University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Marek Rogala
Student number: 277570

Declarative queries on large graphs and their distributed evaluation

Master's Thesis
in COMPUTER SCIENCE

Supervisor
dr Jacek Sroka
Institute of Informatics

September 2014

Supervisor's statement

Hereby I confirm that the present thesis was prepared under my supervision and that it fulfills the requirements for the degree of Master of Computer Science.

Date Supervisor's signature

Author's statement

Hereby I declare that the presented thesis was prepared by me and none of its contents was obtained by means that are against the law.

The thesis has never before been a subject of any procedure of obtaining an academic degree.

Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date Author's signature

Abstract

Distributed computations on graphs are becoming increasingly important with the emergence of large graphs such as social networks and the Web that contain huge amounts of useful information. Currently existing solutions, such as iterated MapReduce, Pregel and GraphLab are relatively difficult and require a significant level of fluency in programming. This thesis presents an implementation of a tool which extends a distributed computations platform, Apache Spark, with the capability of executing queries written in a variant of a declarative query language, Datalog, especially extended to better support graph algorithms. This approach makes it possible to express graph algorithms in a declarative query language, accessible to a broader group of users than typical programming languages, and execute them on an existing infrastructure for distributed computations.

Keywords

Large graphs processing, Declarative query languages, Computations on large datasets, Datalog, SociaLite, Apache Spark, RDD, Hadoop, Pregel

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Computer Science

Subject classification

Information Systems: Query languages

Tytuł pracy w języku polskim

Deklaratywne zapytania na dużych grafach i ich rozproszone wyliczanie

Contents

Chapter 1

Introduction

In recent years, the humanity has created many graph datasets much larger than those available ever before. Those graphs became a very popular object of research. Most notable examples are *the Web graph* – a graph of Internet websites and links between them, and all kinds of social networks. Other interesting graphs include transportation routes, similarity of scientific articles or citations among them.

The graphs mentioned can be a source of a huge amount of useful information. Hence, there is an increasing number of practical computational problems. Some of the analyses carried out are ranking of the graph nodes, e.g. importance of a Web page, determining most influential users in a given group of people, detecting communities with clustering, computing metrics for the whole graph or some parts of it and connection predictions. Usually, such analyses are built on top of standard graph algorithms, such as variations of PageRank [5], shortest paths or connected components.

When dealing with such a large graph, distribution of the computations among many machines is inevitable. The graph size is often too large to fit in one computer's memory. At the same time, performing useful computations on a single machine would take too much time for it to be a feasible solution. Size of the data is growing faster than the computational power of computers, and so is the need for distributing the computations.

In the past, we have seen many tools for efficient distributed large dataset computations, such as Google's MapReduce [6] and its widely used open source counterpart, Apache Hadoop [33], as well as higher-level languages such as PigLatin [40] and Hive [41]. However, they are not well suited for graph computations, as they do not support iteration well.

Recently, there is an outbreak of frameworks and languages for large graphs processing, including industrial systems such as Google's Pregel [7] and its open-source version Apache Giraph [31], Graph Processing System [9], GraphLab [17, 19, 18], Apache Spark with GraphX library [21, 32] and Giraph++ [8].

In the frameworks currently available one needs to implement a graph algorithm in a specified model, for example Pregel's "think like a vertex", using a programming language like Java, Scala or Python. On the other hand, query languages, such as SQL, are a bad fit for graph data because of limited support of iteration. Yet, one of the advantages of query languages over general-purpose programming languages is that they are available for a much broader group of users: they are used not only by programmers, but also by analysts and data scientists. Queries are often optimized by query engines automatically. With the rise of graph computational problems, we need an easier way to extract information from graphs: a query language for effectively expressing data queries typical for graphs.

The Socialite [1, 2] language is one of the most interesting propositions. It is based on a classical query language — Datalog [3]. In Datalog, the problem is expressed in a declarative way as a set of rules. Declarative semantics makes it easy to distribute the computations, since no execution flow is embedded in the program code. It also gives many possibilities for optimizations and approximate evaluation. At the same time, Datalog's support for recursion is crucial, since most graph algorithms are of iterative nature. However, most practical graph algorithms cannot be expressed efficiently in Datalog because of the language limitations. With a few extensions to original Datalog, the most important of which is recursive aggregation, SociaLite makes it easy to write intuitive programs which can be executed very efficiently.

Unfortunately, there is no solid distributed implementation of SociaLite available. Only an undocumented interpreter for a sequential version of the language has been published along with the papers. According to [2], the distributed version of the interpreter was build as an independent implementation based on Hadoop. It is hard to determine when the language could be adopted in the industry. At the same time, papers [1] and [2] which introduced SociaLite contain certain simplifications and are not precise about some important details in definitions and proofs.

The goal of this thesis is to bridge the gap between the theoretical idea for Datalog with recursive aggregation and its practical implementation as well as to draw a path towards its usage in the industry. We show how to translate SociaLite-like declarative programs into programs on the Apache Spark platform and present an implementation of an extension to Spark that enables them to be executed on existing infrastructure and software stack. The extension allows Spark users to perform distributed graph computations using a declarative language without any additional effort to build a dedicated server infrastructure.

Spark [21] is an open-source project providing a general platform for processing large datasets which has gained a huge momentum since the initial white paper in 2010 [20] and inclusion into Apache Incubator in June 2013. Computations in Spark are expressed using an abstraction of distributed memory, called *RDD*. Distinctive features of Spark are the ability to keep cached data in node's memory, which gives impressive speedups over other environments like Hadoop MapReduce, and a powerful API allowing for various usages including MapReduce, machine learning, computations on graphs and stream data processing. In February 2014 Spark became a top-level project of the Apache Foundation, and since July 2014 it has been included in the Cloudera CDH, a popular enterprise platform for Hadoop deployment. Spark is already a stable, well-tested platform which is being intensively developed and can be expected to become a new industry standard in large datasets processing. For these reasons, it has been chosen as the most promising platform for the implementation.

The thesis consists of six chapters. In Chapter 2 we recall definitions of Datalog and its evaluation methods while Chapter 3 contains an overview of the RDD and Pregel computation models. In Chapter 4 we describe the recursive aggregation extension to Datalog introduced by SociaLite and provide formal definitions and general-case proofs which the original papers lack. Chapter 5 describes the translation procedure from Datalog with recursive aggregation to the RDD model and its implementation as a SparkDatalog extension for Apache Spark. In Chapter 6 we summarize the results of this work.

1.1. Basic definitions

We start by giving some basic definitions which will be used in this paper.

The languages considered in the paper operate on databases which consist of facts over relations identified by relation names, for example R, P, Tc, Path or Edge.

The relations contain facts which are tuples of values from a countable infinite set **dom** called the *domain*. The programs that we will consider use variables from a set **var**, which is disjoint from **dom**.

The elements of **dom** are called *constants*, whereas the elements of **var** are called *(free) variables*. In examples and definitions we will use strings starting with a lowercase letter as variables, for example: a, b, x, $length$, $dist$. We will use numbers and strings starting with an uppercase letter as constants, for example 1, 2, A, B, $Alice$ and Bob.

Definition 1.1.1. A *database schema* is a tuple (N, ar), where N is the set of *relation names* and $ar : N \rightarrow \mathbb{N}^+$ assigns *arities* to relation names.

For a database schema $\sigma = (N, ar)$ and a relation name R we will write $R \in \sigma$ as a shorthand for $R \in N$. If $R \in \sigma$, we say that R is a relation name in σ with arity $ar(R)$.

Given a database schema σ, let R be a relation name in σ with arity n. A *fact* over R is an expression $R(x_1, \ldots, x_n)$, where each $x_i \in \mathbf{dom}$. A fact is sometimes written in the form of $R(v)$ where $v \in \mathbf{dom}^n$ is a tuple. The relation name is sometimes omitted when it can be understood from the context; a fact is then written simply as a tuple.

A *relation* or *relation instance* over R is defined as a set of facts over R.

A *database* or *database instance* over database schema σ is a union of relations over R, where $R \in \sigma$. We typically use bold uppercase letters for databases, for example **I**, **J** or **K**. A database is *finite* if all its relations are finite.

We denote the set of all possible relations over a relation name R by $inst(R)$. Similarly, we denote the set of all possible databases over a database schema σ by $inst(\sigma)$.

Definition 1.1.2 (Valuation)**.** For a set $V \subseteq \mathbf{var}$ of free variables, a function $\nu : V \rightarrow \mathbf{dom}$ is called a *valuation* of V.

Valuation is naturally extended to $\mathbf{dom} \cup \mathbf{var}$ as an identity function on constants. It is also extended to a function from tuples over $\mathbf{dom} \cup \mathbf{var}$ to tuples over **dom** by applying the valuation to each element of the tuple.

Example 1.1.1. Let $\sigma = (\{\textsc{Edge}, \textsc{Parent}\}, ar)$, where $ar(\textsc{Edge}) = 3, ar(\textsc{Parent}) = 2$, be a database schema.

$\textsc{Edge}(1, 2, 17)$ and $\textsc{Edge}(1, 3, 5)$ are facts over $\textsc{Edge}$, whereas $\textsc{Parent}(Alice, Bob)$ and $\textsc{Parent}(Bob, Chris)$ are facts over $\textsc{Parent}$.

$I = \{\textsc{Edge}(1,\ 2,\ 17), \textsc{Edge}(1,\ 3,\ 5)\}$ is a relation instance over $\textsc{Edge}$.

$\mathbf{K} = \{\textsc{Edge}(1,\ 2,\ 17), \textsc{Parent}(\text{Alice, Bob}), \textsc{Parent}(\text{Bob, Chris})\}$ is a database instance over σ.

For variables $x, y, z \in \mathbf{dom}$, ν such that $\nu(x) = 1, \nu(y) = 7, \nu(z) = Alice$ is a valuation of $\{x, y, z\}$. For constants ν is by definition identity: $\nu(Bob) = Bob, \nu(7) = 7$. As natural extension, ν can be applied to tuples: $\nu(x, Bob, 9, z) = (1, Bob, 9, Alice)$.

Definition 1.1.3 (Fix-point)**.** If f is a function $f : D \rightarrow D$ and $f(x) = x$ for any $x \in D$, then x is called a *fix-point* of f.

Definition 1.1.4 (Pre-order)**.** A binary relation $\leq$ over a set P is a *pre-order* if it is reflexive and transitive, i.e. for each $x, y, z \in P$ the following properties are satisfied:

- $x \leq x$,
- if $x \leq y$ and $y \leq z$ then $x \leq z$.

Definition 1.1.5 (Partial order)**.** A binary relation $\leq$ over a set P is a *partial order* if it is reflexive, antisymmetric and transitive, i.e. for each $x, y, z \in P$ the following properties are satisfied:

- $x \leq x$,
- if $x \leq y$ and $y \leq x$ then $x = y$,
- if $x \leq y$ and $y \leq z$ then $x \leq z$.

Example 1.1.2. Reachability relation in a directed graph G, such that $x \leq y$ iff there is a path from x to y in G, is a pre-order. This relation is clearly reflexive and transitive, but it does not have to be antisymmetric — if x and y are different vertices contained in one cycle, then $x \leq y$ and $y \leq x$, but $x \neq y$.

If G is additionally required to be acyclic, then the reachability relation is guaranteed to be antisymmetric, so it is a partial order.

Definition 1.1.6 (Monotonicity)**.** Function $f : D \to C$ is monotone with respect to pre-order $\sqsubseteq$ iff $x \sqsubseteq y \to f(x) \sqsubseteq f(y)$ for each $x, y \in D$.

Chapter 2

Datalog

In this chapter we describe the basic Datalog language and its typical extended versions.

Languages based on relational algebra and relational calculus, like SQL, are widely used and researched as query languages for relational databases. This dates back to Edgar F. Codd's relational model [12] introduced in 1970. Unfortunately, such languages leave some simple operations that they cannot handle. Examples of such problems are transitive closure of a graph or distances from a vertex to all other vertices. Although new implementations of SQL permit recursion, it is usually not suitable for expressing complex graph algorithms in an easy and succinct way.

Datalog [3] is a query language which allows for solving those problems due to the availability of recursion. Without the recursion, Datalog (with negation) has the same expressive power as both relational algebra and relational calculus. The language appeared around 1978 and was inspired by the logical programming paradigm. Recently, there has been an increasing interest in Datalog research as well as in its industrial implementations. Datalog is typically extended with negation and simple, non-recursive aggregation.

Let us begin with an example of a problem which cannot be solved in relational calculus, but can be easily solved in Datalog.

Let us suppose that we have a database with a binary relation $\textsc{Edge}$. The database represents graph G: $\textsc{Edge}(a, b)$ means that there is an edge in G between vertices a and b. Given a selected vertex s, we would like to find all vertices in G that are reachable from s.

Unfortunately it can be proven that this kind of query is not expressible in relational calculus [3], unless we make some additional assumptions about G. Intuitively, what is necessary to answer such queries is some kind of conditional iteration or recursion, which is the most important feature of Datalog.

2.1. History

Datalog is not credited to any particular researchers since it originated as an extension or restriction of various other languages, including logic programming languages. It emerged as a separate area of research around 1977. Professor David Maier is believed to be the author of the name *Datalog*.

Datalog is described in detail in classical books on databases theory, such as *Foundations of Databases* [3].

The language has been proven to be useful in various fields like program analysis [22] and network systems [23, 24]. It is also used to formally define computational problems which can be solved with different models and frameworks, allowing for the comparison of those frameworks and their optimizations [11].

Some of the most important fields of research concerning Datalog are the optimizations in programs evaluation, e.g. magic sets [45] and subsumptive queries [46] as well as extensions to the language [42, 43, 44].

Recently there also has been an increasing interest in applications of Datalog in industry. Two examples worth mentioning are LogicBlox and Datomic. LogicBlox [29] provides a high performance database which can be queried with a Datalog variant called LogiQL. Datomic [30], which uses Datalog as a query language, is a distributed database with an innovative architecture featuring immutable records and temporal queries.

2.2. Introduction to Datalog

Before we formally define Datalog syntax and semantics in the further sections, let us take a look at an example program in this language.

As before, let us assume that the database contains a relation Edge representing a graph and Edge(a, b) means that there is an edge between vertices a and b. Figure 2.1 presents a program that computes relation Tc containing a transitive closure of the relation Edge.

$$\begin{array}{lcl}
\textsc{Tc}(a,b) & :- & \textsc{Edge}(a,b). \\
\textsc{Tc}(a,b) & :- & \textsc{Tc}(a,c), \textsc{Edge}(c,b).
\end{array}$$

Figure 2.1: Datalog query for computing transitive closure of a graph.

This program contains two rules. The first one states that if there is an edge between a and b, then there also is such an edge in the transitive closure. The second rule says that if there is a connection in the transitive closure between a and some c and there is an edge between c and b in the original graph, then there also exists a connection in transitive closure between a and b. This is where recursion occurs, as Tc appears on both sides of the second rule.

For example, let Edge contain the following facts:

$$\textsc{Edge}(1,2) \qquad \textsc{Edge}(2,3) \qquad \textsc{Edge}(3,4) \qquad \textsc{Edge}(2,5)$$

The result computed by the program is:

$$\begin{array}{cccc}
\textsc{Tc}(1,2) & \textsc{Tc}(1,4) & \textsc{Tc}(2,3) & \textsc{Tc}(2,5) \\
\textsc{Tc}(1,3) & \textsc{Tc}(1,5) & \textsc{Tc}(2,4) & \textsc{Tc}(3,4)
\end{array}$$

Both rules can add new facts to the Tc relation, since each of them has Tc on its left side. We can view Tc as an output of this program. On the other hand, Edge appears only on the right side of the rules. The program can not produce any new Edge facts, so Edge must be given to the program as an input. As we can see, the program defines a function from an instance of relation Edge into an instance of relation Tc. We will define this more formally in the present chapter.

2.2.1. Differences between Datalog and Prolog

Datalog was largely inspired by logic programming languages such as Prolog. Datalog's and Prolog's syntaxes are very similar. Despite the close relation between Datalog and logic programming languages, there are some significant differences:

- In Prolog, one can use complex terms as arguments to predicates, for example $p(s(x), y)$, which is not permitted in Datalog, where the only allowed arguments are constants and variables.
- One of the important language constructs in Prolog is the cut operator "!", which allows for expressing negation. There is no such operator in Datalog. While some versions of Datalog have the notion of negation, it is different than the cut operator known from Prolog.
- Datalog requires the rules to be *safe*, which means that every variable mentioned in a rule must also be mentioned at least once in a non-negated, non-arithmetic sense.
- Unlike Prolog, the order of rules and subgoals in Datalog does not change the program semantics.

Datalog is less expressive than Prolog. In contrast to Prolog, it is not Turing-complete. This obviously restricts the set of problems which can be solved with Datalog, but at the same time it allows more to be reasoned about Datalog programs and gives more possibilities to optimize their evaluation.

2.3. Datalog syntax

Let us define Datalog programs and rules.

Definition 2.3.1 (Rule)**.** A *rule* is an expression of the form:

$$\mathrm{R}(x) : -R_1(x_1), \ldots, R_n(x_n).$$

where $n \geq 1$, $R, R_1, \ldots, R_n$ are names of relations and $x, x_1, \ldots x_n$ are tuples of free variables or constants. Each tuple $x, x_1, \ldots x_n$ must have the same arity as its corresponding relation.

The sign $: -$ splits the rule into two parts: the leftmost part, i.e. $R(x)$, is called the *head* of the rule, while the rightmost part, i.e. $R_1(x_1), \ldots, R_n(x_n)$, is called the *body* of the rule. The elements of the body separated by commas are called *subgoals*. The head and subgoals are called *atoms*. Each atom consists of a *predicate*, i.e. the relation name and *arguments*.

Intuitively, if there is a free variable in the head, it should also appear in at least one of the subgoals, so that the value for that variable can be determined by matching the corresponding subgoal. Rules satisfying this property are called *safe*.

Definition 2.3.2 (Rule safety)**.** A rule is *safe* iff each free variable appearing in its head also appears in at least one of the subgoals.

A program in Datalog consists of some number of rules, all of which are required to be safe. The order of the rules is irrelevant.

Definition 2.3.3 (Program in Datalog)**.** A *program* in Datalog is a finite set of safe rules.

By *adom*(P) we denote the set of constants appearing in the rules of P.

The *schema* of program P is the set of all relation names occurring in P and is denoted by *sch*(P).

Definition 2.3.4 (Extensional and intensional relations)**.** The rules of a Datalog program P divide the relations into two disjoint classes:

- *extensional* relations, i.e. relations that occur only in the subgoals, but never in the head of the rules in P,
- *intensional* relations occurring in the head of at least one of the rules in P.

The set of extensional relations is called the *extensional database* or the *edb*, whereas the set of intensional relations is called the *intensional database* or the *idb*. For program P, the *extensional database schema*, denoted by *edb*(P), is the set of all extensional relation names. Similarly, the *intensional database schema*, denoted by *idb*(P), is the set of all intensional relation names.

Example 2.3.1. As an example, let us consider the program P presented in Figure 2.2.

$$\begin{array}{lll}
\textsc{Mother}(\mathit{parent}, \mathit{child}) & :- & \textsc{Parent}(\mathit{parent}, \mathit{child}), \textsc{Woman}(\mathit{parent}). \\
\textsc{Father}(\mathit{parent}, \mathit{child}) & :- & \textsc{Parent}(\mathit{parent}, \mathit{child}), \textsc{Man}(\mathit{parent}). \\
\textsc{Ancestor}(\mathit{ancestor}, \mathit{child}) & :- & \textsc{Parent}(\mathit{ancestor}, \mathit{child}). \\
\textsc{Ancestor}(\mathit{ancestor}, \mathit{child}) & :- & \textsc{Ancestor}(\mathit{ancestor}, \mathit{parent}), \textsc{Parent}(\mathit{parent}, \mathit{child}).
\end{array}$$

Figure 2.2: Datalog program for computing ancestors based on a database with relations Parent, Woman and Man.

Let Parent(p, c) mean that p is c's parent and Woman(x) and Man(x) state whether person x is a woman or a man, respectively. As will be clear after defining the program semantics in 2.4, this program computes child's father, mother and all its ancestors that can be derived.

*edb*and *idb*for P are the following:

$$\begin{aligned}
\mathit{edb}(P) &= \{\textsc{Parent}, \textsc{Man}, \textsc{Woman}\} \\
\mathit{idb}(P) &= \{\textsc{Mother}, \textsc{Father}, \textsc{Ancestor}\}
\end{aligned}$$

Parent, Woman and Man are *edb*relations, because there are no rules for these relations. All of their contents must be provided as an input. On the other hand, Mother, Father and Ancestor are *idb*relations, since there are rules for computing them. Only one of them, Ancestor, is recursively defined.

2.4. Datalog semantics

A Datalog program is essentially a function from database instances over *edb*(P) into database instances over *sch*(P). We assume that all *edb* relations are provided in the input. *idb* relations are always empty at the start of a computation and can only be populated by the rules.

The exact semantics of a Datalog program can be defined using one of three equivalent approaches.

In the *model theoretic* definition, we consider the rules of program P to be logical properties of the desired solution. From all possible instances of the intensional database we choose those, which are a *model* for the program, i.e. satisfy all the rules. The smallest such model is defined to be the semantics of P.

The second approach is *proof theoretic*, in which a fact is included in the result if and only if it can be derived, i.e. proven using the rules. There are two strategies for obtaining proofs for facts: *bottom up*, in which we start from all known facts and incrementally derive all provable facts, and *top down*, which starts from a fact to be proven and seeks for rules and facts that can be used to prove it.

The third approach, on which we focus in this thesis is the *least fix-point semantics*, defining the program result as the least fix-point of some function. In this definition, a program is evaluated by iteratively applying the function until a fix-point is reached. This is very similar to the bottom-up evaluation strategy of the proof-theoretic approach.

2.4.1. Fix-point semantics

In this section we give the fix-point semantics for Datalog programs. A central notion in this definition is the *immediate consequence* operator. Intuitively, this operator adds to the database new facts that could be immediately derived using one of the rules. In order to define the immediate consequence operator, let us first introduce the notion of an instantiation of a rule.

Definition 2.4.1 (Instantiation)**.** Given a rule $\mathrm{R}(x) : -R_1(x_1), \ldots, R_n(x_n).$, if ν is a valuation of variables appearing in this rule, then we obtain an *instantiation* of this rule with replacing each tuple in the rule by its value $\nu(t)$:

$$\mathrm{R}(\nu(x)) : -R_1(\nu(x_1)), \ldots, R_n(\nu(x_n)).$$

Example 2.4.1. If $Anna, Chris, Patrick$ are some values in the domain, then:

$$\textsc{Ancestor}(Anna, Chris) : -\textsc{Ancestor}(Anna, Patrick), \textsc{Parent}(Patrick, Chris).$$

is an instantiation of the rule:

$$\textsc{Ancestor}(ancestor, child) : -\textsc{Ancestor}(ancestor, parent), \textsc{Parent}(parent, child).$$

Immediate consequence operator

Given some database $\mathbf{K}$ and a program P, we can infer facts using the rules in P and the contents of $\mathbf{K}$. This procedure is formally defined using the *immediate consequence operator*.

Definition 2.4.2 (Immediate consequence operator)**.** For a program P and a database instance $\mathbf{K}$ over $sch(P)$, we say that a fact $R(v)$ is an *immediate consequence* for $\mathbf{K}$ and P, iff $R(v) \in \mathbf{K}$ or there exists an instantiation $R(v) : -R_1(v_1), \ldots, R_n(v_n)$ of a rule in P such that $R_i(v_i) \in \mathbf{K}$ for each $i = 1 \ldots n$. The *immediate consequence operator* for a Datalog program P is a function $T_P : inst(sch(P)) \to inst(sch(P))$, such that:

$$T_P(\mathbf{K}) = \{R(v) : R(v) \text{ is an immediate consequence for } \mathbf{K} \text{ and } P\}.$$

Lemma 2.4.1. *Operator T_P for any Datalog program P is a monotone function with respect to inclusion order.*

Proof. Given any $\mathbf{I}, \mathbf{J} \in \mathit{inst}(\mathit{sch}(P))$ such that $\mathbf{I} \subseteq \mathbf{J}$, let $R(v)$ be a fact in $T_P(\mathbf{I})$. By definition, $R(v)$ is an immediate consequence for $\mathbf{I}$ and P, so either $R(v)$ is in $\mathbf{I}$ or there exists an instantiation $R(v) : -R_1(v_1), \ldots, R_n(v_n)$ of a rule in P such that $R_i(v_i) \in \mathbf{I}$ for each $i = 1 \ldots n$. In the first case, $R(v) \in \mathbf{I} \subseteq \mathbf{J}$, so $R(v) \in \mathbf{J}$. In the second case, each $R_i(v_i) \in \mathbf{I} \subseteq \mathbf{J}$, so the instantiation also exists in $\mathbf{J}$. Hence, $R(v)$ is also an immediate consequence of $\mathbf{J}$, and thus $R(v) \in T_P(\mathbf{J})$. Since $R(v)$ was arbitrarily chosen, we have $T_P(\mathbf{I}) \subseteq T_P(\mathbf{J})$ and so T_P is a monotone function with respect to $\subseteq$. ■

Semantics of a Datalog program

The output of a Datalog program P on some finite input database $\mathbf{K}$, denoted by $P(\mathbf{K})$, is defined as the minimum fix-point of T_P that contains $\mathbf{K}$.

Theorem 2.4.2. *For a Datalog program P and a finite database instance $\mathbf{K}$ over $\mathit{edb}(P)$, there exists a finite minimum fix-point of T_P containing $\mathbf{K}$.*

Proof. As it holds that $\mathit{adom}(P) \cup \mathit{adom}(\mathbf{K})$ and the database schema $\mathit{sch}(P)$ are all finite and rules of P are safe, they do not introduce any other values, and so there is a finite number n of database instances over $\mathit{sch}(P)$ that can be reached by iteratively applying T_P to $\mathbf{K}$. Hence, because of the monotonicity of T_P, the sequence $\{T_P^i(\mathbf{K})\}_i$ reaches a fix-point: $T_P^n(\mathbf{K}) = T_P^{n+1}(\mathbf{K})$. Let us denote this fix-point by $T_P^*(\mathbf{K})$.

The definition of T_P implies that $\mathbf{K} \subseteq T_P(\mathbf{K})$. Because of monotonicity of T_P, we have inductively that $T_P^i(\mathbf{K}) \subseteq T_P^{i+1}(\mathbf{K})$. Hence, we have:

$$\mathbf{K} \subseteq T_P(\mathbf{K}) \subseteq T_P^2(\mathbf{K}) \subseteq T_P^3(\mathbf{K}) \subseteq \cdots \subseteq T_P^*(\mathbf{K})$$

We will now prove that $T_P^*(\mathbf{K})$ is the minimum fix-point of T_P containing $\mathbf{K}$. Let us suppose that $\mathbf{J}$ is a fix-point of T_P containing $\mathbf{K}$ as a subset, i.e. $\mathbf{K} \subseteq \mathbf{J}$. By applying T_P n times to both sides of the inequality and observing that $T_P^n(\mathbf{J}) = \mathbf{J}$ as $\mathbf{J}$ is a fix-point, we have $T_P^*(\mathbf{K}) = T_P^n(\mathbf{K}) \subseteq \mathbf{T}_P^n(\mathbf{J}) = \mathbf{J}$. Hence, $T_P^*(\mathbf{K})$ is the minimum fix-point of T_P containing $\mathbf{K}$. ■

Example 2.4.2. Let us recall the program P from Figure 2.2, which computes ancestors based on a database with relations Parent, Woman and Man.

Given the following *edb*database instance $\mathbf{K}$:

Parent(*Anna*, *Bill*)	Woman(*Anna*)
Parent(*Bill*, *Chris*)	Woman(*Eva*)
Parent(*Anna*, *David*)	Man(*Bill*)
Parent(*Chris*, *Eva*)	Man(*Chris*)
	Man(*David*)

The minimal fix-point of T_P containing $\mathbf{K}$ is:

Parent(*Anna*, *Bill*)	Man(*Bill*)	Ancestor(*Anna*, *Bill*)
Parent(*Bill*, *Chris*)	Man(*Chris*)	Ancestor(*Bill*, *Chris*)
Parent(*Anna*, *David*)	Man(*David*)	Ancestor(*Anna*, *David*)
Parent(*Chris*, *Eva*)	Mother(*Anna*, *Bill*)	Ancestor(*Chris*, *Eva*)
Woman(*Anna*)	Mother(*Anna*, *David*)	Ancestor(*Anna*, *Chris*)
Woman(*Eva*)	Father(*Bill*, *Chris*)	Ancestor(*Anna*, *Eva*)
	Father(*Chris*, *Eva*)	

2.5. Evaluation of Datalog programs

The most straightforward evaluation algorithm for Datalog programs is the iterative evaluation derived from the fix-point definition of semantics. While having a very simple formulation, this method is not efficient in a typical case due to redundant computation. The most basic optimization addressing this problem is the *semi-naive* evaluation, which tries to avoid computations that cannot bring any new facts. Naive and semi-naive evaluations are examples of the bottom-up strategy, where new facts are inferred based on the facts currently known.

There exist also other, more optimized evaluation methods, such as magic sets [45] and subsumptive queries [46]. The top-down strategy [3, 46] is also possible, where queries are answered by making an attempt to prove a fact using available rules.

This section briefly describes the ways to evaluate Datalog programs.

2.5.1. Naive evaluation

In the naive evaluation the computation starts with the initial database containing the *edb*relations and repeatedly applies all the rules until a fix-point is reached.

In pseudocode the algorithm for naive evaluation of a program P on an input $\mathbf{K}$ can be written as presented in Figure 2.3.

```
NAIVE-EVALUATE-DATALOG($P$, $\mathbf{K}$)
1  $I_0 \leftarrow K$
2  $i \leftarrow 0$
3  repeat
4        $i \leftarrow i + 1$
5        $I_i \leftarrow T_P(I_{i-1})$
6  until $I_i = I_{i-1}$
7  return $I_i$
```

Figure 2.3: Naive evaluation algorithm for Datalog.

Example 2.5.1. As an example, let us consider the program shown in Figure 2.4 which computes a transitive closure of a binary relation R.

$$\begin{aligned} \textsc{Tc}(x,y) &:- \mathrm{R}(x,y). \\ \textsc{Tc}(x,y) &:- \textsc{Tc}(x,z), \textsc{Tc}(z,y). \end{aligned}$$

Figure 2.4: Program computing transitive closure of relation R.

Given $K = \{\mathrm{R}(1,2), \mathrm{R}(2,3), \mathrm{R}(3,4), \mathrm{R}(4,5)\}$, the values produced in subsequent iterations are:

$$\begin{aligned}
I_1 &\leftarrow \{\mathrm{R}(1,2), \mathrm{R}(2,3), \mathrm{R}(3,4), \mathrm{R}(2,5), \textsc{Tc}(1,2), \textsc{Tc}(2,3), \textsc{Tc}(3,4), \textsc{Tc}(4,5)\} \\
I_2 &\leftarrow \{\mathrm{R}(1,2), \mathrm{R}(2,3), \mathrm{R}(3,4), \mathrm{R}(2,5), \textsc{Tc}(1,2), \textsc{Tc}(2,3), \textsc{Tc}(3,4), \textsc{Tc}(4,5), \\
&\quad \textsc{Tc}(1,3), \textsc{Tc}(2,4), \textsc{Tc}(3,5)\} \\
I_3 &\leftarrow \{\mathrm{R}(1,2), \mathrm{R}(2,3), \mathrm{R}(3,4), \mathrm{R}(2,5), \textsc{Tc}(1,2), \textsc{Tc}(2,3), \textsc{Tc}(3,4), \textsc{Tc}(4,5), \\
&\quad \textsc{Tc}(1,3), \textsc{Tc}(2,4), \textsc{Tc}(3,5), \textsc{Tc}(1,4), \textsc{Tc}(2,5)\}, \textsc{Tc}(1,5)\}
\end{aligned}$$

2.5.2. Semi-Naive evaluation

A straightforward implementation of the definition of T_P is to perform a natural join on subgoal relations and then a projection to head variables. Example 2.5.1 shows that such implementation may be inefficient, because most of the facts are computed more than once.

A simple observation is that the immediate consequence operator T_P for any program P is *inflationary*, i.e. it possibly adds facts to the database, but can never remove any fact. In other words, $T_P(\mathbf{I}) \supseteq \mathbf{I}$ for any I. As a consequence, in an iterative evaluation which uses T_P, the database instance $\mathbf{I}_i$ inferred in step i is a superset of any database instance $\mathbf{I}_j$ that was derived in a previous step $j < i$. To name this property, we say that such semantics is *inflationary.*

The *semi-naive evaluation* is the most basic optimization used in Datalog evaluation, in which T_P is computed in a more efficient way. It comes from the following observation: in a Datalog program, if some rule Q produces fact $R(t)$ based on database instance I_i in the i-th iteration of the naive evaluation algorithm, then this rule will produce this fact in each subsequent iteration, because of the inflationary semantics of the language. The goal of this optimization is to avoid such computations after producing the fact for the first time. This is achieved by joining only subgoals in the body of each rule which have at least one new answer produced in the previous iteration.

Let T_P^Δ denote a function that evaluates the rules of program P so that at least one new fact is used in the application of a rule. This function needs to know which facts are the new ones, so it takes two arguments: I — the full database instance and Δ — the database instance containing the facts that were added in the last iteration. Note that this function does not necessarily return facts from I, so we will need to add them to the newly computed facts in order to get the same result as T_P, i.e. for each i, $T_P(I_i) = I_i \cup T_P^\Delta(I_i, \Delta_i)$. Figure 2.5 presents the algorithm for semi-naive evaluation of Datalog program P on input database $\mathbf{K}$.

Example 2.5.2. Let us consider the program and input from Example 2.5.1. The facts computed by the Semi-naive evaluation in subsequent iterations would be the following:

$$C_1 \leftarrow \{\textsc{Tc}(1,2), \textsc{Tc}(2,3), \textsc{Tc}(3,4), \textsc{Tc}(4,5)\}$$
$$C_2 \leftarrow \{\textsc{Tc}(1,3), \textsc{Tc}(2,4), \textsc{Tc}(3,5)\}$$
$$C_3 \leftarrow \{\textsc{Tc}(1,4), \textsc{Tc}(2,5), \textsc{Tc}(1,5)\}$$
$$C_4 \leftarrow \{\textsc{Tc}(1,5)\}$$

```
Semi-naive-Evaluate-Datalog($P$, $\mathbf{K}$)
 1  $I_0 \leftarrow K$
 2  $\Delta_0 \leftarrow K$
 3  $i \leftarrow 0$
 4  repeat
 5        $i \leftarrow i + 1$
 6        $C_i \leftarrow T_P^{\Delta}(I_{i-1}, \Delta_{i-1})$
 7        $I_i \leftarrow C_i \cup I_{i-1}$
 8        $\Delta_i \leftarrow I_i - I_{i-1}$
 9  until $\Delta_i = \emptyset$
10  return $I_i$
```

Figure 2.5: Semi-naive evaluation algorithm for Datalog.

Semi-naive evaluation does not ensure that each fact will be computed once, e.g. Tc(1, 5) was computed more than once — in the third iteration because of $\text{Tc}(1, 3), \text{Tc}(3, 5)$ and in the fourth iteration because of $\text{Tc}(1, 4), \text{Tc}(4, 5)$ — but it eliminates a significant portion of redundant computation.

2.5.3. Other strategies

Naive evaluation and semi-naive evaluation are examples of the bottom-up approach, where we start with the initial database instance and gradually extend it with facts that can be inferred until a fix-point is reached.

An opposite approach is possible as well. In top-down evaluation which originates in logic programs evaluation, we start with the query. For example, we would like to find all values of x, for which $\text{Tc}(3, x)$ is true. We can use the first rule: for $\text{Tc}(3, x)$ we would need $\text{R}(3, x)$. The only such fact is $\text{R}(3, 4)$ for $x = 4$. We can also use the second rule, which leaves us with finding y such that $\text{Tc}(3, y)$, which yields $y \in \{4\}$ by the first rule. Then, we need to find x such that $\text{Tc}(4, x)$, which by the first rule yields $x \in \{5\}$. The final result is thus $x \in \{4, 5\}$

An advantage of the top-down approach is that it does not have to compute the whole database. Instead, it computes only the actually necessary facts.

This can be also achieved in bottom-up evaluation by using optimization techniques such as *magic sets* [45, 3] and *subsumptive queries* [46]. They involve transforming the relations and rules into a new program, which evaluation using the bottom-up approach essentially simulates evaluation using a top-down algorithm. Magic sets is a classical technique, while subsumptive queries is an example of a new development in the field, published in 2011.

2.6. Typical extensions

Despite recursion, pure Datalog's expressive power is still not enough for many practical applications. Datalog is often extended with:

- arithmetic predicates, such as $\leq$,
- arithmetic functions, like addition and multiplication,
- negation,

- non-recursive aggregation.

It this section we will briefly describe these extensions.

2.6.1. Arithmetic predicates

If we assume that all values in a given column of a relation are numeric, it may often be useful to write Datalog programs that incorporate arithmetic comparisons between such values.

Let us consider the following example. We have a database of employees consisting of two relations Boss and Salary : Boss(a, b) means that employee a is a direct boss of employee b and Salary(a, s) means that the salary of employee a is s. All values in the second column of relation Salary are assumed to be numeric. We would like to find all employees that earn more than their direct boss.

$$\begin{array}{ll} \textsc{Boss}(a,b) & \textsc{Salary}(a,10) \\ \textsc{Boss}(b,c) & \textsc{Salary}(b,15) \\ \textsc{Boss}(b,d) & \textsc{Salary}(c,5) \\ & \textsc{Salary}(d,20) \end{array}$$

The problem is solved by the following query with arithmetic comparisons:

$$\begin{aligned} &\textsc{EarnsMoreThanBoss}(\mathit{employee}) :- \\ &\qquad \textsc{Boss}(\mathit{boss}, \mathit{employee}), \textsc{Salary}(\mathit{boss}, \mathit{bs}), \textsc{Salary}(\mathit{employee}, \mathit{es}), \mathit{es} > \mathit{bs}. \end{aligned}$$

Arithmetic comparisons can be thought of as a new kind of predicates, which are infinite built-in relations. Since we introduced implicit infinite relations, we need to adjust the definition 2.3.2 of rule safety.

Definition 2.6.1 (Rule safety)**.** A rule with arithmetic comparisons is *safe* iff each free variable appearing in its head or in any of the comparisons also appears in at least one of the non-comparison subgoals.

This version of the requirement ensures that comparisons do not introduce any new values into the database.

2.6.2. Datalog with negation

Pure version of Datalog permits recursion but provides no negation. Negation allows to answer queries such as "which pairs of the nodes in graph are not connected?". There are several ways of adding negation to Datalog. One of the most prominent of them is the *stratified semantics*, which we will present in this section.

In Datalog with negation, abbreviated Datalog$^{\neg}$, each relational subgoal may be negated, i.e. preceded with the negation symbol “!”. The negated subgoals are called *negative* subgoals, and the rest of the subgoals is called *positive* subgoals. Other types of subgoals, such as arithmetic comparisons, are not allowed to be negated.

Example 2.6.1. Let us consider the program from example 2.5.1 which computes transitive closure Tc of a relation R. The following rule computes the pairs of nodes which are indirectly connected, i.e. are in Tc, but not in R:

$$\textsc{Indirect}(x, y) :- \mathrm{R}(x, y),\ !\textsc{Tc}(x, y).$$

When negative subgoals are permitted, we need to include them in the definition of rule safety.

Definition 2.6.2 (Rule safety)**.** A rule in Datalog$^\neg$with arithmetic comparisons is *safe* iff each free variable appearing in:

- its head,
- any of the comparisons,
- or in any of its negated subgoals

also appears in at least one of the non-negated relational subgoals.

We will first consider a certain class of Datalog$^\neg$programs, called semi-positive programs, for which the semantics of negation is straightforward. We will then move on to a more general version.

Definition 2.6.3 (Semi-positive program)**.** A Datalog$^\neg$program P is *semi-positive*, iff for each rule in P, all its negated subgoals are over $edb(P)$.

For a semi-positive program, any relation used in a negated subgoal is an *edb*relation, so it is constant during the evaluation of P. Such subgoals can be interpreted in the immediate consequence operator as artificial negated *edb*relations, i.e. $!R(x_1, \ldots, x_k)$ holds iff $(x_1, \ldots, x_k) \notin R$. The updated definition of rule safety ensures that such subgoals do not introduce any new values into the database. Thus, semi-positive programs can be evaluated using the fix-point semantics just like positive Datalog programs.

The situation is different when *idb*relations are used in negative subgoals. Let us suppose that we use the naive evaluation. In classical Datalog, all tuples added to the database during the evaluation remain there until its end. However, when negation is allowed, it is not true in general. Let us consider a program which has a rule with a negated subgoal $!R(u)$. Such rule might produce a tuple t in iteration i because some t' is not in R and thus $!R(t')$ is true. When t' is added to relation R in the subsequent iteration though, the rule can no longer produce t. Some versions of negation semantics in Datalog allow for removing tuples from relations during the evaluation [3].

In stratified semantics, we do not allow tuples to be removed from relations. Consequently, the inflationary semantics of Datalog is preserved. To achieve that, we require that if there is a rule for computing relation R_1 that uses R_2 in a negated subgoal, then relation R_2 has to be fully computed before the evaluation of relation R_1. Intuitively, such order of computation is possible if there is no direct or indirect dependency on R_1 in any of the rules for R_2, i.e. R_1 and R_2 are not recursively dependent on each other. This is formalized by the notion of strata.

Definition 2.6.4 (Stratification)**.** Let P be a program in Datalog$^\neg$and $n = \|idb(P)\|$ be the number of *idb*relations in P. A function $\rho : sch(P) \to \{1, ..., n\}$ is called *stratification* of P iff for each rule ϕ in P with head predicate T, the following are true:

1. $\rho(R) \leq \rho(T)$ for each positive relational subgoal $R(u)$ of ϕ,
2. $\rho(R) < \rho(T)$ for each negative relational subgoal $!R(u)$ of ϕ.

A program for which a stratification exists is called *stratifiable.*

ρ corresponds to a partitioning of P into several subprograms $P_1, P_2, \dots, P_n$. Each of those programs is called a *stratum* of P. $\rho(R)$ is called *stratum number* of the relation $R \in \mathit{idb}(P)$. The i-th stratum consists of the rules from P which have a relation with stratum number i in their head. We say that these relations are *defined* in P_i. An example program with a stratification represented as a graph is presented in Figure 2.6.

$$\begin{aligned} \mathrm{P}(x) &:-\ \mathrm{A}(x). \\ \mathrm{Q}(x) &:-\ \mathrm{B}(x). \\ \mathrm{R}(x,y) &:-\ \mathrm{S}(y,x). \\ \mathrm{R}(x,y) &:-\ \mathrm{P}(x). \\ \mathrm{S}(x,y) &:-\ \mathrm{R}(x,y), Q(y). \\ \mathrm{T}(x,y) &:-\ \mathrm{R}(y,y), Q(x). \end{aligned}$$

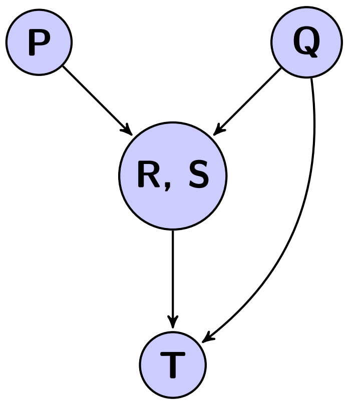

Figure 2.6: A Datalog program and one of its possible stratifications.

Stratification ensures that if a relation R is used in rules of stratum P_i in a positive subgoal, then R must be defined in this stratum or one of the previous strata. Additionally, if a relation is used in stratum P_i in a negated subgoal, then it must be defined in an earlier stratum. It is worth noting that this allows for recursive rules, unless the recursive subgoal is negated.

For each P_i, $\mathit{idb}(P_i)$ consists of relations defined in this stratum, while $\mathit{edb}(P_i)$ contains only relations defined in any of the earlier strata and relations from $\mathit{edb}(P)$. By definition of stratification, the negative subgoals in rules of P_i use only relations in $\mathit{edb}(P_i)$. Hence, each P_i is a semi-positive program and as such, it may be evaluated using the fix-point semantics.

We require the programs in Datalog$^{\neg}$to be stratifiable. If P can be stratified into $P_1, P_2, \dots P_n$, then the output of program P on input $\mathbf{I}$ is defined by applying programs $P_1, P_2, \dots P_n$ in a sequence:

$$P(\mathbf{I}) = P_n(\dots, P_2(P_1(\mathbf{I}))\dots)$$

A program can have multiple stratifications, but it can be shown that $P(\mathbf{I})$ does not depend on which of them is chosen [3].

2.6.3. Arithmetic functions

One of extensions very useful in practice are arithmetic functions. In this extension, there is a new kind of subgoal, an *assignment subgoal*, in the form of:

$$x = y \diamond z$$

where x, y, z are free variables or constants and $\diamond$ is a binary arithmetic operation like addition, subtraction, multiplication, division etc.

To give an adjusted version of the definition of rule safety 2.6.1, let us first define *input variables* and *output variables* of subgoals. Intuitively, values of all input variables need to be determined before a subgoal is evaluated. When a subgoal is evaluated, the values of all its output variables are determined. All variables appearing in a positive relational subgoal are output variables. All variables appearing in a negative relational subgoal are input variables. All variables appearing in a comparison subgoal are output variables. For an assignment subgoal, the variables on the right side of the assignment are input variables, and the one on the left side is an output variable.

Definition 2.6.5 (Rule safety)**.** A rule in Datalog$^\neg$with arithmetic comparisons and assignments is *safe* iff it satisfies the following conditions:

- each free variable appearing in its head also appears as an output variable in at least one of the subgoals,
- the subgoals can be ordered topologically, i.e. there exists an ordering $s_1, \ldots, s_n$ of the subgoals, such that for each i, any *input variable* in subgoal s_i is an *output variable* of at least one subgoal s_j for $j < i$.

Example 2.6.2. As an example, let us suppose we have a graph G defined by a relation Edge where $\textsc{Edge}(v, u, l)$ means that G has an edge from v to u of length $l > 0$. There is also a distinguished source vertex s. An interesting question is what are the minimal distances from s to all other vertices of G. We will return to this question in Section 2.6.4. For now, let us answer a simpler question: supposing that G is a directed acyclic graph, what are the lengths of paths between s and v for each vertex v in G?

The following program answers this question using a straightforward rule of edge relaxation:

$$\begin{array}{lcl}
\textsc{Path}(v, d) & :- & \textsc{Edge}(s, v, d). \\
\textsc{Path}(v, d) & :- & \textsc{Path}(t, d'), \textsc{Edge}(t, v, l), d = d' + l.
\end{array}$$

Figure 2.7: Datalog query for computing all path lengths from a given source.

It is easy to see that arithmetic addition is crucial in this program – it would not be possible to find the path lengths without being able to generate new distance values. As can be seen that both rules satisfy the updated safety definition.

The introduction of arithmetic functions significantly changes the semantics. Similarly to arithmetic comparisons, arithmetic functions can be thought of as built-in infinite relations. The difference is that we do not forbid those relations to introduce new values into the database. Given a program P and a database instance $\mathbf{K}$ over $sch(P)$, rules with arithmetic functions can produce new values, i.e. values that were not present in $adom(P) \cup adom(\mathbf{K})$. In our example, such a situation happens if there is a cycle in G reachable from the source. There is an infinite number of paths from the source to the vertices of the cycle and thus Path would be infinite.

There are different approaches to address this problem, including *finiteness dependencies* and syntactic requirements that imply safety of Datalog programs with arithmetic conditions [13, 14, 15, 16].

For the purpose of this paper, we can simply define the semantics only for Datalog programs that have a finite fixed point. The updated version of Theorem 2.4.2 is as follows.

Theorem 2.6.1. *For a program P and a finite database instance $\boldsymbol{K}$ over edb(P), if there exists $n \geq 0$ such that $T_P^n(\boldsymbol{K})$ is a fix-point of T_P, then it is the minimal fix-point of T_P containing $\boldsymbol{K}$.*

Proof. See the second part of the proof for Theorem 2.4.2. ■

2.6.4. Datalog with non-recursive aggregation

Datalog with negation and arithmetics is already a useful language, but for some queries one more feature is necessary, namely the aggregation using a certain function f. Aggregation works similarly to the Group By clause in SQL. When aggregation is applied to i-th column of a relation, all the facts in the relation are grouped by their values in the remaining columns and for each group the value in i-th column is obtained by applying the aggregation function. Let us consider the following example of relation Rel:

$$(1,5,5) \qquad (1,5,3) \qquad (1,5,4) \qquad (2,3,4) \qquad (2,3,5) \qquad (2,4,6)$$

If aggregation with function Min is applied to the last column of this relation, the result is a new relation Aggregated-Rel

$$\begin{aligned}\textsc{Aggregated-Rel}(1,5,3) &= \textsc{Aggregated-Rel}(1,5,\min\{5,3,4\})\\ \textsc{Aggregated-Rel}(2,3,4) &= \textsc{Aggregated-Rel}(2,3,\min\{4,5\})\\ \textsc{Aggregated-Rel}(2,4,6) &= \textsc{Aggregated-Rel}(1,5,\min\{6\})\end{aligned}$$

A simple version of aggregation can be introduced in Datalog by allowing the rules for aggregated relations to use only *edb*relations in subgoals. The semantics and evaluation is then straightforward. The rules can be evaluated within a single application of the T_P operator and the aggregation can be applied immediately.

This definition can be extended using the stratification method described in the previous section. Semantics for a program is defined if it can be stratified in such a way that each aggregation rule uses in its subgoals only the relations defined in the preceding strata. Aggregation of a relation from the same stratum, i.e. recursive aggregation, is much more complicated and is discussed in Chapter 4.

As an example, let us recall the program in Example 2.7, which for a given graph computes the lengths of all existing paths from the source to other vertices. A typical question is to find the length of the shortest path to each vertex from the source. This question can be answered using aggregation, by computing the minimum of distances for each vertex:

$$\begin{array}{lll}\textsc{Path}(v,d) & :- & \textsc{Edge}(s,v,d).\\ \textsc{Path}(v,d) & :- & \textsc{Path}(t,d'),\textsc{Edge}(t,v,l),d=d'+l.\\ \textsc{MinPath}(t,\textsc{Min}(d)) & :- & \textsc{Path}(t,d).\end{array}$$

Figure 2.8: Datalog query for computing all path lengths from a given source.

The semantics of this syntax is that after inferring all possible facts using the rule for MinPath, this relation should be aggregated using the minimum function. The restriction for this syntax is that if there is aggregation used in a rule for some relation, there can be no other rules for this relation.

In the example presented above, there are two strata, where EDGE is an *edb*relation, PATH belongs to the first stratum and MINPATH belongs to the second stratum. Hence, MINPATH can be computed after computation of PATH is finished.

Chapter 3

The BSP model, Pregel and Apache Spark

In the recent years, several models and many technologies that help efficiently perform iterative computations on large datasets, especially graphs, have been developed.

One of the most modern ones, Apache Spark, allows to express algorithms as transformations of RDDs, which are an abstraction of a distributed dataset. Its distinctive feature is that most of the data during the computation can be kept in worker nodes' memory, which makes it significantly faster than systems like Hadoop MapReduce. It is capable of handling wide range of tasks, including distributed graph computations using the GraphX extension.

Pregel is a computational model designed specifically for large graph computations, introduced in 2010 by Google engineers [7]. Its goal is to streamline implementation of graph algorithms by providing a framework which lets the programmer forget about distributing the computation, implementing the graph topology and addressing fault tolerance issues and focus on the problem at hand. Pregel is based on the classical BSP model [25] for iterative distributed computations.

Since the introduction of Pregel, there have been many systems developed based on this model, most notably an open source implementation of the Pregel model, i.e. Apache Giraph. Pregel model of computations has also been incorporated into other, more general frameworks, including Spark. There have also been extensions to the model, such as Giraph++ [8].

Before Pregel and Spark, there were available graph algorithm libraries such as BGL [26] and GraphBase [28] which were designed for a single computer and thus limited in the scale of problems they could solve, and parallel graph frameworks such as Parallel BGL [27], which did not address issues crucial in large data processing, like fault-tolerance. Graph algorithms also used to be expressed as a series of MapReduce iterations, but this, however, adds a significant overhead due to the need to dump the state of computation to disk after each iteration, given that graph algorithms usually need multiple iterations.

In Section 3.1, the Pregel model is described. The RDD model employed in Apache Spark and the Spark GraphX extension for graph computations, which supports the Pregel model, is covered in Section 3.2.

3.1. Pregel model and its original implementation

The name of the Pregel model comes from its initial proprietary implementation by Google and honors Leonard Euler, a famous Swiss mathematician and physicist and also a pioneer of the graph theory. In 1735, he formulated the first theorem in graph theory: a solution to an old question whether it was possible for a Koenigsberg citizen to take a walk around the city so that he crossed each of the seven city's bridges exactly once. Euler concluded that it was impossible, because the graph bridges form is not what we today call an Euler graph — a graph in which every vertex has an even degree. The name of the river flowing through Koenigsberg and spanned by the famous bridges is Pregel.

The model of computation in Pregel is based on the L. Valiant's Bulk Synchronous Parallel model [25]. The computation is performed in a sequence of *supersteps*. In each superstep, the framework executes on each vertex a *vertex program* provided by the user. Vertices communicate by messages: a message sent by a vertex in superstep S is delivered to its recipient in superstep $S + 1$. Figure 3.1 presents the phases in a single iteration of program execution in Pregel.

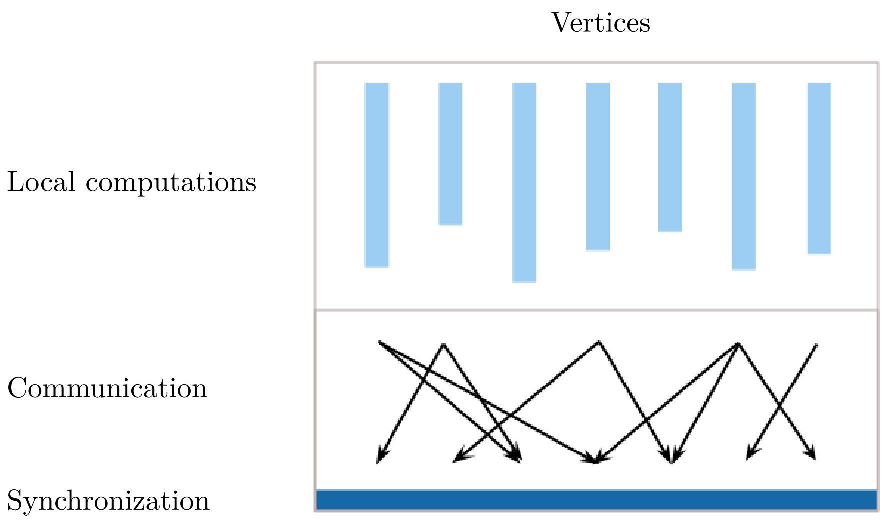

Figure 3.1: One superstep in Pregel as a BSP iteration.

The main concept in implementing algorithms on Pregel is to "think like a vertex". User is required to express the algorithm as a function executed locally on each vertex, where communication between vertices is allowed only across supersteps. Those local functions are then combined by the framework in an efficient way to perform the whole computation. This approach, similar to the MapReduce model, is well suited for distributed computations, since all local functions can be executed in a fully independent way. At the same time, the synchronous structure of computation makes it easier to reason about the semantics of a program than in asynchronous systems and allows for fault-tolerance mechanisms.

A Pregel program takes a directed graph as an input and performs computations that are allowed to modify this graph. Each vertex of the graph has a unique, constant *vertex identifier* and is associated with some *vertex data*, which can be modified during the computation. Vertex also has *outgoing edges*, each of which has a target vertex and some modifiable *edge data*. The algorithm logic is described using the *vertex program*.

A computation is performed as a sequence of *supersteps*. In each superstep the vertex program is concurrently executed on each vertex. The program is the same for each vertex, but can depend on the vertex identifier. The program being executed on vertex V receives messages sent to V in the previous superstep. It can modify the vertex data and the data of its outgoing edges, send messages to other vertices to be delivered in the next superstep and change the topology of the graph by adding or removing vertices or edges. A vertex can send messages not only to its neighbors, but also to other vertices if it knows their identifiers.

The termination criterion is distributed. A vertex may *vote to halt.* Initially, all vertices are in the *active* state. If a vertex votes to halt, its state changes to *inactive.* If an inactive vertex receives a message from another vertex, it is moved back to the active state. The vertex program is executed only on the active vertices and the computation is terminated when all vertices are in the inactive state.

According to the original definition, the result of a computation are the values explicitly output by the vertices, but in most scenarios the graph state after the last superstep is assumed to be the output of the algorithm.

In practice, computations are performed on a number of workers much smaller than the number of vertices in the graph. This allows distributing the vertices between workers in a workload-balanced manner.

Let us consider the following example: for a strongly connected graph with an integer value assigned to each node, compute the minimum of those values. This can be implemented in Pregel using the vertex program presented in Figure 3.2.

```
MAX-VALUE-VERTEX($vertex, superstepNumber, incomingMessages$)
1  $newValue \leftarrow \max(incomingMessages \cup \{vertex.value\})$
2  if $superstepNumber = 0$ or $newValue > vertex.value$
3      $vertex.value \leftarrow newValue$
4      foreach $edge \in vertex.outgoingEdges$
5          SEND-MESSAGE($edge.targetVertex, newValue$)
6  else
7      VOTE-TO-HALT()
```

Figure 3.2: Pregel vertex program for computing maximum value among graph nodes

In the first superstep, a vertex sends its value to all neighbors and votes to halt. Upon the reception of any new values, a vertex is activated and if the received values are greater than the value stored in the vertex, it is updated and messages with the new value are sent. An example computation is presented in Figure 3.3. This picture comes from the original white paper about Pregel [7].

As another example, let us see how single source shortest paths can be computed using Pregel. We assume that the value for each vertex is initially set to ∞. In the first superstep the source vertex updates its value to 0 and sends messages to its neighbors with a new distance. In the following supersteps other vertices update their distances and, if it changed, send messages to their neighbors with a new possible distance. Each active vertex votes to halt in each superstep, so when no more messages with distance updates are sent, the algorithm terminates. This will always happen in a finite number of supersteps, as long as all edges have non-negative lengths. At the end of computation, each vertex has its minimum distance from $SOURCE$ associated with it, or ∞ if it is not reachable.

An important goal in large dataset computations is to achieve fault-tolerance, so that the computation can be continued in case of a failure of some of the machines in the cluster. In Pregel, this is achieved by *checkpointing.* Once in a few supersteps, the workers are required to save their state to the disk. When any of the workers fail, the computation is resumed from the last checkpoint.

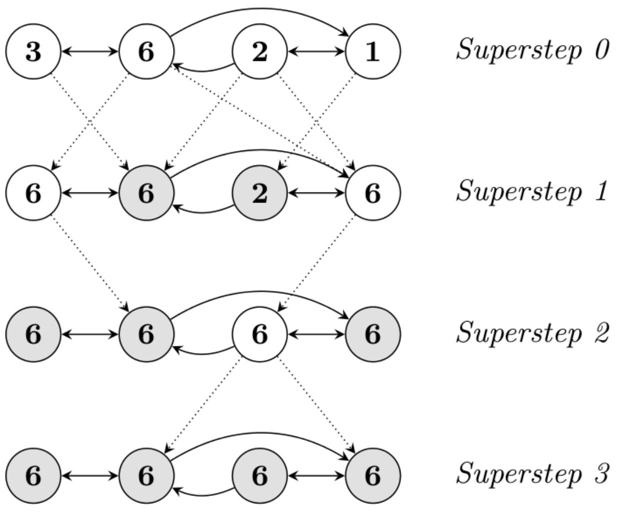

Figure 3.3: Example Pregel computation of maximum value among the nodes. Nodes that voted to halt are marked gray. Source: [7]

```
SHORTEST-PATHS-VERTEX(vertex, superstepNumber, incomingMessages)
1 initialDistance ← if (vertex.id = SOURCE) 0 else $\infty$
2 minDistance ← min(incomingMessages $\cup$ {initialDistance})
3 if newValue < vertex.value
4       vertex.value ← minDistance
5       foreach edge $\in$ vertex.outgoingEdges
6             SEND-MESSAGE(edge.targetVertex, minDistance + edge.length)
7 VOTE-TO-HALT()
```

Figure 3.4: Pregel vertex program for shortest paths from single source to all other vertices

In addition to the general model, Pregel also has some additional features enhancing usability and efficiency, such as *aggregators* for efficiently gathering and broadcasting global values and *combiners* which can reduce the network bandwidth used by merging messages sent from vertices placed on a given node.

The original implementation by Google engineers is proprietary and has never been released to the public. It is written entirely in C++ and tightly connected to the internal Google infrastructure, including the distributed file systems and the execution environment.

3.1.1. Giraph

Giraph [31] is an open-source implementation of the Pregel model. It adds several extensions to the version originally described by Google. Those extensions include introducing the possibility to perform computations on the master node, capabilities to work with support of external memory and removing the single point of failure by adding spare master nodes which can become active when the primary master node fails.

Giraph is built on top of Hadoop [33], the widely-adopted framework for large datasets processing. Hadoop is a platform for big datasets computing at scale, consisting of HDFS — distributed file system, YARN — framework for managing the cluster and scheduling tasks and Hadoop MapReduce — an open implementation of the MapReduce model and libraries for using these elements in other projects.

The development of Giraph was started by Yahoo!. The project was donated to the Apache Foundation as an incubator project in 2011. It became a Top Level Project of the Apache Foundation in 2012. A stable version 1.0 was released in 2013. There are several large companies including Facebook, Twitter and LinkedIn actively using the project and engaged in its development. The most active user is Facebook, which executes Giraph programs on social graphs with up to 10^{12} edges [35]. In 2013 Facebook published an article [35] stating that analyzing so large graphs was impossible with the software available in 2012. The article describes the optimizations and enhancements made in Giraph to enable it to run jobs with this amount of data on their infrastructure:

- Flexible graph input: vertices and edges can be loaded from several sources and the framework takes care of distributing them correctly before the start of computation.
- Multithreaded execution on a single machine which allows for better resource sharing than in case of workers distributed across different machines.
- Memory usage optimization: by serializing the transferred data using primitive types instead of Java objects, the memory usage was significantly reduced, resulting in 10 times lower execution time.
- Sharded aggregators: instead of aggregators being stored and distributed by the master node, they are stored in a distributed way, which scales better for very large datasets.

It is worth mentioning that one of the Google Pregel creators and the primary author of the original white paper about it, Grzegorz Malewicz, is a member of the Facebook's data infrastructure graph processing team which develops Giraph [35].

3.2. Spark

Apache Spark [20, 21] is an open-source framework for distributed data analysis. It was started in 2009 in the AMPLab at the University of California, Berkeley. Since that time it has gained a huge momentum and has a quickly growing community of users and contributors [37]. Since 2009, over 250 individual developers have contributed to Spark, and its permanent contributors come from 12 companies and institutions [32]. Among others, it is used by companies such as Yahoo, IBM, Intel, Alibaba, Cloudera and Databricks.

Spark's computational model supports more specialized models such as MapReduce and Pregel and is also suitable for new applications that these systems do not support, like interactive data mining and stream data processing.

The two key advantages of Spark are:

- its impressive speed: Spark is in some cases even 100 times faster than equivalent computations in Hadoop MapReduce,
- its simplicity and ease of use: it offers APIs in Scala, Java and Python which allow developers to quickly develop programs performing even complicated calculations and a standalone running mode which lets developers set up environment and prototype programs locally without the need to set up a Hadoop cluster.

Similarly to Apache Giraph, Spark is a Top-Level Project of the Apache Foundation. Prior to its promotion as a Top-Level Project in February 2014 [36], it had been in the Apache Incubator program since June 2013.

Spark fits into the Hadoop ecosystem by being able to run on Hadoop clusters without any additional installation and supporting data input from various Hadoop data stores such as HDFS, HBase and Cassandra. It is not tied, however, to Hadoop infrastructure: it can also run as a standalone deployment in a cluster or on other distributed platforms such as Mesos [38] and Amazon EC2 [39].

3.2.1. Resilient Distributed Datasets

The key concept in Spark is the *Resilient Distributed Dataset* (RDD). RDDs are an abstraction of distributed memory, which let the programmer perform distributed computations. They are stored in a way that is transparent to the user and assures fault tolerance.

RDDs provide only a *coarse-grained* interface, i.e. operations that apply to the whole dataset, such as map, filter and join. This allows for achieving fault-tolerance by storing only the history of operations that were used to build a dataset, called its *lineage*, instead of replicating the data to be able to recover it. An additional advantage is that the RDDs do not need to be materialized, unless it is actually necessary. Since parallel computations generally apply some transformations to multiple elements of a dataset, in most cases they can be expressed easily with coarse-grained operation on datasets.

RDDs can be created only in two ways: by loading a dataset from a distributed storage or from another dataset by applying coarse-grained functional operations called *transformations*, such as *map*, *filter* and *join*. For greater efficiency, the user can also control the *persistence* by indicating which RDDs are intended to be used in the future and as such should be kept in the memory and the *partitioning* of each RDD by indicating the key by which the records of RDD should be partitioned across the machines. Finally, the user can perform *actions* on an RDD. Actions return a value or export the dataset to some persistent storage. Available actions include *count*, which returns the number of elements in the RDD, *collect*, which returns the records from the dataset and *save*, which exports the records to an external storage.

Example 3.2.1. Let us suppose we have an RDD *friends*, which consists of pairs of identifiers (a, b) meaning that person b is a friend of person a. We will say that y is a *friend of friend* of x iff there exists z, called a *proxy*, such that z is a friend of x and y is a friend of z. We will call a person with more than 10000 friends of friends an *influencer*. A Spark program which computes the number of all influencers is presented in Figure 3.5.

```
COUNT-INFLUENCERS(friends)
1  connections ← friends.map($(x, y) \mapsto (y, x)$).join(friends)
2  friendsOfFriends ← connections.map($(z, (x, y)) \mapsto (x, y)$).distinct()
3  fofCount ← friendsOfFriends.map($(x, y) \mapsto (x, 1)$).reduceByKey($(m, n) \mapsto m + n$)
4  influencers ← fofCount.filter($(x, c) \mapsto count > 10000$))
5  return influencers.count()
```

Figure 3.5: Example of computation in Spark: computing the number influencers.

The COUNT-INFLUENCERS function first performs a join of *friends* with itself. Since the join transformation joins by the first value in the pair, the elements of *friends* are swapped beforehand. The result is then mapped to remove the information about proxies, which is irrelevant, and duplicates are eliminated with *distinct.* This gives an RDD of pairs (x, y), such that y is a friend of friend of x. For each x, the number of friends of friends is then computed using *map* and *reduceByKey.* The obtained RDD of pairs (x, c), where x is the person and c is its number of friends of friends is then filtered so that only influencers remain. The size of the filtered RDD is returned as the number of influencers. It is worth noting that the transformations used, i.e. *map, filter, distinct* and *reduceByKey*, are not actually executed until the action *count* is performed.

Iterative computation can be achieved in Spark by simply placing the transformations within a loop. For performance reasons, the user will often need to manage which RDDs should be cached in memory and explicitly materialize some of them in each iteration.

3.2.2. Higher level tools

Spark offers a set of high level tools that demonstrate the capabilities of the Resilient Distributed Dataset model. Those tools are implemented as relatively small libraries on top of Spark's core. The following tools are currently available:

- Spark SQL, allowing for seamless integration of SQL queries into Spark programs,
- Spark Streaming, which supports working on on-line streams of data,
- MLlib, which is an implementation of common machine learning algorithms for classification, regression, clustering and dimensionality reduction,
- GraphX, providing an interface for creating efficient graph algorithms, integrated with pre- and post- processing with regular Spark transformations.

3.3. Other frameworks

Some of the other notable examples of systems designed for graph computations include the Graph Processing System, GraphLab and Giraph++.

Graph Processing System [9] is similar to Pregel and Giraph, but offers additional capabilities of expressing computations in a non-vertex-centric way, dynamically repartitioning vertices among workers based on their communication patterns and partitioning of adjacency lists of high-degree vertices. Giraph++ [8] proposes a ”think like a graph” model, as opposed to Pregel's ”think like a vertex”, to allow for algorithm-specific optimizations by giving more control over partitioning of data. In GraphLab [17, 19, 18], computations are expressed using *update functions*, which modify vertices data and schedule further executions of update functions. This approach has several advantages, for example the ability to read the data in adjacent vertices other than by receiving a message from them, but can result in a large overhead due to scheduling.

Chapter 4

SociaLite and Datalog with recursive aggregation

While Datalog allows to express some of graph algorithms in an elegant and succinct way, many practical problems cannot be efficiently solved with Datalog programs.

SociaLite [1, 2] is a graph query language based on Datalog. It allows a programmer to write intuitive queries using declarative semantics, which can often be executed as efficiently as highly optimized dedicated programs. The queries can also be executed in a distributed environment.

The most significant extension over Datalog in SociaLite is the ability to combine recursive rules with aggregation. Under some conditions, such rules can be evaluated incrementally and thus as efficiently as regular recursion in Datalog.

[1] introduces *Sequential SociaLite*, intended to be executed on one machine. We cover its most important feature, namely recursive aggregation, in Section 4.1. Two other extensions over Datalog are tail-nested tables which improve the data layout in the memory and an option to provide hints regarding the execution order. [1] describes the language proposition and possible optimizations, but does not define the language precisely. The paper outlines conditions for the programs to be correct, but only in a simplified way and lacks definitions of some of the notions used, which makes the exact meaning of theorems and proofs unclear.

[2] extends Sequential Socialite to *Distributed SociaLite*, executable on a distributed architecture. The presented version of the interpreter for Distributed SociaLite was built as an independent implementation based directly on Hadoop. Distributed SociaLite, described in section 4.2, introduces a *location operator*, which controls how data is distributed among workers and describes an independent implementation of the language.

In this chapter, we give precise definitions for a Datalog language extended with recursive aggregate functions, which is the main extension over Datalog in SociaLite. We also analyze the conditions that such programs have to satisfy in order to have unambiguous solution.

4.1. Datalog with recursive aggregation

In this section we introduce the recursive aggregation extension to Datalog. Since SociaLite consists of several extensions to Datalog, not only of recursive aggregation, we will call the language defined here *Datalog with recursive aggregation*, abbreviated Datalog^{RA} .

4.1.1. Motivation

Most graph algorithms are based on some kind of iteration or recursive computation. Examples of such algorithms are the Dijkstra algorithm for single source shortest paths and PageRank. Although simple recursion can be expressed easily in Datalog, it is usually difficult or impossible to express such algorithms in Datalog efficiently, as it would require computing much more intermediate results than actually needed to obtain the solution. We will explain this on an example: a simple program that computes the shortest paths from a source node.

A straightforward program in Datalog extended with non-recursive aggregation for computing single source shortest paths starting from node 1 is presented in Figure 4.2. Due to the limitations of Datalog, this program computes all possible path lengths from node 1 to other nodes in the first place, and after that for each node the minimal distance is chosen. Not only does this approach result in a bad performance, but the program executes indefinitely if a loop in the graph is reachable from the source node.

$$\begin{array}{lll} \textsc{Path}(t, d) & :- & \textsc{Edge}(1, t, d). \\ \textsc{Path}(t, d) & :- & \textsc{Path}(s, d_1), \textsc{Edge}(s, t, d_2), d = d_1 + d_2. \\ \textsc{MinPath}(t, \textsc{Min}(d)) & :- & \textsc{Path}(t, d). \end{array}$$

Figure 4.1: Datalog query for computing the shortest paths from node 1 to other nodes.

4.1.2. A program in Datalog^{RA}

Datalog^{RA} allows aggregation to be combined with recursion under some conditions. This allows for writing straightforward programs for problems such as single source shortest paths, which finish execution in finite time and are often much more efficient than Datalog programs. An example Datalog^{RA} program that computes shortest paths from vertex 1 to all vertices is presented in Figure 4.2.

$$\begin{array}{l} \textsc{Edge}(\text{int } \mathit{src}, \text{int } \mathit{sink}, \text{int } \mathit{len}) \\ \textsc{Path}(\text{int } \mathit{target}, \text{int } \mathit{dist} \text{ aggregate } \textsc{Min}) \end{array}$$

$$\begin{array}{lll} \textsc{Path}(t, d) & :- & \textsc{Edge}(1, t, d). \\ \textsc{Path}(t, d) & :- & \textsc{Path}(s, d_1), \textsc{Edge}(s, t, d_2), d = d_1 + d_2. \end{array}$$

Figure 4.2: Datalog^{RA} query for computing the shortest paths from node 1 to other nodes

The relation PATH is declared so that for each *target* the values in *dist* column are aggregated using minimum operator MIN, i.e. for each *target* the minimum of the corresponding *dist* values is chosen by applying MIN to the set of those values.

A Datalog^{RA} program P is a Datalog program, with additional aggregation function defined for selected columns of some relations.

For each relation name $R \in idb(P)$, one column $aggcol_R(P) \in 1, \ldots ar_R$ can be chosen and assigned an aggregation function $aggfun_R(P)$. This column is called the *aggregated column*. The remaining columns are called the *qualifying columns*. Intuitively, after each step of computation, we group the facts in the relation by the qualifying columns and within each group we aggregate the values in the aggregated column using $aggfun_R(P)$. Value $aggcol_R(P) = \mathbf{none}$ means that R is a regular relation with no aggregation. Program P can usually be clearly determined from the context. We will then simply write $aggcol_R, aggfun_R$ instead of $aggcol_R(P), aggfun_R(P)$.

To simplify the notation, we assume that if a relation has an aggregated column, then it is always the last one: $aggcol_R = ar_R$.

The aggregation function needs to fulfill certain requirements that will be stated in the next section.

Syntactically, we require that each *idb*relation is declared at the top of the program. In the declaration of a relation, an aggregated column can be specified by adding keyword *aggregate* and the name of the aggregate function next to the column declaration. This syntax allows for providing multiple rules for each aggregated relation in the program.

An example of a program in Datalog^{RA} is shown in Figure 4.3.

P(int a, int b aggregate F)
R(int src, int $sink$, int len)

P(x, y)	$:-$	$P(x, z),\ R(x, y, z).$
R(x, y, z)	$:-$	$R(x, y, y),\ R(x, z, z).$

Figure 4.3: Example program in Datalog^{RA} .

4.1.3. Aggregation-aware order over databases for inflationary Datalog^{RA}

While being very useful, recursive aggregation rules not always have an unambiguous solution. This is the case only under some conditions on the rules and the aggregation function itself.

Typically, Datalog programs semantics is defined using the fixed point of the immediate consequence operator T_P. This definition assumes that T_P is inflationary with respect to the inclusion order on database instances. This requirement means that T_P only adds facts to the database instance, but never removes them. This is also the reason for which program 4.1 is inefficient: the inflationary semantics forces all suboptimal distances to vertices to be kept in the database and as such, used in subsequent iterations.

When recursive aggregate functions are allowed, the semantics is not inflationary with respect to the inclusion order. A fact in the database can be replaced with a different one because a better aggregated value has appeared. However, an inflationary T_P operator is necessary to prove that the fixed point semantics always gives a unique solution. In order to define semantics for Datalog^{RA} in terms of a fixed point, we need to use a different order on database instances than the regular set inclusion order.

In this section, we describe the idea introduced in [1]. However, the original description lacks precision and details. Here we give the definitions in a precise way.

First, we define what a *join operation* is and show the order that it induces. Then, we show that if the aggregation function is a join operation and corresponding rules are monotone with respect to the order induced by this operation, then the result of the program is unambiguously defined by the least fixed-point. We also show that it can be computed efficiently using the semi-naive evaluation.

[1] uses notions of *meet operation* and greatest fixed point instead. We chose the dual notions of join operation and least upper bound, as we find that they give more clear definitions and proofs.

Join operation and induced ordering

We start by recalling the definitions of idempotency, commutativity and associativity for binary operations.

Definition 4.1.1. A binary operation $\odot : X \times X \to X$ is:

- *idempotent* iff $x \odot x = x$ for each $x \in X$,
- *commutative* iff $x \odot y = y \odot x$ for each $x, y \in X$,
- *associative* iff $(x \odot y) \odot z = x \odot (y \odot z)$ for each $x, y, z \in X$.

A core concept in Datalog^{RA} is the *join operation.* Join operations have the basic properties that are sufficient for performing unambiguous aggregation.

Definition 4.1.2 (Join operation)**.** A binary operation is a *join* operation iff it is idempotent, commutative and associative.

We usually denote a join operation with the symbol $\sqcup$. An example of a join operation is the maximum of two numbers.

Example 4.1.1. $\max(a, b)$ for $a, b \in \mathbb{N}$ is a join operation; it is:

- idempotent — $\max(a, a) = a$,
- commutative — $\max(a, b) = \max(b, a)$,
- associative — $\max(a, \max(b, c)) = \max(\max(a, b), c)$.

Similarly, the minimum of two numbers is also a join operation. On the contrary, $+$ is not a join operation, since it is not idempotent: $1 + 1 \neq 1$.

Since join operation is associative, we can skip parentheses and write: $a_1 \sqcup a_2 \sqcup \cdots \sqcup a_n$ instead of: $a_1 \sqcup (a_2 \sqcup (\cdots \sqcup a_n) \ldots)$.

Join operation can be extended to finite sets of values in a straightforward way:

$$\sqcup(\{a_1, \ldots, a_n\}) = a_1 \sqcup a_2 \sqcup \cdots \sqcup a_n$$

Order induced by a join operation

A join operation over a set P induces a partial order on P that has some useful properties. In particular, P with this ordering is a semi-lattice. We start by recalling the definitions of a least upper bound and a semi-lattice.

Definition 4.1.3 (Upper bound)**.** Given a set S with a partial order $\leq$ over S, c is an *upper bound* of S iff $\forall_{a \in S} a \leq c$.

Definition 4.1.4 (Least upper bound)**.** Given a set S with a partial order $\leq$ over S, $m \in S$ is the *least upper bound* of S iff it is an upper bound of S and for each upper bound c of S, $m \leq c$.

Definition 4.1.5 (Join semi-lattice)**.** A set P with a partial order $\leq$ over P is a *join semi-lattice* iff every two elements of P have a least upper bound with respect to $\leq$.

For any two elements, their least upper bound is called a *join* of those elements.

A join operation $\sqcap$ induces a partial order $\leq_\sqcap$ over its domain, such that $a \leq_\sqcap b \iff b = a \sqcap b$. This relation is indeed a partial order, as it is:

- reflexive — since $\sqcap$ is idempotent, it holds that $a = a \sqcap a$, so $a \leq_\sqcap a$,
- transitive — if $a \leq_\sqcap b$ and $b \leq_\sqcap c$, then $b = a \sqcap b$ and $c = b \sqcap c$ and by combining we have that $c = a \sqcap b \sqcap c = a \sqcap c$, so $a \leq_\sqcap c$,
- antisymmetric – if $a \leq_\sqcap b$ and $b \leq_\sqcap a$, then $b = a \sqcap b$ and $a = b \sqcap a$, so $a = b$, because $\sqcap$ is commutative.

A join operation $\sqcap$ defines a semi-lattice with its induced order $\leq_\sqcap$, because the result of the operation for any two elements is the least upper bound of those elements with respect to $\leq_\sqcap$.

For example, the join operation max over natural number induces the partial order $\leq$. For any two $a, b \in \mathbb{N}$, $\max(a, b)$ is their least upper bound with respect to $\leq$.

Aggregation operator g_R

An important step in the evaluation of a Datalog^{RA} program is grouping the facts in an instance of each relation and performing the aggregation within each group. We can put that into a formal definition as function g_R. g_R takes as an input a relation instance that may contain multiple facts with the same values in qualifying columns and performs the aggregation on the aggregated column within each such group.

Definition 4.1.6 (Aggregation operator over relations)**.** For non-aggregated relations, g_R is an identity function. For a relation R of arity $k = ar(R)$ with an aggregated column, let us define $g_R : \mathit{inst}(R) \to \mathit{inst}(R)$ as:

$$g_R(I) = \Big\{(x_1, \ldots, x_{k-1}, t) : (x_1, \ldots, x_{k-1}, x_k) \in I \wedge t = \mathit{aggfun}_R(\{y : (x_1, \ldots, x_{k-1}, y) \in I\})\Big\}$$

Pre-order over relation instances

We can prove that there exists a unique least fixed point for any Datalog program. The fundamental fact needed for this proof is that Datalog semantics is inflationary, that is during the iterative evaluation of any Datalog program, if the state of a relation is I_1 in some step and I_2 at a later step, we know that $I_1 \subseteq I_2$. In Datalog^{RA} this property no longer holds. A fact in I_1 can be replaced with a different fact with a lower value in the aggregated column. To be able to define the semantics of programs in Datalog^{RA} using a least fixed point, we need to use a custom order on relation instances. This order is built based on the function g_R.

Definition 4.1.7. Let R be a relation name and $k = ar(R)$. Let us define comparison $\sqsubseteq_R$ on relation instances as follows:

$$I_1 \sqsubseteq_R I_2 \iff \begin{cases} \forall_{(q_1,\ldots,q_{k-1},v)\in g_R(I_1)} \exists_{(q_1,\ldots,q_{k-1},v')\in g_R(I_2)} v \leq_{aggfun_R} v' & \text{if } aggcol_R = k \\ I_1 \subseteq I_2 & \text{if } aggcol_R = \textbf{none} \end{cases}$$

The relation satisfies the reflexivity and transitivity requirements, so it is a pre-order.

Lemma 4.1.1. *For any R and $inst(R)$, $\sqsubseteq_R$ is a pre-order over $inst(R)$.*

Proof. If R does not have an aggregated column, $\sqsubseteq_R$ is the same as the inclusion order $\subseteq$, which is a partial order.

If R does have an aggregated column, then:

- $\sqsubseteq_R$ is reflexive: for each R, $inst(R)$ and $A \in inst(R)$, since $\leq_{aggfun_R}$ is reflexive, we have that $\forall_{(q_1,\ldots,q_{k-1},v)\in g_R(A)} \Big((q_1, \ldots, q_{k-1}, v) \in g_R(A) \text{ and } v \leq_{aggfun_R} v \Big)$. Hence, $A \sqsubseteq_R A$.

- $\sqsubseteq_R$ is transitive: if $A \sqsubseteq_R B$ and $B \sqsubseteq_R C$, then by definition of $\sqsubseteq_R$ we have that:

$$\forall_{(q_1,\ldots,q_{n-1},a)\in g_R(A)} \exists_{(q_1,\ldots,q_{n-1},b)\in g_R(B)}\ a \leq_{aggfun_R} b, \text{ and}$$
$$\forall_{(q_1,\ldots,q_{n-1},b)\in g_R(B)} \exists_{(q_1,\ldots,q_{n-1},c)\in g_R(C)}\ b \leq_{aggfun_R} c$$

  By combining it follows that $\forall_{(q_1,\ldots,q_{n-1},a)\in g(A)} \exists_{(q_1,\ldots,q_{n-1},c)\in g(C)} a \leq_{aggfun_R} c$, since $\leq_{aggfun_R}$ is a partial order and thus transitive. Therefore, $A \sqsubseteq_R C$. ■

Example 4.1.2. Let R be a relation with arity 3, with the last column aggregated using the join operation max. For the operation max in $\mathbb{N}$, $\leq_{\max}$ is the usual order $\leq$. The following hold:

- $\{(1, 2, 3)\} \sqsubseteq_R \{(1, 2, 5)\}$, because $3 \leq 5$,

- $\{(1, 2, 3)\} \sqsubseteq_R \{(1, 2, 5), (1, 7, 2)\}$, because $3 \leq 5$,

- $\{(1, 2, 5)\} \sqsubseteq_R \{(1, 2, 3), (1, 2, 8)\}$, since $g_R(\{(1, 2, 3), (1, 2, 8)\}) = \{(1, 2, 8)\}$ and $5 \leq 8$,

- $A = \{(1, 2, 3), (2, 8, 1)\}$ and $B = \{(1, 2, 5), (1, 7, 2)\}$ cannot be compared, because $\forall_x (1, 7, x) \notin A$ and $\forall_x (2, 8, x) \notin B$,

- for any R an empty relation instance $\emptyset$ is smaller under $\sqsubseteq_R$ than any other relation instance,

- if $I = \{(1, 2, 3), (1, 2, 8)\}$ and $J = \{(1, 2, 3), (1, 2, 7)\}$, we have that $I \sqsubseteq_R J$ and $J \sqsubseteq_R I$, but clearly $I \neq J$.

The last point of example 4.1.2 shows that $\sqsubseteq_R$ is not necessarily a partial order over the set of all relation instances $inst(R)$, because it is not antisymmetric.

Partial order over relation instances after aggregation

We already know that $\sqsubseteq_R$ is a pre-order over the set of all possible relation instances $\mathit{inst}(R)$, but because of the lack of antisymmetry, it is not guaranteed to be a partial order. However, if we restrict to the relation instances resulting from applying aggregation, the relation is antisymmetric.

For any R let Z_{R} denote the set of relation instances that can be obtained by application of g_{R} to some instance of R:

$$Z_{\mathrm{R}} = \{g_{\mathrm{R}}(I) : I \in \mathit{inst}(\mathrm{R})\}$$

We will refer to elements of Z_{R} as *relations after aggregation.*

Lemma 4.1.2. *For any R such that $k = ar(R)$ and $aggcol_{\mathrm{R}} \neq$ **none**, if $I \in Z_R$, then $g_R(I) = I$ and for each $x_1, \ldots, x_{n-1}$ there is at most one x_n such that $(x_1, \ldots, x_{n-1}, x_n) \in I$.*

Proof. Since we know that $I = g_R(I')$ for some I', there is at most one fact $R(x_1, \ldots, x_n)$ in I for each $(x_1, \ldots, x_{n-1})$, because g_R by definition groups facts by qualifying columns $x_1, \ldots, x_{n-1}$ and computes a single aggregated value x_n for each such group. Hence, after the next application of g_R to I the aggregated value for each $x_1, \ldots, x_{n-1}$ is simply x_n, so $g_R(I) = I$. ■

Theorem 4.1.3. *For any R such that $k = ar(R)$, $\sqsubseteq_R$ is a partial order over Z_R.*

Proof. If R does not have an aggregated column, $\sqsubseteq_R$ is the same as inclusion order $\subseteq$, which is a partial order.

If R has an aggregated column, then by Lemma 4.1.1, $\sqsubseteq_R$ is a pre-order over $\mathit{inst}(R)$. This implies that it is also a pre-order over Z_R, since it is a subset of $\mathit{inst}(R)$, so it only remains to be shown that $\sqsubseteq_R$ is antisymmetric over Z_R.

Let A, B be any relation instances from Z_R. Let us suppose that $A \sqsubseteq_R B$ and $B \sqsubseteq_R A$. To prove antisymmetry, we need to show that $A = B$. By definition of $\sqsubseteq_R$, we have that:

$$\forall_{(q_1,\ldots,q_{n-1},a)\in g_R(A)} \exists_{(q_1,\ldots,q_{n-1},b)\in g_R(B)} a \leq_{\mathit{aggfun}_R} b, \text{ and}$$
$$\forall_{(q_1,\ldots,q_{n-1},b)\in g_R(B)} \exists_{(q_1,\ldots,q_{n-1},a)\in g_R(A)} b \leq_{\mathit{aggfun}_R} a$$

Since $A, B \in Z_R$ we know that there exist A', B' such that $A = g_R(A'), B = g_R(B')$. By Lemma 4.1.2 $g_R(g_R(A')) = g_R(A')$ and thus $g_R(A) = g_R(g_R(A')) = g_R(A') = A$, and similarly $g_R(B) = B$, so the formulas above are equivalent to:

$$\forall_{(q_1,\ldots,q_{n-1},a)\in A} \exists_{(q_1,\ldots,q_{n-1},b)\in B}\ a \ \leq_{\mathit{aggfun}_R}\ b, \text{ and}$$
$$\forall_{(q_1,\ldots,q_{n-1},b)\in B} \exists_{(q_1,\ldots,q_{n-1},a)\in A}\ b \ \leq_{\mathit{aggfun}_R}\ a$$

Let $t = R(x_1, \ldots, x_{n-1}, a)$ be any fact in A. We know that there exists b such that $(x_1, ..., x_{n-1}, b) \in B$ and $a \ \leq_{\mathit{aggfun}_R}\ b$. Further, we know that there exists $(x_1, ..., x_{n-1}, a') \in A$ such that $b \ \leq_{\mathit{aggfun}_R}\ a'$. By Lemma 4.1.2, it must hold that $a = a'$, so $a \leq_{\mathit{aggfun}_R} b \leq_{\mathit{aggfun}_R} a$, and because $\leq_{\mathit{aggfun}_R}$ is a partial order we have that $a = b$, so $t \in B$. Therefore, $A \subseteq B$, because t was chosen as an arbitrary element of A. By symmetry of the proof, it also holds that $B \subseteq A$, so $A = B$. This means that $\sqsubseteq_R$ is indeed antisymmetric.

As $\sqsubseteq_R$ is an antisymmetric pre-order over Z_R, it is a partial order over this set. ■

Order over database instances

In regular Datalog, database instances can be compared using the inclusion order. We can extend the custom order $\sqsubseteq_R$ defined on relation instances to an order on database instances in a straightforward way, by comparing the databases relation-by-relation:

Definition 4.1.8. Let σ be a database schema and $\mathbf{K}, \mathbf{L}$ be database instances over σ. Let $R_1, \ldots, R_n$ be relation names in σ. By definition, $\mathbf{K}$ is a union of relation instances $I_1, \ldots I_n$ over $R_1, \ldots, R_n$, respectively. Similarly, $\mathbf{L}$ is a union of relation instances $J_1, \ldots J_n$ over $R_1, \ldots, R_n$, respectively. Let the order $\sqsubseteq_\sigma$ on database instances over σ be defined as:

$$\mathbf{K} \sqsubseteq_\sigma \mathbf{L} \iff \forall_{i=1,\ldots,n}\ I_i \sqsubseteq_{R_i} J_i$$

4.1.4. Semantics for Datalog^{RA} programs

In this subsection we will that the semantics of a Datalog^{RA} program can be unambiguously defined using least fixed point in the introduced order, as long as it satisfies some conditions. Intuitively, the rules should be monotone with respect to the order implied by aggregations.

Let P be a Datalog^{RA} program, with w *idb*relations $R_1, R_2, \ldots, R_w$ of arities $k_1, k_2, \ldots, k_w$, respectively. Program P has the following form:

$R_1(x_1, \ldots, x_{k_1-1}, x_{k_1}$ [aggregate F_1])
...
$R_w(x_1, \ldots, x_{k_w-1}, x_{k_w}$ [aggregate F_w])

$$\begin{array}{lcl}
R_1(x_1, \ldots, x_{k_1}) & :- & Q_{1,1}(x_1, \ldots, x_{k_1}). \\
 & \cdots & \\
R_1(x_1, \ldots, x_{k_1}) & :- & Q_{1,m_1}(x_1, \ldots, x_{k_1}). \\
 & \cdots & \\
R_w(x_1, \ldots, x_{k_w}) & :- & Q_{w,1}(x_1, \ldots, x_{k_w}), \\
 & \cdots & \\
R_w(x_1, \ldots, x_{k_w}) & :- & Q_{w,m_w}(x_1, \ldots, x_{k_w}).
\end{array}$$

The program has m_i rules for computing R_i, for each i. $Q_{1,1}, \ldots, Q_{1,m_i}$ for $i = 1, \ldots, w$ are bodies of these rules, with free variables $x_1, \ldots, x_{k_i}$. Each relation R_i may have an aggregation function $aggfun_{R_i}$ defined for column k_i. Each such $aggfun_{R_i}$ is required to be a join operation.

The subgoals in the rule bodies may refer to any relation of an arbitrary set of *edb*relations $edb(P)$, which are constant during the evaluation, and to any of the *idb*relations, $R_1, \ldots, R_w$. The *edb*relations are not declared in the program.

By definition, P is a program P' in regular Datalog, with *aggfun* additionally defined. P can be treated as a pair $(P', aggfun)$. The immediate consequence operator $T_{P'}$ for the regular Datalog program P' will be the base for defining the semantics for Datalog^{RA} program P. Schema $sch(P)$ for the Datalog^{RA} program P is defined as the schema $sch(P')$ of the corresponding Datalog program P'.

We start by we extending aggregation operator g_R over relation instances to aggregation over database instances. Intuitively, it applies the corresponding aggregation operator g_R to each relation instance I of relation R in a database instance.

Definition 4.1.9 (Aggregation operator over databases)**.** Let P be a program in Datalog^{RA} and $\mathbf{K}$ be a database instance over *sch*(P), and $\mathbf{K}$ be a union of relation instances $I_1, \dots I_n$ over $R_1, \dots, R_n$ respectively, where $R_1, \dots, R_n$ are relation names in *sch*(P). Let the *aggregation operator* G_P for $\mathbf{K}$ be defined as:

$$G_P(\mathbf{K}) = \bigcup_{i=1}^{n} g_{R_i}(I_i)$$

Similarly to relations after aggregation, we will call a database $\mathbf{K}$, such that for some $\mathbf{L}$ $\mathbf{K} = G_P(\mathbf{L})$, a *database after aggregation.*

Lemma 4.1.4. *G_P is monotone with respect to $\sqsubseteq_{sch(P)}$ on all databases over sch(P).*

Proof. Since G_P applies g_R to each relation R in the database and $\sqsubseteq_{sch(P)}$ compares databases relation-by-relation using $\sqsubseteq_R$, it is sufficient to show that g_R is monotone with respect to $\sqsubseteq_R$ for each $R \in$ *sch*(P).

Let R be any relation name in *sch*(P) and A, B be relation instances over R such that $A \sqsubseteq_R B$. If R is not aggregated, then g_R is an identity function, so it is monotone. If R is aggregated, $A \sqsubseteq_R B$ is equivalent to $\forall_{(q_1,\dots,q_{k-1},v) \in g_R(A)} \exists_{(q_1,\dots,q_{k-1},v') \in g_R(B)} v \leq_{aggfun_R} v'$. As it holds by definition of g_R that $g_R(g_R(A)) = g_R(A)$ and $g_R(g_R(B)) = g_R(B)$, we have that $\forall_{(q_1,\dots,q_{k-1},v) \in g_R(g_R(A))} \exists_{(q_1,\dots,q_{k-1},v') \in g_R(g_R(B))} v \leq_{aggfun_R} v'$, which is equivalent to $g_R(A) \sqsubseteq_R g_R(B)$. Therefore, g_R is monotone with respect to $\sqsubseteq_R$ and G_P is monotone with respect to $\sqsubseteq_{sch(P)}$. ∎

We can now define the immediate consequence operator for Datalog^{RA} programs.

Definition 4.1.10 (Immediate consequence operator for Datalog^{RA})**.** The *immediate consequence operator* for a Datalog^{RA} program $P = (P', aggfun)$, where P' is a program in Datalog, is a function $T_P : inst(sch(P)) \to inst(sch(P))$:

$$T_P = G_P \circ T_{P'}$$

The immediate consequence operator can be used to define semantics for a Datalog^{RA} program as its fix-point, similarly to the definition of Datalog's semantics.

Theorem 4.1.5. *Let P be a program in Datalog^{RA} and $P = (P', aggfun)$ where P' is a program in Datalog. Let $\boldsymbol{K}$ be a database instance over edb(P). Let $\sigma =$ sch(P).*

If $T_{P'}$ is monotone with respect to $\sqsubseteq_\sigma$ on databases over σ after aggregation, then there exists a finite minimal fix-point of T_P containing $\boldsymbol{K}$. We denote this fix-point by $P(\boldsymbol{K})$.

Proof. G_P is by Lemma 4.1.4 monotone with respect to $\sqsubseteq_\sigma$ on all databases over σ. Since it is assumed that $T_{P'}$ is monotone with respect to $\sqsubseteq_\sigma$ on databases after aggregation, $T_P = G_P \circ T_{P'}$ is also monotone with respect to $\sqsubseteq_\sigma$, as a composition of two monotone functions.

$\mathbf{K}$ is a database instance over *edb*(P), so it is a database after aggregation, because $\mathbf{K} = G_P(\mathbf{K})$. $T_P^i(\mathbf{K}) = G_P(T_{P'}(T_P^{i-1}(\mathbf{K})))$ is also a database after aggregation for each $i > 0$.

As it holds that *adom*$(P) \cup$ *adom*$(\mathbf{K})$ and the database schema *sch*(P) are all finite, there is a finite number n of database instances over *sch*(P) that can be reached by iteratively applying T_P to $\mathbf{K}$. Hence, because of the monotonicity of T_P on databases after aggregation, we have inductively that $T_P^i(\mathbf{K}) \sqsubseteq_\sigma T_P^{i+1}(\mathbf{K})$ for each $i \geq 0$. Therefore, the sequence $\{T_P^i(\mathbf{K})\}_i$ reaches a fix-point: $T_P^n(\mathbf{K}) = T_P^{n+1}(\mathbf{K})$. Let us denote this fix-point by $T_P^*(\mathbf{K})$.

$$\mathbf{K} \sqsubseteq_\sigma T_P(\mathbf{K}) \sqsubseteq_\sigma T_P^2(\mathbf{K}) \sqsubseteq_\sigma T_P^3(\mathbf{K}) \sqsubseteq_\sigma \cdots \sqsubseteq_\sigma T_P^*(\mathbf{K})$$

We will now prove that this is the minimum fix-point of T_P containing $\mathbf{K}$. Let us suppose that $\mathbf{J}$ is a fix-point of T_P containing $\mathbf{K}$: $\mathbf{K} \sqsubseteq_\sigma \mathbf{J}$. By applying T_P n times to both sides of the inequality, we have that $T_P^*(\mathbf{K}) = T_P^n(\mathbf{K}) \sqsubseteq_\sigma \mathbf{T}_P^n(\mathbf{J}) = \mathbf{J}$. Hence, $T_P^*(\mathbf{K})$ is the minimum fix-point of T_P containing $\mathbf{K}$. ■

Example 4.1.3. As an example, let us recall the program for computing the shortest paths from a selected source to other vertices of a graph, which we repeat in Figure 4.4.

$\textsc{Edge}(\text{int } \mathit{src}, \text{int } \mathit{sink}, \text{int } \mathit{len})$
$\textsc{Path}(\text{int } \mathit{target}, \text{int } \mathit{dist} \text{ aggregate } \textsc{Min})$

$$\begin{array}{lll} \textsc{Path}(t,d) & :- & t = 1, d = 0. \\ \textsc{Path}(t,d) & :- & \textsc{Path}(s,d_1), \textsc{Edge}(s,t,d_2), d = d_1 + d_2. \end{array}$$

Figure 4.4: Datalog^{RA} query for computing the shortest paths from node 1 to other nodes.

In this program, we the following aggregation function for relation Path is defined:

$$\mathit{aggfun}_{\textsc{Path}} = \lambda x, y.\, \min(x, y)$$

$\mathit{aggfun}_{\textsc{Path}}$ is clearly a join operation. It induces a partial order $\geq$: for $a, b \in \mathbb{N}$, $\min(a, b)$ is the least upper bound of a and b with respect to $\geq$.

The corresponding aggregation operator for Path is:

$$g_{\textsc{Path}}(I) = \Big\{(x,t) : (x,t) \in I \wedge t = \min(\{y : (x,y) \in I\})\Big\}$$

The aggregation-aware ordering on instances of Path is:

$$I_1 \sqsubseteq_{\textsc{Path}} I_2 \iff \forall_{(x,d) \in g_{\textsc{Path}}(I_1)} \exists_{(x,d') \in g_{\textsc{Path}}(I_2)} d \geq d'$$

For Edge, the order on relation instances is simply given by inclusion, i.e. $\sqsubseteq_{\textsc{Edge}}$ is by definition $\subseteq$.

The order $\sqsubseteq$ on database instances over $\mathit{sch}(P) = \{\textsc{Path}, \textsc{Edge}\}$ is as follows. Let $\mathbf{K}_1$ and $\mathbf{K}_2$ be any such database instances. By definition, $\mathbf{K}_1 = P_1 \cup E_1$ and $\mathbf{K}_2 = P_2 \cup E_2$, where P_1, P_2 are relations over Path and E_1, E_2 are relations over Edge.

$$K_1 \sqsubseteq K_2 \iff P_1 \sqsubseteq_{\textsc{Path}} P_2 \wedge E_1 \subseteq E_2$$

P can be viewed as a Datalog program P' with *aggfun* additionally defined. As for any Datalog program, the immediate consequence operator for $T_{P'}(\mathbf{K})$ is defined as the set of facts that are immediate consequences for P' and $\mathbf{K}$.

For P to be a Datalog^{RA} program, $T_{P'}$ is required to be monotone with respect to $\sqsubseteq$. As ensuring this is the responsibility of the programmer, we will now show that this is the case.

Proof. Let $\mathbf{K}_1$ and $\mathbf{K}_2$ be database instances after aggregation over $\mathit{sch}(P)$, such that $\mathbf{K}_1 \sqsubseteq \mathbf{K}_2$. By definition, $\mathbf{K}_1 = P_1 \cup E_1$ and $\mathbf{K}_2 = P_2 \cup E_2$, where P_1, P_2 are relations after aggregation over Path and E_1, E_2 are relations over Edge. $\mathbf{K}_1 \sqsubseteq \mathbf{K}_2$ implies that $E_1 \subseteq E_2$ and $P_1 \sqsubseteq_{\textsc{Path}} P_2$. We need to show that $T_{P'}(\mathbf{K}_1) \sqsubseteq T_{P'}(\mathbf{K}_2)$. This by definition is equivalent to $E_1' \subseteq E_2' \wedge P_1' \sqsubseteq_{\textsc{Path}} P_2'$, where E_1', E_2', P_1', P_2' are such that for $i = 1, 2$, $T_{P'}(\mathbf{K}_i) = E_i' \cup P_i'$, P_i' is a relation instance over Path and E_i' is a relation instance over Edge.

For any $\textsc{Edge}(x, y, z) \in E_1'$, it must be true that $\textsc{Edge}(x, y, z) \in E_1$, since there are no rules with $\textsc{Edge}$ in head. Hence, as $E_1 \subseteq E_2$, $\textsc{Edge}(x, y, z) \in E_2$ and consequently $\textsc{Edge}(x, y, z) \in E_2'$, so $E_1' \subseteq E_2'$.

$P_1' \sqsubseteq_{\textsc{Path}} P_2'$ is by definition equivalent to $\forall_{(v,d)\in g_{\textsc{Path}}(P_1')} \exists_{(v,d')\in g_{\textsc{Path}}(P_2')} d \geq d'$. To prove this, let us suppose by contradiction that this does not hold, that is that for some $(v, d) \in g_{\textsc{Path}}(P_1')$, for all d' such that $(v, d') \in g_{\textsc{Path}}(P_2')$, it holds that $d < d'$.

P_1, P_2 are relations after aggregation, so $P_1 = g_{\textsc{Path}}(P_1)$ and $P_2 = g_{\textsc{Path}}(P_2)$. Since $g_{\textsc{Path}}$ groups by the first column and aggregates the second column with min, we know that $(v, d) \in P_1'$. By definition of immediate consequence, this means that at least one of the following is true:

1. $(v, d) \in P_1$,
2. there is a corresponding instantiation of the first rule in $\mathbf{K}_1$,
3. there is a corresponding instantiation of the second rule in $\mathbf{K}_1$.

Let us consider each one of these possibilities.

If $(v, d) \in P_1$, as it holds that $P_1 \sqsubseteq_{\textsc{Path}} P_2$ and $P_1 = g_{\textsc{Path}}(P_1)$, there exists $d^* \leq d$ such that $(v, d^*) \in g_{\textsc{Path}}(P_2)$. By definition of $g_{\textsc{Path}}$ we have that $(v, d^*) \in P_2$. As it follows from the definition of immediate consequence, $(v, d^*) \in P_2'$, so it cannot hold that $d < d'$ for all d' such that $(v, d') \in g_{\textsc{Path}}(P_2')$.

If there exists a corresponding instantiation of the first rule in $\mathbf{K}_1$, i.e. $(v, d) = (1, 0)$, we also have the same instantiation in $\mathbf{K}_2$, so $(1, 0) \in P_2'$ and it also cannot hold that $d < d'$ for all d' such that $(v, d') \in g_{\textsc{Path}}(P_2')$.

The last possibility is that there is a corresponding instantiation of the second rule in $\mathbf{K}_1$, i.e. there exist s, d_1, d_2 such that $d = d_1 + d_2$, $(s, d_1) \in P_1$ and $(s, v, d_2) \in E_1$. Since $\textsc{Path}(s, d_1) \in P_1 = g_{\textsc{Path}}(P_1)$ and $P_1 \sqsubseteq_{\textsc{Path}} P_2$, there exists $d_1^* \leq d_1$ such that $(s, d_1^*) \in g_{\textsc{Path}}(P_2)$. By definition of $g_{\textsc{Path}}$, $(s, d_1^*) \in P_2$. Additionally, $(s, v, d_2) \in E_1 \subseteq E_2$. Therefore, $\textsc{Path}(v, d^*) : -\textsc{Path}(s, d_1^*), \textsc{Edge}(s, v, d_2), d^* = d_1^* + d_2$, where $d^* = d_1^* + d_2 \leq d_1 + d_2 = d$ is an instantiation of the second rule in $\mathbf{K}_2$. This means that $(v, d^*) \in P_2$, so again it cannot hold that $d < d'$ for all d' such that $(v, d') \in g_{\textsc{Path}}(P_2')$.

Since each of the three possibilities results in a contradiction, it follows that $P_1' \sqsubseteq_{\textsc{Path}} P_2'$. Therefore $T_{P'}(\mathbf{K}_1) \sqsubseteq T_{P'}(\mathbf{K}_2)$, which proves that $T_{P'}$ is monotone with respect to $\sqsubseteq$. ■

4.1.5. Evaluation

A straightforward way to evaluate $P(\mathbf{K})$, i.e. to compute the minimal fix-point of T_P containing $\mathbf{K}$, is to iteratively apply T_P to $\mathbf{K}$ until a fix-point is reached. This algorithm, used in Datalog and described in detail in Section 2.5.1, can also be directly applied in Datalog^{RA} . The only difference is that the immediate consequence operator which is used needs to take into account the aggregations.

Semi-naive evaluation

Semi-naive evaluation, the most basic optimization technique used in Datalog evaluation, can be easily adopted in Datalog^{RA} .

In semi-naive evaluation of Datalog, which is described in Section 2.5.2, T_P is computed in a more efficient way than in the naive evaluation. To achieve this, a T_P^Δ function is used. $T_P^\Delta(I, \Delta)$ evaluates the rules of program P on database instance I and the set of new facts from the last iteration Δ, so that at least one new fact is used in the application of each rule. In each iteration, T_P^Δ is applied to the full database and Δ from in the last step and the output is merged with the current database instance.

In the evaluation of a program P in Datalog^{RA} , such that $P = (P', aggfun)$ where P' is a Datalog program, we can use this technique to efficiently compute $T_{P'}$. The full algorithm is presented in pseudocode in Figure 4.5.

```
SEMI-NAIVE-EVALUATE-DATALOGRA($P$, $\mathbf{K}$)
1  $I_0 \leftarrow K$, $\Delta_0 \leftarrow K$
2  $i \leftarrow 0$
3  repeat
4      $i \leftarrow i + 1$
5      $C_i \leftarrow T_P^\Delta(I_{i-1}, \Delta_{i-1})$
6      $I_i \leftarrow G_P(C_i \cup I_{i-1})$
7      $\Delta_i \leftarrow I_i - I_{i-1}$
8  until $\Delta_i = \emptyset$
9  return $I_i$
```

Figure 4.5: Semi-naive evaluation algorithm for Datalog^{RA}

The only difference between this algorithm and the semi-naive evaluation algorithm for Datalog described in Section 2.5.2 is that in each step G_P is applied to the newly computed database instance.

4.1.6. Stratified Datalog^{RA}

Negation can be introduced to Datalog^{RA} in the same way as to regular Datalog — by stratification. Intuitively, if there are no negated recursive calls, then the program can be divided into smaller subprograms, called *strata*, so that there is no cyclic dependency between relations defined in different strata. The whole program can be then evaluated by evaluating the subprograms for each stratum in the topological order. Stratification and semantics for Datalog$^\neg$are described in detail in Subsection 2.6.2.

We have already defined the semantics for Datalog^{RA} without negation. It can be naturally extended to the semantics of semi-positive Datalog^{RA} programs, i.e. programs in Datalog^{RA} that can also use *edb*relations in a negated subgoal. This defines the semantics for an individual stratum.

To obtain the semantics of a general program in Datalog^{RA} with negation, let us notice that stratification can be applied to Datalog^{RA} programs with negation just like to regular Datalog$^\neg$programs. The only difference is that when P is partitioned into the strata $P_1, \ldots, P_n$, each stratum P_i can contain recursive aggregation. In other words, it each P_i is a semi-positive Datalog^{RA} program. Therefore, each P_i is evaluated using the Datalog^{RA} semantics, instead of the regular Datalog semantics.

4.2. Distributed SociaLite

Running a program on a single machine significantly limits the scale of problems which can be addressed. Large datasets will not fit into one computer's memory, and the running time is usually too large for practical applications. It is not possible to perform computations on large datasets, unless they are distributed across multiple machines. This rule also applies to programs in Datalog and its variants such as Datalog^{RA} .

The second paper on SociaLite [2] describes one of the possible ways of distributing evaluation of such programs. It proposes that each relation is distributed across workers based on the values in its first column. The facts from relations are sharded either with a hash function, or by dividing the possible value ranges into equal parts. The first column, by which the sharding is done, is distinguished by writing it separately in square brackets, for example the following rule for each fact $(1, v, d)$ in Edge adds a fact (v, d) to Path:

$$\textsc{Path}[v](d) : -\textsc{Edge}[1](v, d)$$

The evaluation is performed on the workers, which are coordinated by a *master node* and distribute between themselves the facts they need. Fault tolerance is achieved by checkpointing the state of the computation on distributed file system every few evaluation steps.

In Chapter 5 we present an alternative way of parallelizing Datalog evaluation — by translating the program into a set of Apache Spark's native operations on RDDs.

4.3. Differences from original SociaLite formulation

The version of Datalog presented here differs from the original description found in [1, 2].

We presented a precise definition of a Datalog^{RA} program and the well-defined conditions for it to have an unambiguous solution. While the original papers contains these conditions, they lack precise definitions of the notions used, such as the order on database instances.

In our definition, the syntax for declaring the aggregated column and function is to place them in the preamble, while in the original paper the aggregations are placed in the rules' heads. We believe that our approach can enhance the readability of Datalog^{RA} programs.

In [1, 2] recursive aggregate functions are defined using *meet operation*, which is dual to join operation, but induces a partial order in which the outcome of the operation is the greatest lower bound instead of the least upper bound. This requires the semantics of a program to be defined using the greatest fixed point. We decided to use join operations, as it allows the semantics to be defined with the least fixed point, consistently with regular Datalog.

Chapter 5

Datalog^{RA} on Spark

Datalog^{RA} queries executed in a distributed way can prove to be very useful for performing computations on large datasets, especially on graphs. Our goal is to provide an implementation of Datalog^{RA} which will be as close as possible to being practically applicable.

To be truly useful, such solution has to be reliable, provide ways to interact with various distributed storages, ensure proper fault tolerance and require little to no additional work to be run on the existing infrastructure.

Distributed Socialite, covered in section 4.2, executes queries with aggregation in a distributed way by sharding the data based on a selected column of each relation. Its authors presented an independent implementation of the proposed solution. We propose an alternative approach to distributed Datalog^{RA} queries evaluation and present an implementation of the proposed solution.

Instead of developing an independent project, we have chosen to implement Datalog^{RA} as an extension to Apache Spark. Owing to this approach, the implementation can read from and write to all popular Hadoop distributed data stores, including HDFS, HBase and Cassandra [32]. It can be used on any existing cluster that supports Spark, including Hadoop YARN [33], Mesos and standalone Spark clusters [32] and has the industrial-quality fault tolerance and efficiency features developed by hundreds of Apache Spark contributors [34].

Using Spark Datalog API, Datalog^{RA} queries can be integrated seamlessly into Spark programs. One can choose to use Datalog for all computations or only for their parts where it serves best. It is also possible to run Datalog^{RA} queries interactively using the Spark Shell.

In this chapter we describe how Datalog and Datalog^{RA} queries can be translated into operations permitted in the RDD model. This method was used to implement the Datalog API for Spark, allowing for the usage of Datalog queries in Spark programs. The implementation is evaluated in section 5.3.

5.1. Integrating Datalog^{RA} into Spark

Spark programs describe the computation as a series of calls to functions which perform one of the following:

- create an RDD, generally by reading it from a distributed data source such as HDFS,
- transform an RDD or several RDDs into another RDD with methods such as *map*, *filter* or *join*,
- execute an action on an RDD, such as storing it in HDFS or counting its size.

Most of the computation logic is expressed with transformations. Core Spark provides only some basic transformations (such as *map, filter, reduceByKey, union, join, sample, sort*) and actions (e.g. *count, collect, reduce, save*). Additionally, there are several extensions: GraphX, Spark SQL, MLlib. Typically, they extend Spark by providing specialized RDDs and composite data structures as well as additional transformations for them.

The same approach has been chosen for adding the ability to execute Datalog queries. The extension is contained in a module called *Spark Datalog API* or *SparkDatalog*. The main component it provides is the *Database* class which contains a set of relations and can be created from regular RDDs. Database objects are equipped with a method *datalog*, which performs a Datalog query on this database. The result of the query is a new Database, from which individual relations can be extracted as RDDs.

Datalog queries can be applied to any set of input RDDs. The input data can be directly loaded from a distributed storage supported by Spark or computed as a transformation of other RDDs. The user first creates relations, declares their arity, and groups them into a database. Next, any query on this database can be performed using its *datalog* method. If there are no errors in the query, the result is a new database containing all computed relations. Each relation can be extracted from the database by giving its name. Figure 5.1 shows an example of a Datalog^{RA} query in a Spark program using Datalog API for Spark.

```
val edgesRdd = ... // RDD of edge tuples read from disk or computed using Spark

val database = Database(Relation.ternary("Edge", edgesRdd))
val resultDatabase = database.datalog("""
    declare Path(int v, int dist aggregate Min).
    Path(x, d) :- s == 1, Edge(s, x, d).
    Path(x, d) :- Path(y, da), Edge(y, x, db), d = da + db.
""")
val resultPathsRdd = resultDatabase("Path")

... // Save or use resultPathsRdd as any RDD.
```

Figure 5.1: Example of Datalog query for computing single source shortest paths embedded in a Spark program.

5.2. Executing Datalog^{RA} queries in the RDD model

When a Datalog query is performed on the Database object, it needs to be translated into a sequence of Spark transformations which eventually produce a new Database object containing the result of the query. In this section, we show how this translation is made.

5.2.1. Data representation

Both input and output of Datalog programs are represented as Database objects. Database is simply a set of several Relations.

Each Relation object has a name and an RDD of Facts, which in turn are represented as arrays. All Facts in a relation are required to have the same arity — this is ensured when the Relation is created.

Valuation objects are not available to the end user, but they play a significant role in the evaluation. They represent valuations, i.e. functions mapping variables to their values. For performance reasons, they are not represented as maps, but as raw arrays whose layout is determined using the information generated during the analysis phase.

5.2.2. Datalog program execution

On the top level, the algorithm for executing a Datalog program on Spark consists of the following steps:

1. Analysis phase: Initially, the program is parsed and analyzed both syntactically and semantically. All correctness requirements are checked at this stage.
2. Stratification: Analyzed program is then stratified, i.e. divided into a sequence of strata using a standard strongly connected components algorithm.
3. Evaluation phase, which starts by evaluating the first stratum of the program on the input database. Each subsequent stratum is then evaluated using the output of the previous stratum as an input. The output of the last stratum is returned as the output of the whole program.

To evaluate each stratum, the semi-naive evaluation algorithm is used. It maintains the current state of the database and runs iteratively until no changes are made to the database. In each iteration, for each rule in this stratum, all inferable facts are computed. The way this can be achieved on RDDs is described in section 5.2.3. The set of obtained facts is then merged into the current database and a *delta*, i.e. the difference between the current and previous database state is computed. During the merge all necessary aggregations are applied. This step is described in Section 5.2.4.

5.2.3. Single rule evaluation

Evaluation of a single rule plays a crucial role in evaluating a Datalog query on RDDs. Given a single rule, consisting of a head and a sequence of subgoals, and two databases: the full database state after the last step and the delta database, the task is to compute an RDD of all facts that can be inferred. It is done in two steps: first, all valuations satisfying the rule body are computed, and then each such valuation is converted to a fact based on the head of the rule.

The subgoals are sorted topologically during the analysis phase, so that the evaluation can be performed sequentially, subgoal by subgoal. We keep the RDD of all valuations which satisfy the subgoals processed so far. This RDD is updated in each step. The topological order ensures that all variables required by arithmetic subgoals are introduced by an earlier subgoal.

We start with an RDD of valuations containing an empty valuation, i.e. a valuation containing no variable assignments. Next, the subgoals are applied sequentially. Each subgoal is evaluated on the current RDD of valuations. This returns a new RDD of valuations all of which satisfy this subgoal. This new RDD is then used for the next subgoal, until all subgoals are processed.

Before describing how a subgoal is evaluated on an RDD of valuations, we first describe a simplified scenario of processing a subgoal given a single valuation. Then we show how generated valuations are converted into facts and how these general methods can be used to perform either naive or semi-naive evaluation. Finally, we describe how new facts are merged into the existing database.

Evaluating a subgoal on a single valuation

Let us start from the most basic scenario. Let us assume that we already have some valuation v to start with. v simply defines a specific value for some variables. The task is to evaluate a single relational subgoal $R(x_1, \dots, x_n)$ on the starting valuation v, i.e. to find the set of all valuations which satisfy $R(x_1, \dots, x_n)$ and are supersets of v. To do this, we can find all facts in relation R that match v on the *matching variables*, i.e. those which appear both in the valuation and in the subgoal. Each such fact yields an extended valuation consisting of v and the valuation for variables from the subgoal.

For example, given a subgoal $\textsc{Edge}(v, u, d)$ and a valuation $\{v : 5, t : 3\}$, if Edge contains $\textsc{Edge}(5, 1, 2)$, $\textsc{Edge}(5, 3, 4)$ and $\textsc{Edge}(1, 2, 3)$, the result will be $\{\{v : 5, t : 3, u : 1, d : 2\}, \{v : 5, t : 3, u : 3, d : 4\}\}$.

This algorithm could be realized by means of *map* and *filter* transformations on the relation in the subgoal.

Other types of subgoals are more straightforward. Arithmetic comparison subgoals, e.g. $x < y + z$ are simply evaluated to a boolean value and yield $\{v\}$ if the result was true, and an empty set if the result was not true. In case of assignment subgoals, e.g. $x = y + z$, the value of the expression is calculated inserted into the valuation for the given variable. If this causes a conflict with a different value for that variable, an empty set is returned. Otherwise, the result is a singleton containing the new variable.

Evaluating a subgoal on a set of valuations

During the evaluation of a sequence of subgoals, instead of having just one starting valuation, we have an RDD of valuations to start with. The result of evaluating a subgoal on an RDD of valuations should be an RDD with all valuations satisfying the subgoal and being supersets of at least one of the starting valuations.

In case of relational subgoals, to find the result efficiently, the relation is first converted into valuations of variables in the subgoal using a *map* transformation. The resulting RDD is then joined with the starting valuations on the matching variables using the *join* transformation on RDDs and mapped to a new RDD of valuations. This is illustrated in Figure 5.2.

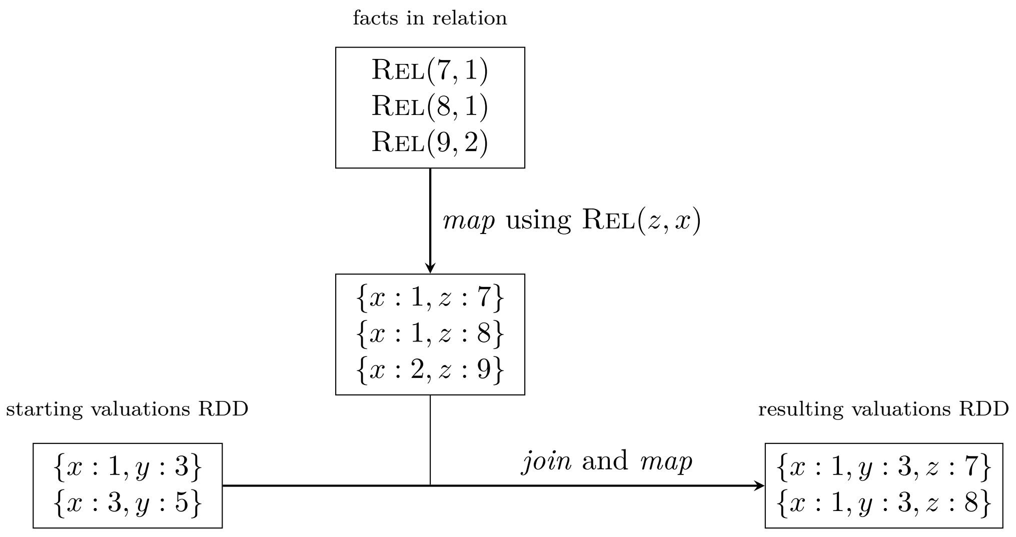

Figure 5.2: Evaluation of a relational subgoal $\textsc{Rel}(z, x)$ on an RDD of valuations.

Arithmetic comparison subgoals are translated into a *filter* transformation on the valuations RDD. They are filtered using a function that checks whether the comparison yields true for that valuation. This is illustrated in Figure 5.3.

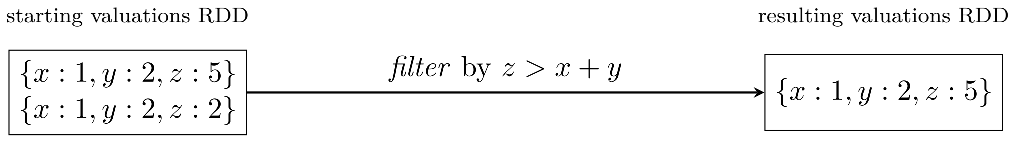

Figure 5.3: Evaluation of an arithmetic comparison subgoal $z > x + y$ on an RDD of valuations.

Assignment subgoals are translated into a *map* transformation on the valuations RDD, combined with flattening of the results. For each valuation, the assignment can be mapped either to a singleton or to an empty set, so the result of applying the mapping is an RDD of sets. Therefore it needs to be flattened so that the result is a plain RDD of valuations. Figure 5.4 illustrates these two steps.

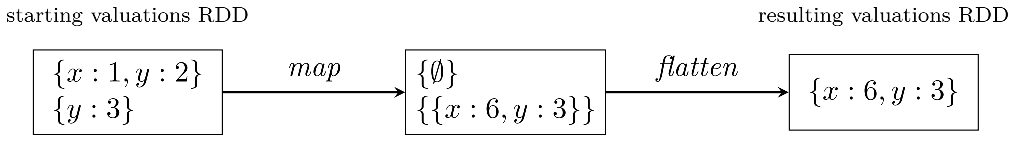

Figure 5.4: Evaluation of an assignment subgoal $x = 2 * y$ on an RDD of valuations.

Processing rule head: converting valuations into facts

Each rule head consists of a relation name and a sequence of variable names, e.g. $\textsc{Path}(v, d)$. Language constraints ensure that each valuation satisfying the rule body contains values for all variables appearing in the head, so a fact can be computed from a valuation by looking up corresponding variables in the valuation. Given an RDD of valuations obtained by evaluating the sequence of subgoals in the rule body, this is done by applying a *map* transformation to this RDD. This is illustrated in Figure 5.5.

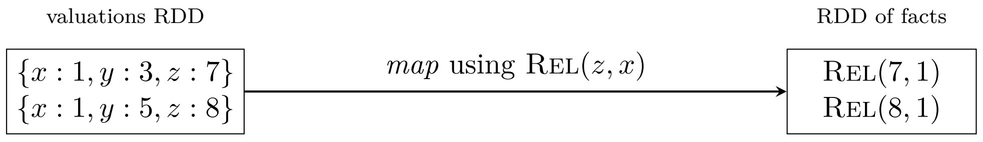

Figure 5.5: Converting valuations into facts based on the rule head.

Naive and semi-naive evaluation

If full database is used to retrieve the contents of the relation in each subgoal, the evaluation procedure described in the previous sections will perform the naive evaluation.

For the semi-naive evaluation to be performed, the delta database needs to be used. Precisely, the rule body is evaluated separately for each subgoal that uses a relation being in the *idb* of the current stratum. The contents of the relation for the selected subgoal is retrieved from the delta database and the full database is used in all other subgoals.

5.2.4. Adding new facts to the database

The procedure covered in section 5.2.3 allows for finding all facts that can be inferred using a single rule. This can be done for each rule within a stratum. The next step in one iteration of evaluation is to merge the results obtained with the existing database.

In case of relations without aggregation, this is achieved by performing the *union* transformation on the RDDs containing new facts for a given relation and the corresponding relation in the current database. It is possible that this introduces duplicated facts, so the *distinct* transformation is used to remove the duplicates.

In case of relations that need to be aggregated, this is slightly different. After performing a union of the new facts and current relation contents, the facts are grouped by the qualifying parameters and the aggregated value is computed by applying the aggregation function. This is done by performing the following three transformations:

1. *map* to split the facts into qualifying parameters and the aggregated values,
2. *reduceByKey* to group by qualifying parameters and apply the aggregation function to the set of aggregated values in each group,
3. *map* to merge the qualifying parameters and the computed value in each group back into facts.

This process is illustrated in Figure 5.6.

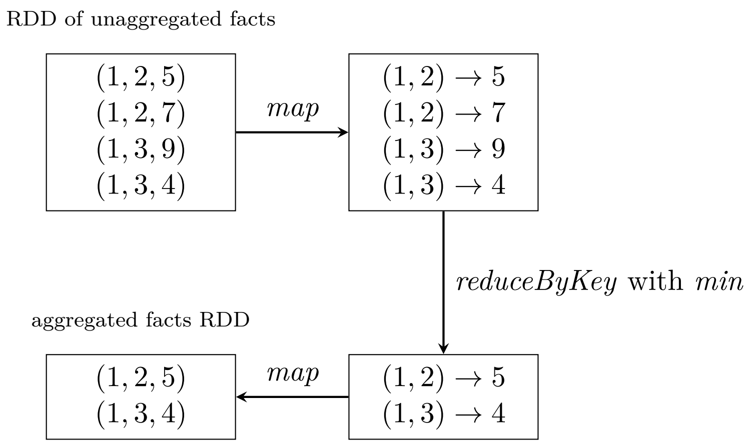

Figure 5.6: Applying *min* aggregation to third column of an RDD of facts.

5.2.5. Handling RDD caching, materialization and lineage

A practical implementation of the evaluation procedure described above requires several RDD-specific issues to be handled:

- reused RDDs need to be marked to be *cached* in order for them to be stored in memory and not recomputed each time they are used. The new full database and delta database are marked for caching after each iteration step,
- results obtained from an iteration and marked to be cached are materialized, i.e. actually computed. This allows for the results of the previous iteration to be *unpersisted*, i.e. removed from cache, as they will no longer be necessary. In each step, after the full database and delta are materialized, their older versions are unpersisted. Unpersisting RDDs helps reduce memory usage and increases performance,
- if many iterations are performed, the procedure can create RDDs with a very long *lineage*. Lineage is stored in RDDs so that they can be restored after a failure of a

worker. Too long lineage can cause errors, so every several iterations all results are checkpointed to persistent storage, which causes the lineage to be cropped,

- most of RDD transformations do not change the partitioning of data, i.e. the way the data is distributed between worker nodes. However, *join* transformations, which are used in the evaluation, increase the number of partitions. The number of partitions too large compared to the size of data can negatively affect performance, so it is reduced when it exceeds a certain limit using the *coalesce* action, which merges certain partitions of an RDD to reduce their number. Currently, the limit is defined manually, but it could be possibly computed based on the amount of data and the number of worker nodes.

5.2.6. Optimizations

The main implemented optimization was the semi-naive evaluation. Additionally, several smaller optimizations have been implemented:

- Usually some of the strata are non-recursive and define only one relation. In this case, it is clear upfront that one iteration is enough for all facts to be inferred, so the evaluation of the stratum is finished after one iteration, instead of performing one more iteration only to notice that there were no changes in the database.

- Rule bodies can start with constant subgoals, e.g. $\text{PATH}(v,t) :- s = 1, \text{EDGE}(s,v,t)$. When subgoals are topologically sorted, such subgoals are placed at the beginning of the subgoals sequence. In order for the evaluation of the constant subgoals not to be repeated, the initial sequence of constant subgoals is evaluated during the analysis phase. The result is then used as an initial valuation when evaluating the rest of the subgoals in each iteration where this rule body is considered.

- If a relation is used in a subgoal, it needs to be converted into an RDD of valuations for joins to be performed. From the perspective of a stratum, however, only the relations defined in this particular stratum can change. Therefore, within each stratum, for each subgoal referring to a relation from another stratum the corresponding valuations are found once and persisted. It makes it possible to avoid repeated work in each of the evaluation iterations in this stratum.

5.3. Experiments

Spark Datalog API has been implemented as a fully working prototype. It has been evaluated on clusters of up to 16 Amazon EC2 worker instances. In this section we provide three sets of experimental results. Datalog^{RA} is not limited to a specific domain and can be used to express various types of distributed computations, but it is primarily intended for computations on large graphs such as social networks. Therefore, we have selected three common graph problems [11, 7] for performance tests of the prototype implementation:

- finding and counting triangles: find all triangles in a graph, i.e. triplets of vertices which form a complete graph; additionally, the total count of triangles should be computed,

- dividing the graph into connected components, i.e. maximal subgraphs in which each pair of vertices is connected by a path,

- computing the shortest paths from a single source to all other vertices.

The performance of the tested implementation was compared with plain Spark programs solving the same problem. The plain Spark implementations were written using Spark core methods and the *pregel* operation provided by the GraphX extension. In addition to performance, an interesting property is the complexity of each solution, which can be roughly measured by the number of lines in each program. To our knowledge, only an implementation of a sequential version of SociaLite was publicly available at the time of our tests. In the future, a comparison with Distributed Socialite will also be interesting.

For each problem, both solutions have been evaluated on Amazon EC2 clusters consisting of $n = 2, 4, 8$ and 16 worker nodes and one master node. Each node was a 2-core 64-bit machine with 7.5 GiB of RAM memory. In all experiments, a social graph of Twitter circles [48], which has 2.4M edges was used.

For both SparkDatalog and plain Spark programs the execution times T_n were measured in each test case, where n is the number of worker nodes. Relative speedups, i.e. the ratio of T_n to T_2 were calculated. Figure 5.7 presents the execution times and the relative speedups compared to the ideal speedup. In the Triangles Count test case the execution time in SparkDatalog is very similar to dedicated Spark. The SparkDatalog versions are slower than dedicated Spark programs in both Shortest Paths and Connected Components, by a factor of approximately 8.5 to 3.5 in Shortest Paths and 4 - 1.7 in Connected Components. The reasons for that are analyzed in section 5.4. The speedups achieved were similar for both versions of each program. This shows that although the implemented solution is slower by some factor, it does parallelize. The speedups were slightly better for the SparkDatalog versions. This is probably because of the fact that the additional overhead in SparkDatalog execution time over dedicated Spark could also get parallelized. In Connected Components and Shortest Paths, the difference in execution times is the least with the greatest number of worker nodes.

	plain Spark	**SparkDatalog**
Connected Components	11	6
Shortest Paths	12	4
Triangles	7	5

Table 5.1: Number of lines of code in programs, excluding data loading and comments.

An important goal for SparkDatalog is to provide programmers and non-programming analysts with the possibility to perform computations by writing declarative queries instead of implementing complicated, lengthy algorithms using the Pregel model or standard RDD transformations. Table 5.1 shows the comparison of lengths between SparkDatalog and plain Spark programs for each of the problems. Datalog versions are 1.4 to 3.5 shorter than dedicated Spark. Most importantly, they are conceptually simpler since they only require a few declarative rules instead of expressing the problem in the vertex-centric Pregel model.

The source codes of programs used in the experiments are presented in Appendix A.

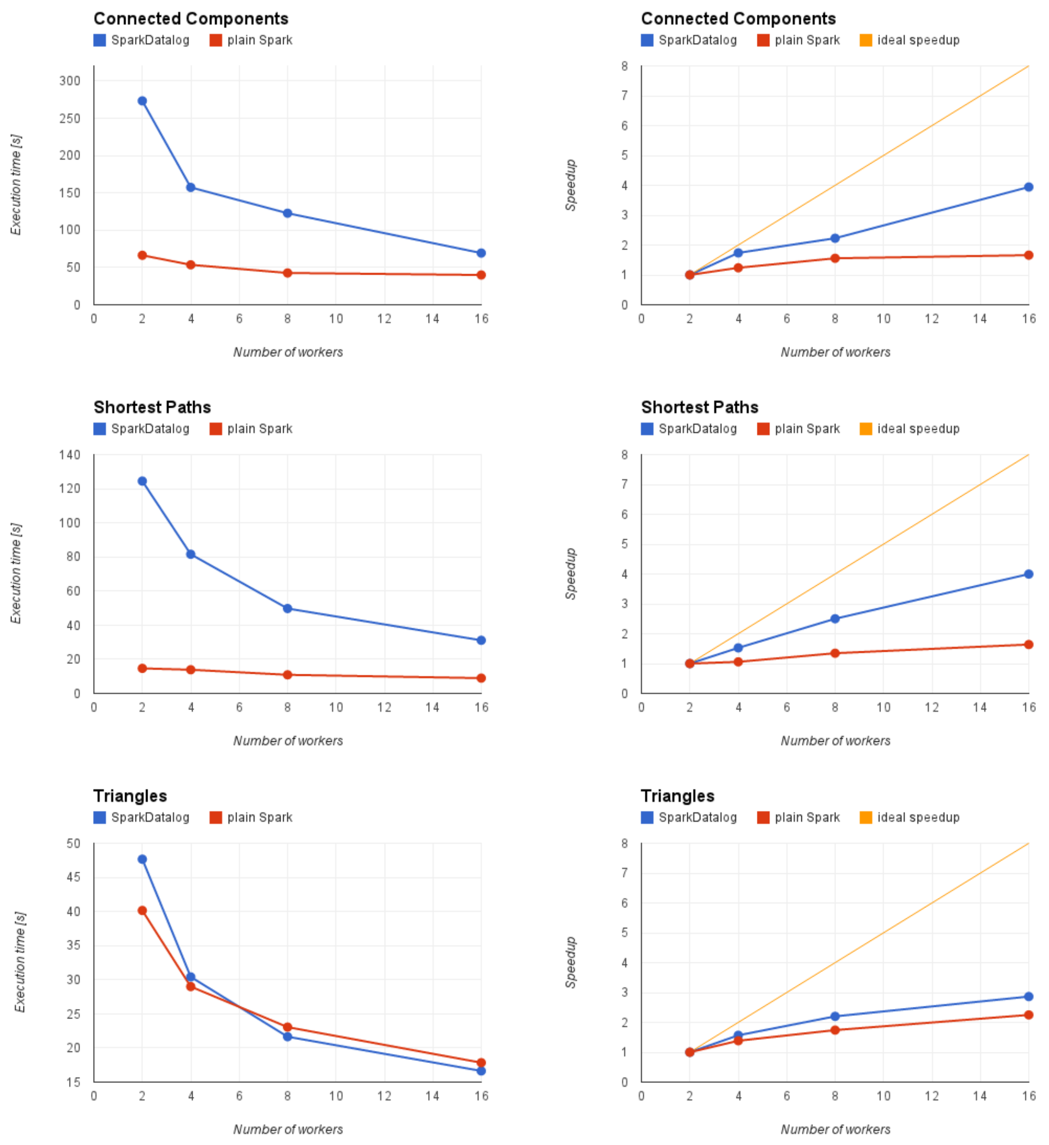

Figure 5.7: Results of experiments.

5.4. Performance differences

The SparkDatalog programs are several times slower than their dedicated Spark counterparts. There are several reasons for that, which can be eliminated or minimized in future versions of SparkDatalog:

1. SparkDatalog uses a general way of converting the Datalog program to a sequence of Spark transformations, which results in a suboptimal execution of the query, whereas the dedicated Spark program executes precisely the necessary operations. This can be resolved or minimized by more work on optimizing the generated execution plan.

2. The internal data representation has a significant impact on the performance. In dedicated Spark programs, native Scala tuples are used. SparkDatalog, on the other hand, currently uses arrays, which are less efficiently serialized and hashed. This results in a significant performance overhead. This could be resolved by representing the most common arities, e.g. 1–20 with tuples and using pre-generated code to work with them.

3. The most costly operation in SparkDatalog execution is the repartitioning of the objects by the hash of their key in order to perform a *join*. The partitioning of the data could be optimized, so that the need to transfer and repartition the data is minimized. This can possibly result in a significant performance gain.

In addition to the above, it is worth adding that since in Datalog problems are expressed in a very high-level, declarative way it offers large possibilities of applying optimizations to query execution, for example the *delta-stepping* technique and approximate evaluation [2]. This can result in significant performance gains over plain Spark programs.

5.5. Further work

There are several areas for further work connected both to the implementation and to the proposed theoretical solution.

Currently, it is the user's responsibility to ensure that the program is correct and, in particular, that the rules are monotone with respect to the order implied by the aggregation function used. A theoretical result or an implementation of a tool helping to determine whether a program satisfies this condition is crucial for the wide adoption of the proposed language.

Clearly, the performance of the solution is noticeably worse than that of dedicated Spark programs, although the speedups achieved are similar. More work is needed on optimizing the way the queries are executed, including the data representation and partitioning. More sophisticated optimizations of the generated execution plan, such as elimination of intermediate relations or the *delta-stepping* technique, could also be implemented in order to further improve performance.

User experience in embedding the Datalog code in Spark programs could also be improved. Specifically, instead of writing the program as a string, which is then parsed, there could be a domain specific language that allowing users to write similar rules directly in the code. This would allow for type-aware syntax highlighting in IDEs, greater type safety checked at the compile time and additional performance optimizations.

Chapter 6

Summary

In this work we have given a precise definition of the Datalog language extended with recursive aggregate functions, abbreviated Datalog^{RA} . This supplements the original idea from the SociaLite language, which was not fully defined in [1, 2]. We have also presented an implementation of the Datalog^{RA} language as a SparkDatalog extension to the Apache Spark distributed computations platform.

Performance of the implementation has been evaluated on clusters with up to 16 worker nodes. The current version of SparkDatalog turned out to be slower than the dedicated Spark programs by a factor of 2 to 8, depending on the problem and cluster size, but scale up in a similar way, giving slightly better speedups. The solution can be further optimized by eliminating the issues that negatively impact the performance as well as implementing additional optimizations to benefit from the high-level, declarative characteristic of the language. This can lead to the same or even better performance than typical Spark programs.

Although not as effective as dedicated programs, the presented implementation allows for a quick adoption by users, including analysts who can use Datalog^{RA} to perform queries on large datasets without being required to implement complicated vertex-centric algorithms. It gives users all benefits of Spark, including fault-tolerance, highly-optimized and thoroughly tested core platform with the possibility to be deployed to the popular Hadoop and Mesos clusters as well as support for most of the distributed data storage standards. This means that in most cases, there are no infrastructure changes required in order to use SparkDatalog on an existing cluster. For more advanced users, SparkDatalog offers the possibility to use Datalog^{RA} queries only for selected parts of their Spark programs to which it fits best.

Appendix A

Source code of programs used in experiments

A.1. Single source shortest paths

Shortest paths - SparkDatalog

```
val query = """ declare Path(int v, int dist aggregate Min).
                Path(x, d) :- s == """ + sourceId + """ , Edge(s, x, d).
                Path(x, d) :- Path(y, da), Edge(y, x, db), d = da + db."""
val resultDatabase = database.datalog(query)
```

Shortest paths - Spark

```
val initialGraph = graph.mapVertices((id, _) => if (id == sourceId) 0.0 else Double.PositiveInfinity)
val sssp = initialGraph.pregel(Double.PositiveInfinity)(
    (id, dist, newDist) => math.min(dist, newDist), // Vertex Program
    triplet => {  // Send Message
      if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
        Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
      } else {
        Iterator.empty
      }
    },
    (a,b) => math.min(a,b)) // Merge Message
val shortestPaths = sssp.vertices.map(v => (v._1, v._2))
```

A.2. Triangles counting

Triangles counting - SparkDatalog

```
val query = """ |declare Triangle(int v, int w, int u).
               |declare Total(int a, int b aggregate Sum).
               |Triangle(x, y, z) :- Edge(x, y), x < y, Edge (y, z), y < z, Edge(x, z).
               |Total(a, c) :- Triangle(x, y, z), a = 1, c = 1. """.stripMargin
val resultDatabase: Database = database.datalog(query)
```

Triangles counting - Spark

```
val canonicalEdges = edgesRdd.filter(Function.tupled(_ < _)).distinct().cache()
val swappedEdgesRdd = canonicalEdges.map(_.swap)
val pathOf2 = swappedEdgesRdd.join(canonicalEdges).map(  case (y, (x, z)) => (x, (y, z))  )
val triangle = pathOf2.join(canonicalEdges)
  .filter( case (x, ((y, z), zp)) => z == zp )
  .map( case (x, ((y, z), zp)) => (x, y, z) )
val count = triangle.count()
```

A.3. Connected components

Connected components - SparkDatalog

```
val query = """ |declare Component(int n, int component aggregate Min).
                |declare ComponentId(int n).
                |Component(n, i) :- Node(n), i = n.
                |Component(n, i) :- Component(p, i), Edge(p, n).
                |ComponentId(id) :- Component(x, id). """.stripMargin
val resultDatabase = database.datalog(query)
```

Connected components - Spark

```
val initialGraph = graph.mapVertices((id, _) => id.toInt)
val connectedComponents = initialGraph.pregel(Int.MaxValue)(
  (id, cmp, newCmp) => math.min(cmp, newCmp), // Vertex Program
  triplet => {  // Send Message
    if (triplet.srcAttr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr))
    } else {
      Iterator.empty
    }
  },
  (a, b) => math.min(a, b)) // Merge Message
```

Bibliography

[1] Jiwon Seo, Stephen Guo, Monica S. Lam, *SociaLite: Datalog extensions for efficient social network analysis*, ICDE 2013: 278-289

[2] Jiwon Seo, Jongsoo Park, Jaeho Shin, Monica S. Lam: *Distributed SociaLite: A Datalog-Based Language for Large-Scale Graph Analysis.* PVLDB 6(14): 1906-1917 (2013)

[3] S. Abiteboul, R. Hull, and V. Vianu: *Foundations of Databases.* Addison-Wesley (1995)

[4] T.J. Ameloot, B. Ketsman, F. Neven, D. Zinn: *Weaker Forms of Monotonicity for Declarative Networking: a more fine-grained answer to the CALM-conjecture*, PODS 2014

[5] S. Brin, L. Page. *The anatomy of a large-scale hypertextual web search engine.* In WWW'98, 1998.

[6] Jeffrey Dean , Sanjay Ghemawat: *MapReduce: simplified data processing on large clusters*, Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, 2004

[7] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski: *Pregel: a system for large-scale graph processing*, Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, 2010

[8] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, J. McPherson: *From "Think Like a Vertex" to "Think Like a Graph"*, Proceedings of the VLDB Endowment, 2013

[9] S. Salihoglun J. Widom: *GPS: A Graph Processing System.* SSDBM, July 2013

[10] U. Meyer, P. Sanders: *Delta-stepping: A parallel single source shortest path algorithm.* ESA, 1998.

[11] Anand Rajaraman, Jeffrey D. Ullman: *Mining of Massive Datasets*, Cambridge University Press, New York, NY, 2011

[12] E. F. Codd: *A relational model of data for large shared data banks*, Communications of the ACM, v.13 n.6, 1970

[13] R. Ramakrishnan, R. Bancilhon, A. Silberschatz: *Safety of recursive horn clauses with infinite relations*, Proc. ACM Symp. on Principles of Database Systems, 1987.

[14] M. Kifer, R. Ramakrishnan, A. Silberschatz: *An axiomatic approach to deciding query safety in deductive databases*, Proc. ACM Symp. on Principles of Database Systems, 1988.

[15] R. Krishnamurthy, R. Ramakrishnan, O. Shmueli: *A framework for testing safety and effective computability of extended Datalog*, Proc. ACM SIGMOD Symp. on the Management of Data, 1988.

[16] Y. Sagiv, M. Y. Vardi: *Safety of datalog queries over infinite databases.* Proc. ACM Symp. on Principles of Database Systems, 1989.

[17] http://graphlab.org

[18] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, Joseph M. Hellerstein: *Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud* PVLDB 2012

[19] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, Joseph M. Hellerstein: *GraphLab: A New Parallel Framework for Machine Learning.* Conference on Uncertainty in Artificial Intelligence, 2010.

[20] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica: *Spark: Cluster Computing with Working Sets*, HotCloud 2010

[21] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica: *Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing.* In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12), 2012

[22] J. Whaley, M. S. Lam: *Cloning-based context-sensitive pointer alias analyses using binary decision diagrams* In PLDI, 2004.

[23] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, R. C. Sears: *Boom analytics: Exploring data-centric, declarative programming for the cloud*, In EuroSys, 2010.

[24] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, R. Sears. *Dedalus: Datalog in time and space.* In Datalog, 2010

[25] Leslie G. Valiant, *A Bridging Model for Parallel Computation.* Comm. ACM 33(8), 1990, 103–111.

[26] Jeremy G. Siek, Lie-Quan Lee, Andrew Lumsdaine: *The Boost Graph Library: User Guide and Reference Manual.* Addison Wesley, 2002.

[27] Douglas Gregor, Andrew Lumsdaine: *The Parallel BGL: A Generic Library for Distributed Graph Computations.* Proc. of Parallel Object-Oriented Scientific Computing (POOSC), July 2005.

[28] Donald E. Knuth: *Stanford GraphBase: A Platform for Combinatorial Computing.* ACM Press, 1994.

[29] http://www.logicblox.com/technology.html, Accessed: September 18th, 2014

[30] http://www.datomic.com, Accessed: September 18th, 2014

[31] http://giraph.apache.com, Accessed: September 18th, 2014

[32] http://spark.apache.com, Accessed: September 18th, 2014

[33] http://hadoop.apache.com, Accessed: September 18th, 2014

[34] https://github.com/apache/spark, Accessed: September 18th, 2014

[35] https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920, Accessed: September 18th, 2014

[36] https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces50, Accessed: September 18th, 2014

[37] http://databricks.com/blog/2013/10/27/the-growing-spark-community.html, Accessed: September 18th, 2014

[38] http://mesos.apache.org, Accessed: September 18th, 2014

[39] http://aws.amazon.com, Accessed: September 18th, 2014

[40] http://pig.apache.org/

[41] http://hive.apache.org/

[42] Mario Alviano, Nicola Leone, Marco Manna, Giorgio Terracina, Pierfrancesco Veltri: *Magic-Sets for Datalog with Existential Quantifiers.* Datalog 2012: 31-43

[43] Mario Alviano, Wolfgang Faber, Nicola Leone, Marco Manna: *Disjunctive datalog with existential quantifiers: Semantics, decidability, and complexity issues.* TPLP 12(4-5): 701-718 (2012)

[44] Francois Bry, Tim Furche, Clemens Ley, Bruno Marnette, Benedikt Linse, Sebastian Schaffert: *Datalog relaunched: simulation unification and value invention*, Proceedings of the First international conference on Datalog Reloaded, March 16-19, 2010, Oxford, UK

[45] Francois Bancilhon, David Maier, Yehoshua Sagiv, Jeffrey D Ullman: *Magic sets and other strange ways to implement logic programs.* In Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems (PODS '86). ACM, New York, NY, USA.

[46] K. Tuncay Tekle, Yanhong A. Liu: *More efficient datalog queries: subsumptive tabling beats magic sets.* In Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD '11). ACM, New York, NY, USA.

[47] Claudio Martella, Roman Shaposhnik, Dionysios Logothetis: *Giraph in Action*, Early access edition, http://www.manning.com/martella/

[48] http://snap.stanford.edu/data/egonets-Twitter.html, Accessed: September 18th, 2014