ :- } \mathsf{HeapAllocation}(\mathsf{H},\mathsf{V}) \tag{6.1}$$

$$\mathsf{VarPointsTo}(\mathsf{H},\mathsf{V2}) \text{ :- } \mathsf{Assign}(\mathsf{V2},\mathsf{V1}), \mathsf{VarPointsTo}(\mathsf{H},\mathsf{V1}) \tag{6.2}$$

We have the following EDBs: !HeapAllocation(H,V)!, which specifies the heap location !H! associated with the variable !V!, and !Assign(V2,V1)!, which specifies the assignment of variable !V1! to !V2!. The IDB !VarPointsTo! computes a reachability query for computing all the variables that point to a particular heap location.

Nitpicking: can we switch the variable order and write $\mathsf{HeapAllocation}(\mathsf{V},\mathsf{H})$? This makes it consistent with $\mathsf{Assign}(\mathsf{V2},\mathsf{V1})$. –Dan.

we mention "negation" below. Can we have a simple example that motivates the need for negation? –Dan.

The first efforts towards using logic programming for a declarative specification of program analysis dates back to 90s (Dawson *et al.*, 1996; Reps, 1993). It was shown by Reps (Reps, 1994) that it is possible to use magic-sets optimization for a demand-driven analysis rather than a exhaustive one. However, all of these work mainly focused on analyzing small programs. The bddbddb system (Lam *et al.*, 2005; Whaley and Lam, 2004) can be considered as the first scalable Datalog-based context-sensitive program analysis tool. This system works by translating Datalog with stratified negation expressions into operations that work on a BDD represenation of relations. The usage of BDDs was motivated by their effectiveness for program analysis (Berndl *et al.*, 2003), which was itself inspired by their successful usage in model checking (Burch *et al.*, 1994). Similarly, Paddle (Lhoták and Hendren, 2008) uses a BDD representation expresses only the core part of program analysis in Datalog; the end-to-end task is represented by extending Java with BDD-based relational support (Lhoták and Hendren, 2004).

Doop (Bravenboer and Smaragdakis, 2009) expressed the entire points-to analysis task in Datalog, as opposed to bddbddb and Paddle. Furthermore, Doop employs an explicit representation as opposed to these systems that

use a BDD represenation. Doop offers one order of magnitude performance improvement over Paddle. This improvement is largely owed to the semi-naive evaluation offered by the LogicBlox (Aref *et al.*, 2015) engine. In addition, Doop uses faster joins by a careful reordering of variables and folding transformation. The latter optimization introduces intermediate IDBs, similar to view materialization, which can guide the query optimizer to prioritize the join between smaller relations. In addition to the LogicBlox engine, another backend based on the Soufflé (Scholz *et al.*, 2016) engine has also been provided for Doop. Soufflé uses C++ template programming to generate C++ code from the Datalog specification of program analysis.

QL (Avgustinov *et al.*, 2016), the successor of CodeQuest (Hajiyev *et al.*, 2006), is an object-oriented dialect of Datalog and has been used for source code analysis. As opposed to the previous systems, QL is mainly for analyzing higher level source code information, such as the subtyping relationship. It has been successfully used for analyzing the source code of NASA's Curiosity Mars Rover. After being commercialized as Semmle, it was acquired by github; thousands of QL queries run everyday to find bugs in code. Consider a Java program where `class A` is at the top of the hierarchy, `class B extends A` below it, and at the bottom we have `class C extends B`. The IDB !SubTypeStar! specifies all subtyping relationships and is computed by the following Datalog program:

$$\mathsf{SubTypeStar}(\mathsf{T},\mathsf{T}) \text{ :- } \mathsf{Type}(\mathsf{T}) \tag{6.3}$$

$$\mathsf{SubTypeStar}(\mathsf{T1},\mathsf{T3}) \text{ :- } \mathsf{SubType}(\mathsf{T1},\mathsf{T2}),\mathsf{SubTypeStar}(\mathsf{T2},\mathsf{T3}) \tag{6.4}$$

We have the following EDBs: !Type(T)!, which specifies that !T! is a type, and !SubType(T1,T2)!, which specifies that !T1! is an immediate subtype of !T2!. The IDB !SubTypeStar! computes the transitive closure over the subtyping relationship, by including the fact that a type is a subtype of itself (i.e., the reflexitivity relationship).

There were also functional-programming-based extensions proposed for Datalog. DataFun (Arntzenius and Krishnaswami, 2016) is a higher-order functional language that represents constraints as functions, and track the monotonicity in the type system. It has been also extended with a semi-naive evaluation engine (Arntzenius and Krishnaswami, 2020).

Flix (Madsen *et al.*, 2016) was initially introduced as an extension of datalog with lattices and functions to support a wider range of program analysis problems such as interprocedural finite distributive subset (IFDS) (Reps *et al.*, 1995) and interprocedural distributive environment (IDE) (Sagiv *et al.*, 1996) analyses. Later on, it was extended with a verification tool to ensure the soundness of program analysis tasks (Madsen and Lhoták, 2018). Afterwards, it has emerged to a generic-purpose programming language where datalog queries are integrated inside it (Madsen and Lhoták, 2020).

Similarly to Flix, IncA (Szabó *et al.*, 2016) is an extension of Datalog with

user-defined lattices. Initially, it targetted an engine speciliazied for incremental graph pattern matching (Ujhelyi *et al.*, 2015) and only supported incremental points-to anlaysis. In order to support incremental recursive aggregation required for interval and string analysis, the backend engine was extended with a variant of DRed algorithm (Gupta *et al.*, 1993), known as DRed_L (Szabó *et al.*, 2018). More recently, to support incremental inter-procedural procedural analysis, the Laddder (Szabó *et al.*, 2021) backend was introduced based on differential dataflow (DDF) (McSherry *et al.*, 2013).

6.3 Program Analysis

Newton method, etc

6.4 Graph computations

Shortest paths, APSP, etc

6.5 Distributed Datalog

Dan S.

This section is ready for review; feel free to edit. The section came out a bit long; we need to decide if we want to keep it as is, or shorten it, and where to keep it. –Dan.

One surprising class of applications of datalog is in distributed computing, for example in the declarative specification of network protocols (Loo *et al.*, 2009; Chu *et al.*, 2007; Condie *et al.*, 2008) or for defining Consistent Replicated Data Structures (CRDT) (Hellerstein and Alvaro, 2020). In these applications the data is distributed over a set of *servers*, which perform local computations and exchange messages. It turns out that minor, but carefully designed extensions of datalog can capture elegantly subtle and difficult concepts in distributed computations. In this section we overview a distributed datalog language, its semantics, and prove the most celebrated result in distributed datalog: the CALM conjecture, which asserts that a query is coordination free if and only if it is monotone. We base our presentation on Dedalus, introduced by Joe Hellerstein in his PODS'2010 keynote talk (Hellerstein, 2010a; Hellerstein, 2010b), whose theoretical properties are explored in (Marczak *et al.*, 2012; Ameloot *et al.*, 2016a). Dedalus is based on earlier declarative datalog variants, such as NDLog (Loo *et al.*, 2009), its extension Overlog (Loo *et al.*, 2005), and BLOOM (Huang *et al.*, 2011).

6.5.1 Syntax

In Dedalus a computation is performed on a network of servers. We will denote by `Servers` the finite set of servers in the network. A *horizontal partitioning* of a relation instance $A(X, Y, \ldots)$ in a family of sets such that $A = \bigcup_{s \in \texttt{Servers}} A_s$. We call A_s the fragment of A stored at server s. Fragments need not be disjoint. If I is the input database instance, then I_s consists of all fragments stored at server s, thus $I = \bigcup_s I_s$. In addition to partitioning, tuples are annotated with time. The time indicates when that tuple was present at a server, and is an important element in a distribute setting, because messages sent from one server to another will always arrive at a later time. Sometimes the space and time attributes are represented explicitly, e.g. by writing $A(s, t, x, y, \ldots)$ (Marczak *et al.*, 2012), sometimes they are dropped altogether (Hellerstein, 2010a). In our discussion we will settle for a compromise: we represent the server explicitly in a subscript, and leave the time implicit.

Deadlus consists of three kinds of rules:

$$\text{Deductive Rule:} \qquad H_S(\bar{X}_0) \text{ :- } (\neg)A_S(\bar{X}_1) \wedge (\neg)B_S(\bar{X}_2) \wedge \ldots \wedge \Phi \tag{6.5}$$

$$\text{Inductive Rule:} \qquad H_S(\bar{X}_0)@\texttt{next} \text{ :- } (\neg)A_S(\bar{X}_1) \wedge (\neg)B_S(\bar{X}_2) \wedge \ldots \wedge \Phi \tag{6.6}$$

$$\text{Asynchronous Rule:} \qquad H_{S'}(\bar{X}_0)@\texttt{async} \text{ :- } (\neg)A_S(\bar{X}_1) \wedge (\neg)B_S(\bar{X}_2) \wedge \ldots \wedge \Phi \wedge \texttt{Server}_S(S' \tag{6.7}$$

We explain these three types of rules. As in standard in datalog$^\neg$, each rule consists of positive and negated atoms, and is required to be safe, i.e. all variables must occur in positive atoms. Φ represents interpreted predicates, e.g. $X = Y$ or $Z > 12$. All atoms in the rule body must reside on the same server, as can be seen by the fact that they are all subscripted by the same variable S. All atoms in a rule body must also have the same timestamp, a fact which is not shown but is assumed implicitly. Consider the first type of rule, the *deductive rule*. By definition, its head atom H_S has the same server S and same timestamp T as the body. A deductive rule represents a local computation, without any communication. The deductive rules are required to be stratified.[10] Next, consider an *inductive rule*. The head atom $H_S(\cdots)@\texttt{next}$ has the same server S, while its time stamp is by definition $T + 1$, where T is the common time stamp of all atoms in the body. One common example of the deductive rule is the following *persistence rule*, which ensures that all

[10] A variant of Dedalus called Dedalus^S (Marczak *et al.*, 2012) requires *all* rules to be stratified. However, the time dimension already introduces a natural stratification, and for that reason we depart here a little from that variant of Deadlus, and ask only for the set of deductive rules to be stratified.

tuples of a relation A persist from one time stamp to the next:

$$\text{Persistence rule:} \qquad A_S(X, Y, \ldots)@\texttt{next} \text{ :- } A_S(X, Y, \ldots) \tag{6.8}$$

Finally, consider the *asynchronous rule*. Here both the server S' and the time stamp T' of the head atom differ from those in the body, S, T. The server S' can be any from the locally known set of servers, $\texttt{Server}_S$, and can possibly be further restricted through additional predicates. Since messages can be delayed arbitrarily by the network, the arrival time T' is not specified, but instead it is chosen nondeterministically as some future time, $T' > T$. The fragment $\texttt{Server}_S$ may contain all servers, in which case the asynchronous rule is a broadcast, or only the neighbors of S, in some network, in which case we assume the network to be connected, so that every server can communicate eventually with every other server.

We note that Dedalus restricts all computations to be done locally, by requiring that all atoms in the body be located at the same server. Systems that preceded Dedalus allowed atoms at different servers to be joined in a rule, but this lead to semantic difficulties, as discussed in Hellerstein, 2010b, Sec.3.5.2. In Dedalus this is not permitted, instead the communication needs to be made explicit through an asynchronous rule.

6.5.2 Semantics

One major appeal of Dedalus is that it can inherit the simple and elegant semantics of datalog. There are, however, four features that distinguish Dedalus from traditional datalog and that require a careful treatment: the spatio-temporal dimension of the data, the nondeterminism introduced by the asynchronous rules, the fact that the domain of time is infinite, and the need to extract an output from an infinitely running program. We discuss Dedalus' semantics here, by addressing these issues. We note that the semantics defined here is a much simplified version of that presented in (Ameloot *et al.*, 2016a): for variations and discussions we refer to (Ameloot *et al.*, 2016a). We denote in this section a Dedalus program by P.

First, in order to model the spatio-temporal dimension of the data, we expand each relational symbol $A(X, Y, \ldots)$ by adding an explicit server and time attribute, so that it becomes $A(S, T, X, Y, \ldots)$. Given the input EDB instances to the program, we consider some horizontal partition at time $T = 0$. The semantics of P may depend on the choice of this initial horizontal partition; we will return to this shortly. We will call a grounded atom of the form $A(s, 0, x, y, \cdots)$ where A is a EDB symbol an *initial EDB atom*. The initial atoms are the inputs to the program.

Second, in order to capture the temporal nondeterminism, for each asynchronous rule (6.7) we choose a function $f(S, T, S', \bar{X}_0) = T'$, which returns

some future time, $T' > T$, and rewrite rule (6.7) to:

$$H(S', T', \bar{X}_0) \text{ :- } (\neg)A(S, T, \bar{X}_1) \wedge (\neg)B(S, T, \bar{X}_2) \wedge \ldots \wedge \Phi \wedge \texttt{Server}_S(S') \wedge T' = f(S, T, S \tag{6.9}$$

We call f a *choice function*. In other words, the choice function f computes (deterministically) the delivery time $T' > T$, as a function of the source server S, the destination server S', the time when the message is sent T, and its content $\bar{X}_0$. We rewrite inductive rules (6.6) similarly, where the choice function f is defined simply as $f(T) \stackrel{\text{def}}{=} T + 1$. Thus, there will be one choice function for each inductive and each asynchronous rule. The semantics of P will also depend on the choice functions.

Third, we handle the infinite time domain by simply truncating it to a finite segment $0, 1, 2, \ldots, t$, then letting $t \to \infty$. Recall that the *grounding of a datalog rule* is a rule obtained by substituting all variables with some constants from the active domain of the database. Given a Dedalus program P, choice functions f, and a time $t \geq 0$, we define $P^{(t)}$ the finite set of all groundings of all its rules, where each time variable is restricted to be $\leq t$. In particular, if the rule is of the form (6.9), then we require that time stamp T' in the head be $\leq t$, which also implies $T < t$, because the choice function always returns a time in the future. Thus, for each $t \geq 0$, the grounded program $P^{(t)}$ consists of a finite set of rules, and is a standard grounded datalog program with negation. The input to this programs consists of all initial EDB atoms, i.e. of the form $A(s, 0, x, y, \cdots)$, where A is an EDB symbol. Finally, we note that $P^{(t)}$ is stratified: indeed, by assumption the deductive rules in P are stratified, and the time variable introduces a natural stratification of the deductive/asynchronous rules, because the time stamp of their head is always strictly greater than the time stamp of their body. Therefore, $P^{(t)}$ has a standard, stratified semantics, which we denote by $I^{(t)}$, and which consists of ground atoms of the form $A(s, t', x, y, \cdots)$, for $t' \leq t$. One can immediately check that the instance $I^{(0)}$ is the result of evaluating the deductive rules on the initial EDB atoms, and that $I^{(t-1)} \subseteq I^{(t)}$ for all $t > 0$. To check the latter, we notice that the only new rules in $P^{(t)}$ that did not exists in $P^{(t-1)}$ have the time stamp t in the head, and these are in a new stratum (or in several new strata), which was not present in $P^{(t-1)}$, because of the natural stratification by time. Therefore $I^{(t-1)}$ and $I^{(t)}$ agree on the tuples that have time stamp $< t$, which implies $I^{(t-1)} \subseteq I^{(t)}$. This justifies the following: we define $J \stackrel{\text{def}}{=} \bigcup_{t \geq 0} I^{(t)}$ and call it a *stable model* of the program P (following terminology in (Marczak *et al.*, 2012; Ameloot *et al.*, 2016a), and in analogy with the stable models in logic programming (Gelfond and Lifschitz, 1988)). We note that J is an infinite object, because it contains infinitely many time stamps t, and it may depend on both the initial horizontal partition of the EDBs, and the choice functions. Intuitively, J represents one possible run of

the program; other runs are possible.

The final question we need to address is to define the output of P, from J. A first attempt is the following. Call a tuple *eventually always true* (Marczak *et al.*, 2012), or *quiescent* (Ameloot *et al.*, 2013), if there exists a server s and time t such that the model J contains the tuple at s at all times $t, t+1, t+2, \ldots$ Let M be the set of eventually-always-true tuples, stripped of their server s and time step t attributes; M is called an *ultimate model* (Marczak *et al.*, 2012). One option would be to define the output of P to be its ultimate model M, a definition that was used in (Marczak *et al.*, 2012; Ameloot *et al.*, 2013). In general, M is not unique, because it depends on the stable model J; we will deal with this by simply requiring P to be *confluent*, meaning that its ultimate model must be unique (Marczak *et al.*, 2012). Checking confluence is undecidable in general[11], but we will ignore this issue, and just assume that P is confluent. However, even in this case, adopting the ultimate model M as the output of P is problematic, because it is unclear how to check whether a tuple is eventually-always-true. For that reason we will define the output of P differently, using ideas from Dedalus^S (Marczak *et al.*, 2012). We assume that the program has one distinguished IDB predicate, `Out`, and require `Out` to be persistent, i.e. we add rule (6.8) for `Out`; thus, every tuple inserted in `Out` is eventually-always-true. Given a stable model J, we define the output to P to consists of all `Out` tuples in J, stripped of their server and time stamp attributes. We call P *consistent* (terminology from (Ameloot *et al.*, 2013)) if `Out` is unique for all stable models J, and will always assume that P is consistent. Thus, P has a well-defined semantics, `Out`.

We end with a brief discussion of Dedalus^S, introduced in (Marczak *et al.*, 2012). Dedalus^S imposes two restrictions: first, the set of all rules must be stratified (not only the deductive rules), and there must be a persistency rule (6.8) for every predicate, EDB or IDB. The authors in (Marczak *et al.*, 2012) prove that every Dedalus^S program is confluent. We preferred to discuss a more permissive language, which requires stratification only for the deductive rules, because time already introduces a natural stratification.

6.5.3 Coordination

A central theme in distributed computing is ensuring global consistency, which often requires coordination between servers. Informally, coordination happens when the servers need to wait in order to synchronize their state. Consensus reaching protocols, like Two Phase Commit or Paxos, use coordination in an essential way in order to ensure that global consistency is reached. Both are expensive protocols, used sparingly by modern distributed systems, only when consistency cannot be guaranteed otherwise. A major result in distributed

[11] See Example 3 and Lemma 1 in (Marczak *et al.*, 2012).

datalog is to identify a rich class of queries that can be computed distributively without coordination. Before we state the result, we illustrate the notion of "coordination", by using two beautiful[12] examples from (Hellerstein and Alvaro, 2020).

Example 6.5.1 (Distributed Deadlock Detection)**.** When a transaction requests a lock and the lock is already taken by another transaction, then the first transaction is put in a wait state, until the lock is released by the second. A *deadlock* occurs if there exists a cycle in the Waits-For graph. For example, $\texttt{TXN}_1$ waits for a lock held by $\texttt{TXN}_2$, which waits for a lock held by $\texttt{TXN}_3$, and the latter waits for a lock held by $\texttt{TXN}_1$. When a deadlock occurs, the database system needs to abort one of the transactions in order to break the deadlock. Deadlock detection is done by checking for a cycle in the `WaitsFor` graph:

$$
\begin{aligned}
\texttt{Reach}_S(X,Y) &\text{ :- } \texttt{WaitsFor}_S(X,Y)\\
\texttt{Reach}_S(X,Y) &\text{ :- } \texttt{Reach}_S(X,Z) \wedge \texttt{WaitsFor}_S(Y,Z)\\
\texttt{Deadlock}_S(X) &\text{ :- } \texttt{Reach}_S(X,X)\\
\texttt{WaitsFor}_{S'}(X,Y)@\texttt{async} &\text{ :- } \texttt{WaitsFor}_S(X,Y) \wedge \texttt{Server}_S(S')
\end{aligned}
$$

The output relation is `Deadlock`, which returns all transactions that are on some cycle. The first three rules compute the transitive closure of `WaitsFor`, the third rule collects all transactions X on a cycle. The last rule sends the local fragment of `WaitsFor` to all other servers S'. Eventually, all servers receive the entire copy of `WaitsFor`, and will be able to determine all transactions that are deadlocked.

Example 6.5.2 (Distributed Garbage Collection)**.** A program has a set of variables that contain pointers to memory locations. The memory locations are distributed across servers, and each may, in turn, have pointers to other memory locations. The task is to identify all unreachable memory locations:

$$
\begin{aligned}
\texttt{Reachable}_S(X) &\text{ :- } \texttt{ProgramVar}_S(X)\\
\texttt{Reachable}_S(Y) &\text{ :- } \texttt{Reachable}_S(X) \wedge \texttt{PointsTo}_S(X,Y)\\
\texttt{Unreachable}_S(X) &\text{ :- } \texttt{MemoryLocation}_S(X) \wedge \neg\texttt{Reachable}_S(X)\\
\texttt{ProgramVar}_{S'}(X)@\texttt{async} &\text{ :- } \texttt{ProgramVar}_S(X) \wedge \texttt{Server}_S(S')\\
\texttt{PointsTo}_{S'}(X,Y)@\texttt{async} &\text{ :- } \texttt{PointsTo}_S(X,Y) \wedge \texttt{Server}_S(S')
\end{aligned}
$$

[12] The examples in (Hellerstein and Alvaro, 2020) are so clear that the paper omits giving any code, but instead describes them only informally. At the risk of taking away some of their glamour we give here datalog rules for each example, to help clarify some of the concepts introduce later.

The first two rules compute the set of reachable memory locations, and the third rule computes its complement. As before, the last two asynchronous rules broadcast their local fragment of the EDB to all other servers. Since at time $T = 0$ each server only has a subset of the data, they may incorrectly determine that some memory locations are unreachable. Later, as they receive more input tuples, these incorrect tuples will be retracted; lets examine this more closely. At time $T = 0$ the servers have only access to their local fragment of the EDBs `ProgramVar` and `PointTo`, and will incorrectly compute an overestimate of `Unreachable`. At times $T = 1, 2, \ldots$ they receive additional fragments of the EDBs, and will recompute `Unreachable`, resulting in ever smaller sets, until, eventually, they receive the entire EDBs and finally compute the correct output. Thus, the ultimate model of this program is unique, and correct, but according to our definition the output, this program is incorrect because its output `Unreachable` is not persistent: tuples inserted at tome time T may be removed at a later time, and we do not allow this. To fix it, one has to modify the program to detect when a server have received the entire EDB, and only then trigger the rule that computes the output. This requires additional coordination.[13]

These two examples differ in a fundamental way: distributed deadlock detection can be done without coordination, but the corrected version of distributed garbage collection does require coordination. In the first case, the servers can start producing outputs without waiting, while in the second case the servers must wait to receive all the data before they can output anything. This raises an important and interesting question: which queries require coordination and which don't?

6.5.4 The CALM Conjecture

In his celebrated PODS keynote talk (Hellerstein, 2010a) and accompanying paper (Hellerstein, 2010b), Joe Hellerstein proposed the following conjecture:

Conjecture 6.5.3 (Consistency And Logical Monotonicity (CALM))**.** A program has an eventually consistent, coordination-free execution strategy if and only if it is expressible in (monotonic) Datalog.

[13] This can be done using the mechanism in Ameloot *et al.*, 2013, Lemma 4.8. When a server S sends an EDB tuple, it includes its own identity S. Upon receiving such a tuple from S, server S' sends the EDB tuple back, as an acknowledgment, and includes both server IDs, S and S'. Server S collects all acknowledgments, and when it detects that some other server S' has acknowledged all tuples, then it sends S' a message acknowledging the completion. When a server S' has received this final acknowledgment from all servers S, then it can proceed to compute `Unreachable`.

The conjecture is well supported by the two earlier examples. Deadlock detection is a monotone query and is coordination-free, while garbage collection is non-monotone and requires coordination.

6.5.5 Monotonicity v.s. Coordination

The CALM conjecture was proven by Ameloot and coauthors in (Ameloot *et al.*, 2013), and later refined in (Ameloot *et al.*, 2016b). There are three important ideas in that line of work. First, the conjecture needs to be stated and proven for a more powerful distributed model than distributed datalog; second, they gave a rigorous definition of coordination, or its complement, coordination-freeness; and third, their results establish a fine-grained connection between variants of monotonicity and variants of distributed policies. In particular, it will turn out that the garbage collection query in Example 6.5.2 *is* coordination free under a suitably restricted notion of coordination freeness.

We start by describing a more powerful distributed computation model than datalog: a network of transducers. Our definition here is a simplified version from (Ameloot *et al.*, 2013; Ameloot *et al.*, 2016b), omitting details that, on closer inspection, turned out to be unnecessary for the main result. A *network of transducers* consists of a set of servers, where each server runs a local transducer. A *transducer* is a machine that has a local state, and a transition function. The *transition function* maps the current state and a set of incoming messages, to a new state, a set of outgoing messages, and a set of output tuples which it inserts in a relation `Out`. Each outgoing message is delivered to each server, asynchronously, at some non-deterministically chosen time in the future. A computation proceeds as follows. The *input* to the network of transducers consists a database instance I of EDB relations. Initially, at time $T = 0$, the input I is horizontally partition across the servers, and each fragment I_s becomes part of the servers' state. At each time T, each server applies its transition function to its current state and all incoming messages delivered at time T, and produces the new state at time $T+1$, a set of outgoing message to be delivered at future times T' (using *choice* functions, as in the semantics of Dedalus), and a set of output tuples to be inserted into `Out`. The output of the transducer consists of all tuples inserted in `Out`, at all time steps.[14] The network is required to be *consistent*: it must compute the same output, independently of the initial distribution of the EDBs, and of the choice functions.

A Dedalus program is a simple example of a network of transducers, where the local state at a server S and time T consists of all its EDB and IDB fragments at that time. A transducer network represents a generalization of this setting.

[14] Since we did not restrict the transducers to be *generic*, `Out` may be infinite.

We can now define a coordination-free transducer network.

Definition 6.5.4. A transducer network is *coordination-free* if it is consistent and, for any set of servers and any input database, there exists a horizontal partition of the database on which the transducer correctly computes `Out` at time $T = 0$.

The intuition is that, if there exists an ideal partition under which the servers can compute the output without waiting for any message, then the transducer is coordination-free. This does not preclude the need for messages, when the initial partition is not the ideal one. We illustrate with our two examples 6.5.1 and 6.5.2. First, we show that the deadlock detection Dedalus program in Example 6.5.1 is coordination free. Consider any set of servers. We partition the `WaitsFor` graph by giving the entire relation to each server. Then, the three deductive rules of the Dedalus program will compute the output relation `Deadlock` correctly at time $T = 0$, as required.

Next, consider the garbage collection program in Example 6.5.2. As written, this program appears to be coordination-free, because we can also give the entire EDB to each server, and the servers will compute the correct output at time $T = 0$. But this is not a correct program, since we require the answer `Unreachable` to be persistent. With this requirement, it is less clear whether garbage collection is coordination free. Assuming we give all data to all servers, will that server *know* that they have the entire database, to output `Unreachable` at time $T = 0$?

In general, the coordination-free property of a network of transducers depends on what assumptions we make about the horizontal partition of the EDBs and what the servers know about it. Three scenarios are considered in (Ameloot *et al.*, 2016b). We call a horizontal partition of the EDB *domain-guided* if there exists a partition of the domain $\mathsf{Dom} = \bigcup_{s \in \mathtt{Servers}} \mathsf{Dom}_s$ such that, for every relation A, the fragment at server s is $A_s \stackrel{\text{def}}{=} \{t \mid t \in A, \mathsf{ADom}(t) \cap \mathsf{Dom}_s \neq \emptyset\}$.

- A *distribution-unaware* transducer network has no information about the initial distribution. The Dedalus programs in Examples 6.5.1 and 6.5.2 are distribution-unaware: we made no assumption on how the input data was distributed.

- A *distribution-aware* network of transducers is one where each server knows which tuples are assigned to it. More precisely, the server s has access to an oracle that, given an EDB relation name A and constants $u, v, w, \ldots$, determines whether the initial horizontal of A would assign the tuple $A(u, v, w, \cdots)$ to s. If the answer is "yes", then by examining its local fragment of A it can determine that either a positive or a

negative atom holds in I:

$$\text{Positive atom:} I \models A(u, v, \cdots) \quad \text{or Negative atom:} I \models \neg A(u, v, \cdots) \tag{6.10}$$

- A *domain-guided* network of transducers is a distribution-aware network where the initial partition of the EDBs is domain-guided. More precisely, the server s has access to an oracle that, given a constant u, determines whether the initial horizontal partition would assign the tuples containing u to s. If the answer is "yes", then by examining its local fragment of the EDB, the sever can determine that the instance I satisfies the following *denial constraint*:

$$t \in I \wedge u \in \mathsf{ADom}(t) \Rightarrow (t = t_1 \vee \cdots \vee t = t_m) \tag{6.11}$$

  where $t_1, \ldots, t_m$ are all tuples in its local fragment that contain the constant u. Denial constraints are more powerful then negative atoms, because each denial constraint implies every negative atom that contains u and is not among $t_1, \ldots, t_m$.

In practice, one finds all three scenarios. For example, if the data is hash-partitioned on a key attribute, then the network is distribution-aware, since a server can use the hash function and check if it is responsible for a tuple with certain key and, if yes, can determine categorically (yes/no) if that tuple is in the database. It turns out that each type of transducer network gives rise to a different notion of coordination-freeness (Ameloot *et al.*, 2016b):

Definition 6.5.5. Consider a query (function) Q that maps an input database I to some output relation `Out`.

- We say that Q is *coordination free* if there exists a distribution-unaware transducer network that computes Q. Denote by $\mathcal{F}_0$ the set of coordination free queries.
- We say that Q is *distribution-aware coordination free* if there exists a distribution-aware transducer network that computes Q. Denote by $\mathcal{F}_1$ the set of distribution-aware coordination-free queries.
- We say that Q is *domain-guided coordination free* if there exists a domain-guided transducer network that computes Q. Denote by $\mathcal{F}_2$ the set of domain-guided coordination-free queries.

It is easy to check that $\mathcal{F}_0 \subseteq \mathcal{F}_1 \subseteq \mathcal{F}_2$. We invite the reader to pause here, and check that the distributed garbage collection query in Example 6.5.2 is in $\mathcal{F}_2$, before reading the proof of Theorem 6.5.7 below; notice that coordination-freeness is a property of the *query*, not of the datalog program. (Hint: the

servers need to use the domain-guided oracle to determine that they have all the data.)

Returning to the Calm conjecture, our goal is to prove that coordination-free queries are the same as monotone queries. But since we have three variants of coordination-freeness, we need similarly three variants of monotonicity. We introduce them next.

Given a finite domain Dom and a database schema, we will denote by $U(\mathsf{Dom})$ the database consisting of all possible tuples consisting of constants in Dom; notice that, if I is any instance, then $U(\mathsf{ADom}(I)) - I$ represent all negated atoms, i.e. all tuples that are missing in I.

Given two instances I, J we say that I is *an induced fragment* of J if $I \subseteq J$ and $(U(\mathsf{ADom}(I)) - I) \subseteq (U(\mathsf{ADom}(J)) - J)$; equivalently, for every tuple t whose constants are in $\mathsf{ADom}(I)$, t is in I iff t is in J. Finally, we say that I is a *component* of J if $I \subseteq J$ and $\mathsf{ADom}(I) \cap \mathsf{ADom}(J - I) = \emptyset$. The following variants of monotonicity where defined in (Ameloot *et al.*, 2016b):

Definition 6.5.6. Let Q be a query mapping an input database I to some output relation `Out`.

- Q is *monotone* if $I \subseteq J$ implies $Q(I) \subseteq Q(J)$. We denote by $\mathcal{M}_0$ the set of monotone queries.
- Q is *domain-distinct monotone*, if, for all I, J where I is an induced fragment of J, $Q(I) \subseteq Q(J)$. We denote by $\mathcal{M}_1$ the set of domain-distinct monotone queries.
- Q is *domain-disjoint monotone* if, for all I, J where I is a component of J, $Q(I) \subseteq Q(J)$. We denote by $\mathcal{M}_2$ the set of domain-disjoint monotone queries.

We give three simple datalog examples for each class:

$$\mathsf{Q}_0(\mathsf{X},\mathsf{Y}) \text{ :- } \mathsf{A}(\mathsf{X},\mathsf{Y}) \wedge \mathsf{B}(\mathsf{X},\mathsf{Y}) \quad \Big| \quad \mathsf{Q}_1(\mathsf{X},\mathsf{Y}) \text{ :- } \mathsf{A}(\mathsf{X},\mathsf{Y}) \wedge \neg\mathsf{C}(\mathsf{X}) \quad \Big| \quad \begin{array}{rl} \mathsf{D}(\mathsf{X}) \text{ :- } & \mathsf{B}(\mathsf{X},\mathsf{Y}) \\ \mathsf{Q}_2(\mathsf{X},\mathsf{Y}) \text{ :- } & \mathsf{A}(\mathsf{X},\mathsf{Y}), \neg\mathsf{D}(\end{array}$$

Q_0 is a join, Q_1 is an anti-semijoin, and Q_2 is an anti-semijoin involving a projection. We invite the reader to check that $Q_i \in \mathcal{M}_i$ for $i = 0, 1, 2$, and that $Q_1 \notin \mathcal{M}_0$ and $Q_2 \notin \mathcal{M}_1$. More examples can be found in (Ameloot *et al.*, 2016b)

We are now ready to state and prove the formal variant of the Calm Conjecture, following (Ameloot *et al.*, 2013; Ameloot *et al.*, 2016b).

Theorem 6.5.7 (The CALM Theorem). $\mathcal{F}_i = \mathcal{M}_i$ for $i = 0, 1, 2$.

In other words, a query is monotone iff it is coordination free, $\mathcal{F}_0 = \mathcal{M}_0$, and this characterization extends to refined notions of coordination freeness

and monotonicity: $\mathcal{F}_1 = \mathcal{M}_1$ and $\mathcal{F}_2 = \mathcal{M}_2$. The proof of the theorem is particularly insightful and we provide it here.

Monotonicity Implies Coordination-Freeness Fix a query Q. We need to describe a network of transducers that computes Q correctly and is coordination free. We start with the case $Q \in \mathcal{M}_0$, and design the transducers as follows. Let I_s be the fragment of the input instance I available at server s. At each step, the transducer computes $Q(I_s)$, inserts the result in `Out`, and sends every tuple in its local fragment I_s to all other servers (similar to an asynchronous rule in Dedalus). When the server receives new tuples, it inserts them in its local fragment I_s. Since at all times $I_s \subseteq I$, we have $Q(I_s) \subseteq Q(I)$ by monotonicity, hence only correct tuples are produced. Eventually each server s will see the entire input, $I_s = I$, then $Q(I_s) = Q(I)$, proving that the transducer network computes Q correctly. Finally, we check that the transducer network is coordination free: for any set of servers, sent the entire input database to each server, $I_s = I$. Then the servers compute `Out` at time $T = 0$.

Consider now the case when $Q \in \mathcal{M}_1$. We modify the previous network of transducers as follows. Given the input database I, denote by $U \stackrel{\text{def}}{=} U(\mathsf{ADom}(I))$: recall that this is the instance consisting of all tuples that can be formed using constants in $\mathsf{ADom}(I)$, in particular, $I \subseteq U$. Our new transducers will store both positive atoms and negative atoms, see (6.10). We denote the set of positive atoms known by server s by I_s and the negative atoms by $(U - I)_s$. At time $T = 0$, we initialize the positive atoms I_s to the fragment of the EDB assigned to s by the horizontal partition, and the negative atoms $(U - I)_s$ to all negative atoms that can be obtained by inquiring the domain-aware oracle using constants in $\mathsf{ADom}(I_s)$. At each time step T all servers broadcast all their known positive and negative atoms. When they receive such atoms they insert them into I_s or $(U - I)_s$ respectively; whenever $\mathsf{ADom}(I_s)$ increases, the server inquires again its domain-aware oracle with the new constants, possibly obtaining new negative atoms, which it inserts into $(U - I)_s$. Notice that the following invariants hold: $I_s \subseteq I$ and $(U - I)_s \subseteq U - I$. At each time T, the server checks whether $I_s \cup (U - I)_s = U(\mathsf{ADom}(I_s))$; in other words, it checks if it has complete information about all tuples with constants in $\mathsf{ADom}(I_s)$. If "yes", then I_s is an induced fragment of I, and the server computes $Q(I_s)$ and inserts it in `Out`; this is correct because $Q(I_s) \subseteq Q(I)$ since Q is domain-distinct monotone. Eventually, every server s will receive all positive atom, i.e. $I_s = I$, and all negative atoms, i.e. $(U - I)_s = U - I$, and it will compute $Q(I_s) = Q(I)$ and insert in into `Out`. Thus, the transducer network is correct. Finally, we argue that it is domain-aware coordination-free. Indeed, given any input I, we simply give the entire input to each server, and define the oracle at each server to always return "true". Then, at time $T = 0$, each server knows all positive atoms, $I_s = I$, and all negative atoms,

$(U - I)_s = U - I$, and therefore condition $I_s \cup (U - I)_s = U(\mathsf{ADom}(I_s))$ holds because $I \cup (U - I) = U = U(\mathsf{ADom}(I))$ by definition. Thus, the server will compute $Q(I)$ at time $T = 0$, as required.

Finally, we consider the case when $Q \in \mathcal{M}_2$. We modify the network of transducers above by replacing the set of negative tuples with a set of denial constraints (6.11): the state of each server consists now of I_s and a set of denial constraints. As before, at time $T = 0$ we initialize I_s to the fragment of the EDB assigned to server s, and initialize the denial constraints with all constraints that can be inferred from the domain-guided oracle using constants in $\mathsf{ADom}(I_s)$. At each step, the servers exchange this information by broadcast, and extended their positive atoms and their denial constraints; when $\mathsf{ADom}(I_s)$ increases, then server s queries its oracle again, using the new constants, for additional denial constraints. At each time, the server checks whether I_s is disjoint from $I - I_s$, as follows: for every constant $u \in \mathsf{ADom}(I_s)$, it checks whether it knows some denial constraint (6.11) with the constant u on the left hand side, and, if so, checks whether I_s contains all tuples $t_1, \ldots, t_m$. If "yes", then[15] $\mathsf{ADom}(I_s) \cap \mathsf{ADom}(I - I_s) = \emptyset$, in other words, I_s is a component of I, and the sever computes $Q(I_s)$ and insert the result in `Out`, which is correct, because $Q(I_s) \subseteq Q(I)$ since the query is domain-disjoint monotone. Eventually, each sever s will learn the entire instance, $I_s = I$, and all denial constraints: at that point each server will compute $Q(I)$ and insert the result in `Out`, thus, the network of transducers is correct. It remains to prove that it is coordination-free. As before, given any instance I, we send the entire input to every server; this is a domain-guided partition, which assigns each constant u in the domain to each server. The servers will learn at time $T = 0$ that their local fragment $I_s = I$ is domain disjoint from the rest (since the rest is $I - I_s = \emptyset$), and output $Q(I)$; notice that they don't know that they have the entire database, but they know that their fragment is a component, which is sufficient for them to output $Q(I_s)$.

The reader may have noticed that the denial constraints can be arbitrarily long, which means that the messages sent by our last network of transducers do not have a fixed type. This is allowed by our simplified definition of a network of transducers. It is possible to modify the transducers to ensure that all messages have a fixed type; we refer the reader to (Ameloot *et al.*, 2016b).

Coordination-Freeness Implies Monotonicity Start with a query $Q \in \mathcal{F}_0$, and consider any two instances I, J such that $I \subseteq J$. We will prove that $Q(I) \subseteq Q(J)$. Fix an output tuple $t \in Q(I)$; it suffices to prove that

[15] Proof: assume the contrary, that there exists a tuple $t \in I - I_s$ s.t. $\mathsf{ADom}(t) \cap \mathsf{ADom}(I_s) \neq \emptyset$, and let u be a constant in $\mathsf{ADom}(t) \cap \mathsf{ADom}(I_s)$. Consider the denial constraint (6.11) that refers to u and for which the server has verified that $t_1, \ldots, t_m \in I_s$. Then we obtain a contradiction, because the constraint implies $t = t_1 \vee \cdots \vee t = t_m$, while we assumed $t \notin I_s$.

$t \in Q(J)$. Consider a coordination-free network of transducers with 2 servers, s_1, s_2. By the definition of coordination-freeness, there exists a horizontal partition $I = I_1 \cup I_2$ on which the network computes $Q(I)$ at time $T = 0$. At least one of the two servers must write the tuple t to `Out`, and we assume w.l.o.g. that this is server s_1. Consider now the larger instance J, and partition it as follows: $J_1 = I_1$ and $J_2 = I_2 \cup (J - I)$. In other words, all extra tuples are "sent away" to server s_2. At time $T = 0$ server s_1 has now exactly the same information that it had when the input was I: hence, it will still output the tuple t, proving that $t \in Q(J)$.

Consider now $Q \in \mathcal{F}_1$. The proof is almost identical to the previous one. Let I be an induced fragment of J, and consider a tuple $t \in Q(I)$. We will prove that $t \in Q(J)$. Consider a partition-aware network of transducers with 2 servers s_1, s_2. Let $I = I_1 \cup I_2$ be the partition under which the servers compute $Q(I)$ at time $T = 0$, and assume w.l.o.g. that t is produced by server s_1. Notice that the two distribution-aware oracles at s_1 and s_2 only answer inquires about tuples with constants in $\mathsf{ADom}(I)$. Consider now the larger instance J. We partition it as above: $J_1 = I_1$, $J_2 = I_2 \cup (J - I)$. We also need to define the distribution-aware oracles at servers s_1 and s_2. Given a tuple $t = A(u, v, w, \cdots)$, if all its constants are in $\mathsf{ADom}(I)$, then oracle at s_1 answers as it did when the input was I, otherwise it will answer "false"; the oracle for s_2 will answer as before in the first case, and will answer "true" in the second case. This is correct, because any new tuple $t \in J - I$ will contain at least one constant $\notin \mathsf{ADom}(I)$, and therefore the two oracles will ensure that this tuple is stored only at server s_2. It follows that at time $T = 0$ server s_1 has exactly the same information as it had when the input was I instead of J, and, therefore, it will output the tuple t as before.

Finally, consider $Q \in \mathcal{F}_2$. The argument here is similar. Assuming that I is a component of J, and $t \in Q(I)$, we consider the domain-aware, coordination-free network of transducers with two servers s_1, s_2 computing $Q(I)$, and assume w.l.o.g. that t is output by server s_1 at time $T = 0$. Given the larger instance J, we extend the domain-guided partition used for I such that all the new constants in $\mathsf{ADom}(J) - \mathsf{ADom}(I)$ are handled by server s_2. Using an argument similar to the one above, we can argue that server s_1 has exactly the same information at time $T = 0$ and the same domain-guided oracle as it had for the input I, and therefore it will still output the tuple t, proving $t \in Q(J)$.

6.5.6 Discussion

Different applications of distributed systems exhibit each of the three variants of transducer networks. For example, in distributed query processing the data is hash-partitioned based on the join attribute, and this is, in our terminology, is *distribution aware*: a server, knowing the hash function, can check if it is responsible for a particular tuple. Even the more complicated hypercube

partitioning (Afrati and Ullman, 2010; Beame *et al.*, 2017) is distribution aware. In contrast, the universal hashing distribution is *domain-guided*: the servers share a hash function $h : \mathsf{Dom} \to \texttt{Servers}$, and each tuple $A(x, y, z, \cdots)$ is sent to all servers $h(x), h(y), h(z), \ldots$ On the other hand, applications that require elasticity, such as a distributed file server, or a distributed key-value store, do not have a fixed set of servers, and rely on an additional level of indirection: there is a *directory server* (usually replicated), which stores the information about the partitioning. Any distributed algorithm in this case is *distribution-unaware*. Thus, all three cases of Theorem 6.5.7 find applications in practice.

Somewhat surprisingly, the query `Unreachable` in Example 6.5.2 is domain-disjoint monotone and, therefore, it is domain-guided coordination-free. However, in practice, distributed stores do not use a domain-guide partition, and in that case garbage collection requires coordination. Another interesting example is the *win-move* query, which was observed in (Zinn *et al.*, 2012) to be in $\mathcal{F}_2$:

$$\texttt{Win}(X) \text{ :- } E(X, Y) \wedge \neg \texttt{Win}(Y) \tag{6.12}$$

The query considers a game between two players who alternate moving a pebble on a graph along the edges: the player who cannot move, loses. The query above computes all winning positions for Player 1. This example has been extensively studied in the database literature: for example Kolaitis (Kolaitis, 1991) has proven that this query cannot be expressed in stratified datalog, even if the input graph is assumed to be acyclic. Returning to the question of coordination, it is easy to see that this query is disjoint-domain monotone: if we add another connected component to the graph, we do not change the winning/losing status of the current nodes. Therefore, this query, too, is domain-guided coordination-free.

Finally, we examine again the original statement of CALM conjecture, which we quoted ad-litteram in Conjecture 6.5.3. It is a statement about datalog, not about general queries. Could monotonicity be the same as coordination free when restricted to datalog queries? More precisely, lets consider the following statement:

> A datalog$^\neg$ query (meaning: stratified datalog) is coordination free iff it is expressible in (montonic) datalog.

Surprisingly, this statement is *false*. To prove this, we consider the following result from (Ketsman and Koch, 2020): there exists a monotone query Q that is expressible in datalog$^\neg$ (i.e. stratified datalog) but not expressible in (monotonic) datalog. Since Q is monotone, it is coordination free by Theorem 6.5.7, thus contradicting the statement above.[16] This highlights how fundamental

[16]The reader may wonder what that query Q is. In a nutshell, Q takes as input

and critical the idea introduced in (Ameloot *et al.*, 2013) is, of extending the notion of distributed computation from datalog to more powerful networks of transducers.

6.6 Knowledge graphs

6.7 Mathematical optimization

6.8 Systems

Cite Paris and Bas paper

6.8.1 Dyna

Amir

Dyna (Eisner and Filardo, 2010) is a declarative language developed for AI workloads including Natural Language Processing (NLP). This language is largely inspired by Datalog, with the several extensions. First, the rules are weighted; the proofs aggregate numerical values. For example, Datalog shows the reachability as follows:

$$\mathsf{Edge}(\mathsf{src}, \mathsf{dest}) \tag{6.13}$$

$$\mathsf{Path}(\mathsf{S}, \mathsf{D}) \text{ :- } \mathsf{Edge}(\mathsf{S}, \mathsf{T}), \mathsf{Path}(\mathsf{T}, \mathsf{D}) \tag{6.14}$$

Test datalog $\mathsf{Edge}(\mathsf{src}, \mathsf{dest})$
Dyna can additionally express a decaying sum of all paths as follows:

$$\mathsf{Edge}(\mathsf{src}, \mathsf{dest}) \mathsf{:=} 10 \tag{6.15}$$

$$\mathsf{Path}(\mathsf{S}, \mathsf{D}) \mathsf{+=} \mathsf{Edge}(\mathsf{S}, \mathsf{T}) + 0.75 * \mathsf{Path}(\mathsf{T}, \mathsf{D}) \tag{6.16}$$

Second, rather than only supporting flat objects, Dyna supports complex objects such as lists. This is achieved by uninterpretted functions (Ceri *et al.*,

a graph, an checks whether G has a perfect matching. The proof in (Ketsman and Koch, 2020) is based on a result by Razborov from 1995 (in Russian) which says that no polynomial size family of monotone circuits exists for computing a perfect matching, yet computing a perfect matching is known to be in PTIME. This, however, does not immediately give us the query Q we seek. On one hand, Razborov's result proves that no monotone datalog program exists for computing Q, but we still need to prove that Q is computable in stratified datalog. This is possible if the datalog program has access to a total order, because then datalog$^\neg$ captures PTIME, however, the resulting program is not necessarily monotone on this order relation: the proof in (Ketsman and Koch, 2020) consists in showing how this program can be designed to be monotone.

1989), which makes the language Turing-complete. For example, the following rule is allowed:

$$\mathsf{Nat}(\mathsf{Succ}(\mathsf{X})) \text{ :- } \mathsf{Nat}(\mathsf{X}) \tag{6.17}$$

Third, there is no more restriction for the variables appearing in the head to appear in the body. This allows for the definition of functions with infinite facts. For example, one can define the following rule that is without any body and checks the equality of two objects:

$$\mathsf{Equals}(\mathsf{X}, \mathsf{X}). \tag{6.18}$$

Finally, as opposed to the stratified extensions of Datalog, recursive rules with aggregation and negation in Dyna can include cycles. This is motivated by the non-stratified patterns for Recurrent Neural Networks (RNNs), iterative optimizations, and dynamic programs.

Because of these extensions, Dyna cannot benefit from the monotonic reasoning and requires a more sophisticated execution model. Forward chaining (Eisner *et al.*, 2005) is the main evaluation strategy for Dyna, which is an iterative process that propagates updates from the body to the head of each rule until convergence. This technique can be considered as the non-batch (record-at-a-time) variant of the semi-naïve evaluation technique.

7

Open Questions

This open-problem chapter is **far** from complete. We need to discuss what to do here

7.1 Semantics

Datalogo and negation?

Can we capture common computations in linear algebra (solve Ax=b, compute eigenvalues, etc.)

7.2 Convergence

many from the Datalogo paper

7.3 Optimizations

Graph computations, Brandes algorithm from straightforward formulation

Tensor decomposition (such as SVD, matrix mult in Strassen-time)

How to find function h

Costing solutions

8

Conclusions

References

Abiteboul, S., R. Hull, and V. Vianu. (1995). *Foundations of Databases*. Addison-Wesley. ISBN: 0-201-53771-0. URL: http://webdam.inria.fr/Alice/.

Abiteboul, S. and V. Vianu. (1991). "Non-Determinism in Logic-Based Languages". *Ann. Math. Artif. Intell.* 3(2-4): 151–186. DOI: 10.1007/BF01530924. URL: https://doi.org/10.1007/BF01530924.

Abo Khamis, M., H. Q. Ngo, and D. Suciu. (2016). "Computing Join Queries with Functional Dependencies". In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by T. Milo and W. Tan. ACM. 327–342. DOI: 10.1145/2902251.2902289. URL: https://doi.org/10.1145/2902251.2902289.

Abo Khamis, M., H. Q. Ngo, and D. Suciu. (2017). "What Do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another?" In: *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*. Ed. by E. Sallinger, J. V. den Bussche, and F. Geerts. ACM. 429–444. DOI: 10.1145/3034786.3056105. URL: https://doi.org/10.1145/3034786.3056105.

Aceto, L., Z. Ésik, and A. Ingólfsdóttir. (2002). "A fully equational proof of Parikh's theorem". In: vol. 36. No. 2. 129–153. DOI: 10.1051/ita:2002007. URL: https://doi-org.gate.lib.buffalo.edu/10.1051/ita:2002007.

Afrati, F. N. and J. D. Ullman. (2010). "Optimizing joins in a map-reduce environment". In: *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*. Ed. by I. Manolescu, S. Spaccapietra, J. Teubner, M. Kitsuregawa, A. Léger, F. Naumann, A. Ailamaki, and F. Özcan. Vol. 426. *ACM International Conference Proceeding Series*. ACM. 99–110. DOI: 10.1145/1739041.1739056. URL: https://doi.org/10.1145/1739041.1739056.

Aho, A. V., C. Beeri, and J. D. Ullman. (1979). "The Theory of Joins in Relational Databases". *ACM Trans. Database Syst.* 4(3): 297–314. DOI: 10.1145/320083.320091. URL: https://doi.org/10.1145/320083.320091.

Aji, S. M. and R. J. McEliece. (2000). "The generalized distributive law". *IEEE Transactions on Information Theory*. 46(2): 325–343. DOI: 10.1109/18.825794. URL: http://dx.doi.org/10.1109/18.825794.

Alviano, M., W. Faber, and M. Gebser. (2023). "Aggregate Semantics for Propositional Answer Set Programs". *Theory Pract. Log. Program.* 23(1): 157–194. DOI: 10.1017/S1471068422000047. URL: https://doi.org/10.1017/S1471068422000047.

Datalog 2.0 2019 - 3rd International Workshop on the Resurgence of Datalog in Academia and Industry co-located with the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019) at the Philadelphia Logic Week 2019, Philadelphia, PA (USA), June 4-5, 2019. (2019). Ed. by M. Alviano and A. Pieris. Vol. 2368. *CEUR Workshop Proceedings*. CEUR-WS.org. URL: http://ceur-ws.org/Vol-2368.

Proceedings of the 4th International Workshop on the Resurgence of Datalog in Academia and Industry (Datalog-2.0 2022) co-located with the 16th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2022), Genova-Nervi, Italy, September 5, 2022. (2022). Ed. by M. Alviano and A. Pieris. Vol. 3203. *CEUR Workshop Proceedings*. CEUR-WS.org. URL: http://ceur-ws.org/Vol-3203.

Ameloot, T. J., J. V. den Bussche, W. R. Marczak, P. Alvaro, and J. M. Hellerstein. (2016a). "Putting logic-based distributed systems on stable grounds". *Theory Pract. Log. Program.* 16(4): 378–417. DOI: 10.1017/S1471068415000381. URL: https://doi.org/10.1017/S1471068415000381.

Ameloot, T. J., B. Ketsman, F. Neven, and D. Zinn. (2016b). "Weaker Forms of Monotonicity for Declarative Networking: A More Fine-Grained Answer to the CALM-Conjecture". *ACM Trans. Database Syst.* 40(4): 21:1–21:45. DOI: 10.1145/2809784. URL: https://doi.org/10.1145/2809784.

Ameloot, T. J., F. Neven, and J. V. den Bussche. (2013). "Relational transducers for declarative networking". *J. ACM.* 60(2): 15:1–15:38. DOI: 10.1145/2450142.2450151. URL: https://doi.org/10.1145/2450142.2450151.

Apt, K. R., H. A. Blair, and A. Walker. (1988). "Towards a Theory of Declarative Knowledge". In: *Foundations of Deductive Databases and Logic Programming*. Ed. by J. Minker. Morgan Kaufmann. 89–148. DOI: 10.1016/b978-0-934613-40-8.50006-3. URL: https://doi.org/10.1016/b978-0-934613-40-8.50006-3.

Aref, M., B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. (2015). "Design and Implementation of the LogicBlox System". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by T. K. Sellis, S. B. Davidson, and Z. G. Ives. ACM. 1371–1382. DOI: 10.1145/2723372.2742796. URL: https://doi.org/10.1145/2723372.2742796.

Arntzenius, M. and N. Krishnaswami. (2020). "Seminaïve evaluation for a higher-order functional language". *Proc. ACM Program. Lang.* 4(POPL): 22:1–22:28. DOI: 10.1145/3371090. URL: https://doi.org/10.1145/3371090.

Arntzenius, M. and N. R. Krishnaswami. (2016). "Datafun: a functional Datalog". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. Ed. by J. Garrigue, G. Keller, and E. Sumii. ACM. 214–227. DOI: 10.1145/2951913.2951948. URL: https://doi.org/10.1145/2951913.2951948.

Atserias, A., M. Grohe, and D. Marx. (2008). "Size Bounds and Query Plans for Relational Joins". In: *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*. IEEE Computer Society. 739–748. DOI: 10.1109/FOCS.2008.43. URL: https://doi.org/10.1109/FOCS.2008.43.

Avgustinov, P., O. de Moor, M. P. Jones, and M. Schäfer. (2016). "QL: Object-oriented Queries on Relational Data". In: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. Ed. by S. Krishnamurthi and B. S. Lerner. Vol. 56. *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. 2:1–2:25. DOI: 10.4230/LIPIcs.ECOOP.2016.2. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2016.2.

Backhouse, R. C. and B. A. Carré. (1975). "Regular algebra applied to path-finding problems". *J. Inst. Math. Appl.* 15: 161–186. ISSN: 0020-2932.

Bagan, G., A. Durand, and E. Grandjean. (2007). "On Acyclic Conjunctive Queries and Constant Delay Enumeration". In: *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*. Ed. by J. Duparc and T. A. Henzinger. Vol. 4646. *Lecture Notes in Computer Science*. Springer. 208–222. DOI: 10.1007/978-3-540-74915-8_18. URL: https://doi.org/10.1007/978-3-540-74915-8%5C_18.

Baget, J., M. Leclère, M. Mugnier, and E. Salvat. (2011a). "On rules with existential variables: Walking the decidability line". *Artif. Intell.* 175(9-10): 1620–1654. DOI: 10.1016/j.artint.2011.03.002. URL: https://doi.org/10.1016/j.artint.2011.03.002.

Baget, J., M. Mugnier, S. Rudolph, and M. Thomazo. (2011b). "Walking the Complexity Lines for Generalized Guarded Existential Rules". In: *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*. Ed. by T. Walsh. IJCAI/AAAI. 712–717. DOI: 10.5591/978-1-57735-516-8/IJCAI11-126. URL: https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-126.

Balbin, I., G. S. Port, K. Ramamohanarao, and K. Meenakshi. (1991). "Efficient Bottom-UP Computation of Queries on Stratified Databases". *J. Log. Program.* 11(3&4): 295–344. DOI: 10.1016/0743-1066(91)90030-S. URL: https://doi.org/10.1016/0743-1066(91)90030-S.

Balbin, I. and K. Ramamohanarao. (1986). "A differential approach to query optimisation in recursive databases". *Tech. rep.* Tech. rept. 86/7. University of Melbourne, Dep. of Computer Science.

Balbin, I. and K. Ramamohanarao. (1987). "A Generalization of the Differential Approach to Recursive Query Evaluation". *J. Log. Program.* 4(3): 259–262. DOI: 10.1016/0743-1066(87)90004-5. URL: https://doi.org/10.1016/0743-1066(87)90004-5.

Bancilhon, F. (1985). "Naive Evaluation of Recursively Defined Relations". In: *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies, Book resulting from the Islamorada Workshop 1985 (Islamorada, FL, USA)*. Ed. by M. L. Brodie and J. Mylopoulos. *Topics in Information Systems*. Springer. 165–178.

Bancilhon, F., D. Maier, Y. Sagiv, and J. D. Ullman. (1986). "Magic Sets and Other Strange Ways to Implement Logic Programs". In: *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24-26, 1986, Cambridge, Massachusetts, USA*. Ed. by A. Silberschatz. ACM. 1–15. DOI: 10.1145/6012.15399. URL: https://doi.org/10.1145/6012.15399.

Datalog in Academia and Industry - Second International Workshop, Datalog 2.0, Vienna, Austria, September 11-13, 2012. Proceedings. (2012). Ed. by P. Barceló and R. Pichler. Vol. 7494. *Lecture Notes in Computer Science*. Springer. ISBN: 978-3-642-32924-1. DOI: 10.1007/978-3-642-32925-8. URL: https://doi.org/10.1007/978-3-642-32925-8.

Beame, P., P. Koutris, and D. Suciu. (2017). "Communication Steps for Parallel Query Processing". *J. ACM*. 64(6): 40:1–40:58. DOI: 10.1145/3125644. URL: https://doi.org/10.1145/3125644.

Beeri, C. and R. Ramakrishnan. (1991). "On the Power of Magic". *J. Log. Program.* DOI: 10.1016/0743-1066(91)90038-Q. URL: https://doi.org/10.1016/0743-1066(91)90038-Q.

Beeri, C. and M. Y. Vardi. (1984). "A Proof Procedure for Data Dependencies". *J. ACM*. 31(4): 718–741. DOI: 10.1145/1634.1636. URL: https://doi.org/10.1145/1634.1636.

Bellomarini, L., D. Benedetto, G. Gottlob, and E. Sallinger. (2022). "Vadalog: A modern architecture for automated reasoning with large knowledge graphs". *Inf. Syst.* 105: 101528. DOI: 10.1016/j.is.2020.101528. URL: https://doi.org/10.1016/j.is.2020.101528.

Bellomarini, L., E. Sallinger, and G. Gottlob. (2018). "The Vadalog System: Datalog-based Reasoning for Knowledge Graphs". *Proc. VLDB Endow.* 11(9): 975–987. DOI: 10.14778/3213880.3213888. URL: http://www.vldb.org/pvldb/vol11/p975-bellomarini.pdf.

Berndl, M., O. Lhoták, F. Qian, L. J. Hendren, and N. Umanee. (2003). "Points-to analysis using BDDs". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003.* Ed. by R. Cytron and R. Gupta. ACM. 103–114. DOI: 10.1145/781131.781144. URL: https://doi.org/10.1145/781131.781144.

Bidoit, N. (1991). "Negation in Rule-Based Database Languages: A Survey". *Theor. Comput. Sci.* 78(1): 3–83. DOI: 10.1016/0304-3975(51)90003-5. URL: https://doi.org/10.1016/0304-3975(51)90003-5.

Bravenboer, M. and Y. Smaragdakis. (2009). "Strictly declarative specification of sophisticated points-to analyses". In: *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*. Ed. by S. Arora and G. T. Leavens. ACM. 243–262. DOI: 10.1145/1640089.1640108. URL: https://doi.org/10.1145/1640089.1640108.

Burch, J. R., E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. (1994). "Symbolic model checking for sequential circuit verification". *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 13(4): 401–424. DOI: 10.1109/43.275352. URL: https://doi.org/10.1109/43.275352.

Calì, A., G. Gottlob, and M. Kifer. (2013). "Taming the Infinite Chase: Query Answering under Expressive Relational Constraints". *J. Artif. Intell. Res.* 48: 115–174. DOI: 10.1613/jair.3873. URL: https://doi.org/10.1613/jair.3873.

Calì, A., G. Gottlob, and T. Lukasiewicz. (2012a). "A general Datalog-based framework for tractable query answering over ontologies". *J. Web Semant.* 14: 57–83. DOI: 10.1016/j.websem.2012.03.001. URL: https://doi.org/10.1016/j.websem.2012.03.001.

Calì, A., G. Gottlob, T. Lukasiewicz, B. Marnette, and A. Pieris. (2010). "Datalog+/-: A Family of Logical Knowledge Representation and Query Languages for New Applications". In: *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*. IEEE Computer Society. 228–242. DOI: 10.1109/LICS.2010.27. URL: https://doi.org/10.1109/LICS.2010.27.

Calì, A., G. Gottlob, and A. Pieris. (2012b). "Towards more expressive ontology languages: The query answering problem". *Artif. Intell.* 193: 87–128. DOI: 10.1016/j.artint.2012.08.002. URL: https://doi.org/10.1016/j.artint.2012.08.002.

Carré, B. (1979). *Graphs and networks*. The Clarendon Press, Oxford University Press, New York. xvi+277. ISBN: 0-19-859622-7.

Ceri, S., G. Gottlob, and L. Tanca. (1989). "What you Always Wanted to Know About Datalog (And Never Dared to Ask)". *IEEE Trans. Knowl. Data Eng.* 1(1): 146–166. DOI: 10.1109/69.43410. URL: https://doi.org/10.1109/69.43410.

Ceri, S. and J. Widom. (1991). "Deriving Production Rules for Incremental View Maintenance". In: *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*. Ed. by G. M. Lohman, A. Sernadas, and R. Camps. Morgan Kaufmann. 577–589. URL: http://www.vldb.org/conf/1991/P577.PDF.

Chan, T. M., V. V. Williams, and Y. Xu. (2021). "Algorithms, Reductions and Equivalences for Small Weight Variants of All-Pairs Shortest Paths". In: *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*. Ed. by N. Bansal, E. Merelli, and J. Worrell. Vol. 198. *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. 47:1–47:21. DOI: 10.4230/LIPIcs.ICALP.2021.47. URL: https://doi.org/10.4230/LIPIcs.ICALP.2021.47.

Chan, T. M., V. V. Williams, and Y. Xu. (2022). "Hardness for triangle problems under even more believable hypotheses: reductions from real APSP, real 3SUM, and OV". In: *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*. Ed. by S. Leonardi and A. Gupta. ACM. 1501–1514. DOI: 10.1145/3519935.3520032. URL: https://doi.org/10.1145/3519935.3520032.

Chandra, A. K. and P. M. Merlin. (1977). "Optimal Implementation of Conjunctive Queries in Relational Data Bases". In: *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*. Ed. by J. E. Hopcroft, E. P. Friedman, and M. A. Harrison. ACM. 77–90. DOI: 10.1145/800105.803397. URL: https://doi.org/10.1145/800105.803397.

Chin, B., D. von Dincklage, V. Ercegovac, P. Hawkins, M. S. Miller, F. J. Och, C. Olston, and F. Pereira. (2015). "Yedalog: Exploring Knowledge at Scale". In: *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*. Ed. by T. Ball, R. Bodík, S. Krishnamurthi, B. S. Lerner, and G. Morrisett. Vol. 32. *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. 63–78. DOI: 10.4230/LIPIcs.SNAPL.2015.63. URL: https://doi.org/10.4230/LIPIcs.SNAPL.2015.63.

Cholak, P. and H. A. Blair. (1994). "The Complexity of Local Stratification". *Fundam. Informaticae*. 21(4): 333–344. DOI: 10.3233/FI-1994-2144. URL: https://doi.org/10.3233/FI-1994-2144.

Chu, D., L. Popa, A. Tavakoli, J. M. Hellerstein, P. A. Levis, S. Shenker, and I. Stoica. (2007). "The design and implementation of a declarative sensor network system". In: *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems, SenSys 2007, Sydney, NSW, Australia, November 6-9, 2007*. Ed. by S. Jha. ACM. 175–188. DOI: 10.1145/1322263.1322281. URL: https://doi.org/10.1145/1322263.1322281.

Condie, T., D. Chu, J. M. Hellerstein, and P. Maniatis. (2008). "Evita raced: metacompilation for declarative networks". *Proc. VLDB Endow.* 1(1): 1153–1165. DOI: 10.14778/1453856.1453978. URL: http://www.vldb.org/pvldb/vol1/1453978.pdf.

Condie, T., A. Das, M. Interlandi, A. Shkapsky, M. Yang, and C. Zaniolo. (2018). "Scaling-up reasoning and advanced analytics on BigData". *Theory Pract. Log. Program.* 18(5-6): 806–845. DOI: 10.1017/S1471068418000418. URL: https://doi.org/10.1017/S1471068418000418.

Cooper, G. F. (1990). "The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks". *Artif. Intell.* 42(2-3): 393–405. DOI: 10.1016/0004-3702(90)90060-D. URL: https://doi.org/10.1016/0004-3702(90)90060-D.

Cousot, P. and R. Cousot. (1977). "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Ed. by R. M. Graham, M. A. Harrison, and R. Sethi. ACM. 238–252. DOI: 10.1145/512950.512973. URL: https://doi.org/10.1145/512950.512973.

Cousot, P. and R. Cousot. (1992). "Comparing the Galois connection and widening/narrowing approaches to abstract interpretation". In: *Programming language implementation and logic programming (Leuven, 1992)*. Vol. 631. *Lecture Notes in Comput. Sci.* Springer, Berlin. 269–295. DOI: 10.1007/3-540-55844-6_142. URL: https://doi-org.gate.lib.buffalo.edu/10.1007/3-540-55844-6_142.

Dannert, K. M., E. Grädel, M. Naaf, and V. Tannen. (2021). "Semiring Provenance for Fixed-Point Logic". In: *29th EACSL Annual Conference on Computer Science Logic, CSL 2021, January 25-28, 2021, Ljubljana, Slovenia (Virtual Conference)*. Ed. by C. Baier and J. Goubault-Larrecq. Vol. 183. *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. 17:1–17:22. DOI: 10.4230/LIPIcs.CSL.2021.17. URL: https://doi.org/10.4230/LIPIcs.CSL.2021.17.

Davey, B. A. and H. A. Priestley. (2002). *Introduction to lattices and order*. Second. Cambridge University Press, New York. xii+298. ISBN: 0-521-78451-4. DOI: 10.1017/CBO9780511809088. URL: https://doi.org/10.1017/CBO9780511809088.

Davey, B. A. and H. A. Priestley. (1990). *Introduction to lattices and order*. Cambridge: Cambridge University Press. ISBN: 9780521367660. URL: http://www.worldcat.org/search?qt=worldcat_org_all&q=0521367662.

Dawson, S., C. R. Ramakrishnan, and D. S. Warren. (1996). "Practical Program Analysis Using General Purpose Logic Programming Systems - A Case Study". In: *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996*. Ed. by C. N. Fischer. ACM. 117–126. DOI: 10.1145/231379.231399. URL: https://doi.org/10.1145/231379.231399.

Dechter, R. (1997). "Bucket Elimination: a Unifying Framework for Processing Hard and Soft Constraints". *Constraints An Int. J.* 2(1): 51–55. DOI: 10.1023/A:1009796922698. URL: https://doi.org/10.1023/A:1009796922698.

Dechter, R. (1999). "Bucket Elimination: A Unifying Framework for Reasoning". *Artif. Intell.* 113(1-2): 41–85. DOI: 10.1016/S0004-3702(99)00059-4. URL: https://doi.org/10.1016/S0004-3702(99)00059-4.

Denecker, M., V. W. Marek, and M. Truszczynski. (2000). In: *Logic-Based Artificial Intelligence*. Ed. by J. Minker. Kluwer Academic Publishers, Boston and Dordrecht. 127–144.

Denecker, M., V. W. Marek, and M. Truszczynski. (2004). "Ultimate approximation and its application in nonmonotonic knowledge representation systems". *Inf. Comput.* 192(1): 84–121. DOI: 10.1016/j.ic.2004.02.004. URL: https://doi.org/10.1016/j.ic.2004.02.004.

Denecker, M., N. Pelov, and M. Bruynooghe. (2001). "Ultimate Well-Founded and Stable Semantics for Logic Programs with Aggregates". In: *Logic Programming, 17th International Conference, ICLP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*. Ed. by P. Codognet. Vol. 2237. *Lecture Notes in Computer Science*. Springer. 212–226. DOI: 10.1007/3-540-45635-X_22. URL: https://doi.org/10.1007/3-540-45635-X%5C_22.

Deutch, D., T. Milo, S. Roy, and V. Tannen. (2014). "Circuits for Datalog Provenance". In: *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*. Ed. by N. Schweikardt, V. Christophides, and V. Leroy. OpenProceedings.org. 201–212. DOI: 10.5441/002/icdt.2014.22. URL: https://doi.org/10.5441/002/icdt.2014.22.

Eisner, J. and N. W. Filardo. (2010). "Dyna: Extending Datalog for Modern AI". In: *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*. Ed. by O. de Moor, G. Gottlob, T. Furche, and A. J. Sellers. Vol. 6702. *Lecture Notes in Computer Science*. Springer. 181–220. DOI: 10.1007/978-3-642-24206-9_11. URL: https://doi.org/10.1007/978-3-642-24206-9%5C_11.

Eisner, J., E. Goldlust, and N. A. Smith. (2005). "Compiling Comp Ling: Weighted Dynamic Programming and the Dyna Language". In: *HLT/EMNLP 2005, Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference, 6-8 October 2005, Vancouver, British Columbia, Canada*. The Association for Computational Linguistics. 281–290. URL: https://aclanthology.org/H05-1036/.

Emden, M. H. van and R. A. Kowalski. (1976). "The Semantics of Predicate Logic as a Programming Language". *J. ACM*. 23(4): 733–742. DOI: 10.1145/321978.321991. URL: https://doi.org/10.1145/321978.321991.

Esparza, J., P. Ganty, S. Kiefer, and M. Luttenberger. (2011). "Parikh's theorem: A simple and direct automaton construction". *Inf. Process. Lett.* 111(12): 614–619. DOI: 10.1016/j.ipl.2011.03.019. URL: https://doi.org/10.1016/j.ipl.2011.03.019.

Esparza, J., S. Kiefer, and M. Luttenberger. (2010). "Newtonian program analysis". *J. ACM*. 57(6): 33:1–33:47. DOI: 10.1145/1857914.1857917. URL: https://doi.org/10.1145/1857914.1857917.

Etessami, K. and M. Yannakakis. (2009). "Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations". *J. ACM*. 56(1): 1:1–1:66. DOI: 10.1145/1462153.1462154. URL: https://doi.org/10.1145/1462153.1462154.

Faber, W., G. Pfeifer, and N. Leone. (2011). "Semantics and complexity of recursive aggregates in answer set programming". *Artif. Intell.* 175(1): 278–298. DOI: 10.1016/j.artint.2010.04.002. URL: https://doi.org/10.1016/j.artint.2010.04.002.

Fagin, R. (1982). "Horn clauses and database dependencies". *J. ACM*. 29(4): 952–985. DOI: 10.1145/322344.322347. URL: https://doi.org/10.1145/322344.322347.

Ferraris, P. (2011). "Logic programs with propositional connectives and aggregates". *ACM Trans. Comput. Log.* 12(4): 25:1–25:40. DOI: 10.1145/1970398.1970401. URL: https://doi.org/10.1145/1970398.1970401.

Fitting, M. (1985). "A Kripke-Kleene Semantics for Logic Programs". *J. Log. Program.* 2(4): 295–312. DOI: 10.1016/S0743-1066(85)80005-4. URL: https://doi.org/10.1016/S0743-1066(85)80005-4.

Fitting, M. (1991). "Bilattices and the Semantics of Logic Programming". *J. Log. Program.* 11(1&2): 91–116. DOI: 10.1016/0743-1066(91)90014-G. URL: https://doi.org/10.1016/0743-1066(91)90014-G.

Fitting, M. (1993). "The Family of Stable Models". *J. Log. Program.* 17(2/3&4): 197–225. DOI: 10.1016/0743-1066(93)90031-B. URL: https://doi.org/10.1016/0743-1066(93)90031-B.

Fitting, M. (2002). "Fixpoint semantics for logic programming a survey". *Theor. Comput. Sci.* 278(1-2): 25–51. DOI: 10.1016/S0304-3975(00)00330-3. URL: https://doi.org/10.1016/S0304-3975(00)00330-3.

Floyd, R. W. (1962). "Algorithm 97: Shortest path". *Commun. ACM.* 5(6): 345. DOI: 10.1145/367766.368168. URL: https://doi.org/10.1145/367766.368168.

Ganguly, S., S. Greco, and C. Zaniolo. (1991). "Minimum and Maximum Predicates in Logic Programming". In: *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 29-31, 1991, Denver, Colorado, USA*. Ed. by D. J. Rosenkrantz. ACM Press. 154–163. DOI: 10.1145/113413.113427. URL: https://doi.org/10.1145/113413.113427.

Garcia-Molina, H., J. D. Ullman, and J. Widom. (2009). *Database systems - the complete book (2. ed.)* Pearson Education. ISBN: 978-0-13-187325-4.

Gelder, A. V. (1989). "The Alternating Fixpoint of Logic Programs with Negation". In: *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania, USA*. Ed. by A. Silberschatz. ACM Press. 1–10. DOI: 10.1145/73721.73722. URL: https://doi.org/10.1145/73721.73722.

Gelder, A. V. (1992). "The Well-Founded Semantics of Aggregation". In: *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California, USA*. Ed. by M. Y. Vardi and P. C. Kanellakis. ACM Press. 127–138. DOI: 10.1145/137097.137854. URL: https://doi.org/10.1145/137097.137854.

Gelder, A. V., K. A. Ross, and J. S. Schlipf. (1988). "Unfounded Sets and Well-Founded Semantics for General Logic Programs". In: *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 21-23, 1988, Austin, Texas, USA*. Ed. by C. Edmondson-Yurkanan and M. Yannakakis. ACM. 221–230. DOI: 10.1145/308386.308444. URL: https://doi.org/10.1145/308386.308444.

Gelder, A. V., K. A. Ross, and J. S. Schlipf. (1991). "The Well-Founded Semantics for General Logic Programs". *J. ACM.* 38(3): 620–650. DOI: 10.1145/116825.116838. URL: https://doi.org/10.1145/116825.116838.

Gelfond, M. and V. Lifschitz. (1988). "The Stable Model Semantics for Logic Programming". In: *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*. Ed. by R. A. Kowalski and K. A. Bowen. MIT Press. 1070–1080.

Gelfond, M. and Y. Zhang. (2014). "Vicious Circle Principle and Logic Programs with Aggregates". *Theory Pract. Log. Program.* 14(4-5): 587–601. DOI: 10.1017/S1471068414000222. URL: https://doi.org/10.1017/S1471068414000222.

Gelfond, M. and Y. Zhang. (2019). "Vicious circle principle, aggregates, and formation of sets in ASP based languages". *Artif. Intell.* 275: 28–77. DOI: 10.1016/j.artint.2019.04.004. URL: https://doi.org/10.1016/j.artint.2019.04.004.

Giannotti, F., S. Greco, D. Saccà, and C. Zaniolo. (1997). "Programming with Non-Determinism in Deductive Databases". *Ann. Math. Artif. Intell.* 19(1-2): 97–125. DOI: 10.1023/A:1018999404360. URL: https://doi.org/10.1023/A:1018999404360.

Giannotti, F. and D. Pedreschi. (1998). "Datalog with Non-Deterministic Choice Computers NDB-PTIME". *J. Log. Program.* 35(1): 79–101. DOI: 10.1016/S0743-1066(97)10004-8. URL: https://doi.org/10.1016/S0743-1066(97)10004-8.

Giannotti, F., D. Pedreschi, and C. Zaniolo. (2001). "Semantics and Expressive Power of Nondeterministic Constructs in Deductive Databases". *J. Comput. Syst. Sci.* 62(1): 15–42. DOI: 10.1006/jcss.1999.1699. URL: https://doi.org/10.1006/jcss.1999.1699.

Goldstein, J. and P. Larson. (2001). "Optimizing Queries Using Materialized Views: A practical, scalable solution". In: *SIGMOD*. DOI: 10.1145/375663.375706. URL: https://doi.org/10.1145/375663.375706.

Goldstine, J. (1977). "A simplified proof of Parikh's theorem". *Discrete Math.* 19(3): 235–239 (1978). ISSN: 0012-365X. DOI: 10.1016/0012-365X(77)90103-0. URL: https://doi-org.gate.lib.buffalo.edu/10.1016/0012-365X(77)90103-0.

Gondran, M. (1975). "Algèbre linéaire et cheminement dans un graphe". *Rev. Française Automat. Informat. Recherche Opérationnelle Sér. Verte*. 9(V-1): 77–99. ISSN: 0376-2165.

Gondran, M. and M. Minoux. (2008). *Graphs, dioids and semirings*. Vol. 41. *Operations Research/Computer Science Interfaces Series*. Springer, New York. xx+383. ISBN: 978-0-387-75449-9.

Gottlob, G., G. Greco, N. Leone, and F. Scarcello. (2016). "Hypertree Decompositions: Questions and Answers". In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by T. Milo and W. Tan. ACM. 57–74. DOI: 10.1145/2902251.2902309. URL: https://doi.org/10.1145/2902251.2902309.

Gottlob, G., S. T. Lee, and G. Valiant. (2009). "Size and treewidth bounds for conjunctive queries". In: *Proceedings of the Twenty-Eigth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2009, June 19 - July 1, 2009, Providence, Rhode Island, USA*. Ed. by J. Paredaens and J. Su. ACM. 45–54. DOI: 10.1145/1559795.1559804. URL: https://doi.org/10.1145/1559795.1559804.

Gottlob, G. and A. Pieris. (2015). "Beyond SPARQL under OWL 2 QL Entailment Regime: Rules to the Rescue". In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. Ed. by Q. Yang and M. J. Wooldridge. AAAI Press. 2999–3007. URL: http://ijcai.org/Abstract/15/424.

Greco, S., D. Saccà, and C. Zaniolo. (1995). "DATALOG Queries with Stratified Negation and Choice: from P to D^P". In: *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11-13, 1995, Proceedings*. Ed. by G. Gottlob and M. Y. Vardi. Vol. 893. *Lecture Notes in Computer Science*. Springer. 82–96. DOI: 10.1007/3-540-58907-4_8. URL: https://doi.org/10.1007/3-540-58907-4%5C_8.

Greco, S. and C. Zaniolo. (1998). "Greedy Algorithms in Datalog with Choice and Negation". In: *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming, Manchester, UK, June 15-19, 1998*. Ed. by J. Jaffar. MIT Press. 294–309.

Greco, S., C. Zaniolo, and S. Ganguly. (1992). "Greedy by Choice". In: *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California, USA*. Ed. by M. Y. Vardi and P. C. Kanellakis. ACM Press. 105–113. DOI: 10.1145/137097.137836. URL: https://doi.org/10.1145/137097.137836.

Green, T. J. (2015). "LogiQL: A Declarative Language for Enterprise Applications". In: *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by T. Milo and D. Calvanese. ACM. 59–64. DOI: 10.1145/2745754.2745780. URL: https://doi.org/10.1145/2745754.2745780.

Green, T. J., S. S. Huang, B. T. Loo, and W. Zhou. (2013). "Datalog and Recursive Query Processing". *Found. Trends Databases*. 5(2): 105–195. DOI: 10.1561/1900000017. URL: https://doi.org/10.1561/1900000017.

Green, T. J., G. Karvounarakis, and V. Tannen. (2007). "Provenance semirings". In: *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*. Ed. by L. Libkin. ACM. 31–40. DOI: 10.1145/1265530.1265535. URL: https://doi.org/10.1145/1265530.1265535.

Green, T. J., D. Olteanu, and G. Washburn. (2015). "Live Programming in the LogicBlox System: A MetaLogiQL Approach". *Proc. VLDB Endow.* 8(12): 1782–1791. DOI: 10.14778/2824032.2824075. URL: http://www.vldb.org/pvldb/vol8/p1782-green.pdf.

Grohe, M. and D. Marx. (2014). "Constraint Solving via Fractional Edge Covers". *ACM Trans. Algorithms.* 11(1): 4:1–4:20. DOI: 10.1145/2636918. URL: https://doi.org/10.1145/2636918.

Gu, Y., A. Polak, V. V. Williams, and Y. Xu. (2021). "Faster Monotone Min-Plus Product, Range Mode, and Single Source Replacement Paths". In: *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*. Ed. by N. Bansal, E. Merelli, and J. Worrell. Vol. 198. *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. 75:1–75:20. DOI: 10.4230/LIPIcs.ICALP.2021.75. URL: https://doi.org/10.4230/LIPIcs.ICALP.2021.75.

Gunawardena, J. (1998). "An introduction to idempotency". In: *Idempotency (Bristol, 1994)*. Vol. 11. *Publ. Newton Inst.* Cambridge Univ. Press, Cambridge. 1–49. DOI: 10.1017/CBO9780511662508.003. URL: https://doi-org.gate.lib.buffalo.edu/10.1017/CBO9780511662508.003.

Gupta, A., I. S. Mumick, and V. S. Subrahmanian. (1993). "Maintaining Views Incrementally". In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*. Ed. by P. Buneman and S. Jajodia. ACM Press. 157–166. DOI: 10.1145/170035.170066. URL: https://doi.org/10.1145/170035.170066.

Hajiyev, E., M. Verbaere, and O. de Moor. (2006). "*codeQuest:* Scalable Source Code Queries with Datalog". In: *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*. Ed. by D. Thomas. Vol. 4067. *Lecture Notes in Computer Science*. Springer. 2–27. DOI: 10.1007/11785477_2. URL: https://doi.org/10.1007/11785477%5C_2.

Halevy, A. Y. (2001). "Answering queries using views: A survey". *VLDB J.* 10(4): 270–294. DOI: 10.1007/s007780100054. URL: https://doi.org/10.1007/s007780100054.

Halpin, T. and S. Rugaber. (2014). *LogiQL: A Query Language for Smart Databases*. 1st. USA: CRC Press, Inc. ISBN: 1482244934.

Hellerstein, J. M. (2010a). "Datalog redux: experience and conjecture". In: *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*. Ed. by J. Paredaens and D. V. Gucht. ACM. 1–2. DOI: 10.1145/1807085.1807087. URL: https://doi.org/10.1145/1807085.1807087.

Hellerstein, J. M. (2010b). "The declarative imperative: experiences and conjectures in distributed logic". *SIGMOD Rec.* 39(1): 5–19. DOI: 10.1145/1860702.1860704. URL: https://doi.org/10.1145/1860702.1860704.

Hellerstein, J. M. and P. Alvaro. (2020). "Keeping CALM: when distributed consistency is easy". *Commun. ACM*. 63(9): 72–81. DOI: 10.1145/3369736. URL: https://doi.org/10.1145/3369736.

Hitzler, P. (2001). "Generalized Metrics and Topology in Logic Programming Semantics". *PhD thesis*. Ireland: Department of Mathematics, National University of Ireland, University College Cork.

Hitzler, P. and M. Wendt. (2005). "A uniform approach to logic programming semantics". *Theory Pract. Log. Program.* 5(1-2): 93–121. DOI: 10.1017/S1471068404002212. URL: https://doi.org/10.1017/S1471068404002212.

Hopkins, M. W. and D. Kozen. (1999). "Parikh's Theorem in Commutative Kleene Algebra". In: *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. IEEE Computer Society. 394–401. DOI: 10.1109/LICS.1999.782634. URL: https://doi.org/10.1109/LICS.1999.782634.

Huang, S. S., T. J. Green, and B. T. Loo. (2011). "Datalog and emerging applications: an interactive tutorial". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*. Ed. by T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis. ACM. 1213–1216. DOI: 10.1145/1989323.1989456. URL: https://doi.org/10.1145/1989323.1989456.

Jordan, H., B. Scholz, and P. Subotic. (2016). "Soufflé: On Synthesis of Program Analyzers". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Ed. by S. Chaudhuri and A. Farzan. Vol. 9780. *Lecture Notes in Computer Science*. Springer. 422–430. DOI: 10.1007/978-3-319-41540-6_23. URL: https://doi.org/10.1007/978-3-319-41540-6%5C_23.

Kam, J. B. and J. D. Ullman. (1976). "Global Data Flow Analysis and Iterative Algorithms". *J. ACM*. 23(1): 158–171. DOI: 10.1145/321921.321938. URL: https://doi.org/10.1145/321921.321938.

Kemp, D. B. and P. J. Stuckey. (1991). "Semantics of Logic Programs with Aggregates". In: *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct. 28 - Nov 1, 1991*. Ed. by V. A. Saraswat and K. Ueda. MIT Press. 387–401.

Ketsman, B. and C. Koch. (2020). "Datalog with Negation and Monotonicity". In: *23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark*. Ed. by C. Lutz and J. C. Jung. Vol. 155. *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. 19:1–19:18. DOI: 10.4230/LIPIcs.ICDT.2020.19. URL: https://doi.org/10.4230/LIPIcs.ICDT.2020.19.

Ketsman, B. and P. Koutris. (2022). "Modern Datalog Engines". *Found. Trends Databases*. 12(1): 1–68. DOI: 10.1561/1900000073. URL: https://doi.org/10.1561/1900000073.

Khamis, M. A., H. Q. Ngo, R. Pichler, D. Suciu, and Y. R. Wang. (2021). "Convergence of Datalog over (Pre-) Semirings". *CoRR*. abs/2105.14435. arXiv: 2105.14435. URL: https://arxiv.org/abs/2105.14435.

Khamis, M. A., H. Q. Ngo, R. Pichler, D. Suciu, and Y. R. Wang. (2022a). "Convergence of Datalog over (Pre-) Semirings". In: *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by L. Libkin and P. Barceló. ACM. 105–117. DOI: 10.1145/3517804.3524140. URL: https://doi.org/10.1145/3517804.3524140.

Khamis, M. A., H. Q. Ngo, R. Pichler, D. Suciu, and Y. R. Wang. (2022b). "Datalog in Wonderland". *SIGMOD Rec.* 51(2): 6–17. DOI: 10.1145/3552490.3552492. URL: https://doi.org/10.1145/3552490.3552492.

Khamis, M. A., H. Q. Ngo, and A. Rudra. (2016). "FAQ: Questions Asked Frequently". In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by T. Milo and W.-C. Tan. ACM. 13–28. DOI: 10.1145/2902251.2902280. URL: https://doi.org/10.1145/2902251.2902280.

Kincaid, Z., T. W. Reps, and J. Cyphert. (2021). "Algebraic Program Analysis". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*. Ed. by A. Silva and K. R. M. Leino. Vol. 12759. *Lecture Notes in Computer Science*. Springer. 46–83. DOI: 10.1007/978-3-030-81685-8_3. URL: https://doi.org/10.1007/978-3-030-81685-8%5C_3.

Kleene, S. C. (1956). "Representation of events in nerve nets and finite automata". In: *Automata studies. Annals of mathematics studies, no. 34*. Princeton University Press, Princeton, N. J. 3–41.

Kleene, S. C. (1952). *Introduction to Metamathematics*. North Holland , Princeton, NJ.

Koch, C., Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. (2014). "DBToaster: higher-order delta processing for dynamic, frequently fresh views". *VLDB J.* 23(2): 253–278. DOI: 10.1007/s00778-013-0348-4. URL: https://doi.org/10.1007/s00778-013-0348-4.

Kohlas, J. and P. P. Shenoy. (2000). "Computation in valuation algebras". In: *Algorithms for uncertainty and defeasible reasoning*. Vol. 5. *Handb. Defeasible Reason. Uncertain. Manag. Syst.* Kluwer Acad. Publ., Dordrecht. 5–39. ISBN: 0-7923-6672-7.

Kohlas, J. and N. Wilson. (2008). "Semiring induced valuation algebras: Exact and approximate local computation algorithms". *Artif. Intell.* 172(11): 1360–1399. DOI: 10.1016/j.artint.2008.03.003. URL: https://doi.org/10.1016/j.artint.2008.03.003.

Kolaitis, P. G. (1991). "The Expressive Power of Stratified Programs". *Inf. Comput.* 90(1): 50–66. DOI: 10.1016/0890-5401(91)90059-B. URL: https://doi.org/10.1016/0890-5401(91)90059-B.

Koller, D. and N. Friedman. (2009). *Probabilistic graphical models. Adaptive Computation and Machine Learning*. MIT Press, Cambridge, MA. xxxvi+1231. ISBN: 978-0-262-01319-2.

Kozen, D. (1990). "On Kleene Algebras and Closed Semirings". In: *Mathematical Foundations of Computer Science 1990, MFCS'90, Banská Bystrica, Czechoslovakia, August 27-31, 1990, Proceedings*. Ed. by B. Rovan. Vol. 452. *Lecture Notes in Computer Science*. Springer. 26–47. DOI: 10.1007/BFb0029594. URL: https://doi.org/10.1007/BFb0029594.

Kozen, D. (1994). "A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events". *Inf. Comput.* 110(2): 366–390. DOI: 10.1006/inco.1994.1037. URL: https://doi.org/10.1006/inco.1994.1037.

Kripke, S. (1976). "Outline of a Theory of Truth". *The Journal of Philosophy*. 72(19): 690–716.

Krishnamurthy, R. and S. A. Naqvi. (1988). "Non-Deterministic Choice in Datalog". In: *Proceedings of the Third International Conference on Data and Knowledge Bases: Improving Usability and Responsiveness, June 28-30, 1988, Jerusalem, Israel*. Ed. by C. Beeri, J. W. Schmidt, and U. Dayal. Morgan Kaufmann. 416–424. DOI: 10.1016/b978-1-4832-1313-2.50038-x. URL: https://doi.org/10.1016/b978-1-4832-1313-2.50038-x.

Kuich, W. (1987). "The Kleene and the Parikh theorem in complete semirings". In: *Automata, languages and programming (Karlsruhe, 1987)*. Vol. 267. *Lecture Notes in Comput. Sci.* Springer, Berlin. 212–225. DOI: 10.1007/3-540-18088-5_17. URL: https://doi-org.gate.lib.buffalo.edu/10.1007/3-540-18088-5_17.

Kuich, W. (1997). "Semirings and formal power series: their relevance to formal languages and automata". In: *Handbook of formal languages, Vol. 1*. Springer, Berlin. 609–677.

Lam, M. S., J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. (2005). "Context-sensitive program analysis as database queries". In: *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, Maryland, USA*. Ed. by C. Li. ACM. 1–12. DOI: 10.1145/1065167.1065169. URL: https://doi.org/10.1145/1065167.1065169.

Lauritzen, S. L. and D. J. Spiegelhalter. (1988). "Local computations with probabilities on graphical structures and their application to expert systems". *Journal of the Royal Statistical Society: Series B (Methodological)*. 50(2): 157–194.

Leeuwen, J. van. (1974). "A generalisation of Parikh's theorem in formal language theory". In: *Automata, languages and programming (Second Colloq., Univ. Saarbrücken, Saarbrücken, 1974). Lecture Notes in Comput. Sci., Vol. 14*. Springer, Berlin. 17–26.

Lehmann, D. J. (1977). "Algebraic Structures for Transitive Closure". *Theor. Comput. Sci.* 4(1): 59–76. DOI: 10.1016/0304-3975(77)90056-1. URL: https://doi.org/10.1016/0304-3975(77)90056-1.

Leone, N., M. Manna, G. Terracina, and P. Veltri. (2012). "Efficiently Computable Datalog∃ Programs". In: *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012*. Ed. by G. Brewka, T. Eiter, and S. A. McIlraith. AAAI Press. URL: http://www.aaai.org/ocs/index.php/KR/KR12/paper/view/4521.

Lhoták, O. and L. J. Hendren. (2004). "Jedd: a BDD-based relational extension of Java". In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*. Ed. by W. W. Pugh and C. Chambers. ACM. 158–169. DOI: 10.1145/996841.996861. URL: https://doi.org/10.1145/996841.996861.

Lhoták, O. and L. J. Hendren. (2008). "Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation". *ACM Trans. Softw. Eng. Methodol.* 18(1): 3:1–3:53. DOI: 10.1145/1391984.1391987. URL: https://doi.org/10.1145/1391984.1391987.

Lifschitz, V. (2008). "Twelve Definitions of a Stable Model". In: *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*. Ed. by M. G. de la Banda and E. Pontelli. Vol. 5366. *Lecture Notes in Computer Science*. Springer. 37–51. DOI: 10.1007/978-3-540-89982-2_8. URL: https://doi.org/10.1007/978-3-540-89982-2%5C_8.

Lipton, R. J., D. J. Rose, and R. E. Tarjan. (1979). "Generalized nested dissection". *SIAM J. Numer. Anal.* 16(2): 346–358. ISSN: 0036-1429. DOI: 10.1137/0716027. URL: https://doi-org.gate.lib.buffalo.edu/10.1137/0716027.

Lipton, R. J. and R. E. Tarjan. (1980). "Applications of a planar separator theorem". *SIAM J. Comput.* 9(3): 615–627. ISSN: 0097-5397. DOI: 10.1137/0209046. URL: https://doi-org.gate.lib.buffalo.edu/10.1137/0209046.

Liu, L., E. Pontelli, T. C. Son, and M. Truszczynski. (2010). "Logic programs with abstract constraint atoms: The role of computations". *Artif. Intell.* 174(3-4): 295–315. DOI: 10.1016/j.artint.2009.11.016. URL: https://doi.org/10.1016/j.artint.2009.11.016.

Liu, Y. A. and S. D. Stoller. (2020). "Founded semantics and constraint semantics of logic rules". *J. Log. Comput.* 30(8): 1609–1668. DOI: 10.1093/logcom/exaa056. URL: https://doi.org/10.1093/logcom/exaa056.

Liu, Y. A. and S. D. Stoller. (2022). "Recursive rules with aggregation: a simple unified semantics". *J. Log. Comput.* 32(8): 1659–1693. DOI: 10.1093/logcom/exac072. URL: https://doi.org/10.1093/logcom/exac072.

Loo, B. T., T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. (2009). "Declarative networking". *Commun. ACM*. 52(11): 87–95. DOI: 10.1145/1592761.1592785. URL: https://doi.org/10.1145/1592761.1592785.

Loo, B. T., T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. (2005). "Implementing declarative overlays". In: *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*. Ed. by A. Herbert and K. P. Birman. ACM. 75–90. DOI: 10.1145/1095810.1095818. URL: https://doi.org/10.1145/1095810.1095818.

Loomis, L. H. and H. Whitney. (1949). "An inequality related to the isoperimetric inequality". *Bull. Amer. Math. Soc.* 55: 961–962. ISSN: 0002-9904.

Luttenberger, M. and M. Schlund. (2013). "Convergence of Newton's Method over Commutative Semirings". In: *Language and Automata Theory and Applications - 7th International Conference, LATA 2013, Bilbao, Spain, April 2-5, 2013. Proceedings*. Ed. by A.-H. Dediu, C. Martín-Vide, and B. Truthe. Vol. 7810. *Lecture Notes in Computer Science*. Springer. 407–418. DOI: 10.1007/978-3-642-37064-9_36. URL: https://doi.org/10.1007/978-3-642-37064-9%5C_36.

Madsen, M. and O. Lhoták. (2018). "Safe and sound program analysis with Flix". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. Ed. by F. Tip and E. Bodden. ACM. 38–48. DOI: 10.1145/3213846.3213847. URL: https://doi.org/10.1145/3213846.3213847.

Madsen, M. and O. Lhoták. (2020). "Fixpoints for the masses: programming with first-class Datalog constraints". *Proc. ACM Program. Lang.* 4(OOPSLA): 125:1–125:28. DOI: 10.1145/3428193. URL: https://doi.org/10.1145/3428193.

Madsen, M., M.-H. Yee, and O. Lhoták. (2016). "From Datalog to flix: a declarative language for fixed points on lattices". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. Ed. by C. Krintz and E. Berger. ACM. 194–208. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908096. URL: https://doi.org/10.1145/2908080.2908096.

Maier, D., A. O. Mendelzon, and Y. Sagiv. (1979). "Testing Implications of Data Dependencies". *ACM Trans. Database Syst.* 4(4): 455–469. DOI: 10.1145/320107.320115. URL: https://doi.org/10.1145/320107.320115.

Maier, D., Y. Sagiv, and M. Yannakakis. (1981). "On the Complexity of Testing Implications of Functional and Join Dependencies". *J. ACM.* 28(4): 680–695. DOI: 10.1145/322276.322280. URL: https://doi.org/10.1145/322276.322280.

Marczak, W. R., P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. (2012). "Confluence Analysis for Distributed Programs: A Model-Theoretic Approach". In: *Datalog in Academia and Industry - Second International Workshop, Datalog 2.0, Vienna, Austria, September 11-13, 2012. Proceedings.* Ed. by P. Barceló and R. Pichler. Vol. 7494. *Lecture Notes in Computer Science.* Springer. 135–147. DOI: 10.1007/978-3-642-32925-8_14. URL: https://doi.org/10.1007/978-3-642-32925-8%5C_14.

Marek, V. W. and J. B. Remmel. (2004). "Set Constraints in Logic Programming". In: *Logic Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, FL, USA, January 6-8, 2004, Proceedings.* Ed. by V. Lifschitz and I. Niemelä. Vol. 2923. *Lecture Notes in Computer Science.* Springer. 167–179. DOI: 10.1007/978-3-540-24609-1_16. URL: https://doi.org/10.1007/978-3-540-24609-1%5C_16.

Marek, V. W. and M. Truszczynski. (1991). "Autoepistemic Logic". *J. ACM.* 38(3): 588–619. DOI: 10.1145/116825.116836. URL: https://doi.org/10.1145/116825.116836.

Mascellani, P. and D. Pedreschi. (2002). "The Declarative Side of Magic". In: *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II.* Ed. by A. C. Kakas and F. Sadri. Vol. 2408. *Lecture Notes in Computer Science.* Springer. 83–108. DOI: 10.1007/3-540-45632-5_4. URL: https://doi.org/10.1007/3-540-45632-5%5C_4.

Master, J. (2021). "The open algebraic path problem". In: *9th Conference on Algebra and Coalgebra in Computer Science.* Vol. 211. *LIPIcs. Leibniz Int. Proc. Inform.* Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern. Art. No. 20, 20.

Mazuran, M., E. Serra, and C. Zaniolo. (2013). "Extending the power of datalog recursion". *VLDB J.* 22(4): 471–493. DOI: 10.1007/s00778-012-0299-1. URL: https://doi.org/10.1007/s00778-012-0299-1.

McCarthy, J. (1984). "Applications of Circumscription to Formalizing Common Sense Knowledge". In: *Proceedings of the Non-Monotonic Reasoning Workshop, Mohonk Mountain House, New Paltz, NY 12561, USA, October 17-19, 1984*. American Association for Artificial Intelligence (AAAI). 295–324.

McSherry, F., D. G. Murray, R. Isaacs, and M. Isard. (2013). "Differential Dataflow". In: *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org. URL: http://cidrdb.org/cidr2013/Papers/CIDR13%5C_Paper111.pdf.

Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers. (2011). Ed. by O. de Moor, G. Gottlob, T. Furche, and A. J. Sellers. Vol. 6702. *Lecture Notes in Computer Science*. Springer. ISBN: 978-3-642-24205-2. DOI: 10.1007/978-3-642-24206-9. URL: https://doi.org/10.1007/978-3-642-24206-9.

Motik, B., Y. Nenov, R. Piro, and I. Horrocks. (2019). "Maintenance of datalog materialisations revisited". *Artif. Intell.* 269: 76–136. DOI: 10.1016/j.artint.2018.12.004. URL: https://doi.org/10.1016/j.artint.2018.12.004.

Motik, B., Y. Nenov, R. Piro, I. Horrocks, and D. Olteanu. (2014). "Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems". In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*. Ed. by C. E. Brodley and P. Stone. AAAI Press. 129–137. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8505.

Mumick, I. S., H. Pirahesh, and R. Ramakrishnan. (1990). "The Magic of Duplicates and Aggregates". In: *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*. Ed. by D. McLeod, R. Sacks-Davis, and H. Schek. Morgan Kaufmann. 264–277. URL: http://www.vldb.org/conf/1990/P264.PDF.

Mumick, I. S. and O. Shmueli. (1993). “Finiteness Properties of Database Queries”. In: *Advances in Database Research - Proceedings of the 4th Australian Database Conference, ADC ’93, Griffith University, Brisbane, Queensland, Australia, February 1-2, 1993*. Ed. by M. E. Orlowska and M. P. Papazoglou. World Scientific. 274–288.

Nappa, P., D. Zhao, P. Subotic, and B. Scholz. (2019). “Fast Parallel Equivalence Relations in a Datalog Compiler”. In: *28th International Conference on Parallel Architectures and Compilation Techniques, PACT 2019, Seattle, WA, USA, September 23-26, 2019*. IEEE. 82–96. DOI: 10.1109/PACT.2019.00015. URL: https://doi.org/10.1109/PACT.2019.00015.

Ngo, H. Q. (2018). “Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems”. In: *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*. Ed. by J. V. den Bussche and M. Arenas. ACM. 111–124. DOI: 10.1145/3196959.3196990. URL: https://doi.org/10.1145/3196959.3196990.

Ngo, H. Q., E. Porat, C. Ré, and A. Rudra. (2012). “Worst-case optimal join algorithms: [extended abstract]”. In: *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*. Ed. by M. Benedikt, M. Krötzsch, and M. Lenzerini. ACM. 37–48. DOI: 10.1145/2213556.2213565. URL: https://doi.org/10.1145/2213556.2213565.

Ngo, H. Q., C. Ré, and A. Rudra. (2013). “Skew strikes back: new developments in the theory of join algorithms”. *SIGMOD Rec.* 42(4): 5–16. DOI: 10.1145/2590989.2590991. URL: https://doi.org/10.1145/2590989.2590991.

Nguyen, D. T., M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra. (2015). “Join Processing for Graph Patterns: An Old Dog with New Tricks”. In: *Proceedings of the Third International Workshop on Graph Data Management Experiences and Systems, GRADES 2015, Melbourne, VIC, Australia, May 31 - June 4, 2015*. Ed. by J. L. Larriba-Pey and T. L. Willke. ACM. 2:1–2:8. DOI: 10.1145/2764947.2764948. URL: https://doi.org/10.1145/2764947.2764948.

Nielson, F., H. R. Nielson, and C. Hankin. (1999). *Principles of program analysis*. Springer-Verlag, Berlin. xxii+450. ISBN: 3-540-65410-0. DOI: 10.1007/978-3-662-03811-6. URL: https://doi-org.gate.lib.buffalo.edu/10.1007/978-3-662-03811-6.

Olteanu, D. (2020). "The Relational Data Borg is Learning". *Proc. VLDB Endow.* 13(12): 3502–3515. DOI: 10.14778/3415478.3415572. URL: http://www.vldb.org/pvldb/vol13/p3502-olteanu.pdf.

Olteanu, D. and M. Schleich. (2016). "Factorized Databases". *SIGMOD Rec.* 45(2): 5–16. DOI: 10.1145/3003665.3003667. URL: https://doi.org/10.1145/3003665.3003667.

Olteanu, D. and J. Závodný. (2015). "Size Bounds for Factorised Representations of Query Results". *ACM Trans. Database Syst.* 40(1): 2:1–2:44. DOI: 10.1145/2656335. URL: https://doi.org/10.1145/2656335.

Palopoli, L. (1992). "Testing Logic Programs for Local Stratification". *Theor. Comput. Sci.* 103(2): 205–234. DOI: 10.1016/0304-3975(92)90013-6. URL: https://doi.org/10.1016/0304-3975(92)90013-6.

Parikh, R. J. (1966). "On context-free languages". *J. Assoc. Comput. Mach.* 13: 570–581. ISSN: 0004-5411. DOI: 10.1145/321356.321364. URL: https://doi-org.gate.lib.buffalo.edu/10.1145/321356.321364.

Pearl, J. (1982). "Reverend Bayes on Inference Engines: A Distributed Hierarchical Approach". In: *Proceedings of the National Conference on Artificial Intelligence, Pittsburgh, PA, USA, August 18-20, 1982*. Ed. by D. L. Waltz. AAAI Press. 133–136. URL: http://www.aaai.org/Library/AAAI/1982/aaai82-032.php.

Pearl, J. (1989). *Probabilistic reasoning in intelligent systems - networks of plausible inference. Morgan Kaufmann series in representation and reasoning*. Morgan Kaufmann.

Pelov, N., M. Denecker, and M. Bruynooghe. (2004). "Partial Stable Models for Logic Programs with Aggregates". In: *Logic Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, FL, USA, January 6-8, 2004, Proceedings*. Ed. by V. Lifschitz and I. Niemelä. Vol. 2923. *Lecture Notes in Computer Science*. Springer. 207–219. DOI: 10.1007/978-3-540-24609-1_19. URL: https://doi.org/10.1007/978-3-540-24609-1%5C_19.

Pelov, N., M. Denecker, and M. Bruynooghe. (2007). "Well-founded and stable semantics of logic programs with aggregates". *Theory Pract. Log. Program.* 7(3): 301–353. DOI: 10.1017/S1471068406002973. URL: https://doi.org/10.1017/S1471068406002973.

Pichler, R. and S. Skritek. (2013). "Tractable counting of the answers to conjunctive queries". *J. Comput. Syst. Sci.* 79(6): 984–1001. DOI: 10.1016/J.JCSS.2013.01.012. URL: https://doi.org/10.1016/j.jcss.2013.01.012.

Pilling, D. L. (1973). "Commutative regular equations and Parikh's theorem". *J. London Math. Soc. (2).* 6: 663–666. ISSN: 0024-6107. DOI: 10.1112/jlms/s2-6.4.663. URL: https://doi-org.gate.lib.buffalo.edu/10.1112/jlms/s2-6.4.663.

Pivoteau, C., B. Salvy, and M. Soria. (2012). "Algorithms for combinatorial structures: Well-founded systems and Newton iterations". *J. Comb. Theory, Ser. A.* 119(8): 1711–1773. DOI: 10.1016/j.jcta.2012.05.007. URL: https://doi.org/10.1016/j.jcta.2012.05.007.

Przymusinska, H. and T. C. Przymusinski. (1988). "Weakly Perfect Model Semantics for Logic Programs". In: *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes).* Ed. by R. A. Kowalski and K. A. Bowen. MIT Press. 1106–1120.

Przymusinski, T. C. (1988a). "On the Declarative Semantics of Deductive Databases and Logic Programs". In: *Foundations of Deductive Databases and Logic Programming.* Ed. by J. Minker. Morgan Kaufmann. 193–216. DOI: 10.1016/b978-0-934613-40-8.50009-9. URL: https://doi.org/10.1016/b978-0-934613-40-8.50009-9.

Przymusinski, T. C. (1988b). "Perfect Model Semantics". In: *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes).* Ed. by R. A. Kowalski and K. A. Bowen. MIT Press. 1081–1096.

Przymusinski, T. C. (1989a). "Every Logic Program Has a Natural Stratification And an Iterated Least Fixed Point Model". In: *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania, USA*. Ed. by A. Silberschatz. ACM Press. 11–21. DOI: 10.1145/73721.73723. URL: https://doi.org/10.1145/73721.73723.

Przymusinski, T. C. (1989b). "On the Declarative and Procedural Semantics of Logic Programs". *J. Autom. Reason.* 5(2): 167–205. DOI: 10.1007/BF00243002. URL: https://doi.org/10.1007/BF00243002.

Przymusinski, T. C. (1990). "The Well-Founded Semantics Coincides with the Three-Valued Stable Semantics". *Fundam. Inform.* 13(4): 445–463.

Ramakrishnan, R. and J. Gehrke. (2003). *Database management systems (3. ed.)* McGraw-Hill. ISBN: 978-0-07-115110-8.

Ramakrishnan, R. and D. Srivastava. (1994). "Semantics and Optimization of Constraint Queries in Databases". *IEEE Data Eng. Bull.* URL: http://sites.computer.org/debull/94JUN-CD.pdf.

Reps, T. W. (1993). "Demand Interprocedural Program Analysis Using Logic Databases". In: *Applications of Logic Databases*. Ed. by R. Ramakrishnan. *The Kluwer International Series in Engineering and Computer Science 296*. Kluwer. 163–196.

Reps, T. W. (1994). "Solving Demand Versions of Interprocedural Analysis Problems". In: *Compiler Construction, 5th International Conference, CC'94, Edinburgh, UK, April 7-9, 1994, Proceedings*. Ed. by P. Fritzson. Vol. 786. *Lecture Notes in Computer Science*. Springer. 389–403. DOI: 10.1007/3-540-57877-3_26. URL: https://doi.org/10.1007/3-540-57877-3%5C_26.

Reps, T. W., S. Horwitz, and S. Sagiv. (1995). "Precise Interprocedural Dataflow Analysis via Graph Reachability". In: *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*. Ed. by R. K. Cytron and P. Lee. ACM Press. 49–61. DOI: 10.1145/199448.199462. URL: https://doi.org/10.1145/199448.199462.

Reps, T. W., E. Turetsky, and P. Prabhu. (2016). "Newtonian program analysis via tensor product". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by R. Bodík and R. Majumdar. ACM. 663–677. DOI: 10.1145/2837614.2837659. URL: https://doi.org/10.1145/2837614.2837659.

Ross, K. A. (1994). "Modular Stratification and Magic Sets for Datalog Programs with Negation". *J. ACM*. 41(6): 1216–1266. DOI: 10.1145/195613.195646. URL: https://doi.org/10.1145/195613.195646.

Ross, K. A. and Y. Sagiv. (1992). "Monotonic Aggregation in Deductive Databases". In: *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California, USA*. Ed. by M. Y. Vardi and P. C. Kanellakis. ACM Press. 114–126. ISBN: 0-89791-519-4. DOI: 10.1145/137097.137852. URL: https://doi.org/10.1145/137097.137852.

Rote, G. (1990). "Path problems in graphs". In: *Computational graph theory*. Vol. 7. *Comput. Suppl.* Springer, Vienna. 155–189. DOI: 10.1007/978-3-7091-9076-0_9. URL: https://doi-org.gate.lib.buffalo.edu/10.1007/978-3-7091-9076-0_9.

Saccà, D. and C. Zaniolo. (1990). "Stable Models and Non-Determinism in Logic Programs with Negation". In: *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee, USA*. Ed. by D. J. Rosenkrantz and Y. Sagiv. ACM Press. 205–217. ISBN: 0-89791-352-3. DOI: 10.1145/298514.298572. URL: https://doi.org/10.1145/298514.298572.

Sagiv, S., T. W. Reps, and S. Horwitz. (1996). "Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation". *Theor. Comput. Sci.* 167(1&2): 131–170. DOI: 10.1016/0304-3975(96)00072-2. URL: https://doi.org/10.1016/0304-3975(96)00072-2.

Salomaa, A. (1966). "Two Complete Axiom Systems for the Algebra of Regular Events". *J. ACM*. 13(1): 158–169. DOI: 10.1145/321312.321326. URL: https://doi.org/10.1145/321312.321326.

Scholz, B., H. Jordan, P. Subotic, and T. Westmann. (2016). "On fast large-scale program analysis in Datalog". In: *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*. Ed. by A. Zaks and M. V. Hermenegildo. ACM. 196–206. DOI: 10.1145/2892208.2892226. URL: https://doi.org/10.1145/2892208.2892226.

Shafer, G. R. and P. P. Shenoy. (1991). "Local computation in hypertrees". URL: https://kuscholarworks.ku.edu/handle/1808/143.

Shafer, G. R. and P. P. Shenoy. (1990). "Probability propagation". In: vol. 2. No. 1-4. 327–351. DOI: 10.1007/BF01531015. URL: https://doi.org/10.1007/BF01531015.

Sharir, M. and A. Pnueli. (1978). *Two approaches to interprocedural data flow analysis*. New York, NY: New York Univ. Comput. Sci. Dept. URL: https://cds.cern.ch/record/120118.

Shenoy, P. P. and G. Shafer. (1988). "Axioms for probability and belief-function proagation". In: *UAI '88: Proceedings of the Fourth Annual Conference on Uncertainty in Artificial Intelligence, Minneapolis, MN, USA, July 10-12, 1988*. Ed. by R. D. Shachter, T. S. Levitt, L. N. Kanal, and J. F. Lemmer. North-Holland. 169–198.

Shkapsky, A., M. Yang, and C. Zaniolo. (2015). "Optimizing recursive queries with monotonic aggregates in DeALS". In: *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. Ed. by J. Gehrke, W. Lehner, K. Shim, S. K. Cha, and G. M. Lohman. IEEE Computer Society. 867–878. ISBN: 978-1-4799-7964-6. DOI: 10.1109/ICDE.2015.7113340. URL: https://doi.org/10.1109/ICDE.2015.7113340.

Solar-Lezama, A., L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. (2006). "Combinatorial sketching for finite programs". In: *ASPLOS*. DOI: 10.1145/1168857.1168907. URL: https://doi.org/10.1145/1168857.1168907.

Son, T. C. and E. Pontelli. (2007). "A Constructive semantic characterization of aggregates in answer set programming". *Theory Pract. Log. Program.* 7(3): 355–375. DOI: 10.1017/S1471068406002936. URL: https://doi.org/10.1017/S1471068406002936.

Staudt, M. and M. Jarke. (1996). "Incremental Maintenance of Externally Materialized Views". In: *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*. Ed. by T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda. Morgan Kaufmann. 75–86. URL: http://www.vldb.org/conf/1996/P075.PDF.

Sudarshan, S. and R. Ramakrishnan. (1991). "Aggregation and Relevance in Deductive Databases". In: *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*. Ed. by G. M. Lohman, A. Sernadas, and R. Camps. Morgan Kaufmann. 501–511. URL: http://www.vldb.org/conf/1991/P501.PDF.

Szabó, T., G. Bergmann, S. Erdweg, and M. Voelter. (2018). "Incrementalizing lattice-based program analyses in Datalog". *Proc. ACM Program. Lang.* 2(OOPSLA): 139:1–139:29. DOI: 10.1145/3276509. URL: https://doi.org/10.1145/3276509.

Szabó, T., S. Erdweg, and G. Bergmann. (2021). "Incremental whole-program analysis in Datalog with lattices". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by S. N. Freund and E. Yahav. ACM. 1–15. DOI: 10.1145/3453483.3454026. URL: https://doi.org/10.1145/3453483.3454026.

Szabó, T., S. Erdweg, and M. Voelter. (2016). "IncA: a DSL for the definition of incremental program analyses". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. Ed. by D. Lo, S. Apel, and S. Khurshid. ACM. 320–331. DOI: 10.1145/2970276.2970298. URL: https://doi.org/10.1145/2970276.2970298.

Tarjan, R. E. (1976). "Graph theory and Gaussian elimination". New York.

Tarjan, R. E. (1975). "Efficiency of a Good But Not Linear Set Union Algorithm". *J. ACM*. 22(2): 215–225. DOI: 10.1145/321879.321884. URL: https://doi.org/10.1145/321879.321884.

Tarjan, R. E. (1981a). "A Unified Approach to Path Problems". *J. ACM.* 28(3): 577–593. DOI: 10.1145/322261.322272. URL: https://doi.org/10.1145/322261.322272.

Tarjan, R. E. (1981b). "Fast Algorithms for Solving Path Problems". *J. ACM.* 28(3): 594–614. DOI: 10.1145/322261.322273. URL: https://doi.org/10.1145/322261.322273.

Ujhelyi, Z., G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró. (2015). "EMF-IncQuery: An integrated development environment for live model queries". *Sci. Comput. Program.* 98: 80–99. DOI: 10.1016/j.scico.2014.01.004. URL: https://doi.org/10.1016/j.scico.2014.01.004.

Ullman, J. D. (1989). *Principles of Database and Knowledge-Base Systems, Volume II.* Computer Science Press. ISBN: 0-7167-8162-X.

Veldhuizen, T. L. (2014). "Triejoin: A Simple, Worst-Case Optimal Join Algorithm". In: *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014.* Ed. by N. Schweikardt, V. Christophides, and V. Leroy. OpenProceedings.org. 96–106. DOI: 10.5441/002/icdt.2014.13. URL: https://doi.org/10.5441/002/icdt.2014.13.

Vianu, V. (2021). "Datalog Unchained". In: *PODS'21: Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Virtual Event, China, June 20-25, 2021.* Ed. by L. Libkin, R. Pichler, and P. Guagliardo. ACM. 57–69. DOI: 10.1145/3452021.3458815. URL: https://doi.org/10.1145/3452021.3458815.

Wainwright, M. J. and M. I. Jordan. (2008). "Graphical Models, Exponential Families, and Variational Inference". *Found. Trends Mach. Learn.* 1(1-2): 1–305. DOI: 10.1561/2200000001. URL: https://doi.org/10.1561/2200000001.

Wang, Y. R., M. A. Khamis, H. Q. Ngo, R. Pichler, and D. Suciu. (2022a). "Optimizing Recursive Queries with Progam Synthesis". In: *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022.* Ed. by Z. G. Ives, A. Bonifati, and A. E. Abbadi. ACM. 79–93. DOI: 10.1145/3514221.3517827. URL: https://doi.org/10.1145/3514221.3517827.

Wang, Y. R., M. A. Khamis, H. Q. Ngo, R. Pichler, and D. Suciu. (2022b). "Optimizing Recursive Queries with Program Synthesis". *CoRR*. abs/2202.10390. arXiv: 2202.10390. URL: https://arxiv.org/abs/2202.10390.

Warshall, S. (1962). "A Theorem on Boolean Matrices". *J. ACM*. 9(1): 11–12. DOI: 10.1145/321105.321107. URL: https://doi.org/10.1145/321105.321107.

Whaley, J. and M. S. Lam. (2004). "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams". In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*. Ed. by W. W. Pugh and C. Chambers. ACM. 131–144. DOI: 10.1145/996841.996859. URL: https://doi.org/10.1145/996841.996859.

Willsey, M., C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha. (2021). "egg: Fast and extensible equality saturation". *Proc. ACM Program. Lang.* (POPL). DOI: 10.1145/3434304. URL: https://doi.org/10.1145/3434304.

Wu, Z., G. Eadon, S. Das, E. I. Chong, V. Kolovski, M. Annamalai, and J. Srinivasan. (2008). "Implementing an Inference Engine for RDFS/OWL Constructs and User-Defined Rules in Oracle". In: *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*. Ed. by G. Alonso, J. A. Blakeley, and A. L. P. Chen. IEEE Computer Society. 1239–1248. DOI: 10.1109/ICDE.2008.4497533. URL: https://doi.org/10.1109/ICDE.2008.4497533.

Yannakakis, M. (1981). "Algorithms for Acyclic Database Schemes". In: *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. IEEE Computer Society. 82–94.

Zaniolo, C., A. Das, J. Gu, Y. Li, M. Li, and J. Wang. (2019). "Monotonic Properties of Completed Aggregates in Recursive Queries". *CoRR*. abs/1910.08888. arXiv: 1910.08888. URL: http://arxiv.org/abs/1910.08888.

Zaniolo, C., M. Yang, A. Das, A. Shkapsky, T. Condie, and M. Interlandi. (2017). "Fixpoint semantics and optimization of recursive Datalog programs with aggregates". *Theory Pract. Log. Program.* 17(5-6): 1048–1065. DOI: 10.1017/S1471068417000436. URL: https://doi.org/10.1017/S1471068417000436.

Zaniolo, C., M. Yang, M. Interlandi, A. Das, A. Shkapsky, and T. Condie. (2018). "Declarative BigData Algorithms via Aggregates and Relational Database Dependencies". In: *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018*. Ed. by D. Olteanu and B. Poblete. Vol. 2100. *CEUR Workshop Proceedings*. CEUR-WS.org. URL: http://ceur-ws.org/Vol-2100/paper2.pdf.

Zhang, N. L. and D. Poole. (1994). "A simple approach to Bayesian network computations". In: *Proc. of the Tenth Canadian Conference on Artificial Intelligence*.

Zhang, Y., Y. R. Wang, O. Flatt, D. Cao, P. Zucker, E. Rosenthal, Z. Tatlock, and M. Willsey. (2023). "Better Together: Unifying Datalog and Equality Saturation". *Proc. ACM Program. Lang.* 7(PLDI). DOI: 10.1145/3591239. URL: https://doi.org/10.1145/3591239.

Zimmermann, U. (1981). "Linear and combinatorial optimization in ordered algebraic structures". *Ann. Discrete Math.* 10: viii+380.

Zinn, D., T. J. Green, and B. Ludäscher. (2012). "Win-move is coordination-free (sometimes)". In: *15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012*. Ed. by A. Deutsch. ACM. 99–113. DOI: 10.1145/2274576.2274588. URL: https://doi.org/10.1145/2274576.2274588.