Przetwarzanie dużych grafów za pomocą Apache Giraph

Marek Rogala

Wrzesień 2014

Streszczenie

Słowa kluczowe

graph queries, Datalog

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D. Software
D.127. Blabalgorithms
D.127.6. Numerical blabalysis

Tytuł pracy w języku angielskim

Processing large graphs with Apache Giraph

Spis treści

Introduction

In recent years, the humanity has created many graph datasets much larger than those avaiable ever before. Those graphs became a very popular object of research. Most notable examples include *the Web graph* – a graph of internet websites and links between them, all kinds of social networks. There are also graphs such as 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 graps metioned 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 (ex. importance of a Web page, determining most influential users in given group of people), clustering (detecting communities), computing metrics for the whole graph or some parts of it (ex. connectivity measures) and connection predictions. Usually, such analyses are built on top of standard graph algorithms, such as PageRank-like procedures, shortest paths or connected components.

In the past, we have seen many tools for efficient distributed large dataset computations, starting from Google's MapReduce [5] 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 [6], its open-source version Apache Giraph and Spark GraphX, Giraph++ [7].

The frameworks currently avaiable allow you to implement a graph algorithm in a specified model (for example Pregel's "think like a vertex"). On the other hand, query languages, such as SQL, are a bad fit for graph data since because of limited support of iteration. 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. Declatarive 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, support for recursion is crucial, since most graph algorithms have iterative nature. However, most of practical graph algorithms can not be expressed efficiently in Datalog because of the language limitiations. 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 avaiable. The interpreter published by the authors is undocumented and contains many bugs. It is hard to imagine the langauge being implemented in the industry in the forseeable future. The 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 Giraph "think-like-a-vertex" programs and introduce a compiler that enables SociaLite programs to be executed on existing Giraph infrastructure. This allows users of Hadoop to write and execute SociaLite programs without any additional effort to build a dedicated server infrastructure for that.

The work consists of six chapters. In 1 we recall definitions of Datalog and its evaluation methods while 2 contains an introduction to the Pregel computation model. In 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 Giraph programs implemented in the S2G compiler, which is described in 5. In 6 we sketch the possible future work and the path to industrial implementation of the language using the S2G compiler.

Rozdział 1

Datalog

In this chapter we define the Datalog language.

1.1. Definitions

@TODO: te definicje są roboczo przepisane z [4], którzy z kolei podają je za [3].

1.1.1. Queries and instances

Let us assume that ***dom*** is an infinite set of data values.

- A *database schema* σ is a collection of relation names R where every R has arity $ar(R) \geqslant 0$.
- We call $R(\overline{d})$ a *fact* when R is a relation name and $\overline{d}$ is a tuple in ***dom***.
- We say that a fact $R(d_1, \ldots, d_k)$ is *over* a database schema σ if $R \in \sigma$ and $ar(R) = k$.
- A *(database) instance* I over σ is a finite set of facts over σ.
- We denote by $adom(I)$ the set of all values that occur in facts of I. When $I = \mathbf{f}$, we simply write $adom(\mathbf{f})$ rather than $adom(\{\mathbf{f}\})$.
- By $|I|$ we denote the number of facts in I.

1.1.2. Datalog with negation

Let **var** be the universe of variables, disjoint from **dom**. An atom is of the form $R(u_1, \ldots, u_k)$ where R is a relation name and each $u_i \in$ **var**. We call R the predicate. A *literal* is an atom (*positive atom*) or a negated atom (*negative atom*)

We recall Datalog with negation [3], abbreviated Datalog$^\neg$. Formally, a Datalog$^\neg$ rule ϕ is a quadruple $(head_\phi, pos_\phi, neg_\phi, ineq_\phi)$ where $head_\phi$ is an atom; pos_ϕ and neg_ϕ are sets of atoms; $ineq_\phi$ is a set of inequalities $(u = v)$ with $u, v \in var$ and the variables of ϕ all occur in pos_ϕ. The components $head_\phi$, pos_ϕ and neg_ϕ are called respectively the *head*, the *positive body atoms* and the *negative body atoms*. We refer to $pos_\phi \cup neg_\phi$ as the *body atoms*. Note, neg_ϕ contains just atoms, not negative literals. Every Datalog$^\neg$ rule ϕ must have a head, pos_ϕ must be non-empty and neg_ϕ may be empty. If $neg_\phi = \emptyset$ then ϕ is called positive.

Of course, a rule ϕ may be written in the conventional syntax. For instance, if $head_\phi = T(u,v)$, $pos_\phi = \{R(u,v)\}$, $neg_\phi = \{S(v)\}$, and $ineq_\phi = u \neq v$, with $u, v \in var$, then we can write ϕ as:

$$T(u,v) : -R(u,v), \neg S(v), u \neq v$$

.

The set of variables of ϕ is denoted $vars(\phi)$. A rule ϕ is said to be over schema σ if for each atom $R(u_1, \ldots, u_k) \in head\phi \cup pos\phi \cup neg\phi$, the arity of R in ϕ is k. A Datalog program P over σ is a set of Datalog eg rules over σ. We write $sch(P)$ to denote the (minimal) database schema that P is over. We define $idb(P) \subset sch(P)$ to be the database schema consisting of all relations in rule-heads of P. We abbreviate $edb(P) = sch(P)\ idb(P)$. As usual, the abbreviation "idb" stands for "intensional database schema" and "edb" stands for "extensional database schema". A valuation for a rule ϕ in P w.r.t. an instance I over $edb(P)$, is a total function $V : vars(\phi) \to dom$. The application of V to an atom $R(u_1, \ldots, u_k)$ of ϕ, denoted $V(R(u_1, \ldots, u_k))$, results in the fact $R(a_1, \ldots, a_k)$ where $a_i = V(u_i)$ for each $i \in 1, ..., k$. This is naturally extended to a set of atoms, which results in a set of facts. The valuation V is said to be satisfying for ϕ on I if $V(pos_\phi) \subset I, V(neg\phi) \cap I = \emptyset$, and $V(u) \neq V(v)$ for each $(u \neq v) \in ineq_\phi$. If so, ϕ is said to derive the fact $V(head_\phi)$.

1.1.3. Positive and semi-positive Datalog

A Datalog $^\neg$ program P is positive if all rules of P are positive. We say that P is semi-positive if for each rule $\phi \in P$, the atoms of neg_ϕ are over $edb(P)$. We now give the semantics of a semi-positive Datalog $^\neg$ program P. First, let T_P be the immediate consequence operator that maps each instance J over $sch(P)$ to the instance $J = J \cup A$ where A is the set of facts derived by all possible satisfying valuations for the rules of P on J. Let I be an instance over $edb(P)$. Consider the infinite sequence I_0, I_1, I_2, etc, inductively defined as follows: $I_0 = I$ and $I_i = T_P(I_{i-1})$ for each $i \geqslant 1$. The output of P on input I, denoted $P(I)$, is defined as $\bigcup_j I_j$; this is the *minimal fixpoint* of the T_P operator.

1.1.4. Stratified semantics

We say that P is syntactically stratifiable if there is a function $\rho : sch(P) \to 1, ..., |idb(P)|$ such that for each rule $\phi \in P$, having some head predicate T , the following conditions are satisfied:

1. $\rho(R) \leqslant \rho(T)$ for each $R(\overline{u}) \in pos_\phi \cap idb(P)$
2. $\rho(R) < \rho(T)$ for each $R(\overline{u}) \in neg_\phi \cap idb(P)$.

For $R \in idb(P)$, we call $\rho(R)$ the stratum number of R. Intuitively, ρ partitions P into a sequence of semi-positive Datalog $^\neg$ programs $P_1, ..., P_k$ with $k \leqslant |idb(P)|$ such that for each $i = 1, \ldots, k$, the program P_i contains the rules of P whose head predicate has stratum number i. This sequence is called a syntactic stratification of P . We can now apply the stratified semantics to P : for an input I over $sch(P)$, we first compute the fixpoint $P_1(I)$, then the fixpoint $P_2(P_1(I))$, etc. The output of P on input I, denoted $P(I)$, is defined as $P_k(P_{k-1}(\ldots P_1(I)\ldots))$. It is well known that the output of P does not depend on the chosen syntactic stratification (if more than one exists). Not all Datalog $^\neg$ programs are syntactically stratifiable. By $stratifiedDatalog$ we refer to all Datalog$^\neg$ programs which are syntactically stratifiable.

1.2. Evaluation strategies for Datalog

bottom-up, top-down

Rozdział 2

Pregel/Giraph

@TODO: Description of Pregel model and info about the Giraph project, also mention other extensions like Giraph++.

Rozdział 3

SociaLite

SociaLite ([1], [2]) is a graph query language based on Datalog. While Datalog allows to express some of graph algorithms in an elegant and succint 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 be executed in a distributed environment with no need for the programmer to worry about distributing the computation or managing the communication protocols.

The language consists of a few extensions to Datalog, most significant of which 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 new 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*, shows 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 sematics 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 [8]. In SociaLite, this technique can be applied automatically to a certain class of recursive aggregate programs.

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 computaion, 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

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 Page Rank. 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 shortests 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.

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

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 staightforward 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.

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

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

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

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 least fixed point of instance inclusion. This requires that the subsequent computation interations 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 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 sematics in terms of least 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 $aggcol_R \in 1, \dots ar_R$ 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 $aggcol_R = \textbf{none}$ 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.

$$
\begin{array}{lll}
\mathrm{P}(\text{int } a, \text{int } b \text{ aggregate F}) & & \\
\mathrm{R}(\text{int } src, \text{int } sink, \text{int } len) & & \\
\mathrm{P}(x_1, \ldots, x_{ar_P}) & :- & Q_{P,1}(x_1, \ldots, x_{ar_P}) \\
 & \ldots & \\
\mathrm{P}(x_1, \ldots, x_{ar_P}) & :- & Q_{P,m}(x_1, \ldots, x_{ar_P}) \\
\mathrm{R}(x_1, \ldots, x_{ar_R}) & :- & Q_{R,1}(x_1, \ldots, x_{ar_R}) \\
 & \ldots & \\
\mathrm{R}(x_1, \ldots, x_{ar_R}) & :- & Q_{R,m}(x_1, \ldots, x_{ar_R})
\end{array}
$$

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

Aggregation function g_R

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

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 transistive: 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 transistive, 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: 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 SociaLite program, with only one *idb* relation R of arity k in the form of:

$$\begin{aligned} \mathrm{R}(x_1,\ldots,x_{k-1},\mathrm{F}(y)) \quad &:- \quad Q_1(x_1,\ldots,x_{k-1},y) \\ &\ldots \\ &:- \quad Q_m(x_1,\ldots,x_{k-1},y). \end{aligned}$$

To simplify the notation, by $\sqsubseteq$ let us denote $\sqsubseteq_F$.
Let us define:

- $f : \mathbf{dom^k} \to \mathbf{dom^k}$ – the function that evaluates the rules $Q_1,\ldots Q_m$ based on the contents of the relation given in the argument and returns its input extended with the set of generated tuples
- $g : \mathbf{dom^k} \to \mathbf{dom^k}$ – the function that performs grouping and aggregation.
- $h = g \circ f$

Theorem 3.1.2. *If f is monotone with respect to $\sqsubseteq$, i.e. $R_1 \subseteq R_2 \rightarrow f(R_1) \subseteq f(R_2)$, 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. $R^* = h(R^*)$
2. $R^* \sqsubseteq R$ *for all* R *such that* $R = h(R)$

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 $\sqsubseteq$.

Let us notice that $\emptyset$ is greater under $\sqsubseteq$ than any other element. TODO: show that (very easy)

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

@TODO: open questions:

- How we compute recursive functions with non-meet aggregation operators?

3.2. Tail-nested tables

3.3. Distributed SociaLite

3.4. Delta stepping in Distributed SociaLite

3.5. Approximate evaluation in Distributed SociaLite

Rozdział 4

Translating SociaLite programs to Giraph programs

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

[6] 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

[7] 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

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

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