Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Marek Rogala
Nr albumu: 277570

Deklaratywne zapytania na dużych grafach i ich rozproszone wyliczanie

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dra Jacka Sroki
Instytut Informatyki

Wrzesień 2014

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

W pracy przedstawiono metodę tłumaczenia programów zapisanych w Datalogu i jego rozszerzonych wersjach do programów w modelu obliczeniowym Google Pregel, oraz implementację prototypowego kompilatora wykorzystującego tę metodę do uruchamiania takich programów na platformie Apache Spark. Takie podejście pozwala na wykonywanie obliczeń na istniejących architekturach do obliczeń rozproszonych za pomocą deklaratywnego języka zapytań, znacznie prostszego niż dotychczas dostępne języki dla tych architektur.

Słowa kluczowe

Przetwarzanie dużych grafów, deklaratywne języki zapytań, obliczenia na dużych zbiorach danych, Datalog, SociaLite, Apache Spark, Hadoop.

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

Information Systems: Query languages

Tytuł pracy w języku angielskim

Declarative queries on large graphs and their distributed evaluation

Spis treści

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's Hadoop [33], as well as higher-level languages such as PigLatin [39] and Hive [40]. However, those 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 — *"the"* – tutaj powinno być? wydaje mi się że nie... 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 have 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 implementation of SociaLite available. The interpreter published by the authors is undocumented and contains many bugs. It is hard to imagine it being adopted in the industry in the foreseeable future. At the same time, papers [1] and [2] which introduced SociaLite contain certain simplifications and are not specific about some important details in definitions and proofs.

The goal of this thesis is to bridge the gap between the theoretical idea for SociaLite and a practical implementation and to draw a path towards its usage in the industry. We show how to translate SociaLite declarative programs into Pregel "think-like-a-vertex"programs and introduce a compiler that enables SociaLite programs to be executed on existing infrastructure. This allows its users to write and execute SociaLite programs without any additional effort to build a dedicated server infrastructure for that.

We present an experimental implementation of the SociaLite language on the Apache Spark platform. Spark [21] is an open-source project which provides 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. Distinctive features of Spark are the to ability 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 in since July 2014 it is included in the Cloudera CDH, a popular enterprise platform for Hadoop deployment. Spark is already a *bez a* ale to jest *a* od *a platform* 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 implementation platform for the S2P compiler.

The thesis consists of six chapters. In Chapter 1 we recall definitions of Datalog and its evaluation methods while Chapter 2 contains an introduction to the Pregel computation model. In Chapter 3 we describe the extensions introduced by SociaLite and provide formal definitions and general-case proofs which the original papers lack. Chapter 4 shows the translation procedure from SociaLite to Pregel programs implemented in the S2P compiler, which is described in Chapter 5. In Chapter 6 we sketch the possible future work and the path to industrial implementation of the language using the S2P compiler.

0.1. 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 relations identified by relation names, for example R, P, Tc, Path or Edge. Relation names will be usually denoted by a *R*. *wymień litery / oznaczenia jakie będą używane* — czy to o to chodziło?

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**.

Elements of **dom** are called *constants*, whereas 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. *wymień oznaczenia* czy o to chodziło?

jak się ma database schema do database? Najpierw definuję schemę a potem database (instance). Zmieniłem teraz w definicji *database instance* na po prostu *database*. Czy coś tu jeszcze powinienem zmienić?

Definition 0.1.1. *najpierw potrzebna def relacji i arity* Zmieniłem tutaj trochę i powinno być bardziej precyzyjnie, proszę o sprawdzenie.

A *database schema* is a tuple (N, ar), where N is some set and $ar : N \to \mathbb{N}^+$ is a function. The elements of N are called *relation names* R. For each relation name R, the value $ar(R)$ is called the *arity* of R.

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 $R(v)$ where $v \in \mathbf{dom}^n$ is a tuple.

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

A *database* $\mathbf{I}$ over database schema σ is a union $a \to an$ sprawdziłem że jest *a union* of relations over R, where $R \in \sigma$.

Definition 0.1.2. For a given set $V \subseteq \mathbf{var}$ of free variables, a function $\nu : V \to \mathbf{dom}$ is called a *valuation* over V.

Definition 0.1.3. If f is a function $f : D \to D$ and $f(x) = x$ for an $x \in D$, then x is called a *fix-point* of f.

TODO Powerset $\mathcal{P}$ TODO inst(sigma) TODO monotone

Definition 0.1.4.

Rozdział 1

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 can not handle. Examples of such problems are transitive closure of a graph or distances from a vertex to all other vertices.

Datalog is a language that enhances relational calculus with recursion, which allows for solving those problems. It appeared around 1978 and is inspired by logical programming paradigm. Recently, there is an increasing interest in Datalog research as well as its implementations in industry. Datalog is typically extended with negation and simple, non-recursive aggregation.

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

Let us suppose that we have a database with a binary relation Edge. The database represents a graph G: if $\textsc{Edge}(a, b)$ 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, unless we have some additional assumptions about G, is seems difficult to answer this query using languages like SQL. It can be proven that this kind of query is not expressible in the relational calculus [3]. Intuitively, what is necessary to answer such queries is some kind of conditional iteration or recursion, which is the most important feature of Datalog.

1.1. 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. It is believed that professor David Maier is 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], network systems [23, 24]. It is also used to formally define computational problems which can be solved with different models and frameworks, allowing for comparison of those frameworks and their optimizations [11].

Some of the most important fields of research concerning Datalog are optimizations in

programs evaluation (magic sets [44], subsumptive queries [45]) and extensions to the language [41, 42, 43].

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

1.2. Introduction to Datalog

Before we formally define Datalog syntax and semantics, 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 $\textsc{Edge}(a, b)$ means that there is an edge between vertices a and b. The following program computes relation Tc containing a transitive closure of relation Edge.

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

Rysunek 1.1: 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 also there is such 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 at the same time 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 is used: Tc appears on both sides of the second rule.

For example, let Edge contain the following tuples:

Edge
$(1, 2)$
$(2, 3)$
$(3, 4)$
$(2, 5)$

The result of the program is:

Tc
$(1, 2)$
$(1, 3)$
$(1, 4)$
$(1, 5)$
$(2, 3)$
$(2, 4)$
$(2, 5)$
$(3, 4)$

As we can see, the program defines a function from the an instance of relation EDGE into an instance of relation TC.

In the following sections, we will define Datalog's syntax and semantics in a more formal way.

1.3. Datalog syntax

Let us formally Datalog programs and rules.

Definition 1.3.1. A *rule* is an expression of the form:

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

where $n \geqslant 1$, $R, R_1, \ldots, R_n$ 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 the 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 body separated by commas are called *subgoals*. Head and the subgoals are called *atoms*. Each atom consists of a *predicate*, i.e. the relation name and *arguments*.

Definition 1.3.2. A rule is *safe* if the each free variable appearing in its head also appears in at least one of the subgoals.

Definition 1.3.3. 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 1.3.4. 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 are called the *edb* or *extensional database*, whereas the set of intensional relations is called *idb* or *intensional database*. For a 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.

A Datalog program is essentially a function from database instances over $edb(P)$ into database instances over $idb(P)$.

Definition 1.3.5. 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 by replacing each variable t in the rule by its value $\nu(t)$:

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

Example 1.3.1. As an example, let us consider the following program P:

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

Assuming that $\textsc{Parent}(p, c)$ means that p is c's parent and $\textsc{Woman}(x)$ and $\textsc{Man}(x)$ tell whether person x is a woman or a man, this program computes child's father, mother and all its ancestors that can be derived.

edb and *idb* for P are the following:

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

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

If $Anna, Chris, Patrick$ are some values in the domain, an example of an instantiation of the last rule is:

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

1.3.1. Differences between Datalog and Prolog

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 domain elements or variables: $p(x, y)$.
- in Prolog, there is a cut operator which is not present in Datalog. While some versions of Datalog have the notion of negation, but it is still different than the cut operator.
- Datalog requires the rules to be *safe*, which means that every variable mentioned in a rule must be also mentioned at least once in a non-negated, non-arithmetic sense

1.4. Datalog semantics

Semantics of a Datalog program can be defined using one of three different 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.

A second approach is *proof theoretic*, in which a fact is included in the result if and only if it can be derived, or 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.

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

1.4.1. Fix-point semantics

In this section we show the fix-point semantics for Datalog programs. A central notion in this definition is the *immediate consequence* operator. Intuitively, that operator adds to the database new facts that could be immediately derived using one of the rules.

Given a Datalog program P, let $\mathbf{K}$ be a database instance over $sch(P)$.

We say that a fact $R(v)$ is an *immediate consequence* for $\mathbf{K}$ and P if $R(v) \in \mathbf{K}$ 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 $T_P : inst(sch(P)) \to inst(sch(P))$:

$$T_P(\mathbf{K}) = \{F : F \text{ is a fact over } sch(P) \text{ and } F \text{ is an immediate consequence for } \mathbf{K} \text{ and } P\}$$

Lemma 1.4.1. *Operator T_P for any Datalog program P is a monotone function with respect to inclusion order.*

Proof. Given any $\mathbf{I}, \mathbf{J} \in inst(sch(P))$ such that $\mathbf{I} \subseteq \mathbf{J}$, let F be a fact in $T_P(\mathbf{I})$. By definition, F is an immediate consequence of $\mathbf{I}$, so either F is in $\mathbf{I}$ or it there exists an instantiation $F : -F_1, \ldots, F_n$ of a rule in P such that $F_i \in \mathbf{I}$ for each $i = 1 \ldots n$. In the first case $F \in \mathbf{I} \subseteq \mathbf{J}$, so $F \in \mathbf{J}$. In the second case, each $F_i \in \mathbf{I} \subseteq \mathbf{J}$, so the instantiation also exists in $\mathbf{J}$. Hence, F is also an immediate consequence of $\mathbf{J}$, and thus $F \in T_P(\mathbf{J})$. Since F was arbitrarily chosen, we have that $T_P(\mathbf{I}) \subseteq T_P(\mathbf{I})$ and T_P is a monotone function with respect to $\subseteq$.

Theorem 1.4.2. *For any P and an instance $\boldsymbol{K}$ over $edb(P)$, there exists a finite minimal fix-point of T_P containing $\boldsymbol{K}$.*

Proof. 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 that:

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

$adom(P) \cup adom(\mathbf{K})$ and the database schema $sch(P)$ of P are all finite, so there is a finite number n of database instances over $sch(P)$ using those values. Hence, 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 $T_P^*(\mathbf{K})$.

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 $\mathbf{K}$: $\mathbf{K} \subseteq \mathbf{J}$. By applying T_P n times to both sides of the inequality, we have that $T_P^*(\mathbf{K}) = T_P^n(\mathbf{K}) \subseteq \mathbf{T}_P^n(\mathbf{J} = \mathbf{J}$. Hence, $T_P^*(\mathbf{K})$ is the minimum fix-point of T_P containing $\mathbf{K}$.

Example 1.4.1. Let us recall the example program P from the previous section:

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

Given the following *edb* database instance **K**:

$$
\begin{array}{ll}
\textsc{Parent}(Anna, Bill) & \textsc{Woman}(Anna)\\
\textsc{Parent}(Bill, Chris) & \textsc{Woman}(Eva)\\
\textsc{Parent}(Anna, David) & \textsc{Man}(Bill)\\
\textsc{Parent}(Chris, Eva) & \textsc{Man}(Chris)\\
 & \textsc{Man}(David)
\end{array}
$$

The minimal fix-point of T_P containing **K** is:

$$
\begin{array}{lll}
\textsc{Parent}(Anna, Bill) & \textsc{Man}(Bill) & \textsc{Ancestor}(Anna, Bill)\\
\textsc{Parent}(Bill, Chris) & \textsc{Man}(Chris) & \textsc{Ancestor}(Bill, Chris)\\
\textsc{Parent}(Anna, David) & \textsc{Man}(David) & \textsc{Ancestor}(Anna, David)\\
\textsc{Parent}(Chris, Eva) & \textsc{Mother}(Anna, Bill) & \textsc{Ancestor}(Chris, Eva)\\
\textsc{Woman}(Anna) & \textsc{Mother}(Anna, David) & \textsc{Ancestor}(Anna, Chris)\\
\textsc{Woman}(Eva) & \textsc{Father}(Bill, Chris) & \textsc{Ancestor}(Anna, Eva)\\
 & \textsc{Father}(Chris, Eva) &
\end{array}
$$

1.5. 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 being having very simple formulation, this method is not efficient in a typical case due to excessive redundant computation. The most basic optimization adressing this problem is *semi-naive* evaluation, which tries to avoid computations that can not 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 are also other, more optimized evaluation methods, such as Magic Sets and Subsumptive queries as well. A top-down strategy 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.

1.5.1. Naive evaluation

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

In pseudocode, if T_P is the immediate consequence operator, the algorithm for evaluation of a program P on an input **K** can be written as:

$P(\mathbf{K}) = \{$
  $I_0 \leftarrow K$
  $i \leftarrow 0$
  do
    $i \leftarrow i + 1$
    $I_i \leftarrow T_P(I_{i-1})$
  while $I_i \neq I_{i-1}$
  return I_i
$\}$

Example 1.5.1. As an example, let us consider the following program, which computes a transitive closure of a binary relation R:

$$\textsc{Tc}(x, y) : -\mathrm{R}(x, y). \tag{1.1}$$
$$\textsc{Tc}(x, y) : -\textsc{Tc}(x, z), \textsc{Tc}(z, y). \tag{1.2}$$

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:

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

1.5.2. Datalog is inflationary

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 any of the 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*.

1.5.3. Semi-Naive evaluation

A straightforward implementation of T_P definition is to perform a natural join on subgoal relations and a projection to head variables. Example 1.5.1 shows that such implementation may be inefficient, because most of the facts is computed more than once.

Semi-naive evaluation is the most basic optimization used in Datalog evaluation, in which T_P in an optimized way. It comes from the following observation: in a Datalog program, if some rule Q produced a fact $R(t)$ based on database instance I_i in the i-th iteration of the naive evaluation algorithm, then this rule with produce this fact in each subsequent iteration, because of the inflationary semantics of the language. The goal of this optimization is to avoid those 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 rules of program P so that at least one new fact is used in 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, a 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 facts newly computed to get the same result as T_P: $T_P(I_i) = I_i \cup T_P^{\Delta}(I_i, \Delta_i)$ for each i. The following pseudocode presents the algorithm for semi-naive evaluation:

$P(\mathbf{K}) = \{$
  $I_0 \leftarrow K$ $\Delta_0 \leftarrow K$
  $i \leftarrow 0$
  do
    $i \leftarrow i + 1$
    $C_i \leftarrow T_P^{\Delta}(I_{i-1}, \Delta_{i-1})$
    $I_i \leftarrow C_i \cup I_{i-1}$
    $\Delta_i \leftarrow I_i - I_{i-1}$
  while $\Delta_i \neq \emptyset$
  return I_i
}

Example 1.5.2. Let us consider the program and input from Example 1.5.1. The facts computed by the Semi-naive evaluation in subsequent iterations would be:

$$C_1 \leftarrow \{\text{Tc}(1,2), \text{Tc}(2,3), \text{Tc}(3,4), \text{Tc}(4,5)\}$$

$$C_2 \leftarrow \{\text{Tc}(1,3), \text{Tc}(2,4), \text{Tc}(3,5)\}$$

$$C_3 \leftarrow \{\text{Tc}(1,4), \text{Tc}(2,5), \text{Tc}(1,5)\}$$

$$C_3 \leftarrow \{\text{Tc}(1,5)\}$$

Semi-naive evaluation does assure that each fact will be computed once, e.g. Tc(1, 5) was computed more than once, but it eliminates a significant portion of redundant computation.

1.5.4. 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 find x such that $\text{Tc}(4, x)$, which by the first rule yields $x \in \{5\}$. The final result is thus $x \in \{4\}$

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

This can be also achieved in bottom-up evaluation by using optimization techniques such as *Magic sets* [44, 3] and *Subsumptive queries* [45]. 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.

1.6. 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 $\leqslant$
- arithmetic functions, like addition and multiplication
- negation
- non-recursive aggregation

It this section we will briefly describe these extensions.

1.6.1. Arithmetic predicates

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

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

Boss
(a, b)
(b, c)
(b, d)

Salary
(a, 10)
(b, 15)
(c, 5)
(d, 20)

The following query with arithmetic comparisons solves this problem:

$$\text{EarnsMoreThanBoss}(employee) :-$$
$$\text{Boss}(boss, employee), \text{Salary}(boss, bs), \text{Salary}(employee, es), es > bs.$$

We can think of arithmetic comparisons as a new kind of predicates, which are infinite built-in relations. Since we introduced implicit infinite relations, we need to adjust the definition of rule safety 1.3:

Definition 1.6.1. A rule with arithmetic comparisons is *safe* if 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 assures that comparisons do not introduce any new values into the database.

1.6.2. Arithmetic functions

Addition of arithmetic functions is a next step after arithmetic comparisons. 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.

An adjusted version of definition of rule safety 1.6.1 is:

Definition 1.6.2. A rule in Datalog with arithmetic comparisons and assignments is *safe* if each free variable appearing in:

- its head,
- any of the comparisons
- or on the right side of any of the assignment subgoals

also appears in at least one of the relational subgoals or on the left side of an assignment subgoal.

Example 1.6.1. 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 come back to this question in section 1.6.4. For now, let us answer a simpler question: supposing that G is a directed acyclic graph, for each vertex v in G, what are the lengths of paths between s and v?

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

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

Rysunek 1.2: Datalog query for computing all path lengths from a given source

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

Introduction of arithmetic functions significantly changes the semantics. Similarly to arithmetic comparisons, arithmetic functions can be interpreted an 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 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 on 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 semantics for Datalog programs that have a finite fixed point. The updated version of Theorem 1.4.1 is as follows.

Theorem 1.6.1. *For any P and an instance $\boldsymbol{K}$ over $edb(P)$, if there exists $n \geqslant 0$ such that $T_P^n(\boldsymbol{K})$ is a fix-point of T_P, then is it the minimal fix-point of T_P containing $\boldsymbol{K}$.*

Proof. See second part of the proof for Theorem 1.4.1.

1.6.3. 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, or 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. Arithmetic comparisons and assignments are not allowed to be negated.

Example 1.6.2. Let us consider the program for computing trasitive closure TC of a relation R from example 1.5.1. A following rule computes the pairs of nodes which are indirectly connected, i.e. are in TC, but not in R:

$$\text{INDIRECT}(x, y) : -\text{R}(x, y),\ !\text{TC}(x, y).$$

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

Definition 1.6.3. A rule in Datalog$^\neg$ with arithmetic comparisons and arithmetic assignments is *safe* if each free variable appearing in:

- its head,
- any of the comparisons,
- on the right side of any of the assignment subgoals
- or in any of its negated subgoals

also appears in at least one of the non-negated relational subgoals or on the left side of an assignment subgoal.

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

Definition 1.6.4. A Datalog$^\neg$ program P is *semi-positive* if for each rule in P, all its negated subgoals are over $edb(P)$.

For a semi-positive program, any relation used in a negative sense is an *edb* relation, so it is constant during the evaluation of P. Negation could be eliminated from P by introducing artificial negated *edb* relations. 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 assume 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 a 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 can be fully computed before evaluation of relation R_1. Intuitively, this 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 from each other. This is formalized this by the notion of strata.

Definition 1.6.5. Let P be a program in Datalog$^\neg$ and $n = \mathsf{idb(P)}$ be then number of *idb* relations in P. A function $\rho : sch(P) \to 1, ..., n$ is called *stratification* of P if such that for each rule ϕ in P with head predicate T, the following are true:

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

Definition 1.6.6. A program that has stratification is called *stratifiable*.

gm For each relation $R \in idb(P)$, $\rho(R)$ is called its *stratum number*.

ρ corresponds to a partitioning of P into several subprograms $P_1, P_2, \ldots, P_n$. Each of those programs is called a *stratum* of 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 those relations are *defined* in P_i.

Stratification assures that if a relation R is used in rules of stratum P_i, 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, $idb(P_i)$ consists of the relations defined in this stratum, while $edb(P_i)$ may contain only relations defined in earlier strata or relations from $edb(P)$. By definition of stratification, the negative subgoals in rules of P_i use only relations in $edb(P_i)$. 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 $P_1, P_2, \ldots P_n$, then the output of program P on input $\mathbf{I}$ is defined by applying programs $P_1, P_2, \ldots P_n$ in a sequence:

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

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

DO ROZWAŻENIA tutaj można by dać ilustrację grafu zależności

1.6.4. Datalog with non-recursive aggregation

Datalog with negation and arithmetics is already a useful language, but for some queries one more feature is necessary: aggregation using a certain function f. Aggregation works similarly to GroupBy 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:

Rel
$(1, 5, 5)$
$(1, 5, 3)$
$(1, 5, 4)$
$(2, 3, 4)$
$(2, 3, 5)$
$(2, 4, 6)$

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

Aggregated-Rel
$(1, 5, 3) = (1, 5, \min\{5, 3, 4\})$
$(2, 3, 4) = (2, 3, \min\{4, 5\})$
$(2, 4, 6) = (1, 5, \min\{6\})$

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: such 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 in a simple way 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 relations defined in preceding strata. Aggregation of a relation from the same stratum, i. e. recursive aggregation, is much more complicated and is discussed in Chapter 3.

For an example, let us recall the program from example 1.2, which computes for a given graph the lengths of all existing paths from source to other vertices. An interesting question in often what is the shortest path to each vertex. This question can be answered using aggregation, by computing the minimum of distances for each vertex:

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

Rysunek 1.3: Datalog query for computing all path lengths from a given source

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

In this example, there are two strata: 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.

Rozdział 2

The Pregel model for graph computations and its implementations

Pregel is a computational model designed 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 instead focus on the problem at hand.

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

Since introduction of Pregel, there have been many systems developed based on this model, most notably Apache Giraph, the open source implementation of the Pregel model. Pregel has been included in other, more general frameworks such as Spark as one of the available APIs. There have also been extensions to the model, such as Giraph++ [8].

In the first section of this chapter, the Pregel model is described. The subsequent sections cover the most important open-source implementations of the model: Apache Giraph and Apache Spark's GraphX.

2.1. 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 a Königsberg citizen could take a walk around the city so that he crossed each of the seven city's bridges exactly once. Euler concluded that it is 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 river that flows through Königsberg and which the famous bridges spanned is Pregel.

The model of computation in Pregel is based on the L. Valiant's Bulk Synchronous Parallel model. 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 $S + 1$.

DO ROZWAŻENIA Może warto dać tutaj dragram ilustrujący BSP, z obliczeniami, komunikacją i barierą?

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 what happens in the MapReduce model, is well suited for distributed computations, since all local functions can be executed completely independently. 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, and *outgoing edges*. Each such edge 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 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 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. 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.

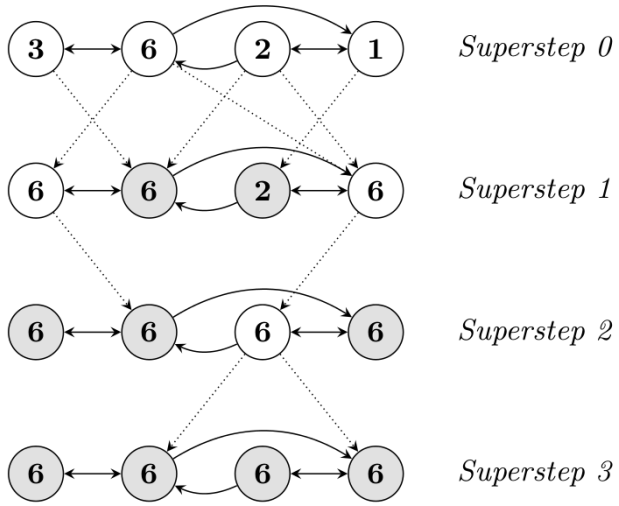
Let us consider the following example: for a strongly connected graph with an integer value assigned to each node, compute the minimum of those values. This can be implemented in Pregel using the following vertex program:

In the first superstep, a vertex sends its value to all neighbors, and votes to halt. Upon 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 on figure 2.1. This figure comes from the original white paper about Pregel [7].

For 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 send messages

```
vertexProgram(vertex, superstepNumber, incomingMessages){
    newValue ← min(incomingMessages ∪ {vertex.value});
    if superstepNumber = 0 or newValue < vertex.value then
        foreach edge ∈ vertex.outgoingEdges do
            sendMessage(edge.targetVertex, newValue);
        vertex.value ← newValue;
    else
        voteToHalt();
}
```

Rysunek 2.1: Pregel vertex program for computing maximum value among graph nodes



Rysunek 2.2: Example Pregel computation of maximum value among the nodes. Nodes that voted to halt are marked gray. Source: [7]

to their neighbors with a new possible distance. Each 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.

```
vertexProgram(vertex, superstepNumber, incomingMessages){
    initialDistance ← if (vertex.id = SOURCE) 0 else ∞
    minDistance ← min(incomingMessages ∪ {initialDistance});
    if newValue < vertex.value then
        vertex.value ← minDistance
        foreach edge ∈ vertex.outgoingEdges do
            sendMessage(edge.targetVertex, minDistance + edge.length);
    voteToHalt();
}
```

Rysunek 2.3: Pregel vertex program for shortest paths from *SOURCE* to all other vertices

An important goal in large dataset computations is to achieve fault-tolerance, so 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 fails, the computation is resumed from the last checkpoint.

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 send from vertices placed on a given node.

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

2.2. Giraph

Giraph [31] is an open-source implementation of the Pregel model. It implements the original Pregel loosely, adding a few extensions to the version originally described by Google. Those extensions include introducing a possibility to perform computations on the master node, capabilities to work with a support of external memory and removing the single-point-of-failure (SPoF) 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, Hadoop MapReduce — an open implementation of the MapReduce model and libraries for using these elements in other projects.

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 using the project extensively and engaged in the development. The most active user is Facebook, which executes Giraph programs on social graphs with up to $10^12 edges$ [34]. In 2013, Facebook published an article [34] 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 be able 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. This eliminates the need for preprocessing the graph with a MapReduce job so that each vertex data is in the same place as all its edges.
- Multithreaded execution of 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 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 Google Pregel creators and the primary author of the original white paper about it, Grzegorz Malewicz, is one of the members of Facebook's data infrastructure graph processing team which develops Giraph [34].

2.3. Spark

Apache Spark [20, 21] is an open-source framework for distributed data analytics. It started in 2009 in the AMPLab at University of California, Berkeley. Since that time it has gained a huge momentum and has quickly growing community of users and contributors [36]. Since 2009, over 250 individual developers contributed to Spark, and its permanent contributors come from 12 companies and institutions [32]. Among others, it is being used by companies such as Yahoo, IBM, Intel, Alibaba, Cloudera and Databricks.

Spark's computational model is able to express programs in 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, 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 Apache Hadoop

Similarly to Apache Giraph, Spark is a Top-Level Project of the Apache Foundation. Previously to promotion as a Top-Level Project in February 2014 [35], it was an 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 [37] and Amazon EC2 [38].

2.3.1. Resilient Distributed Datasets

The key concept in Spark is the *RDD* or *Resilient Distributed Dataset*. 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 assure fault tolerance.

Other distributed memory abstractions, such as distributed key-value stores and databases, provide a fine-grained interface to the memory, like reads and updates of particular cells in a table. In contrast to such systems, RDDs provide only a *coarse-grained* interface: operations that are applied 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 materialised, unless it is actually necessary. Since parallel computations generally apply some transformation 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: 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* of an RDD by indicating which RDDs are intended to be used in the future and as such should be materialised in the memory and the *partitioning* of an 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 of exports the dataset to some persistent storage. Avaiable 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.

2.3.2. 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. Currently available tools are:

- Spark SQL, allowing to seamlessly integrate 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

2.3.3. Pregel API

One of the elements of the GraphX library is a general Pregel API, which allows for expressing programs written in the Pregel model and working on RDDs. Chapter 5 describes an implementation of a Datalog API for Spark, which employs the method for translating Datalog and SociaLite programs to the Pregel model and uses Spark's Pregel API.

2.4. Other frameworks

DO ROZWAŻENIA Tutaj można by opisać inne frameworki: GPS, GraphLab, Giraph++. Ale może nie potrzeba?

Rozdział 3

SociaLite

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 then be executed in a distributed environment.

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 describe recursive aggregation, which is the most important feature in Sequential SociaLite, in Section 3.1.

[2] extends Sequential Socialite to *Distributed SociaLite*, executable on a distributed architecture. It introduces a *location operator*, which determines how the data and computations can be distributed. The programmer does not have to think about how to distribute the data between machines or manage the communication between them. He only specifies an abstract *location* for each fact in the data, and the data and computations are automatically sharded. Distributed SociaLite is covered in section 3.3

The declarative semantics of Datalog and SociaLite does not specify how the programs should be evaluated. This property enables applying various optimizations in the evaluation. An example of such optimization is shown in Section 3.4. TODO Czy dam radę to zrobić?

In distributed computations on large graphs, an approximate result is often enough. Usually we can observe the *long tail* phenomenon in the computation, where a good approximate solution is achieved quickly, but it takes a long time to get to the optimal one. Declarative query language such as SociaLite can be easily extended to return approximate results very fast, in a way that is described in Section 3.5.

3.1. Datalog with recursive aggregation

In this section we introduce the recursive aggregation extension from SociaLite. Since the original 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} .

3.1.1. Motivation

Most graph algorithms are essentially some kind of iteration or recursive computation. Simple recursion can be expressed easily in Datalog. However, in many problems the computation results are gradually refined in each iteration, until the final result is reached. Examples of such algorithms are the Dijkstra algorithm for single source shortest paths or PageRank. Usually, it is difficult or impossible to express such algorithms in Datalog efficiently, as it would require computing much more intermediate results than it is actually needed to obtain the solution. We will explain that on an example: a simple program that computes shortest paths from a source node.

A straightforward program in Datalog with nonrecursive aggregation for computing single source shortest paths (starting from node 1) is presented in Figure ex:ssspsocialite. Due to 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 this approach results in bad performance, but causes the program to execute infinitely if a loop in the graph is reachable from the source node. TODO czy to jest w Datalogu? trzeba wyjaśnić że to już jest rozszerzenie a teraz tylko chodzi o sposób wyliczania i zapętlanie

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

Rysunek 3.1: Datalog query for computing shortest paths from node 1 to other nodes

Datalog^{RA} allows aggregation to be combined with recursion under some conditions. This allows us to write straightforward programs for such problems, which finish execution in finite time and often are much more efficient than Datalog programs. An example Datalog^{RA} program that computes single source shortest paths is presented below. The relation Path is declared so that for each *target* the values in *dist* column are aggregated using minimum operator.

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

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

Rysunek 3.2: SociaLite query for computing shortest paths from node 1 to other nodes

3.1.2. A program in Datalog^{RA}

A Datalog^{RA} program is a Datalog program, with additional aggregation function defined for selected columns of some relations. This function needs to fulfill requirements that will be stated in the next section.

For each relation name R, there can be one column $aggcol_R \in 1, \ldots ar(R)$ chosen for which an aggregation function $aggfun_R$ is provided. This column is called the *aggregated column*. The rest of the 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 $aggfun_R$. Value $aggcol_R = \textbf{none}$ means that R is a regular relation with no aggregation.

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

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

An example of a program in Datalog^{RA} in show on Figure 3.1.2.

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

$$
\begin{array}{lll}
\mathrm{P}(x_1, \ldots, x_{ar_P}) & :- & Q_{P,1}(x_1, \ldots, x_{ar_P}) \\
 & \ldots & \\
\mathrm{P}(x_1, \ldots, x_{ar_P}) & :- & Q_{P,m}(x_1, \ldots, x_{ar_P}) \\
\mathrm{R}(x_1, \ldots, x_{ar_R}) & :- & Q_{R,1}(x_1, \ldots, x_{ar_R}) \\
 & \ldots & \\
\mathrm{R}(x_1, \ldots, x_{ar_R}) & :- & Q_{R,m}(x_1, \ldots, x_{ar_R})
\end{array}
$$

Rysunek 3.3: Structure of a program in Datalog^{RA} .

3.1.3. 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 operato T_P. This definition assumes that T_P is inflationary with respect to inclusion order on database instances. This requirement means that T_P only adds facts to the database instance, but never removes facts from it. This is also the reason for which program 3.1 is inefficient: the inflationary semantics forces all suboptimal distances to nodes 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 new aggregated value appeared. However, an inflationary T_P operator is necessary for the proof that the fixed point semantics always gives a unique solution. In order to define semantics for Datalog^{RA} in terms of fixed point, we need to use a different order on database instances than the regular set inclusion order.

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

Join operation and induced ordering

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

Definition 3.1.1. A binary operation $\odot : X \times X \to X$ is:

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

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

Definition 3.1.2. A binary operation is a *join* operation if it is idempotent, commutative and associative. TODO Maybe citation needed?

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

Example 3.1.1. $\max(a, b)$ 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, minimum of two numbers is a join operation. On the contrary, $+$ is not a meet operation, since it is not idempotent: $1 + 1 \neq 1$.

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 partial order and semi-lattice.

Definition 3.1.3. A binary relation $\leqslant$ over a set P is a *partial order* if it is reflexive, antisymmetic and transitive, i. e. for each $x, y, z \in P$ the following properties are satisfied:

- $x \leqslant x$
- if $x \leqslant y$ and $y \leqslant x$ then $x = y$
- if $x \leqslant y$ and $y \leqslant z$ then $x \leqslant z$

Definition 3.1.4. A set P with a partial order $\leqslant$ over P is a *join semilattice* if every two elements of P have a least upper bound with respect to $\leqslant$. For any two elements, their least upper bound is called a *join* of those elements.

A join operation $\sqcup$ defines a semi-lattice: it induces a partial order $\leqslant_\sqcap$ over its domain, such that the result of the operation for any two elements is the least upper bound of those elements with respect to $\leqslant_\sqcap$

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

Aggregation operation 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 within each such group performs the aggregation on the aggregated column.

Definition 3.1.5. For a relation R of arity $k = ar(R)$, let us define $g_R : \mathcal{P}(\mathbf{dom^k}) \to \mathcal{P}(\mathbf{dom^k})$:

$$g_R(I) = \begin{cases} \Big\{(x_1, \ldots, x_{k-1}, t) : (x_1, \ldots, x_{k-1}, x_k) \in I \wedge \\ \qquad t = \mathit{aggfun}_R(\{y : (x_1, \ldots, x_{k-1}, y) \in I\})\Big\} & \text{if } \mathit{aggcol}_R \neq \textbf{none} \\ I & \text{otherwise} \end{cases}$$

If R has an aggregated column, g_R groups the facts in relation instance I by qualifying parameters and performs the aggregation using aggfun_R. For non-aggregated relations, g_R is an identity function.

Order over relation instances

In Datalog, we can prove that there is a unique least fixed point for any program. The fundamental fact needed for this proof is that it is inflationary: during 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 different fact with a lower value in the aggregated column. To be able to define semantics of programs in Datalog^{RA} using least fixed point, we need to use a custom order on relation instances. This order is built based on the function g_R.

Definition 3.1.6. Let R be a relation name and $k = ar(R)$ Let us define comparison $\sqsubseteq_R$ on relation instances as follows:

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

Remark. If R does not have an aggregated column, $\sqsubseteq_R$ is simply the inclusion order $\subseteq$.

Example 3.1.2. Let R be a relation with arity 3, with the last column aggregated using join operation max. We recall that for max, $\leqslant_{\max}$ is the usual order $\leqslant$.

- $\{(1, 2, 3)\} \sqsubseteq_R \{(1, 2, 5)\}$, because $3 \leqslant 5$
- $\{(1, 2, 3)\} \sqsubseteq_R \{(1, 2, 5), (1, 7, 2)\}$, because $3 \leqslant 5$
- $\{(1, 2, 3), (1, 2, 8)\} \sqsubseteq_R \{(1, 2, 5)\}$, because $g_R(\{(1, 2, 3), (1, 2, 8)\}) = \{(1, 2, 3)\}$ and $3 \leqslant 5$

- $\{(1,2,3),(2,8,1)\}$ and $\sqsubseteq_R \{(1,2,5),(1,7,2)\}$ are not comparable
- $\emptyset \sqsubseteq_R \{(1,2,3)\}$

We can easily see that for any R an empty relation instance $\emptyset$ is smaller under $\sqsubseteq_R$ than any other relation instance.

$\sqsubseteq_R$ is not necessarily a partial order over the set of relation instances $\mathcal{P}(\mathbf{dom}^k)$, because it is not antisymmeric. In example 3.1.3, 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 relation satisfies the two remaining requirements for partial order: reflexivity and trasitivity. Such a relation is called a *pre-order*.

Lemma 3.1.1. *For any R, $\sqsubseteq_R$ is a pre-order over $\mathcal{P}(\mathbf{dom}^{ar(R)})$.*

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 does have an aggregated column, then:

- $\sqsubseteq_R$ is reflexive: for each R, we have that $\forall_{(q_1,\ldots,q_{k-1},v)\in g(R)} \exists_{(q_1,\ldots,q_{k-1},v)\in g(R)}\ v \leqslant_{aggfun_R} v$ because $\leqslant_{aggfun_R}$ is reflexive. Hence, $R \sqsubseteq_R R$.
- $\sqsubseteq_R$ is transitive: if $A \sqsubseteq_R B$ and $B \sqsubseteq_R C$, then by definition of $\sqsubseteq_R$ we have that:

$$\forall_{(q_1,\ldots,q_{n-1},a)\in g(A)} \exists_{(q_1,\ldots,q_{n-1},b)\in g(B)}\ a \leqslant_{aggfun_R} b$$

$$\forall_{(q_1,\ldots,q_{n-1},b)\in g(B)} \exists_{(q_1,\ldots,q_{n-1},c)\in g(C)}\ b \leqslant_{aggfun_R} c$$

  $\leqslant_{aggfun_R}$ is transitive, because it is a partial order, so $\forall_{(q_1,\ldots,q_{n-1},a)\in g(A)} \exists_{(q_1,\ldots,q_{n-1},c)\in g(C)} a \leqslant_{aggfun_R} c$, which means that $A \sqsubseteq_R C$.

Order over relation instances after aggregation

We already know that $\sqsubseteq_R$ is a pre-order over the set of all possible relation instances $\mathcal{P}(\mathbf{dom}^{ar(R)})$, but because of lack of antisimmetry, it is not guaranteed to be a partial order. However, if we restrict to the relation instances that are possible after aggregation is applied, the relation is antisimmetric.

For any R such that $k = ar(R)$, let Z_R denote the set of relation instances that can be obtained by applying g_R to any relation instance:

$$Z_R = \{g_R(I) : I \in \mathcal{P}(\mathbf{dom}^k)\}$$

Lemma 3.1.2. *For any R such that $k = ar(R)$ and $aggcol_R \neq$ **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 $R(x_1,\ldots,x_n) \in I$.*

Proof. $I = g_R(I')$ for some I'. By definition of g_R, there is at most one fact $R(x_1,\ldots,x_n)$ in $g_R(I')$ for each $(x_1,\ldots,x_{n-1})$. Hence, in application of g_R to I the aggregated value for each $x_1,\ldots,x_{n-1}$ is simply x_n, so $g_R(I) = I$.

Lemma 3.1.3. *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 does have an aggregated column, then by Lemma 3.1.3 it is a pre-order over $\mathcal{P}(\mathbf{dom}^k) \supseteq Z_R$, so it only remains to be shown that $\sqsubseteq_R$ is antisimmetric.

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 antisimmetry, we need to show that $A = B$. By definition of $\sqsubseteq_R$, we have that:

$$\forall_{(q_1,\ldots,q_{n-1},a)\in g(A)} \exists_{(q_1,\ldots,q_{n-1},b)\in g(B)}\ a \leqslant_{aggfun_R} b$$
$$\forall_{(q_1,\ldots,q_{n-1},b)\in g(B)} \exists_{(q_1,\ldots,q_{n-1},a)\in g(A)}\ b \leqslant_{aggfun_R} a$$

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 3.1.3, $g_R(A) = g_R(g_R(A')) = g_R(A') = A$, and similarly $g_R(B) = B$, so we can formulas above are equivalent to:

$$\forall_{(q_1,\ldots,q_{n-1},a)\in A} \exists_{(q_1,\ldots,q_{n-1},b)\in B}\ a \leqslant_{aggfun_R} b$$
$$\forall_{(q_1,\ldots,q_{n-1},b)\in B} \exists_{(q_1,\ldots,q_{n-1},a)\in A}\ b \leqslant_{aggfun_R} a$$

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 $a \leqslant_{aggfun_R} b$. Futher, we know that there exists $(x_1, ..., x_{n-1}, a') \in A$ such that $b \leqslant_{aggfun_R} a'$. By Lemma 3.1.3, it must hold that $a = a'$. Since $a \leqslant_{aggfun_R} b \leqslant_{aggfun_R} a'$, we have that $a = b$, so $t \in B$. Therefore, $A \subseteq B$, because t was chosen as any element of A. Because of symmetry of the proof, it also holds that $B \subseteq A$, so $A = B$. This means that $\sqsubseteq_R$ is indeed antisimmetric.

$\sqsubseteq_R$ is an antisimmetric pre-order over Z_R, so 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 defined on relation instances to an order on database instances in a straightforward way, by comparing the databases relation-by-relation:

Definition 3.1.7. 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:

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

Remark. Note that if there is no aggregation, all relation instances are simply compared using the regular inclusion order. Since relation instances for different relation names are always disjoint, $\sqsubseteq_\sigma$ is the same as $\subseteq$ in this case.

3.1.4. Semantics for Datalog^{RA} programs

In this subsection we will show that the semantics of a Datalog^{RA} program can be unambiguously defined using least fixed point of the newly introduced order, as long as it satisfies some conditions.

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. P has the form of:

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 $x_1, \ldots, x_{k_i}$. Relation R_i may have an aggregation

$R_1(x_1, \ldots, x_{k-1}, x_k \text{ [aggregate } F_1])$
$R_w(x_1, \ldots, x_{k-1}, x_k \text{ [aggregate } F_w])$

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

function $aggfun_{R_i}$ defined for column $aggcol_{R_i}$. Each such $aggfun_{R_i}$ is required to be a join operator. $aggcol_{R_i}$ is allowed to be any column in $1, 2, \ldots, k_i$, but to simplify the notation we assume it may be only the last column k_i.

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, $R_1, \ldots, R_w$.

By definition, P is a program P' in regular Datalog, with *aggcol* and *aggfun* additionally defined. P can be treated as a triple $(P', aggcol, aggfun)$. Hence, exists an immediate consequence operator T'_P for regular Datalog part of program P. This operator will be the base for defining P's semantics in Datalog^{RA} .

Another important building block in Datalog^{RA} semantics is an aggregation operator on database instances. Intuitively, it applies of the corresponding aggregation operator g_R to each relation instance I of relation R in a database instance.

Definition 3.1.8. TODO moze lepiej G_P ? Let σ be a database schema and $\mathbf{K}$ be a database instance 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. Let the *aggregation operator* G_σ for $\mathbf{K}$ be defined as:

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

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

Definition 3.1.9. The *immediate consequence operator* for a Datalog^{RA} program $P = (P', aggcol, aggfun)$, where P' is a program in Datalog, is a function $T_P : inst(sch(P)) \to inst(sch(P))$:

$$T_P = G_{sch(P)} \circ T_{P'}$$

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

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

Proof. G_σ is monotone with respect to $\sqsubseteq_\sigma$. Since we assumed that $T_{P'}$ is monotone with respect to $\sqsubseteq_\sigma$, $T_P = G_\sigma \circ T_{P'}$ is also monotone with respect to $\sqsubseteq_\sigma$ as a composition of two monotone functions.

Because of monotonicity of T_P, we have inductively that $T_P^i(\mathbf{K}) \sqsubseteq_\sigma T_P^{i+1}(\mathbf{K})$ for each $i \geqslant 0$. Therefore:

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

$adom(P) \cup adom(\mathbf{K})$ and the database schema $sch(P)$ of P are all finite, so there is a finite number n of database instances over $sch(P)$ using those values. Hence, 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 $T_P^*(\mathbf{K})$.

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 $\mathbf{K}$.

TODO Czy ten dowód to nie jest zbyt mocne powtórzenie 1.4.1?

3.1.5. 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 also in Datalog and described in detail in Section 1.5.1, can be directly applied also in Datalog^{RA} . The only difference is the immediate consequence operator which is used: to evaluate Datalog^{RA} programs, one needs to use the T_P operator for Datalog^{RA} , which takes into account the aggregation to be applied.

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 1.5.3, T_P 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 rules of program P on database instance I and a set of new facts from last iteration Δ, so that at least one new fact is used in application of each rule. Output of T_P^Δ is then merged with the current database instance.

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

$P(\mathbf{K}) = \{$
  $I_0 \leftarrow K$ $\Delta_0 \leftarrow K$
  $i \leftarrow 0$
  do
    $i \leftarrow i + 1$
    $C_i \leftarrow T_P^\Delta(I_{i-1}, \Delta_{i-1})$
    $I_i \leftarrow G_\sigma(C_i \cup I_{i-1})$
    $\Delta_i \leftarrow I_i - I_{i-1}$
  while $\Delta_i \neq \emptyset$
  return I_i
$\}$

The only difference from the algorithm for Datalog without aggregation described in Section 1.5.3 is that is each step G_σ is applied to the newly computed database instance.

3.1.6. Datalog^{RA} with negation

TODO Basically we do the same thing as in regular Datalog – stratification

3.2. Tail-nested tables

Another important extension in SociaLite are *tail nested tables*, which optimize the memory layout so that it can be accessed in a faster way. While being very useful in practice, this optimization is not crucial for running such programs on distributed architecture. TODO I don't want to have this in the compiler, but maybe describe here?

3.3. Distributed SociaLite

TODO koniecznie

3.4. Delta stepping in Distributed SociaLite

TODO moze?

3.5. Approximate evaluation in Distributed SociaLite

TODO moze?

Rozdział 4

Translating SociaLite programs to Pregel

```
SociaLite to Giraph compilation plan

Input: AST from parser
Goal output: Giraph program

AST is in the form:
Program (
[Declarations]
[Rules]
)

Put all declarations in Env. There are 3 kinds of declarations:
constant value
sharded table
non-sharded table
Sharded tables define ranges of vertices on which they will be stored. -> compute number
Non-sharded tables will be stored on Master node.
Verify semantic validity of each rule according to Enviroment.
Build rules graph.
*Divide rules into strata based on negations.
Find recursive cycles and for each rule identify in which recursive cycles it is.
*Check that each recursive cycle is within one strata (error otherwise)
add artificial rules and tables so that all computation is based on local data, and the o
We require that each table appears only once on the left side (but can have many alternat
For each regular rule:
result[u](args) :-     conditions_set_1;
   conditions_set_2;
   ...
   conditions_set_n.
emit:
if new_data not present in dest_table:
save dest_table(new_data)
for each conditions_set:
```

```
if dest_table in conditions_set:
try find new results from this rule using new_data
if found, send them to destination vertex u
For each aggregation in recursive cycle:
result[u](args, AGREGATE(arg)) :-  conditions_set_1;
    conditions_set_2
    ...
    conditions_set_n.
check whether it is a meet operation and monotonic. If yes, then emit code similar to the or
If not, then additionally before looking for new results send out messages to invalidate pre
For each global (non-sharded) aggregation, generate a Giraph aggregator for it.



Basic idea: one vertex for each value of (some) variables in statements.
Vertex remembers all statements which have this value on either variable.

Example:
Let us have the following statements:
EDGE(1, 2, 10000), EDGE(2, 3, 5000), EDGE(1, 3, 20000),
PATH(1, 2, 10000), PATH(2, 3, 5000), PATH(1, 3, 15000)
We would then have vertices 1, 2, 3:
1 { EDGE(1, 2, 10000), EDGE(1, 3, 20000), PATH(1, 2, 10000), PATH(1, 3, 15000)
2 { EDGE(1, 2, 10000), EDGE(2, 3, 5000), PATH(1, 2, 10000), PATH(2, 3, 20000)
3 { EDGE(1, 3, 20000), EDGE(2, 3, 5000), PATH(1, 3, 15000), PATH(2, 3, 20000)
What values need to have their vertices depends on the program, but we probably don't need v

DOUBLE_EDGE(p, r) :- EDGE(p, q), EDGE(q, r)


Datalog/Socialite
Pregel idea
Shortest Paths, Datalog


PATH(t, d) :- EDGE(1, t, d).


PATH(t, d) :- PATH(s, d1), EDGE(s, t, d2), d = d1 + d2.



MINPATH(t, $MIN(d)) :- PATH(t, d).



One vertex for each t value. Vertex initially has all EDGE(_, t, _) and EDGE(_, t, _) values
```

```
Vertex initializes its PATH(t, d) value according to the rule.


Repeat until no change: each vertex tries to match the rule as s vertex, and sends out ne

Each vertex computes min of its PATH values and stores its MINPATH.
Shortest Paths, SociaLite

EDGE (int src: 0..10000, (int sink, int len)).
PATH (int sink: 0..10000, int dist).


PATH(t, $MIN(d)) :- EDGE(1, t, d);

                 :- PATH(s, d1), EDGE(s, t, d2), d = d1 + d2.


One vertex for each t value. Vertex initially has all EDGE(_, t, _) and EDGE(_, t, _) val

Vertex initializes its PATH(t, d) value according to the rule.

Repeat until no change: each vertex tries to match the rule as s vertex, and sends out ne

Note 1: this is basically the standard version of Shortest Paths in Pregel. Different tha
Connected Components, SociaLite

int N = 1768195.
EDGE (int src: 0..N, (int sink)).
NODES (int n: 0..N).
COMP (int n: 0..N, int root).
COMPIDS (int id).
COMPCOUNT (int cnt).

COMP(n, $MIN(i))  :- NODES(n), i = n;

:- COMP(p, i), EDGE(p, n).


COMPIDS(id) :- COMP(_, id).

COMPCOUNT($SUM(1)) : - COMPIDS(id).
```

```
Set COMP(n, n)

Repeat until no change: always after updating the minimum, vertex p sends new values to all

Each vertex rewrites the value.

Compute SUM using Pregel aggregation mechanism.
Triangles, SociaLite

int N = 1768195.
EDGE (int src: 0..N, (int sink)) orderby sink .
TRIANGLE (int x, int y, int z).
TOTAL (int cnt).

TRIANGLE(x, y, z) : - EDGE(x, y), x < y, EDGE(y, z), y < z, EDGE(x, z).



TOTAL($SUM(1)) :- TRIANGLE(x, y, z).




Two Pregel supersteps: 1) Intermediate join of EDGE(x, y), x < y, EDGE(y, z), by vertex y
2) Join of EDGE_TMP(x, z), y < z, EDGE(x, z) by vertex x (or z)

Compute SUM using Pregel aggregation mechanism.
```

Rozdział 5

Implementation

Rozdział 6

Summary

Bibliografia

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

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

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

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

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

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

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

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

[42] 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)

[43] 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

[44] 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.

[45] 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.

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

TODO http://giraph.apache.org/literature.html