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

TODO – graph queries, Datalog

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

TODO — SD H.2.3 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 as well as all kinds of social networks. Other interesting graphs include transportation routes, similarity of newspaper of scientific articles or citations among them.

With increasing computational power and memory space, we can expect more and more real-life graphs to became subject to computation. We can also expect the existing graphs, such as the Web or social networks, to grow in all aspects.

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 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 so large graphs, distribution of the computations among many machines is inevitable. The graph size is often be 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. Sizes of the graphs are growing faster than the computational power of a computer, 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, as well as higher-level languages such as PigLatin and Hive. 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], its open-source version Apache Giraph and Spark GraphX, Giraph++ [8].

The frameworks currently available allow one 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. 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 the 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 language — Datalog [3]. In Datalog, the problem is expressed in a declarative way as a set of rules. Declarative semantics makes it easy to distribute the computations,

since no execution flow is embedded in the program code. It also gives many possibilities for optimizations and approximate evaluation. At the same time, Datalog's support for recursion is crucial, since most graph algorithms have iterative nature. However, most practical graph algorithms can not be expressed efficiently in Datalog because of the language limitations. With a few extensions to original Datalog, 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 implemented 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 work 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.

This works presents an experimental implementation of the SociaLite language on the Apache Spark platform. Spark 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 [16] and inclusion into Apache Incubator in June 2013. Distinctive features of Spark are the ability keep cached data in nodes 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 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 work 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

Let us give 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. The databases can contain any value from an countable infinite set ***dom*** called the *domain*. The programs use variables from a set **var**, which is disjoint from **dom**.

Definition 0.1.1. A *database schema* is a collection of relation names R with arity $ar(R) \geqslant 0$ defined for each relation.

Given a database schema σ, let R is a relation in σ with arity n. A *fact* over R is an expression $R(x_1, \ldots, x_n)$, where each $x_i \in \boldsymbol{dom}$. 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 instance* I over database schema σ is a union of relations over R, where $R \in \sigma$.

Elements of $\boldsymbol{dom}$ are called *constants*, whereas elements of **var** are called *(free) variables*.

Definition 0.1.2. 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.

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 [11] 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 [17], network systems [18, 19]. 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 [10].

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

programs evaluation (magic sets [31], subsumptive queries [32]) and extensions to the language [28, 29, 30].

Recently there is also an increasing interest in applications of Datalog in industry. Two examples worth mentioning are LogicBlox and Datomic. LogicBlox [24] delivers a high performance database which can be queried with a Datalog variant called LogiQL. Datomic [25] 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. @TODO: jakis fajny przyklad programu, jakie jest edb, jakie jest idb, instantiation jakiejs reguly

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 function symbols, which are not permitted in Datalog
- in Prolog, there is a cut operator which is not present in Datalog (some versions of Datalog have the notion of negation, but it is still different from the cut operator)
- Datalog programs focus on querying possibly large databases with a limited number of rules, whereas in logic programming the initial data is usually embedded in the source code of the program.

@TODO: moze przyklady

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 will focus in this work 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 $\in \mathbf{K}$ 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 $T_P : inst(sch(P)) \rightarrow inst(sch(P))$:

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

Lemma 1.4.1. *Operator T_P for any Datalog program P is a monotone function.*

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.

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 $\mathbf{K}$.

@TODO: example

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 $\mathbf{K}$ 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
$\}$

1.5.2. Semi-Naive evaluation

Semi-naive evaluation is the most basic optimization used in Datalog evaluation. 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 $I_j \supseteq I_i$ for $j > i$. 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.

@TODO: example

1.5.3. Other strategies

@TODO: Bottom-up, top-down, Magic sets, subsumptive queries.

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 : 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:

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

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?

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

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

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

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 [12, 13, 14, 15].

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. @TODO: maybe some example of a problem which needs negation. 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.

@TODO: example

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

Definition 1.6.3. A rule in Datalog$^\neg$, 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 $edb(P)$. @TODO: example

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 **??**.

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*.

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 $P_1, P_2, \dots P_n$, then the output of program P on input $\mathbf{I}$ is defined by applying programs $P_1, P_2, \dots P_n$ in a sequence:

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

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

@TODO: Może ilustracja precedence graph

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 a certain column of a relation, all the facts in the relation are grouped by their values in the remaining columns and for each 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 only to aggregate *edb* relations. The semantics and evaluation is then straightforward. 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 can be stratified in such a way that each aggregation is applied to a relation defined in a preceding stratum. Aggregation of a relation from the same stratum, i. e. recursive aggregation, is much more difficult 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

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

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

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:

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.

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[21] and GraphBase [23] designed for a single computer, and this limited in the scale of problems they could solve, and parallel graph frameworks such Parallel BGL [22], 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 $S + 1$.

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.

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:

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

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

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 lower than the value stored in the vertex, it is updated, and messages with the new value are sent. An example computation is presented on the image below.

@TODO: obrazek o obliczaniu min-value jak z pracy o pregelu

For another example, let us see how single source shortest paths can be computed using Pregel.

@TODO: najkrótsze ścieżki - przykład

@TODO: krótko o własnej implementacji Google'a

2.2. Giraph

2.3. Spark

Spark with GraphX

2.4. Other extensions

Giraph++

Rozdział 3

SociaLite

@TODO: zastanów się co w jakiej kolejności jest najważniejsze (i zrozumiałe dla czytelnika)

SociaLite ([1, 2]) is a graph query language based on Datalog. 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 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, consisting of two main extensions: *recursive aggregate functions* and *tail-nested tables*. Recursive aggregate functions are the most important feature in Socialite – in 3.1 we present a complete definition and proofs of correctness of that extension, which are missing in [1]. Tail-nested tables are a much more straightforward extension – an optimization of data layout in memory. They are described in 3.2

[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 row in the data, and the data and computations are automatically sharded. Distributed SociaLite is covered in section 3.3

Additionally, thanks to the declarative semantics of Datalog and SociaLite, it is possible to provide an important optimization: the *delta stepping* technique, which is an effective way of parallelizing the Dijkstra algorithm [9]. In SociaLite, this technique can be applied automatically to a certain class of recursive aggregate programs. @TODO: zostaw sobie TODO i zastanówi się czy to dasz radę zrobić (potrzebny tu komentarz) Być może będzie oddzielny rozdział o optymalizacji i te akapity wylądują tam (że oni robili to i to, a ty to i to, bo bardziej pasowało). Wtedy tu wystarczy wspomnieć że w socialite je mają.

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. In SociaLite, by simply stopping the computation, we can obtain an approximate solution found so far. [2] also shows a method which can significantly reduce memory requirements by storing the intermediate results in an approximate way using Bloom filters. Those topics are covered in

section 3.5 @TODO: to np. nie będzie jasne dopóki nie wiemy więcej

3.1. Datalog with recursive aggregate functions

In this section we introduce the recursive aggregate functions extension from SociaLite. Since the original SociaLite consists of several extensions to Datalog, we will call the language defined here *Datalog with recursive aggregate functions*, 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 Datalog program for computing single source shortest paths (starting from node 1) is presented below. 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.

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 instance inclusion. This requires that the subsequent computation iterations only add tuples to the database instance, but never remove tuples from the instance. This is the reason for which program 3.1 is inefficient. When recursive aggregate functions are allowed, this is not the

EDGE(int *src*, int *sink*, int *len*)
PATH(int *target*, int *dist* aggregate MIN)

$\text{PATH}(1, 0).$
$\text{PATH}(t, d) \quad :- \quad \text{PATH}(s, d_1), \text{EDGE}(s, t, d_2), d = d_1 + d_2.$

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

case: a tuple in the instance can be replaced with a different one because a new aggregated value appeared. Consequently, in order to define SociaLite programs semantics in terms of fixed point, we need to use a different order on database instances.

First, we will define a meet operation 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 how it can be computed efficiently.

3.1.2. Meet operation and induced ordering

Definition 3.1.1. A binary operation is a *meet* operation if it is idempotent, commutative and associative.

@TODO: Maybe remind definitions of semi-lattice and partial order?

Order induced by a meet operation

A meet operation $\sqcap$ defines a semi-lattice: it induces a partial order $\preceq_\sqcap$ 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 $\preceq_\sqcap$

Example 3.1.1. $\max(a, b)$ for $a, b \in \mathbb{N}$ is a meet operation; it is:

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

It induces the partial order $\leqslant$: for any two $a, b \in \mathbb{N}$, $\max(a, b)$ is their least upper bound with respect to $\leqslant$.

On the contrary, $+$ is not a meet operation, since it is not idempotent: $1 + 1 \neq 1$.

3.1.3. A program in Datalog^{RA}

A Datalog^{RA} program is a Datalog program, with additional aggregation function defined for some of the relations: For each relation R, there can be one column $aggcol_R \in 1, \ldots ar_R$ chosen for which an aggregation function $aggfun_R$ is provided. The rest of the columns are called the *qualifying columns*. Intuitively, after each step of computation, we group the tuples in the relation by the qualifying columns and aggregate the column $aggcol_R$ using $aggfun_R$. Value $aggcol_R =$ **none** means that R is a regular relation with no aggregation.

For simplicity, 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 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.

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} .

Aggregation operation g_R

An important step in the evaluation of a Datalog^{RA} program is grouping the tuples 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, which takes a relation instance which may contain multiple tuples with the same set of qualifying parameters and performs the aggregation.

Definition 3.1.2. For a relation R of arity $ar_R = k$, in let us define $g : \mathbf{dom^k} \to \mathbf{dom^k}$:

$$
g_R(I) = \begin{cases} \{(x_1, \ldots, x_{k-1}, aggfun_R(\{y : (x_1, \ldots, x_{k-1}, y) \in I\}) : (x_1, \ldots, x_{k-1}, x_k) \in I\} & \text{if } aggcol_R \neq \textbf{none} \\ I & \text{otherwise} \end{cases}
$$

If R has an aggregated column, g_R groups the tuples 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 on 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 during the evaluation of a Datalog program, if the state of a relation is I_1 at some point and I_2 at some other point, we know that $I_1 \subseteq I_2$. In Datalog^{RA} this property no longer holds: a tuple in I_1 can be replaced with different tuple 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.

Definition 3.1.3. Let R be a relation. Let us define comparison $\sqsubseteq_R$ on relation instances as follows:

$$
I_1 \sqsubseteq_R I_2 \iff \forall_{(q_1,\ldots,q_{n-1},v)\in g_R(I_1)} \exists_{(q_1,\ldots,q_{n-1},v')\in g_R(I_2)} v \preceq_{aggfun_R} v' \text{ if } aggcol_R \neq \textbf{none} \quad (3.1)
$$

$$
I_1 \sqsubseteq_R I_2 \iff \forall_{(q_1,\ldots,q_n)\in g_R(I_1)} \exists_{(q_1,\ldots,q_n)\in g_R(I_2)} otherwise \quad (3.2)
$$

Note. If R does not have an aggregated column, $g_R(I) = I$ for any I, so $\sqsubseteq_R$ is simply the inclusion order $\subseteq$.

Lemma 3.1.1. *For any R, $\sqsubseteq_R$ is a partial order.*

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 $\forall_{(q_1,\ldots,q_{n-1},v)\in g(R)}\exists_{(q_1,\ldots,q_{n-1},v)\in g(R)} v \preceq_{aggfun_R} v$ because $\preceq_{aggfun_R}$ is reflexive. Hence, $R \sqsubseteq_R R$.
- $\sqsubseteq_R$ is antisimmetric @TODO: No, it is not antisimmetric, so this is not a partial order — how to deal with that?
- $\sqsubseteq_R$ is transitive: if $A \sqsubseteq_R B$ and $B \sqsubseteq_R C$, then $\forall_{(q_1,\ldots,q_{n-1},a)\in g(A)}\exists_{(q_1,\ldots,q_{n-1},b)\in g(B)} a \preceq_{aggfun_R} b$ and $\forall_{(q_1,\ldots,q_{n-1},b)\in g(B)}\exists_{(q_1,\ldots,q_{n-1},c)\in g(C)} b \preceq_{aggfun_R} c$.

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

@TODO: Because of lack of antisimmetry it is only a preorder, not a partial order —¿ how to deal with that?

@TODO: dodatkowy komentarz?

Example 3.1.2. Let R be a relation with arity 3, with the last column aggregated using meet operation max. We recall that for max, $\preceq_{\max}$ 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.

3.1.4. Semantics and evaluation - one relation case

In this section we will show that the semantics of a Datalog^{RA} program can be unambiguously defined using least fixed point, as long as it satisfies some conditions. To simplify the reasoning, we will restrict our attention to programs with only one *idb* relation. In the following section we extend the definitions and theorems presented here to the general case of many *idb* relations.

Let P be a Datalog^{RA} program, with only one *idb* relation R of arity k in the form of:

$Q_1, \ldots, Q_m$ are rule bodies with free variables $x_1, \ldots, x_k$. They may contain references to any of the *edb* relations, which are constant during the evaluation or to the only *idb* relation,

$$\begin{array}{lll} \mathrm{R}(x_1, \ldots, x_k) & :- & Q_1(x_1, \ldots, x_k) \\ & \ldots & \\ \mathrm{R}(x_1, \ldots, x_k) & :- & Q_m(x_1, \ldots, x_k) \end{array}$$

R. We denote evaluation of rule body Q in the context of an instance I of the relation Q as $E_I(Q)$.

Since we consider only one relation R, we can simplify the notation: let $\sqsubseteq$ denote $\sqsubseteq_R$ and let g denote the aggregation operation g_R for relation R, as defined in Definition 3.1.2.

Let us define $f : \mathbf{dom^k} \to \mathbf{dom^k}$ as the function that evaluates the rules $Q_1, ...Q_m$ based on the given instance of the relation R and and returns its input extended with the set of generated tuples:

$$f(I) = I \cup \bigcup_{i=1..m} E_I(Q_i)$$

Let $h = g \circ f$.

Theorem 3.1.2. *If f is monotone with respect to $\sqsubseteq$, i.e. $R_1 \subseteq R_2 \to f(R_1) \subseteq f(R_2)$, and there exists $n \geqslant 0$, such that $h^n(\emptyset) = h^{n+1}(\emptyset)$, then $R^* = h^n(\emptyset)$ is the least fixed point of h, that is:*

1. *$R^* = h(R^*)$, i.e. R^* is a fix-point*
2. *$R^* \sqsubseteq R$ for all R such that $R = h(R)$, i.e. R^* is smaller than any other fix-point*

@TODO: Extract the definition of fix-point – it does not have to be here, but where to put it?

Proof:

g is monotone with respect to $\sqsubseteq$. Since we assumed that f is monotone with respect to $\sqsubseteq$, $h = g \circ f$ is also monotone with respect to $\sqsubseteq$.

We know that $\emptyset$ is smaller under $\sqsubseteq$ than any other element.

Let us suppose that I' is any fix-point of h. We know that $\emptyset \sqsubseteq I'$. Applying h to both sides of the inequality n times, we have that $I^* = h^n(\emptyset) \sqsubseteq h^n(I') = I'$, thanks to the monotonicity of h with respect to $\sqsubseteq$. Therefore, the inductive fixed point I^* is the least fixed point of h.

In pseudocode, the evaluation algorithm is straightforward:

$I_0 \leftarrow \emptyset$

$i \leftarrow 0$

do

  $i \leftarrow i + 1$

  $I_i \leftarrow h(I_{i-1})$

while $I_i \neq I_{i-1}$

Rysunek 3.4: Naive evaluation algorithm for Datalog^{RA} programs with one *idb* relation.

@TODO: open questions:

- How we compute recursive functions with non-meet aggregation operators? – I can forbid that for now...

3.1.5. Semantics and evaluation - multiple relations case

In this section we extend Datalog^{RA} semantics from 3.1.4 to a general case of possibly many *idb* relations.

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.

$$
\begin{array}{lll}
\mathrm{R}_1(x_1, \ldots, x_{k_1}) & :- & Q_{1,1}(x_1, \ldots, x_{k_1}) \\
 & \ldots & \\
\mathrm{R}_1(x_1, \ldots, x_{k_1}) & :- & Q_{1,m_1}(x_1, \ldots, x_{k_1}) \\
 & \ldots & \\
\mathrm{R}_w(x_1, \ldots, x_{k_w}) & :- & Q_{w,1}(x_1, \ldots, x_{k_w}) \\
 & \ldots & \\
\mathrm{R}_w(x_1, \ldots, x_{k_w}) & :- & Q_{w,m_w}(x_1, \ldots, x_{k_w})
\end{array}
$$

For $i = 1, \ldots, w$, $Q_{i,1}, \ldots, Q_{i,m}$ are rule bodies with free variables $x_1, \ldots, x_{k_w}$. They may contain references to the *idb* relations $R_1, R_2, \ldots R_w$ and the *edb* relations, which are constant during the evaluation. We denote evaluation of rule body Q in the context of an instance $I_1, \ldots I_w$ of the relation $R_1, \ldots, R_w$ as $E_(I_1, \ldots, I_w)(Q)$.

For $i = 1, \ldots, w$, $Q_{i,1}, \ldots, Q_{i,m}$, let us define $f_i : P(\mathbf{dom^{k_1}}) \times \cdots \times \mathbf{P}(\mathbf{dom^{k_w}}) \to \mathbf{P}(\mathbf{dom^{k_i}})$ as the function that evaluates the rules $Q_{i,1}, ...Q_{i,m_w}$ for relation R_i based on the given instances of the relations $R_1, \ldots, R_w$:

$$f_i(I_1, \ldots I_w) = I_i \cup \bigcup_{i=1..m_w} E_(I_1, \ldots, I_w)(Q_i)$$

Let $h_i = g_{R_i} \circ f_i$. Let $h(I_1, \ldots, I_w) = (h_1(I_1, \ldots, I_w), \ldots, h_w(I_1, \ldots, I_w))$

Let $\sqsubseteq = \sqsubseteq_{R_1} \times \cdots \sqsubseteq_{R_w}$.

Theorem 3.1.3. *If h is monotone with respect to $\sqsubseteq$, i.e. $I \subseteq I' \to h(I) \subseteq f(I')$, and there exists $n \geqslant 0$, such that $h^n(\emptyset) = h^{n+1}(\emptyset)$, then $I^* = h^n(\emptyset)$ is the least fixed point of h, that is:*

1. *$I^* = h(I^*)$, i.e. I^* is a fix-point*
2. *$I^* \sqsubseteq I$ for all I such that $I = h(I)$, i.e. I^* is smaller than any other fix-point*

@TODO: Extract the definition of fix-point – it does not have to be here, but where to put it?

Proof:

We know that $\emptyset$ is smaller under $\sqsubseteq$ than any other element.

Let us suppose that I' is any fix-point of h. We know that $\emptyset \sqsubseteq I'$. Applying h to both sides of the inequality n times, we have that $I^* = h^n(\emptyset) \sqsubseteq h^n(I') = I'$, thanks to the monotonicity of h with respect to $\sqsubseteq$. Therefore, the inductive fixed point I^* is the least fixed point of h.

In pseudocode, the evaluation algorithm is straightforward:

@TODO: open questions:

$I_0 \leftarrow (\emptyset, \ldots, \emptyset)$

$i \leftarrow 0$

do

$\quad i \leftarrow i + 1$

$\quad I_i \leftarrow h(I_{i-1})$

while $I_i \neq I_{i-1}$

Rysunek 3.5: Naive evaluation algorithm for Datalog^{RA} programs with multiple *idb* relations.

- How we compute recursive functions with non-meet aggregation operators? – I can forbid that for now...

3.1.6. Semi-naive evaluation – one relation case

Semi-naive evaluation is the most basic optimization used in Datalog evaluation. In comes from the following observation: in a Datalog program, if some rule Q produced a tuple t based on database instance I_i in the i-th iteration of the naive evaluation algorithm, then this rule with produce this tuple in each subsequent iteration, because $I_j \supseteq I_i$ for $j > i$. The goal of this optimization is to avoid such redundant computation. It is achieved by joining only subgoals in the body of each rule which have at least one new answer produced in the previous iteration.

This optimization can be applied in Datalog^{RA} as well. To achieve this, we need to define function f and the evaluation operation $E_I(Q)$ in a different way.

3.1.7. 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

3.4. Delta stepping in Distributed SociaLite

3.5. Approximate evaluation in Distributed SociaLite

Rozdział 4

Translating SociaLite programs to Pregel

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

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

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

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

[13] 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.

[14] 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.

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

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

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

[18] 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.

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

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

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

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

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

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

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

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

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

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

[29] 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)

[30] 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

[31] 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.

[32] 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.