INSTITUT FÜR INFORMATIK
der Ludwig-Maximilians-Universität München

Parallel Datalog on Pregel

Bachelor Thesis

Matthias Maiterth

Advisor:	Harald Zauner
Supervising professor:	Prof. Dr. François Bry
Filing date:	October 22, 2012

Erklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe.

München, den 22. Oktober 2012 Matthias Maiterth

Abstract

With the rise of Web 2.0, the efficient processing and analysis of Big Data (e.g. graphs of social networks) has gained strong interest, and still is an active research issue.

For such analysis tasks, Datalog with its concise syntax and clean semantics is a suitable language to express facts and rules concerning Big Data.

Google's MapReduce is the de facto standard for processing Big Data, but for performing recursive computations (as is the case when evaluating Datalog programs), Pregel is better suited.

Pregel was developed for graph processing at Google, and provides a data parallel, vertex-centric approach for computations on a compute cluster.

The focus of this thesis lies on elaborating and implementing a data parallel algorithm for the semi-naive evaluation of Datalog programs using Pregel.

Acknowledgements

First and foremost, I would like to thank Harald Zauner for offering me a highly interesting topic for this thesis. His support throughout the whole work has been very important and encouraging.

I would like to thank Prof. Dr. François Bry for his helpful advises and insights. The appointments with him throughout the work were an important guideline and built a common thread throughout my work.

I would also like to express my thanks to the other members of the Teaching and Research Unit Programming and Modelling Languages at the Department for Informatics at the LMU.

My greatest thanks to my parents, my friends and the people I love for their support.

Contents

Chapter 1

Motivation: Evaluation of Huge Datalog Programs

With ever growing amounts of data, e.g. graphs of social networks or the web itself, scalable processing of this data has become of utmost importance. Distributed systems have proven capable of processing data of such size.

Datalog [1] provides a concise syntax and clean semantics to define relations in networks of arbitrary size and also allows the definition of logical rules to infer relations that are not yet explicitly represented in the raw data. The versatility and expressive power of Datalog, while at the same time staying simple and understandable, has allowed Datalog not only to stay a hot topic in academia, but also to gain a foothold in industry. Logicblox, for example, uses Datalog to provide cloud based solutions for Big Data enterprise applications in the domains of decision automation, analytics, and planning [2]. SAP is another notable company that uses Datalog to discover, represent, and reconstruct business networks from raw data (mined from a network) [3]. In contrast to primarily interactive systems like the ones mentioned above, we focus on computing the minimal model (or the deductive closure) of a Datalog program, which can be used for further processing later on.

Recent work [4] on the semi-naive evaluation of Datalog based on MapReduce [5] has shown that a query cannot be evaluated by a single MapReduce job or a small number of MapReduce jobs combined. In general, evaluation of a single query requires a large number of MapReduce jobs combined in sequence. This chaining of large numbers of MapReduce jobs results in an overhead that would only be negligible when chaining just a few MapReduce jobs. Consider two MapReduce jobs A and B chained in sequence on the same compute cluster. When A is finished, it writes its output data to a distributed file system. For the cluster to execute B on A's output, it first has to distribute the `map`- and `reduce`-functions of job B and then read and partition B's input from the file system. This interaction with the distributed file system, communication of the user program, and partitioning of data would not pose a problem for a single MapReduce job. But it becomes expensive, when it has to be done repeatedly for a large number of MapReduce jobs.

The overhead of the repeated tasks of interacting with the distributed file system, partitioning the input data, and communicating the user program does not only apply when performing a semi-naive evaluation of Datalog, but more generally to all kinds of complex computations wherever the dependencies within the input data are more complex than what can be expressed by a single instance of the user-defined functions `map` and `reduce`.

Since no communication amongst the mappers or reducers is supported by MapReduce, its dataflow model is in general ill-suited for problems with complex dependencies in their input data.

Coming back to evaluating Datalog with MapReduce, especially Datalog programs with recursive rules would in general require a considerably long sequence of MapReduce jobs until no more new knowledge can be derived. Even specialized MapReduce implementations like HaLoop [6], that reduce part of the overhead caused by chaining MapReduce jobs together, cannot eliminate the overhead caused by the mismatch between MapReduce's programming model and input data with complex dependencies (as is the case with most Datalog programs).

In this work, we elaborate an algorithm for evaluating Datalog on a distributed system using Pregel [7], a framework for large-scale graph processing developed at Google. The objective of the algorithm proposed is an efficient computation of the facts implied by a Datalog program relying on data parallelism using a dedicated compute cluster.

In contrast to the dataflow model of MapReduce, Pregel is closely oriented on the bulk-synchronous parallel (BSP) model [8]. A Pregel computation consists of several distributed processes that process data in parallel according to a user program. These processes communicate with each other via messages, and each of them is responsible for a specific partition of the input graph. The expressiveness and flexibility of Pregel's computational model allows us to implement a distributed, semi-naive evaluation algorithm for Datalog as a single Pregel job.

For the above mentioned reasons, the Pregel-based approach does not introduce the overhead, which cannot be avoided by any MapReduce-based approach.

In Chapter 2, we give a short introduction to Datalog. In Chapter 3, a detailed introduction to Pregel is given including the Apache implementation: Giraph [9]. The concept and implementation of the Datalog evaluation using Pregel is presented in Chapter 4. In Chapter 5, we compare our implementation with DEDALUS [10], another distributed implementation of Datalog. In Chapter 6, we conclude with an outlook on further developments and possible applications.

An experimental evaluation of the algorithm on large Datalog programs utilizing a compute cluster has not been conducted, as it was not part of this thesis. This is to be done in future work.

Chapter 2

Datalog

In Section 2.1, we give a short introduction to Datalog. In Section 2.2, the semi-naive evaluation is presented. This evaluation algorithm is adapted for distributed evaluation in Chapter 4.

2.1 Introduction to Datalog

Datalog is a language to represent knowledge in a fragment of first-order logic. To be able to combine Datalog and the distributed graph processing framework Pregel, we first introduce Datalog itself as it is described in [1, pp. 96–128] and [11, pp. 11–12]. To understand the structure of Datalog programs, in the following, we examine the program of Figure 2.1.

We begin by introducing atoms: An *atom* consists of a predicate symbol and a list of arguments. Each argument of an atom is either a variable or a constant; function symbols of arities greater than zero are not allowed.

A Datalog program is a finite set of rules. The left part of a rule, before the ":-", is a single atom called the *head* of the rule. The right part, called the *body* of the rule, can consist of an arbitrary number (≥ 0) of atoms (separated by `&`), also referred to as *subgoals*. These atoms can also be simple arithmetic comparisons formed with predi-

```
sibling(X,Y):- parent(X,Z) & parent(Y,Z) & X≠Y.
parent('Isabella','Ella').
parent('Ella','Ben').
parent('Daniel','Ben').
```

Figure 2.1: Example Datalog program for computing a `sibling` relation, given `parent` EDB facts.

cates like $=$, $\neq$, and $\geq$, which are referred to as *built-in* predicates. For a rule to be *safe*, each of its variables has to be limited. The limited variables of a rule are defined as follows:

A variable appearing in a rule r is *limited* iff (short for: if and only if) (i) it appears as argument of an ordinary (i.e. not built-in) subgoal of r, or (ii) it is equated to a constant (in a subgoal of r), or (iii) it is equated to a limited variable of r (in a subgoal of r) [1, p.105]. In the following, only safe Datalog programs (i.e. Datalog programs consisting of safe rules only) are considered.

By *facts*, we denote rules with empty body (written as only the head of the rule, leaving out the ":-", see lines 2-4 of Figure 2.1).

A relation can either be an extensional database (EDB) relation, or an intensional database (IDB) relation. An EDB relation is given by a set of facts with the same predicate symbol (which can be seen as the name of the relation), while IDB relations are defined via rules.

Predicate symbols are used for labeling relations. The attributes that name columns in relational algebra are missing, though. The identification of attributes is possible by following the order of the attributes.

For the relation `parent` an EDB relation could look like lines 2-4 of Figure 2.1. This relation can be interpreted as follows: " 'Isabella' has parent 'Ella', 'Ella' has parent 'Ben', and 'Daniel' has parent 'Ben' ".

We use the proof-theoretic interpretation of rules and disallow negation, as in pure Datalog. This means that we derive facts from rules given by a Datalog program or a database, according to [1, p. 97].

If we want to infer facts using a rule, we can use all facts derived from EDB and IDB relations. When replacing the variables with constants and each subgoal of the body is satisfied, the head is true. In other words, a new fact is derived. Let us evaluate rule 1 of the Datalog program in Figure 2.1:

```
sibling(X,Y):- parent(X,Z) & parent(Y,Z) & X≠Y.
```

We can insert the facts from above and infer the following:

```
sibling('Ella','Daniel'):- parent('Ella','Ben') &
    parent('Daniel','Ben') & 'Ella'≠'Daniel'
```

The two predicates for `parent` and the built-in predicate are satisfied. This makes the body true and thus the head is true accordingly.

2.2 Semi-naive Evaluation

In this section, we introduce the semi-naive evaluation as defined in Figure 2.2. To begin with, we define the notion of a substitution:

`//Input:` F `(set of Datalog EDB facts),`
`//` R `(set of Datalog rules with non-empty body)`
$\Delta_{old} := F$
while $\Delta_{old} \neq \emptyset$
  $\Delta_{new} :=$ `EVAL-INCR`(R, F, Δ_{old})
  $F := F \cup \Delta_{new}$
  $\Delta_{old} := \Delta_{new}$
`output` F

Figure 2.2: Semi-naive evaluation of a Datalog program

> A *substitution* σ is a function, written in postfix notation, that maps terms to terms and is
>
> - homomorphous, i.e. $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$ for compound terms and $c\sigma = c$ for constants
> - identical almost everywhere, i.e., $\{x \mid x \text{ is a variable and } x\sigma \neq x\}$ is finite.
>
> [11, Def. 48]

A substitution σ can thus be represented as a finite set $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ where $\{x_1, \ldots, x_n\}$ is the finite set of variables on which σ is not identical and $x_i\sigma = t_i$ for $1 \leq i \leq n$.

For the semi-naive evaluation, we define `EVAL-INCR`(R, F, Δ_{old}), where R is a set of Datalog rules (each with non-empty body), and F and Δ_{old} are sets of facts, with $\Delta_{old} \subset F$.

$$\begin{aligned}\texttt{EVAL-INCR}(R, F, \Delta_{old}) :=&\\ &\{h\sigma \mid \exists(h \text{ :- } b_1 \ \& \ \ldots \ \& \ b_n) \in R(\\ &\exists\sigma, \text{where } \sigma \text{ is a substitution}, (\exists k(1 \leq k \leq n \wedge\\ &b_k\sigma \in \Delta_{old} \wedge \forall l(1 \leq l \leq n \wedge l \neq k \Rightarrow b_l\sigma \in F))))\}\\ &\setminus F\end{aligned}$$

`EVAL-INCR` generates a new set of facts, obtained from the heads of rules r of the form 'h :- b_1 & ... & b_n' from R, where for each rule r all subgoals are satisfied the following way: At least one subgoal is satisfied using a fact from Δ_{old} and the remaining subgoals are satisfied using facts from F. Those facts, that already appear in F are removed from this new set, and the remaining facts are returned as the result of `EVAL-INCR`.

`EVAL-INCR` can be used in the semi-naive evaluation of Datalog, as shown in Figure 2.2.

The idea behind the semi-naive algorithm is evaluating all rules of a Datalog program, using only the most recent addition of facts Δ_{old} for one subgoal and all facts F for the remaining subgoals to generate a new set of facts Δ_{new}. These new facts are again used for the next evaluation round, until now more new facts can be inferred.

The input for the semi-naive algorithm is a set of Datalog rules R with non-empty body and a set of EDB facts F.

- Initially, we initialize Δ_{old} with the set of facts F.
- After that, the tasks from line 5-7 are repeated until Δ_{old} is empty:
  - Δ_{old} represents all facts that have been derived in the previous round. (In the first round, these are the input facts F.)
  - `EVAL-INCR` is evaluated using R, F, and Δ_{old} as arguments, and saving the result in Δ_{new}.
  - F is assigned the union $F \cup \Delta_{new}$. Hence, F contains all facts that have been derived so far.
  - Δ_{new} is saved to Δ_{old} since it represents the most recent addition of facts, which is used for evaluating the next round.
- Finally, the set of facts F is returned.

In Chapter 4, we map this evaluation algorithm to Pregel.

Chapter 3

Pregel

This chapter presents Pregel [7], a framework for the efficient processing of large (directed) graphs on a compute cluster.

A Pregel computation is a sequence of supersteps, in each of which a vertex can process messages from the previous superstep, send messages to other vertices, and modify its own state, thereby changing the state of the graph as a whole.

Pregel has been developed at Google, and is inspired by the bulk-synchronous parallel (BSP) model [8]. According to [8], the BSP model is defined as the combination of the following three attributes:

- A number of *components* for performing computations and memory functions.
- A *router* for delivering messages between components.
- *Supersteps* for synchronizing the computations performed by components. During a superstep, each component performs a task consisting of local computation and message arrivals and transmissions. Once the superstep has been completed by all components, the components proceed to the next superstep (barrier synchronization [8]).

A BSP computation consists of a sequence of such supersteps.

Similar to MapReduce [5], Google's implementation of Pregel is not publicly available, but there exist several open source implementations, e.g. GoldenOrb [12] and Apache Giraph [9].

This chapter is organized as follows: Section 3.1 gives a conceptual overview of Pregel, while Section 3.2 presents more details of the framework. Section 3.3 presents the open source implementation Apache Giraph. In Section 3.4, a general example application is given to visualize Giraph's method of operation.

3.1 General Concept of Pregel

As this section presents a conceptual overview of Pregel, it does not address how Pregel computations are actually executed on a distributed system. Hence, the terms "graph" and "vertex" used below are not to be confused with compute nodes in a compute cluster.

The input to a Pregel computation consists of a directed graph and a user program (see Section 3.2) containing a user-defined function called `compute()`.

A Pregel computation consists of a sequence of so-called supersteps. In each superstep, the framework calls the same (user-provided) `compute()`-function in parallel on each (active) vertex of the directed graph. Note that, a Pregel computation has only one `compute()`-function. `compute()` can essentially be used to conduct local computations which generally result in a change of the local state of the vertex or communication with other vertices by sending messages. Once every vertex is finished with the `compute()`-function, the next superstep starts. All messages are synchronized by these supersteps, i.e. a message issued at superstep S will be available at its destination vertex at superstep $S + 1$.

A Pregel computation terminates once all vertices of the directed graph are inactive and no further messages are to be passed. Initially, each vertex is active. If a vertex V calls `voteToHalt()`, V will be inactive in the next superstep (i.e. the framework won't call its `compute()`-function) unless some vertex issued a message to V in the current superstep, since messages always activate the receiving vertex in the next superstep.

The output of a Pregel computation is the resulting directed graph after the computation has terminated.

3.2 Presentation of Pregel

This section gives a more detailed presentation of Pregel than was given in the conceptual overview of the last section.

The input to a Pregel computation consists of a directed graph G (consisting of vertices and directed edges) and a user program. The user program contains a vertex class, which includes the `compute()`-

function, and might contain additional classes, e.g. for realizing aggregators and combiners (see details below).

Each vertex of G consists of

- a unique vertex ID,
- a flag indicating whether the vertex is active or not,
- a vertex value,
- a set of incoming messages, and
- a set of outgoing edges.

Each directed edge of G is associated with its source vertex and consists of an edge value and a target vertex ID.

A Pregel computation consists of a sequence of supersteps.

In superstep 0, after the input graph has been read, each vertex is active (i.e. its active flag is set to true) and its set of incoming messages is empty.

In each superstep S, the Pregel framework invokes the user-provided `compute()`-function on each active vertex V, conceptually in parallel. From within `compute()`, it is possible

- to read each component the vertex V consists of,
- to write V's vertex value,
- to read the number of the current superstep,
- to call `voteToHalt()`, which will set V to inactive in the next superstep,
- to issue messages to vertices that are addressed by their vertex ID,
- to modify the set of outgoing edges and their edge values, and
- to pass values to aggregators (see details below) and to read the aggregated values.

Each superstep in Pregel is concluded by barrier synchronization, which affects both computation and communication as follows:

- Concerning computation, barrier synchronization means that each vertex waits until all other vertices have finished the `compute()`-function.

- Concerning communication, barrier synchronization affects messages, the `voteToHalt()`-function, and aggregators:

  - A message issued at superstep S will not be available at its destination vertex until the next superstep $S + 1$.
  - Calling `voteToHalt()` will not render a vertex V inactive in the current, but in the next superstep (unless a message is available for V in the next superstep).

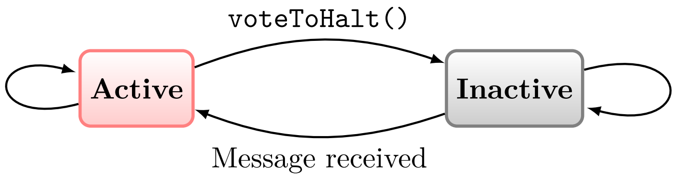


Figure 3.1: Vertex State Machine [7]

- When using aggregators, the result of aggregating values, which have been passed to an aggregator in the current superstep, is not visible until the next superstep.

A Pregel computation terminates once all vertices of the directed graph are inactive and no further messages are to be passed.

Figure 3.1 illustrates how the state of a vertex can change from active to inactive, and vice versa: A change from active to inactive only occurs, when the vertex itself calls `voteToHalt()`. However, if a vertex receives a message it is guaranteed that the vertex is active in the next superstep. Thus, only when no more messages are to be passed, the existence of no further active vertices can be guaranteed and the computation will terminate. Otherwise, a new superstep follows.

The output of a Pregel computation is the resulting directed graph after the computation has terminated. In order to generate this output, each vertex contributes its vertex ID, its vertex value, and its set of outgoing edges, as specified in the user program.

Aggregators

Aggregators have already been mentioned above and can be used to introduce a global value that each vertex can read and modify by passing values to the aggregator. For example, an aggregator using the boolean `and` operation to aggregate values can be used to detect if some condition is satisfied for all vertices that pass values to that aggregator.

Master/Worker Model

In the following, we describe how the vertices of the directed graph G are mapped to a distributed system.

Similar to MapReduce [5], the Pregel framework is realized as master/worker model. When a Pregel computation is executed on a compute cluster, one of its compute nodes is designated as the master, and all other compute nodes act as workers, which are coordinated by the master. At the beginning of a Pregel computation, the directed graph G is read from a distributed file system.

All vertices of the directed graph G are partitioned via a hash function, which maps each vertex ID to an integer $\in [0, \ldots, N-1]$, where N is the number of partitions (which is determined by the master). The master then assigns one or more partitions to each worker.

After the master instructed each worker to perform a superstep, each worker uses one thread per partition and invokes the `compute()`-function on each active vertex of that partition. In addition, each worker delivers the messages, that have been issued in the previous superstep, to their destination vertices before `compute()` is executed.

Each worker notifies the master once it is finished with the superstep, also telling it the number of active vertices in the next superstep. The master then either instructs the workers to perform the next superstep, or to save their partitions to an output file located on a distributed file system when the computation terminates (this is the case iff each worker reported that none of its vertices will be active in the next superstep).

Combiners, Fault Tolerance, Topology Mutation

The presentation of Pregel is not complete in that it does only mention, but not elaborate on the following features (which were not necessary for the semi-naive, distributed implementation of Datalog presented in Chapter 4):

- Combiners can be used to reduce the message overhead in some cases, if only some aggregated value is of interest, but not the individual value of each single message.
- Pregel offers mechanisms for fault tolerance, e.g. when one or more workers fail.
- The topology of the input graph can be modified (i.e. adding or removing vertices) during the course of a Pregel computation.

Please refer to [7] for further information concerning these features.

Short comparison to MapReduce

To round out the presentation of Pregel, we concisely compare Pregel and MapReduce [5], with regard to their suitability for processing recursive graph algorithms on large graphs.

In the case of MapReduce, in general a sequence of MapReduce jobs will be needed to process a recursive graph algorithm, see [13]. As a consequence, the entire graph needs to be passed from one MapReduce job to the next. Considerable overhead is caused, since the entire graph is passed from one MapReduce job to the next by means of a distributed file system.

Pregel, in contrast, avoids this overhead by reading the input graph only once, partitioning its vertices and then assigning one or more partitions to each worker. As each vertex has a state and can communicate with other vertices by messages, a recursive graph algorithm can in general be processed by only one Pregel job.

3.3 Open Source Implementation: Apache Giraph

In this section, we introduce Giraph [9], an open source implementation of Pregel, which is developed by the Apache Software Foundation.

Technically, Giraph is based on Apache Hadoop [14]. However, only the mapping phase of a single Hadoop job is used to realize a Giraph job. Zookeeper [15] is used for fault tolerance, and the input and output file of a Giraph job are located on the Hadoop Distributed File System (HDFS), which is part of Apache Hadoop.

Conceptually, Giraph is strongly oriented towards the Pregel paper [7]. In the following, we address those concepts of Giraph which can not be found in [7], but are important for the implementation of the distributed evaluation of Datalog with Giraph.

- Worker context: The Pregel paper does not mention any functions that are executed at worker level (recall that `compute()` is executed at vertex level). In Giraph, each worker has a worker context, which includes the following user-defined functions. These can only access values that are not vertex-specific, and can e.g. be used to manage aggregators:
  - Each worker calls `preApplication()` before the beginning of superstep 0. This function can be used to create and initialize (i.e. specify an initial value) aggregators, which can then be accessed from within the worker context (in particular from within the following three functions).
  - After the last superstep has finished but before the result of the Giraph job is written to an output file, each worker calls `postApplication()`. It is mostly used for logging purposes.

– Each worker calls `preSuperstep()` before the beginning of each superstep, in particular before it calls `compute()` on any vertex in any of its partitions. At the time `preSuperstep()` is invoked, it is guaranteed that all aggregated values are consistent across all workers – after that, the local value of an aggregator on one worker may differ from that on another worker, as no synchronization takes place until the end of the superstep.

– Each worker calls `postSuperstep()` at the end of each superstep. This function is mostly used for logging purposes, as the values having been passed to an aggregator during the superstep have not been processed yet (i.e. the local value of an aggregator may differ from worker to worker, as explained above).

- Configuration parameters: The Pregel paper does not mention how configuration parameters like the number of workers, the worker context itself, and the format of the input and output files is specified. Details about how Giraph was configured for the implementation of a distributed, semi-naive evaluation of Datalog are given in Section 4.4.

Please visit the Giraph wiki [16] and the Giraph API [17] for references.

3.4 Application: Single-Source Shortest Paths

In order to round out the picture of Pregel and Giraph, in the following, we present an example application of Giraph to the single-source shortest path (SSSP) problem, inspired by [7]. The SSSP problem consists in finding a shortest path between a single, distinguished source vertex and every other vertex in a graph, where the (non-negative) cost of an edge is given by its edge value.

Figure 3.2 shows a simplified (i.e. abstracted from technical details) distributed algorithm in the spirit of Dijkstra's sequential algorithm, implemented in Giraph.

The input graph for this algorithm is supposed to be directed, with each edge having a non-negative integer as its value. The vertex IDs are arbitrary strings, and the ID of the distinguished source vertex is **a**. Each vertex value is initially set to ∞, which means that the vertex has not yet been reached from the distinguished source vertex.

```
public void compute(Iterator<Integer> msgIterator) {
  int minDist = getVertexId().equals("a")? 0 : ∞;

  while (msgIterator.hasNext())
    minDist = Math.min(minDist, msgIterator.next());

  if (minDist < getVertexValue()) {
    setVertexValue(minDist);
    for (Edge<String,Integer> edge: getOutEdgeMap().values())
      sendMsg(edge.getDestVertexId(), minDist + edge.getEdgeValue());
  }

  voteToHalt();
}
```

Figure 3.2: (Simplified) Implementation of the SSSP problem, inspired by [7]

In the following, we use the first graph from Figure 3.3 as input graph and apply the Giraph job given by Figure 3.2 to it. In Figure 3.3, each active vertex is depicted in red color, and each inactive vertex in gray color. Furthermore, each vertex is divided into two parts: the upper part contains the vertex ID, whereas the lower part contains the vertex value (which is initially set to ∞, as explained above).

- At the beginning of superstep 0, the state of the graph is identical to the input graph, as no computation has been conducted yet. Hence, no messages are received at superstep 0.
  Within superstep 0, the vertex value of the distinguished source vertex **a** is updated from ∞ to 0, which issues messages along **a**'s outgoing edges: one message to **b** with value 1 (as the edge **a**$\rightarrow$**b** has cost 1), and another message to **c** with value 4 (as the edge **a**$\rightarrow$**c** has cost 4).

- At the beginning of superstep 1, **a** is inactive (depicted in gray color), as each vertex calls `voteToHalt()` at the end of `compute()` (see Figure 3.2), and as **a** does not receive any message.
  However, **b** and **c** are active, since they receive the messages that **a** issued during superstep 0. For both **b** and **c**, the received value (1 and 4, respectively) is smaller than their current vertex value (∞ in both cases) – i.e. a path with lower cost than the previously known was discovered. Hence, **b** and **c** update their vertex value to 1 and 4, respectively, and propagate this update along their outgoing edges: **b** issues a message to **c** with value 3 (as **b** can be reached from **a** by a path of cost 1, and the edge **b**$\rightarrow$**c** has cost 2), and **c** issues a message to **b** with value 7.

- At the beginning of superstep 2, **b** and **c** are active since they receive the messages issued during superstep 1.
  In the case of **c**, the vertex value is updated from 4 to 3, which again is propagated along **c**'s outgoing edges: **c** issues a message to **b** with value 6 (as **c** can now be reached from **a** by a path of cost 3, and the edge **c**→**b** has cost 3).
  In the case of **b**, the vertex value stays the same (hence, no messages are issued by **b**), as **b** could already be reached from **a** by a path of cost 1, and the newly discovered path has cost 7.

- At the beginning of superstep 3, only **b** is active as it receives a message with value 6. However, the vertex value of **b** stays unchanged for the same reason as in superstep 2. Hence, as **b** (i) is the only active vertex, (ii) did not issue any message in this superstep, and (iii) calls `voteToHalt()` at the end of `compute()`, the computation terminates (in the next superstep, each vertex would be inactive).

The graph of the last superstep from Figure 3.3 (without the information about active and inactive vertices) represents the result of the computation: the cost of the shortest path from **a** to **a** is 0, from **a** to **b** is 1, and from **a** to **c** is 3.

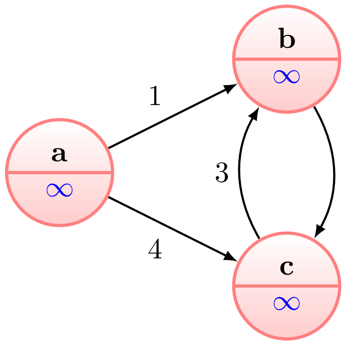
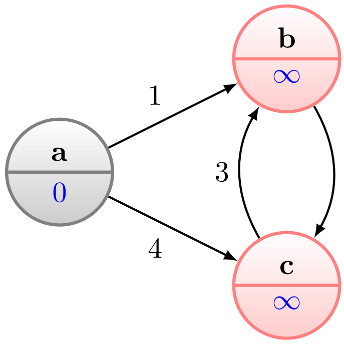
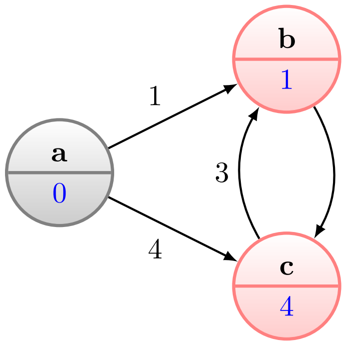
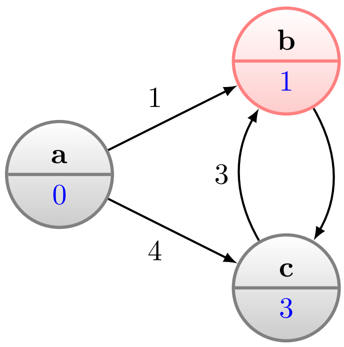
Superstep	State of the graph at the beginning of the superstep	Messages received
0		none
1		– **b** receives 1 from **a** – **c** receives 4 from **a**
2		– **b** receives 7 from **c** – **c** receives 3 from **b**
3		– **b** receives 6 from **c**

Figure 3.3: Giraph job from Figure 3.2 applied to the graph of superstep 0

Chapter 4

Implementing Datalog using Pregel

This chapter represents the main part of this work: Implementing a distributed algorithm for the semi-naive evaluation of Datalog using Pregel.

4.1 Motivation on how to implement Datalog using Pregel

This section motivates how the semi-naive evaluation of Datalog (see Figure 2.2) is mapped to Pregel.

In general, there are two ways how problems can be parallelized across multiple processors:

- each processor applies different functions to the same data, which is known as *task parallelism*, and
- each processor applies the same function to different data, which is known as *data parallelism*.

By design (as each active vertex invokes the same `compute()`-function in each superstep, and as the input data is usually distributed among the vertices), Pregel is better suited for data parallelism than for task parallelism. Nevertheless, task parallelism can also be realized in Pregel if each vertex has access to the same data, and if the `compute()`-function consists of several cases, where the case to be executed is determined by e.g. the vertex ID (e.g. realized by an `if-then-else`, or a `switch-case` statement).

According to [18, p. 125], the degree of achievable task parallelism does not grow much with the size of the problem to be solved, while the degree of achievable data parallelism usually does.

For these reasons, a data parallel mapping of the (sequential) semi-naive algorithm for the evaluation of Datalog (see Figure 2.2) to Pregel seems more worthwhile than a task parallel mapping in the case of large Datalog programs with a considerable amount of facts.

Given a Datalog program P that consists of a set of rules R (with non-empty body) and a set of EDB facts F, a data parallel mapping to Pregel implies that each Pregel vertex should perform the same computation within each superstep. In the case of the semi-naive evaluation (see Figure 2.2), this computation solely consists of applying the whole set of rules R to the already derived facts. In order for each vertex to conduct this computation, we choose to store the whole set of rules R as part of the vertex value of each vertex.

In addition, applying a data parallel mapping also implies that the remaining part of the Datalog program P, namely the EDB facts F are distributed among the vertices. This poses the question of the granularity of such a distribution. If each vertex held all facts of a specific EDB or IDB relation, the degree of parallelism achievable would probably be low when evaluating Datalog programs with few, but large relations. For this reason, we choose a very fine-grained approach for distributing the facts among the vertices: for each constant c appearing in P, a vertex with ID c is created that (in its vertex value) stores all those EDB and IDB facts, where c appears as argument. To illustrate, a fact `p(a,b)` would be stored both in the vertex value of vertex `a` and in the vertex value of vertex `b`. This replication may seem disadvantageous at first sight, but it allows to directly address the corresponding vertex when subgoals containing at least one constant are to be processed, e.g. `p(a,X)` and `p(Y,b)` can directly answered by vertex `a` and `b` respectively, without any indirection step being necessary.

To briefly summarize: The data parallel mapping of a Datalog program P to Pregel as presented above consists in

- creating a vertex for every constant c occurring in P and storing (in its vertex value) all those facts from all (EDB and IDB) relations, where c appears as argument, and
- storing the complete set of rules (with non-empty body) R from P in the value of each vertex.

Next, the question is how the function `EVAL-INCR` of Figure 2.2 can be evaluated in a data parallel way, i.e. such that each vertex performs the same computation within each superstep.

In each round, the computation to be performed consists in evaluating `EVAL-INCR`(R, F, Δ_{old}), which for all rules $r \in R$, tries to derive new facts by satisfying at least one subgoal of r with a fact from Δ_{old}, and all remaining subgoals with facts from F.

As consequence of the data parallel mapping discussed above, the facts from F are distributed among the vertices. However, in order to satisfy all subgoals of a Datalog rule r, in general facts stored at different vertices will have to be taken into account. Hence, some kind of communication between the vertices is necessary. In Pregel, communication between vertices is realized via messages, which are affected by Pregel's barrier synchronization: A message issued at superstep S will not be available at its destination vertex until the next superstep $S+1$ (see Section 3.2). This implies, that in general, more than one superstep is necessary for the computation of one round of the semi-naive evaluation, i.e. for evaluating one call of `EVAL-INCR`.

To solve this issue, rules are processed on a per-subgoal base, i.e. the vertices try to collaboratively satisfy all subgoals, one after the other. To illustrate, consider a rule `r(X,Z):- p(X,Y) & q(Y,Z).` that is processed at vertex `a` whose vertex value contains the fact `p(a,b)`. This fact is used to satisfy and afterwards remove the first subgoal of the rule, leading to a new rule `r(a,Z):- q(b,Z).`. As the remaining subgoal contains the constant `b`, this rule is sent to vertex `b` for further processing (note that vertex `b` stores all facts where `b` appears as argument). Vertex `b` now proceeds in the same way, and tries to satisfy `q(b,Z)` with one of its facts, say `q(b,c)`. The resulting rule `r(a,c):- .` has an empty body, i.e. it is a fact and is stored at the vertex values of vertices `a` and `c`.

Now, consider another rule `r(X,Y):- p(X) & q(Y).` that expresses the cross product of relations `p` and `q`. No matter which vertex satisfies the subgoal `p(X)`, the resulting rule will have `q(Y)` as its only subgoal. `q(Y)` however can be satisfied by any fact having `q` as its predicate symbol, and these facts are stored in a distributed manner across all vertices (e.g. `q(a)` is stored at vertex `a`, `q(b)` is stored at vertex `b`, and so forth). Hence, each vertex needs to able to send rules (i.e. messages) to all vertices, including itself. This is realized by introducing outgoing edges from each vertex to all vertices in Pregel, used for sending messages as broadcast.

```
//Input: F (set of Datalog EDB facts),
//       R (set of Datalog rules with non-empty body)
Δ_old := F
while Δ_old ≠ ∅
   Δ_new := EVAL-INCR-DIST(R, F, Δ_old)
   F := F ∪ Δ_new
   Δ_old := Δ_new
output F
```

Figure 4.1: Distributed semi-naive evaluation of a Datalog program

Finally, the data parallel mapping of a Datalog program P to Pregel consists in

- creating a vertex for every constant c mentioned in P and storing (in its vertex value) all those facts from all (EDB and IDB) relations, where c appears as argument,
- storing the complete set of rules (with non-empty body) R from P in the vertex value of each vertex,
- introducing outgoing edges from each vertex to all vertices, used for broadcast messages.

In the following section, we present the concept of parallelizing the semi-naive evaluation from Figure 2.2 on a distributed system based on Pregel.

4.2 Distributed Semi-naive Evaluation

In this section, we present the concept of the distributed semi-naive evaluation, with reference to the algorithm shown in Figure 4.1.

The input for the distributed semi-naive evaluation is the same as for the non-distributed algorithm: a Datalog program P, consisting of a set of rules (with non-empty body) R and a set of EDB facts F. Let C be the set of constants appearing in P.

As a first step, P has to be distributed, as already described in Section 4.1. Since we use Pregel, we generate a graph consisting of one vertex for each constant $c \in C$ with the following vertex structure:

- Vertex ID: c
- State-flag: Active (default, see Section 3.2).
- Vertex value: Consisting of the whole set of rules R and the set of facts F_c from F where c appears as argument. Note that $\bigcup_{c \in C} F_c = F$.

- Incoming messages: Empty set (default, see Section 3.2).
- Outgoing edges: Outgoing edges to all vertices (recall that these are used for broadcast messages)

The state of the vertices changes with each performed superstep and finally represents the result of the evaluation after the last superstep, in a distributed manner.

In the following, we realise the algorithm of Figure 4.1 within Pregel:

- In line 3, Δ_{old} is initialized with the set of EDB facts F. Note that, as explained above, F is stored in a distributed manner across all vertices, by design of the initial vertex values: $F = \bigcup_{c \in C} F_c$. The same applies to Δ_{old}, i.e. a vertex c stores $\Delta_{old,c}$, the set of all facts from Δ_{old} where c occurs as argument.

- Next, we discuss the `while` loop (lines 4-7):

  - In line 5, `EVAL-INCR-DIST(`R`,`F`,`Δ_{old}`)` is evaluated, where R is the set of Datalog rules with non-empty body, F is the set of all facts that have been derived in previous rounds, and Δ_{old} is the set of facts that has been derived in the previous round. `EVAL-INCR-DIST` is computed in a sequence of supersteps using messages for communication between the vertices, which is described in detail below. The result of `EVAL-INCR-DIST` is the set Δ_{new}, consisting of all facts that have been derived in the current round, but not in one of the previous rounds. Δ_{new} is stored in a distributed manner across all vertices: a vertex c stores $\Delta_{new,c}$, the set of all facts from Δ_{new} where c occurs as argument.
  - In line 6, the union $F \cup \Delta_{new}$ is formed. This set is stored and computed in a distributed manner, as each vertex forms the union locally: $F_c := F_c \cup \Delta_{new,c}$ (recall that $\bigcup_{c \in C} F_c = F$ and $\bigcup_{c \in C} \Delta_{new,c} = \Delta_{new}$).
  - In line 7, Δ_{old} is assigned the value of Δ_{new} for the next round. This assignment is also done in a distributed manner: each vertex c sets $\Delta_{old,c}$ to the value of $\Delta_{new,c}$.

  The `while` loop (see line 4) terminates, when no facts for Δ_{new} are inferred in `EVAL-INCR-DIST`. This, in consequence, violates the condition of the loop (as $\Delta_{old} = \Delta_{new} = \emptyset$).

- The output F (see line 8) is distributed across all vertices. Collecting the distributed output is handled by design of the Pregel system: After the last superstep, all vertex values are written into an output file on the distributed file system. The union over all vertex values contains the set of all rules R (with non-empty body), and the desired result $F = \bigcup_{c \in C} F_c$.

Next, we discuss the the local execution of `EVAL-INCR-DIST` on an arbitrary vertex with ID c.

We first define a helper function `EVAL-DIST(`R'`,`F'`)`, which receives a set of rules R' with non-empty body and a set of facts F' as arguments, and returns the following set:

$$\begin{aligned}\{h\sigma \text{ :- } b_1\sigma \ \& \ \ldots \ \& \ b_n\sigma \mid {} & \exists(h \text{ :- } b_1 \ \& \ \ldots \ \& \ b_n) \in R'(\\ & \exists k(1 \leq k \leq n \wedge \\ & \exists \sigma, \text{ where } \sigma \text{ is a substitution, } (b_k\sigma \in F')))\}\end{aligned}$$

In the algorithm of Figure 4.1, one call of `EVAL-INCR-DIST` corresponds to one round of the semi-naive evaluation. As noted above, the evaluation of `EVAL-INCR-DIST` requires a sequence of supersteps S. We now examine one arbitrary superstep of S:

Initially, we filter the incoming messages for rules with empty body (i.e. facts), which are added to $\Delta_{new,c}$. The remaining messages are denoted by $R_{in,c}$. We have to consider two cases:

- If the current superstep is the first of a round, we define $\Delta'_{old,c}$ to be the set of facts from $\Delta_{old,c}$, where c appears as first argument, and evaluate `EVAL-DIST`$(R, \Delta'_{old,c})$.
- Otherwise (if the current superstep is not the first superstep of a round), we evaluate `EVAL-DIST`$(R_{in,c}, F_c)$.

In any case, the result of `EVAL-DIST` is denoted by $R_{ver,c}$.

Let $R_{out,c}$ be an empty set. All rules $r \in R_{ver,c}$ need to be verified. For this, all variable-free subgoals of r are checked:

- If the variable-free subgoal is ordinary and contains c as argument, we check if the subgoal is contained in F_c as fact. If this is the case, the subgoal is removed from r, else the rule r is discarded (i.e. not added to $R_{out,c}$).
- If the variable-free subgoal is ordinary and does not contain c as argument, r is not altered.
- If the variable-free subgoal is built-in and evaluates to `true`, the subgoal is removed from r. If it evaluates to `false`, r is discarded (i.e. not added to $R_{out,c}$).

```
path(X,Y) :- edge(X,Y).
path(X,Z) :- path(X,Y) & path(Y,Z).
edge(a,b).
edge(b,c).
edge(c,a).
```

Figure 4.2: Example Datalog program for computing the transitive closure over a cyclic base relation

After all variable-free subgoals of r have been checked, r is added to $R_{out,c}$, unless it was discarded.

As a last step, the rules in $R_{out,c}$ are issued as messages. The recipients for the messages are obtained in the following way: If the body of a rule is empty, the recipients are all vertices whose ID appears as argument in the head of the rule. Otherwise, the ordinary subgoals are searched for constants. If constants were found, the rule is sent to all vertices whose ID is equal to one of the constants. Otherwise (if the ordinary subgoals contain no constants), the rule is broadcasted, as any vertex might be able to satisfy one of its subgoals. After issuing the messages, the superstep ends.

The next superstep of S starts. `EVAL-INCR-DIST` terminates when no more messages are issued in a superstep. After the last superstep of S, Δ_{new} is stored in a distributed manner across all vertices: $\Delta_{new} = \bigcup_{c \in C} \Delta_{new,c}$.

This concludes the concept of the distributed semi-naive evaluation using Pregel. In the next section, we illustrate our approach with a simple example, followed by a section, which presents the Java implementation of the algorithm.

4.3 Evaluation by Example

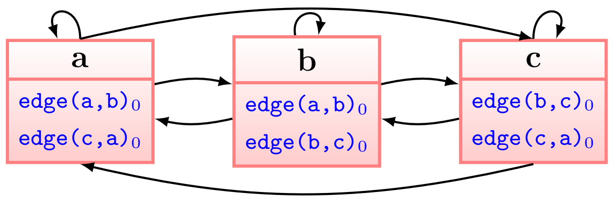
In this section, we carry out our algorithm for the distributed evaluation of Datalog programs on a simple, yet illustrative example program shown in Figure 4.2.

First, we begin by creating the vertex structure. The only constants appearing in the Datalog program in Figure 4.2 are `a`, `b`, and `c`, hence we create vertices with IDs `a`, `b`, and `c`. Each vertex stores all those facts from Figure 4.2, where its ID appears as argument. Therefore, vertex `a` stores the facts `edge(a,b)` and `edge(c,a)`, vertex `b` stores the facts `edge(a,b)` and `edge(b,c)`, and vertex `c` stores the facts `edge(b,c)` and `edge(c,a)`. Furthermore, each vertex stores the two rules shown in lines 1 and 2 of Figure 4.2, and each vertex has outgoing edges to all vertices.

In the following tables, the left column depicts the state of the graph, and the right column contains all messages that are available at the corresponding superstep. For reasons of clarity, the rules in lines 1 and 2 of Figure 4.2 are omitted from the vertices in the left column – nevertheless, they are part of the vertex value of each vertex. Each fact that is stored in a vertex value has an index indicating the number of the round, in which this fact has been derived (e.g. `path(a,a)`$_3$ means that the fact `path(a,a)` has been derived in round 3). Note that in the tables below, the facts of Δ_{old} are always those facts indexed with the number of the previous round.

In the following descriptions, we use the notation introduced in Section 4.2. Additionally, by "rule 1" and "rule 2", we denote the rule in line 1 and 2 of Figure 4.2, respectively.

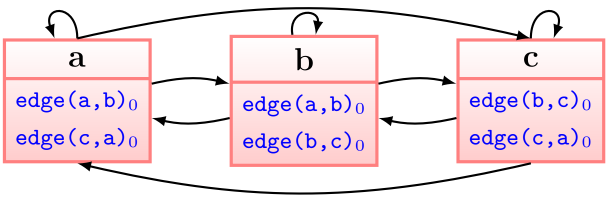
Superstep 0 (round 1)

<table>
<tr><th>State of the graph at the beginning</th><th>Messages received</th></tr>
<tr><td>
</td><td>none</td></tr>
</table>

The first superstep, superstep 0, is the first superstep of round 1. In this case, each vertex v just uses those facts from $\Delta_{old,v}$ where its vertex ID appears as first argument, e.g. vertex `a` just uses `edge(a,b)`, but does not use `edge(c,a)`. Let's continue to describe the computation that vertex `a` performs: Regarding rule 1, the subgoal `edge(X,Y)` can be satisfied by `edge(a,b)`, leading to the rule `path(a,b):- edge(a,b).` which is added to $R_{ver,\texttt{a}}$. Regarding rule 2, no subgoal can be satisfied, as no `path`-facts have been derived yet. Verifying the only rule in $R_{ver,\texttt{a}}$ leads to the fact `path(a,b):- .`, which is added to $R_{out,\texttt{a}}$, and sent to vertices `a` and `b`.

The computation at the remaining vertices `b` and `c` is performed in an analogous manner and results in the fact `path(b,c):- .` and `path(c,a):- .`, respectively. Each of these two facts is also sent to its corresponding vertices.

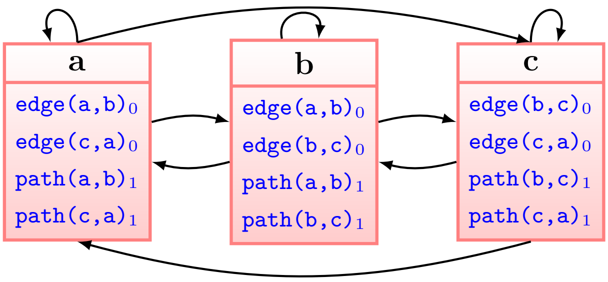
Superstep 1 (round 1)

State of the graph at the beginning	Messages received
	**a:** `path(a,b). path(c,a).` **b:** `path(a,b). path(b,c).` **c:** `path(b,c). path(c,a).`

Superstep 1 also belongs to round 1, as messages have been issued in superstep 0. These messages are now available at their destination vertices, as depicted in the right column of the above table. Let's again look at vertex `a`: it receives the facts `path(a,b)` and `path(c,a)` and adds these to $\Delta_{new,\mathtt{a}}$, which is stored in `a`'s vertex value. No rules with non-empty subgoal have been received, so $R_{in,\mathtt{a}}$ is empty, therefore `EVAL-DIST` produces an empty output and no messages are issued.

Analogously, vertex `b` and `c` receive only facts and store them in $F_{\mathtt{b}}$ and $F_{\mathtt{c}}$, respectively.

Superstep 2 (round 2)

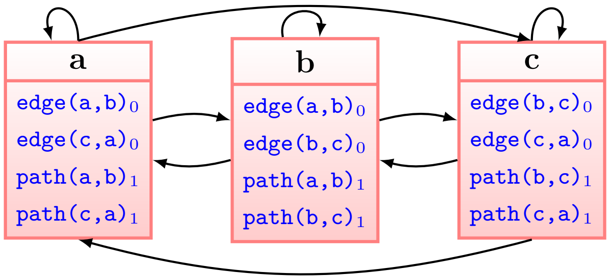
State of the graph at the beginning	Messages received
	none

As none of the vertices issued a message in superstep 1, `EVAL-INCR-DIST` terminates and each vertex v adds all facts from $\Delta_{new,c}$ to F_c and sets $\Delta_{old,c}$ to the value of $\Delta_{new,c}$. Since $\Delta_{old} \neq \emptyset$, a new round starts (round 2). As superstep 2 is the first superstep of this new round, for each vertex v only those facts in $\Delta_{old,v}$ are considered, where v appears as first argument.

For vertex `a`, the computation proceeds as follows: `path(a,b)` is the only fact in $\Delta_{old,\mathtt{a}}$ with `a` as first argument. This fact can't satisfy the subgoal of rule 1, but it can satisfy both subgoals `path(X,Y)` and `path(Y,Z)` of rule 2. For the first subgoal `path(X,Y)` the resulting rule is `path(a,Z):- path(a,b) & path(b,Z).`, and for the second subgoal it is `path(X,b):- path(X,a) & path(a,b).`. In the verification step, the satisfied subgoal `path(a,b)` is removed from both rules, which results in the rules `path(a,Z):- path(b,Z).` and `path(X,b):- path(X,a)..` These are sent to vertex `b` and `a`, respectively, for further processing.

Again, the computation at the remaining vertices `b` and `c` is performed in an analogous manner.

Superstep 3 (round 2)

<table>
<tr><th>State of the graph at the beginning</th><th>Messages received</th></tr>
<tr><td>
</td><td>a: <code>path(c,Z):- path(a,Z).</code>
<code>path(X,b):- path(X,a).</code>
b: <code>path(a,Z):- path(b,Z).</code>
<code>path(X,c):- path(X,b).</code>
c: <code>path(b,Z):- path(c,Z).</code>
<code>path(X,a):- path(X,c).</code></td></tr>
</table>

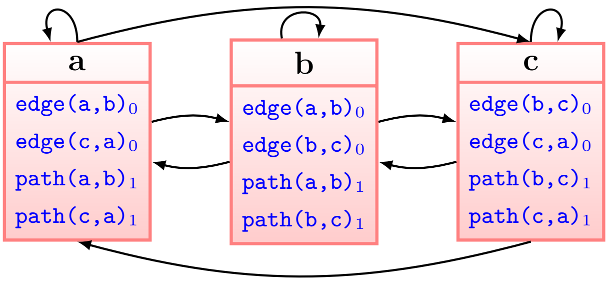
As messages have been issued in superstep 2, superstep 3 still belongs to the same round as superstep 2 (round 2). In contrast to superstep 1, the messages received at the current superstep 3 are no facts, but rules with non-empty body.

In the case of vertex `a`, the computation proceeds as follows: As both incoming messages `path(c,Z):- path(a,Z).` and `path(X,b):- path(X,a).` have non-empty body, they are added to $R_{in,\texttt{a}}$. Vertex `a` can use all facts from $F_{\texttt{a}}$ for satisfying the subgoals of these two rules.

Concerning the first rule, the subgoal `path(a,Z)` can be only satisfied by the fact `path(a,b)`. Concerning the second rule, the subgoal `path(X,a)` can be only satisfied by `path(c,a)`. After verification, both rules result in the fact `path(c,b):- .`, which is sent to vertices `b` and `c`.

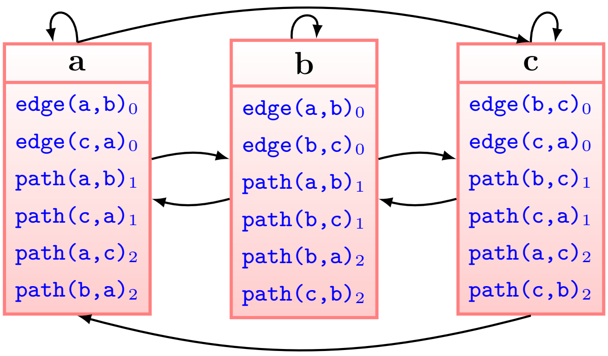
For the vertices `b` and `c`, the derived facts are `path(a,c)` and `path(b,a)`, respectively.

Superstep 4 (round 2)

<table>
<tr><th>State of the graph at the beginning</th><th>Messages received</th></tr>
<tr><td>
</td><td>a: <code>path(a,c). path(b,a).</code>
b: <code>path(b,a). path(c,b).</code>
c: <code>path(a,c). path(c,b).</code></td></tr>
</table>

The computation in superstep 4 is similar to that of superstep 1: As messages have been issued in superstep 3, superstep 4 still belongs to round 2 and just facts are received as incoming messages, each of which is stored in the set $\Delta_{new,v}$ in the vertex value of the corresponding vertex v.

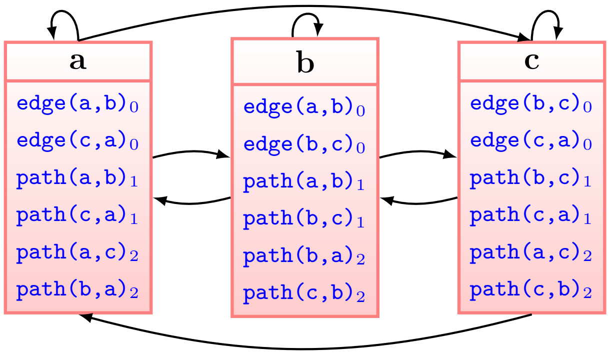
Superstep 5 (round 3)

<table>
<tr><th>State of the graph at the beginning</th><th>Messages received</th></tr>
<tr><td>
</td><td>none</td></tr>
</table>

As none of the vertices issued a message in superstep 4, `EVAL-INCR-DIST` terminates and each vertex v adds all facts from $\Delta_{new,c}$ to F_c and sets $\Delta_{old,c}$ to the value of $\Delta_{new,c}$. Since $\Delta_{old} \neq \emptyset$, a new round is started (round 3), and hence superstep 5 is the first superstep in this new round. This implies, that for each vertex v only those facts in $\Delta_{old,v}$ are considered, where v appears as first argument.

In the case of vertex `a`, the computation proceeds as follows. The only fact in $\Delta_{old,\texttt{a}}$ with `a` as first argument is `path(a,c)`. This fact can't satisfy the subgoal of rule 1, but it can satisfy both subgoals `path(X,Y)` and `path(Y,Z)` of rule 2. For the first subgoal `path(X,Y)`, the resulting rule is `path(a,Z):- path(a,c) & path(c,Z).`, and for the second subgoal, it is `path(X,c):- path(X,a) & path(a,c).`. In the verification step, the satisfied subgoal `path(a,c)` is removed from both rules, which results in the rules `path(a,Z):- path(c,Z).` and `path(X,c):- path(X,a).`. These are sent to vertex `c` and `a` for further processing, respectively.

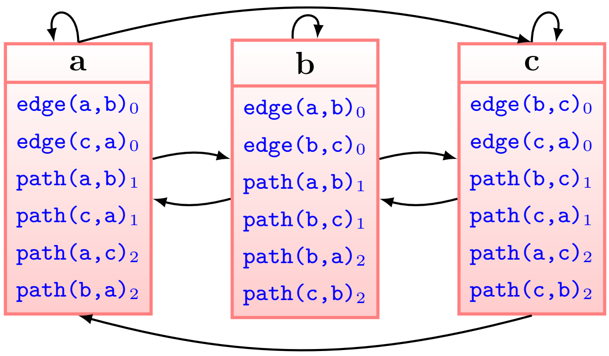
Superstep 6 (round 3)

State of the graph at the beginning	Messages received
	**a:** `path(b,Z):- path(a,Z).` `path(X,c):- path(X,a).` **b:** `path(c,Z):- path(b,Z).` `path(X,a):- path(X,b).` **c:** `path(a,Z):- path(c,Z).` `path(X,b):- path(X,c).`

The computation of superstep 6 is similar to that of superstep 3: each vertex v uses facts from F_v to satisfy a subgoal in the incoming rules.

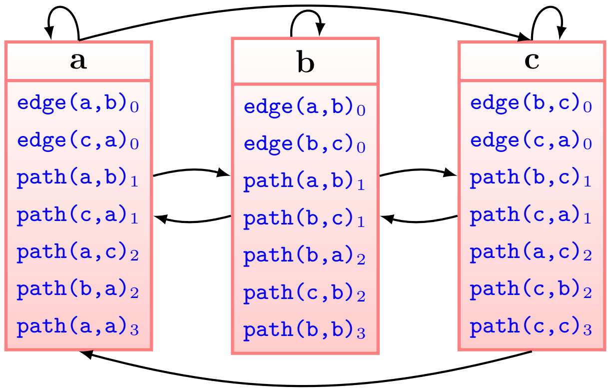
Let's again look at vertex `a`. Its first incoming rule `path(b,Z):- path(a,Z).` leads to the facts `path(b,b)` and `path(b,c)`, and its second incoming rule leads to the facts `path(c,c)` and `path(b,c)`. All of these facts are sent to the corresponding vertices.

Superstep 7 (round 3)

State of the graph at the beginning	Messages received
	**a:** `path(a,a).` `path(a,b).` `path(c,a).` **b:** `path(a,b).` `path(b,b).` `path(b,c).` **c:** `path(b,c).` `path(c,a).` `path(c,c).`

The computation in superstep 7 is again similar to that of superstep 1: superstep 7 still belongs to round 3 and just facts are received as incoming messages, each of which is stored in the set $\Delta_{new,v}$ in the vertex value of the corresponding vertex v.

Superstep 8 (round 4)

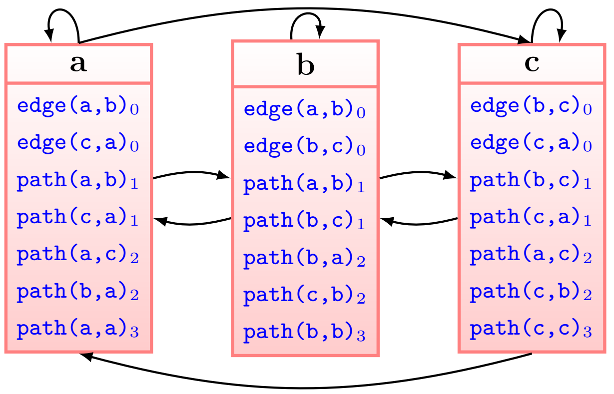
<table>
<tr><th>State of the graph at the beginning</th><th>Messages received</th></tr>
<tr><td>
</td><td>none</td></tr>
</table>

As no message was issued in superstep 7, `EVAL-INCR-DIST` terminates and each vertex v adds all facts from $\Delta_{new,c}$ to F_c and sets $\Delta_{old,c}$ to the value of $\Delta_{new,c}$. Since $\Delta_{old} \neq \emptyset$, a new round is started (round 4), and hence superstep 8 is the first superstep in this new round. This implies, that for each vertex v only those facts in $\Delta_{old,v}$ are considered, where v appears as first argument.

In the case of vertex `a`, `p(a,a)` is the only fact that can be used to satisfy the subgoals of rule 2, which finally generates the rules `path(a,Z):- path(a,Z).` and `path(X,a):- path(X,a).` and which both are sent to vertex `a` for further processing.

As all rules issued in this superstep are not able to derive new facts (since the only subgoal of the body is equal to the head), these rules could be discarded in an optimized version of the algorithm.

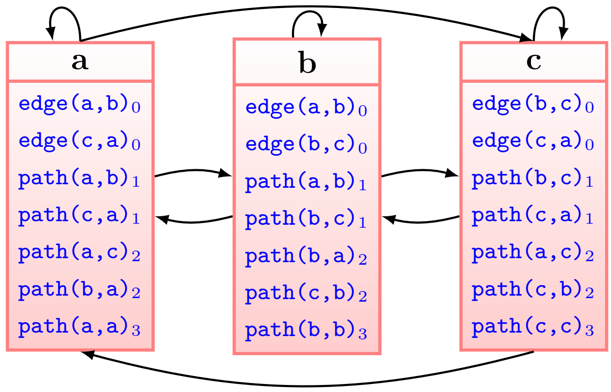
Superstep 9 (round 4)

<table>
<tr><th>State of the graph at the beginning</th><th>Messages received</th></tr>
<tr><td>
</td><td>a: <code>path(a,Z):- path(a,Z).</code>
<code>path(X,a):- path(X,a).</code>
b: <code>path(b,Z):- path(b,Z).</code>
<code>path(X,b):- path(X,b).</code>
c: <code>path(c,Z):- path(c,Z).</code>
<code>path(X,c):- path(X,c).</code></td></tr>
</table>

The computation of superstep 9 is again similar to that of superstep 3:

each vertex v uses facts from F_v to satisfy a subgoal in each incoming rule.

Superstep 10 (round 4)

State of the graph at the beginning	Messages received
	**a:** `path(a,a).` `path(a,b).` `path(a,c).` `path(b,a).` `path(c,a).` **b:** `path(a,b).` `path(b,a).` `path(b,b).` `path(b,c).` `path(c,b).` **c:** `path(a,c).` `path(b,c).` `path(c,a).` `path(c,b).` `path(c,c).`

The computation in superstep 10 is again similar to that of superstep 1: superstep 10 still belongs to round 4, and just facts are received as incoming messages. However, none of these facts is “new”, i.e. all received facts have already been derived in previous rounds and are already contained in F. Hence, `EVAL-INCR-DIST` returns the empty set, and in consequence, the `while` loop terminates as its condition is not satisfied anymore.

The result of the computation is contained in the vertex values of `a`, `b`, and `c`, and represents the minimal model of the Datalog program from Figure 4.2.

4.4 Java Implementation

In this section, we give an overview of the Java implementation of the algorithm in Figure 4.1. The complete source-code can be found in the Appendix.

The implementation is based on Apache Giraph version 0.1 [9] and Hadoop version 0.20.203.0 [14]. In addition to that, we make use of the parser component of IRIS [19], an open source Datalog reasoner.

The `main()`-function is located in the class `PDoP` (Appendix A), and calls `run()`. In the `run()`-function, the Giraph job is configured and started. The configuration consists of the following tasks:

- Setting the worker context
  (class `PDoPWorkerContext`, Appendix B)
- Specifying the input format
  (class `PDoPInputFormat`, Appendix F)
- Specifying the output format
  (class `PDoPOutputFormat`, Appendix G)
- Setting the worker configuration (Specifying the number of workers, and fault tolerance)

Setting the worker context

The worker context (class `PDoPWorkerContext`, Appendix B) specifies the following four functions `preApplication()`, `postApplication()`, `preSuperstep()`, and `postSuperstep()`:

- `preApplication()`: In this function, aggregators are registered and initialized. For our application, we use four aggregators:
  - `VoteAggregator` is used to check if all vertices are finished with the current round, and whether a new round can start. The `VoteAggregator` is realized by a boolean aggregator that uses the boolean `and` operation as method of aggregation (class `AndAggregator`, Appendix C).
  - `LastTimeAggregator` is used to check if a new round was started in the previous superstep, which indicates the first superstep of a round. The `LastTimeAggregator` is realized by a boolean aggregator that uses the boolean `or` operation as method of aggregation (class `OrAggregator`, Appendix D).
  - `TerminateAggregator` is used to detect when the algorithm is finished. In simple Giraph jobs, `voteToHalt()` is called at the end of every superstep, thus making the job stop when no more messages are issued, but in our case no more messages just indicate that a new round can start. For this reason we need this aggregator to indicate when to terminate (utilizing the `OrAggregator`, Appendix D).
  - `RoundAggregator` (Appendix E) is used as the round counter. The round counter indicates how often the `while` loop of the evaluation algorithm has been entered so far. This is useful if we want to identify in which round a fact was derived, and gives the possibility to differentiate facts of Δ_{old}, Δ_{new}, and F according to a specific round. (The `RoundAggregator` is

useful for debugging purposes, but the algorithm could be implemented without it.)

- `preSuperstep()`: This function contains the logic for evaluating the aggregated values. Placing the logic in `preSuperstep()` ensures, that every worker has the same aggregator values before calling `compute()` on its active vertices.

- `postSuperstep()` and `postApplication()` are not used for the implementation of our algorithm.

Specifying the input and output format

The input and output format are specified by the following classes:

- The class `PDoPInputFormat` uses the class `PDoPReader` to read the input file from the HDFS, which contains one line for each vertex represented in the following structure:

  ```
  [<ID> <RULES><FACTS> <EDGES>]
  ```

  where `<ID>` is the vertex ID, `<RULES>` are all rules with non-empty body, `<FACTS>` are only those facts concerning the specific vertex (as specified in Section 4.2), and `<EDGES>` are all vertex IDs (separated by commas) indicating the destination for the edges. Rules and facts are given in Datalog syntax, as specified by IRIS.

- The class `PDoPOutputFormat` uses the class `PDoPWriter` to write the results of each vertex to an output file on the HDFS.

Having finished the configuration of the Giraph job, we can start the evaluation, which brings us to the central function: `compute()`.

The `compute()`-function

The structure of the `compute()`-function is as follows:

1. If the value of the `TerminateAggregator` is `true`, the vertex calls `voteToHalt()` and the computation terminates.

2. The vertex value is parsed using the class `RuleFactQuerySplitter` (Appendix J).

3. The incoming messages are parsed and filtered using the `RuleFactQuerySplitter` and the `Filter` classes (Appendix K). `Filter` uses the interface `Condition` (Appendix L) together with

its implementations for the type of facts to be generated (Appendices: M,N,O). With `Filter`, we can generate $\Delta_{old,v}$, $\Delta_{new,v}$, and F_v from all facts that are available to a vertex v, as needed (see Section 4.2).

4. Next, the evaluation using the appropriate rules according to Section 4.2 is executed by utilizing the class `PartialEvaluation` (Appendix P). If the evaluation generated new rules, these are sent as messages.

5. As a last step, the `VoteAggregator` is called (with a positive vote, if no messages were sent), and the vertex value is updated.

Finally, in order to represent relations, the class `SimplePDoPRelation` (Appendix Q) has been created, based on the class `SimpleRelation` from the IRIS reasoner, as the latter does not offer a public constructor.

For further details refer to the Appendix, as this section gave just an overview of our implementation.

Chapter 5

Related Work

Besides this work, another approach to the distributed evaluation of Datalog is DEDALUS [10]. In the following, we give a short introduction to its key ideas.

DEDALUS is a language that builds upon Datalog and includes negation, aggregation, and a choice construct. In order to capture time, DEDALUS uses a `successor` predicate with `successor(X,Y)` being true iff $\mathtt{X} = \mathtt{Y} + 1$. In a well-formed DEDALUS rule, every subgoal must use the same variable $\mathcal{T}$ at its rightmost attribute, and the variable $\mathcal{S}$ at its rightmost head attribute. $\mathcal{S}$ and $\mathcal{T}$ describe time, and $\mathcal{S}$ must be restricted in one of the following two ways:

> 1. A rule is deductive if $\mathcal{S}$ is bound to the value $\mathcal{T}$; that is, the body contains the subgoal $\mathcal{S} = \mathcal{T}$.
> 2. A rule is inductive if $\mathcal{S}$ is the successor of $\mathcal{T}$; that is, the body contains the subgoal successor($\mathcal{T}$, $\mathcal{S}$).
>
> [10]

With these supplements, the two key features of DEDALUS are achieved: mutable state and asynchronous processing and communication.

Mutable State

Mutable state is achieved by introducing predicates `p_pos` and `p_neg` for each EDB predicate `p` in a DEDALUS program, together with the following rules

$$\texttt{p\_pos}(\texttt{A}_1\texttt{,A}_2\texttt{,}\ldots\texttt{,A}_n\texttt{,}\mathcal{S}) \texttt{ :- } \texttt{p}(\texttt{A}_1\texttt{,A}_2\texttt{,}\ldots\texttt{,A}_n\texttt{,}\mathcal{T})\texttt{, } \mathcal{S} = \mathcal{T}\texttt{.} \tag{5.1}$$

$$\begin{aligned} \texttt{p\_pos}(\texttt{A}_1\texttt{,A}_2\texttt{,}\ldots\texttt{,A}_n\texttt{,}\mathcal{S}) \texttt{ :- } & \texttt{p\_pos}(\texttt{A}_1\texttt{,A}_2\texttt{,}\ldots\texttt{,A}_n\texttt{,}\mathcal{T})\texttt{,} \\ & \neg\texttt{p\_neg}(\texttt{A}_1\texttt{,A}_2\texttt{,}\ldots\texttt{,A}_n\texttt{,}\mathcal{T})\texttt{,} \\ & \texttt{successor}(\mathcal{T}\texttt{, } \mathcal{S})\texttt{.} \end{aligned} \tag{5.2}$$

Rule 5.1 is a deductive rule that derives a `p_pos`-fact for each `p`-fact, but also allows rule 5.2 to derive additional `p_pos`-facts. Rule 5.2 is an inductive rule that captures the notion of mutable state: as long as no corresponding `p_neg`-fact exists, a `p_pos`-fact at time $\mathcal{T}$ will also be true at the next point in time $\mathcal{S}$.

Asynchronous Processing and Communication

By introducing a `choice` construct, asynchronous processing and communication is possible. `choice` allows to model non-determinism, e.g. as it occurs in unreliable networks with delay, loss of or out-of-order messages. A `choose` subgoal looks as follows:

$$\texttt{choose}(\mathbf{X}\texttt{,}\mathbf{Y})\texttt{,}$$

where $\mathbf{X} := (\texttt{X}_1\texttt{,X}_2\texttt{,}\ldots\texttt{,X}_n)$ and $\mathbf{Y} := (\texttt{Y}_1\texttt{,Y}_2\texttt{,}\ldots\texttt{,Y}_m)$. This subgoal enforces the functional dependency $\mathbf{X} \rightarrow \mathbf{Y}$: for each value of $\mathbf{X}$, a value for $\mathbf{Y}$ has to be "chosen" non-deterministically.

With these additions, implementing Lamport clocks and reliable broadcasts is possible in Dedalus (which is not possible in pure Datalog), see [10].

Comparison of Dedalus with our implementation

Dedalus and our implementation are similar in the following aspects:

- Concerning the distribution, both in our implementation and in Dedalus, each compute node stores the whole set of rules, while the facts are distributed across the compute nodes.
- Concerning the evaluation, Dedalus also makes use of the semi-naive algorithm for efficiently evaluating a single timestep.

However, they differ in the following aspects:

- Our implementation focuses on pure Datalog, while Dedalus is an extension thereof, including (among others) an explicit notion of time, i.e. a fact $\texttt{p}(\texttt{A}_1\texttt{,}\ldots\texttt{,A}_n\texttt{,}\mathcal{T})$ is interpreted as the fact $\texttt{p}(\texttt{A}_1\texttt{,}\ldots\texttt{,A}_n)$ being true at timestep $\mathcal{T}$.

- Dedalus is not synchronized by design, i.e. it may happen that facts from the future are used to derive facts in the past. Our implementation, however, is synchronized by design of the underlying framework Pregel.

Chapter 6

Conclusion and Future Work

Not only in the case of social networks, companies are interested in data they can use to generate profit. Thus, as a logical consequence, collecting this data (e.g. user activity) in data warehouses has become common practise. However, the efficient evaluation of this data is not yet that well established [20]. Knowledge discovery in big data sets is gaining ever growing importance, with the side effect of a renewed interest in Datalog. Recent work [21] has successfully connected these three cornerstones: Datalog, Machine Learning, and Big Data. This gives us sufficient claims that the efficient evaluation of arbitrary Datalog programs on a compute cluster is of great value, not only to academia.

In this work, we developed a data parallel, semi-naive evaluation algorithm for Datalog programs that utilizes the distributed graph processing framework Pregel. We have shown the method of operation of this distributed algorithm, and are fairly confident that this approach scales well for Big Data applications – even if an experimental evaluation remains to be done, as it is outside the scope of this bachelor's thesis.

When querying a data set multiple times, computing the minimal model just once and using it for query answering is more efficient than evaluating each query separately, e.g. in a top-down manner. With our approach, we are able to generate the minimal model using Pregel, which was explicitly designed for practical computing problems concerning large graphs [7].

In future work, the distributed, semi-naive evaluation algorithm for Datalog developed in this thesis can be extended to Datalog$^{\neg}$ by adding stratified negation, which can be easily realized. As this implementation can be seen as a proof of concept, no special emphasis has been put on optimization, yet. For example, Pregel combiners (see Section 3.2) can be used in further work to reduce the messaging overhead caused by some redundant messages. In addition, messages that are not able to derive new knowledge (e.g. `p(X):- p(X).`) need not to be considered at all.

List of Figures

Appendix

Appendix A:

pdop.PDoP.java

```java
package pdop;

import java.io.IOException;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;

import org.apache.giraph.graph.EdgeListVertex;
import org.apache.giraph.graph.GiraphJob;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BooleanWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat
    ;
import org.apache.hadoop.mapreduce.lib.output.
    FileOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.mahout.math.Arrays;
import org.deri.iris.api.basics.IAtom;
import org.deri.iris.api.basics.IPredicate;
import org.deri.iris.api.basics.IRule;
import org.deri.iris.api.basics.ITuple;
import org.deri.iris.api.terms.ITerm;
import org.deri.iris.factory.Factory;
import org.deri.iris.storage.IRelation;

import pdop.giraph.graph.PDoPWorkerContext;
import pdop.giraph.graph.aggregator.AndAggregator;
import pdop.giraph.graph.aggregator.OrAggregator;
import pdop.giraph.graph.aggregator.RoundAggregator;
import pdop.giraph.io.PDoPInputFormat;
import pdop.giraph.io.PDoPOutputFormat;
import pdop.iris.PartialEvaluation;
```

```java
import pdop.iris.RuleFactQuerySplitter;
import pdop.iris.SimplePDoPRelation;
import pdop.iris.filter.Filter;
import pdop.iris.filter.conditions.EqualCondition;
import pdop.iris.filter.conditions.LessEqualCondition;
import pdop.iris.filter.conditions.TrueCondition;

import com.google.common.base.Preconditions;

/*
 * Parallel Datalog on Pregel
 * (Giraph implementation after SimpleShortestPathsVertex example)
 */

public class PDoP extends EdgeListVertex<Text, Text, Text, Text> implements
    Tool {

  @Override
  public void compute(Iterator<Text> msgIterator) throws IOException {

    RoundAggregator roundAggregator = (RoundAggregator) getAggregator("RoundAggregator");
    AndAggregator voteAggregator = (AndAggregator) getAggregator("VoteAggregator");
    OrAggregator lastTimeAggregator = (OrAggregator) getAggregator("LastTimeAggregator");
    OrAggregator terminateAggregator = (OrAggregator) getAggregator("TerminateAggregator");

    if (terminateAggregator.getAggregatedValue().get()) {
      System.out.println("Finished after " + getSuperstep()
          + " Supersteps. In Round "
          + roundAggregator.getAggregatedValue().get());
      voteToHalt();
    } else {
      boolean voteForNextRound = true;

      RuleFactQuerySplitter splitter = new RuleFactQuerySplitter(
          getVertexValue().toString());
      List<IRule> rules = splitter.getRules();
      Map<IPredicate, IRelation> facts = splitter.getFacts();

      if (getSuperstep() == 0) {
        // add timestamps to first facts
        Map<IPredicate, IRelation> factsWtime = new HashMap<IPredicate, IRelation>();
        for (IPredicate pred : facts.keySet()) {
          factsWtime.put(
              Factory.BASIC.createPredicate(
```

```
          pred.getPredicateSymbol() + "_0",
          pred.getArity()), facts.get(pred));
      }
      facts = factsWtime;
      /*
       * all facts look like this now: a_0('a','b'). These are set to
       * VertexValue at the end of compute
       */
    }

    StringBuilder ruleBuilder = new StringBuilder();
    while (msgIterator.hasNext()) {
      ruleBuilder.append(msgIterator.next().toString());
    }

    RuleFactQuerySplitter msgSplitter = new RuleFactQuerySplitter(
        ruleBuilder.toString());

    List<IRule> msgRules = msgSplitter.getRules();
    List<IRule> partialRules = new LinkedList<IRule>();

    Map<IPredicate, IRelation> msgedFacts = new HashMap<IPredicate, IRelation>();
    Map<IPredicate, IRelation> truncatedFacts = Filter.filter(facts,
        new TrueCondition());
    long timeStamp = roundAggregator.getAggregatedValue().get();

    // check msgedRules for real facts and partial Rules
    for (IRule msgRule : msgRules) {
      if (msgRule.getBody().size() == 0) { // add as fact
        IAtom headAtom = msgRule.getHead().get(0).getAtom();
        ITuple headTuple = headAtom.getTuple();
        IPredicate factPred = headAtom.getPredicate();
        IRelation tuplesOfPredicate = truncatedFacts.get(factPred);
        if (tuplesOfPredicate == null
            || !(tuplesOfPredicate.contains(headTuple))) {
          // fact not there: add.
          // otherwise fact is there: do nothing.
          String factPredSymbol = factPred.getPredicateSymbol()
              + "_" + timeStamp;
          IPredicate newFactPred = Factory.BASIC.createPredicate(
              factPredSymbol, headTuple.size());
          IRelation factRel;
          if (msgedFacts.containsKey(newFactPred)) {
            factRel = msgedFacts.get(newFactPred);
          } else {
```

```java
            factRel = new SimplePDoPRelation();
          }
          factRel.add(headTuple);
          msgedFacts.put(newFactPred, factRel);
        }
      } else { // add rule as partialRule
        partialRules.add(msgRule);
      }
    }
    facts.putAll(msgedFacts);

    /*
     * Vertex value is set with these rules and facts. Following only
     * messages are generated with these rules, but no local Rules or
     * facts are changed.
     */

    if (getSuperstep() == 0) {
      // First evaluation of rules
      voteForNextRound = evaluateThis(truncatedFacts, rules);
    } else {
      // if last time was increased use deltafacts and evalutate with
      // all rules!
      if (lastTimeAggregator.getAggregatedValue().get()) {
        Map<IPredicate, IRelation> deltaFacts = Filter.filter(
            facts, new EqualCondition(roundAggregator
                .getAggregatedValue().get() - 1));
        // remove facts with vertex ID not appearing as first
        // argument from deltaFacts! (not yet implemented)
        voteForNextRound = evaluateThis(deltaFacts, rules);
      } else {
        // if last time was not increased use all facts from the
        // previous rounds and evaluate only with partial Rules!
        Map<IPredicate, IRelation> previousFacts = Filter.filter(
            facts, new LessEqualCondition(roundAggregator
                .getAggregatedValue().get() - 1));
        voteForNextRound = evaluateThis(previousFacts, partialRules);
      }
    }
    voteAggregator.aggregate(new BooleanWritable(voteForNextRound));

    // Update vertex value
```

```java
      StringBuilder builder = new StringBuilder();
      for (IRule rule : rules) {
        builder.append(rule);
      }
      for (IPredicate pred : facts.keySet()) {
        for (int i = 0; i < facts.get(pred).size(); i++) {
          builder.append(pred);
          builder.append(facts.get(pred).get(i));
          builder.append(".");
        }
      }
      setVertexValue(new Text(builder.toString()));
    }
  }

  /*
   * evaluates the Rules with the given facts, collects all the messages and
   * eliminates duplicates Sends them out in the end.
   */
  public boolean evaluateThis(Map<IPredicate, IRelation> facts,
      List<IRule> rules) {
    boolean voteForNextRound = false; // only set this to true, when no
                        // message was added
    Map<IRule, Set<ITerm>> cleanedMessages = new HashMap<IRule, Set<ITerm>>();
    Set<IRule> broadcasts = new HashSet<IRule>();
    for (IRule rule : rules) {
      PartialEvaluation eval = new PartialEvaluation(facts, rule);
      Map<IRule, Set<ITerm>> partialRules = eval.getPartialRules();
      for (IRule msg : partialRules.keySet()) {
        if (!cleanedMessages.containsKey(msg)) {
          cleanedMessages.put(msg, new HashSet<ITerm>());
        }
        cleanedMessages.get(msg).addAll(partialRules.get(msg));
      }
      broadcasts.addAll(eval.getBroadcastRules());
    }
    for (IRule msg : cleanedMessages.keySet()) {
      for (ITerm recipient : cleanedMessages.get(msg)) {
        sendMsg(new Text(recipient.getValue().toString()),
            new Text(msg.toString()));
      }
    }
    for (IRule broadcast : broadcasts) {
      sendMsgToAllEdges(new Text(broadcast.toString()));
    }
    if (cleanedMessages.isEmpty() && broadcasts.isEmpty()) {
      voteForNextRound = true; // no messages were sent
```

```java
    }
    return voteForNextRound;
  }

  @Override
  public int run(String[] argArray) throws Exception {
    System.out.println(Arrays.toString(argArray));

    Preconditions.checkArgument(argArray.length >= 3,
        "run:␣Must␣have␣3␣arguments␣<input␣path>␣<output␣path>␣"
            + "<#␣of␣workers>");

    GiraphJob job = new GiraphJob(getConf(), getClass().getName());
    job.setVertexClass(getClass());
    job.setWorkerContextClass(PDoPWorkerContext.class);
    job.setVertexInputFormatClass(PDoPInputFormat.class);
    job.setVertexOutputFormatClass(PDoPOutputFormat.class);
    // hadoop paths!
    FileInputFormat.addInputPath(job, new Path(argArray[0]));
    FileOutputFormat.setOutputPath(job, new Path(argArray[1]));
    job.setWorkerConfiguration(Integer.parseInt(argArray[2]),
        Integer.parseInt(argArray[2]), 100.0f);
    return job.run(true) ? 0 : -1;
  }

  /*
   * Use from commandline with the following: > inputfolder outputfolder
   * number_of_workers
   */
  public static void main(String[] args) throws Exception {
    System.exit(ToolRunner.run(new PDoP(), args));
  }
}
```

Appendix B:
pdop.giraph.graph.PDoPWorkerContext.java

```java
package pdop.giraph.graph;

import org.apache.giraph.graph.WorkerContext;
import org.apache.hadoop.io.BooleanWritable;
import org.apache.hadoop.io.LongWritable;

import pdop.giraph.graph.aggregator.AndAggregator;
```

```java
import pdop.giraph.graph.aggregator.RoundAggregator;
import pdop.giraph.graph.aggregator.OrAggregator;

/**
 * Worker Context needed for Aggregators
 */
public class PDoPWorkerContext extends WorkerContext {

  @Override
  public void preApplication() throws InstantiationException,
      IllegalAccessException {
    registerAggregator("RoundAggregator", RoundAggregator.class);
    registerAggregator("VoteAggregator", AndAggregator.class);
    registerAggregator("LastTimeAggregator", OrAggregator.class);
    registerAggregator("TerminateAggregator", OrAggregator.class);

    RoundAggregator roundAggregator = (RoundAggregator) getAggregator("RoundAggregator");
    AndAggregator voteAggregator = (AndAggregator) getAggregator("VoteAggregator");
    OrAggregator lastTimeAggregator = (OrAggregator) getAggregator("LastTimeAggregator");
    OrAggregator terminateAggregator = (OrAggregator) getAggregator("TerminateAggregator");

    roundAggregator.setAggregatedValue(new LongWritable(0));
    voteAggregator.setAggregatedValue(new BooleanWritable(true));
    lastTimeAggregator.setAggregatedValue(new BooleanWritable(false));
    terminateAggregator.setAggregatedValue(new BooleanWritable(false));
  }

  @Override
  public void postApplication() {
  }

  /*
   * Logic for rounds in preSuperstep, since values in aggregators are not
   * synchronized if used in postSuperstep.
   *
   * Use postSuperstep for logging only!
   */
  @Override
  public void preSuperstep() {

    RoundAggregator roundAggregator = (RoundAggregator)
```

```
        getAggregator("RoundAggregator");
    AndAggregator voteAggregator = (AndAggregator)
        getAggregator("VoteAggregator");
    OrAggregator lastTimeAggregator = (OrAggregator)
        getAggregator("LastTimeAggregator");
    OrAggregator terminateAggregator = (OrAggregator)
        getAggregator("TerminateAggregator");

    if (voteAggregator.getAggregatedValue().get()) {
      roundAggregator.setAggregatedValue(new LongWritable(
          roundAggregator
          .getAggregatedValue().get() + 1));
      if (lastTimeAggregator.getAggregatedValue().get()) {
        terminateAggregator
            .setAggregatedValue(new BooleanWritable(true));
      } else {
        lastTimeAggregator
            .setAggregatedValue(new BooleanWritable(true));
      }
    } else {
      lastTimeAggregator.setAggregatedValue(new
          BooleanWritable(false));
      voteAggregator.setAggregatedValue(new BooleanWritable(
          true));
    }

    useAggregator("RoundAggregator");
    useAggregator("VoteAggregator");
    useAggregator("LastTimeAggregator");
    useAggregator("TerminateAggregator");
  }

  @Override
  public void postSuperstep() {
  }
}
```

Appendix C:
pdop.giraph.graph.aggregator.AndAggregator.java

```
package pdop.giraph.graph.aggregator;

import org.apache.giraph.graph.Aggregator;
import org.apache.hadoop.io.BooleanWritable;

public class AndAggregator implements Aggregator<
    BooleanWritable> {

  boolean voteAggregator;

```

```java
  @Override
  public void aggregate(BooleanWritable value) {
    voteAggregator = voteAggregator && value.get();
  }

  @Override
  public void setAggregatedValue(BooleanWritable value) {
    voteAggregator = value.get();

  }

  @Override
  public BooleanWritable getAggregatedValue() {
    return new BooleanWritable(voteAggregator);
  }

  @Override
  public BooleanWritable createAggregatedValue() {
    return new BooleanWritable();
  }
}
```

Appendix D:
pdop.giraph.graph.aggregator.OrAggregator.java

```java
package pdop.giraph.graph.aggregator;

import org.apache.giraph.graph.Aggregator;
import org.apache.hadoop.io.BooleanWritable;

public class OrAggregator implements Aggregator<
    BooleanWritable> {

  boolean voteAggregator;

  @Override
  public void aggregate(BooleanWritable value) {
    voteAggregator = voteAggregator || value.get();
  }

  @Override
  public void setAggregatedValue(BooleanWritable value) {
    voteAggregator = value.get();

  }

  @Override
  public BooleanWritable getAggregatedValue() {
    return new BooleanWritable(voteAggregator);
  }
```

```java

  @Override
  public BooleanWritable createAggregatedValue() {
    return new BooleanWritable();
  }
}
```

Appendix E:
pdop.giraph.graph.aggregator.RoundAggregator.java

```java
package pdop.giraph.graph.aggregator;

import org.apache.giraph.graph.Aggregator;
import org.apache.hadoop.io.LongWritable;

public class RoundAggregator implements Aggregator<
    LongWritable> {

  long roundcounter;

  /**
   * used as aggregate(newvalue);
   */
  @Override
  public void aggregate(LongWritable value) {
    roundcounter = value.get();
  }

  @Override
  public void setAggregatedValue(LongWritable value) {
    roundcounter = value.get();

  }

  @Override
  public LongWritable getAggregatedValue() {
    return new LongWritable(roundcounter);
  }

  @Override
  public LongWritable createAggregatedValue() {
    return new LongWritable();
  }
}
```

Appendix F:
pdop.giraph.io.PDoPInputFormat.java

```java
package pdop.giraph.io;

import java.io.IOException;

import org.apache.giraph.graph.VertexReader;
import org.apache.giraph.lib.TextVertexInputFormat;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.TaskAttemptContext;

public class PDoPInputFormat extends
    TextVertexInputFormat<Text, Text, Text, Text> {

  @Override
  public VertexReader<Text, Text, Text, Text>
      createVertexReader(
      InputSplit split, TaskAttemptContext context) throws
          IOException {
    return new PDoPReader(
        textInputFormat.createRecordReader(split, context));
  }
}
```

Appendix G:
pdop.giraph.io.PDoPOutputFormat.java

```java
package pdop.giraph.io;

import java.io.IOException;

import org.apache.giraph.graph.VertexWriter;
import org.apache.giraph.lib.TextVertexOutputFormat;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.RecordWriter;
import org.apache.hadoop.mapreduce.TaskAttemptContext;

public class PDoPOutputFormat extends TextVertexOutputFormat
    <Text, Text, Text> {

  @Override
  public VertexWriter<Text, Text, Text> createVertexWriter(
      TaskAttemptContext context) throws IOException,
      InterruptedException {
    RecordWriter<Text, Text> recordWriter = textOutputFormat
        .getRecordWriter(context);
    return new PDoPWriter(recordWriter);
  }
```

```
}
```

Appendix H:
pdop.giraph.io.PDoPReader.java

```
package pdop.giraph.io;

import java.io.IOException;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.Map;

import org.apache.giraph.graph.BasicVertex;
import org.apache.giraph.graph.BspUtils;
import org.apache.giraph.lib.TextVertexInputFormat.TextVertexReader;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.RecordReader;

import pdop.PDoP;

/**
 * for {@link PDoP} should support input files given in the following form:
 * [VertexID \tab rules.facts.]
 */
public class PDoPReader extends TextVertexReader<Text, Text, Text, Text> {

  /**
   * Constructor with the line record reader.
   *
   * @param lineRecordReader
   *            Will read from this line.
   */
  public PDoPReader(RecordReader<LongWritable, Text> lineRecordReader) {
    super(lineRecordReader);
  }

  @Override
  public BasicVertex<Text, Text, Text, Text> getCurrentVertex()
      throws IOException, InterruptedException {

    BasicVertex<Text, Text, Text, Text> vertex = BspUtils
        .<Text, Text, Text, Text> createVertex(getContext()
            .getConfiguration());

```

```java
    String line = getRecordReader().getCurrentValue().toString();
    StringBuilder builder = new StringBuilder(line);
    builder.deleteCharAt(0);
    builder.deleteCharAt(builder.length() - 1);

    String[] lineSegments = builder.toString().split("\t");
    try {
      Text vertexId = new Text(lineSegments[0]);
      Text vertexVal = new Text();
      vertexVal.set(lineSegments[1]);

      Map<Text, Text> vertexEdge = new HashMap<Text, Text>();
      String[] edgeVertices = lineSegments[2].split(",");
      for (String edgeVertex : edgeVertices) {
        vertexEdge.put(new Text(edgeVertex), new Text());
      }
      LinkedList<Text> vertexMsg = null;
      vertex.initialize(vertexId, vertexVal, vertexEdge, vertexMsg);

    } catch (IllegalArgumentException e) {
      throw new IllegalArgumentException(
          "next: Couldn't get vertex from line" + line, e);
    }
    return vertex;
  }

  @Override
  public boolean nextVertex() throws IOException, InterruptedException {
    return getRecordReader().nextKeyValue();
  }
}
```

Appendix I:
pdop.giraph.io.PDoPWriter.java

```java
package pdop.giraph.io;

import java.io.IOException;

import org.apache.giraph.graph.BasicVertex;
import org.apache.giraph.lib.TextVertexOutputFormat.TextVertexWriter;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.RecordWriter;

public class PDoPWriter extends TextVertexWriter<Text, Text,
```

```
    Text> {
  public PDoPWriter(RecordWriter<Text, Text> lineRecordWriter) {
    super(lineRecordWriter);
  }

  /**
   * Output should be in the following form again! [VertexID \tab
   * rules.facts.]
   */
  @Override
  public void writeVertex(BasicVertex<Text, Text, Text, ?> vertex)
      throws IOException, InterruptedException {
    StringBuilder builder = new StringBuilder();
    builder.append("[");
    builder.append(vertex.getVertexId().toString());
    builder.append("\t");
    builder.append(vertex.getVertexValue().toString());
    builder.append("]");

    getRecordWriter().write(new Text(builder.toString()), null);
  }
}
```

Appendix J:
pdop.iris.RuleFactQuerySplitter.java

```
package pdop.iris;

import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;

import org.deri.iris.api.basics.IPredicate;
import org.deri.iris.api.basics.IQuery;
import org.deri.iris.api.basics.IRule;
import org.deri.iris.compiler.Parser;
import org.deri.iris.storage.IRelation;

public class RuleFactQuerySplitter {

  private Map<IPredicate, IRelation> facts = new HashMap<IPredicate, IRelation>();
  private List<IRule> rules = new LinkedList<IRule>();
  private List<IQuery> queries = new LinkedList<IQuery>();

```

```java
  /**
   * The program is split into rules and facts and querries
       in the
   * constructor. Use getters to obtain these afterwards.
   */
  public RuleFactQuerySplitter(String program) {
    try {
      Parser parser = new Parser();
      parser.parse(program);
      facts = parser.getFacts();
      rules = parser.getRules();
      queries = parser.getQueries();
    } catch (Exception e) {
      System.out.println("Parser Error in RuleFactSplitter: 
          " + e);
    }
  }

  public Map<IPredicate, IRelation> getFacts() {
    return facts;
  }

  public List<IRule> getRules() {
    return rules;
  }

  public List<IQuery> getQueries() {
    return queries;
  }
}
```

Appendix K:
pdop.iris.filter.Filter.java

```java
package pdop.iris.filter;

import java.util.HashMap;
import java.util.Map;

import org.deri.iris.api.basics.IPredicate;
import org.deri.iris.factory.Factory;
import org.deri.iris.storage.IRelation;

import pdop.iris.SimplePDoPRelation;
import pdop.iris.filter.conditions.Condition;

public class Filter {
  /**
   * Filters facts according to given roundcounter,
       timestamp of the facts and
```

```java
   * condition. Used to create DeltaFacts, "oldFacts" and
      trunkated Facts.
   * (all of these are trunkated by default)
   *
   * @param roundcounter
   * @param facts
   * @param condition
   * @return filtered facts according to condition (
      trunkated)
   */

  public static Map<IPredicate, IRelation> filter(
      Map<IPredicate, IRelation> facts, Condition c) {

    Map<IPredicate, IRelation> filteredFacts = new HashMap<
        IPredicate, IRelation>();

    for (IPredicate factPred : facts.keySet()) {
      String pred = factPred.getPredicateSymbol();
      String[] predWOTime = pred.split("_");
      filteredFacts.put(
          Factory.BASIC.createPredicate(predWOTime[0],
              factPred.getArity()), new SimplePDoPRelation()
                  );
    }

    for (IPredicate filteredPred : filteredFacts.keySet()) {
      SimplePDoPRelation relAcc = new SimplePDoPRelation();
      for (IPredicate factPred : facts.keySet()) {
        String[] split = factPred.getPredicateSymbol().split
            ("_");
        if (split[0].equals(filteredPred.getPredicateSymbol
            ())) {
          if (c.isSatisfied(Long.valueOf(split[1]))) {
            IRelation factRel = facts.get(factPred);
            for (int i = 0; i < factRel.size(); i++) {
              relAcc.add(factRel.get(i));
            }
          }
        }
      }
      filteredFacts.put(filteredPred, relAcc);
    }
    return filteredFacts;
  }
}
```

Appendix L:
pdop.iris.filter.conditions.Condition.java

```java
package pdop.iris.filter.conditions;

public interface Condition {

  public boolean isSatisfied(long round);

}
```

Appendix M:
pdop.iris.filter.conditions.EqualCondition.java

```java
package pdop.iris.filter.conditions;

public class EqualCondition implements Condition {

  long value;

  public EqualCondition(long v) {
    value = v;
  }

  public boolean isSatisfied(long round) {
    return round == value;
  }
}
```

Appendix N:
pdop.iris.filter.conditions.LessEqualCondition.java

```java
package pdop.iris.filter.conditions;

public class LessEqualCondition implements Condition {

  long value;

  public LessEqualCondition(long v) {
    value = v;
  }

  public boolean isSatisfied(long round) {
    return round <= value;
  }
}
```

Appendix O:
pdop.iris.filter.conditions.TrueCondition.java

```java
package pdop.iris.filter.conditions;

public class TrueCondition implements Condition {
  public boolean isSatisfied(long round) {
    return true;
  }
}
```

Appendix P:
pdop.iris.PartialEvaluation.java

```java
package pdop.iris;

import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;

import org.deri.iris.api.basics.IAtom;
import org.deri.iris.api.basics.ILiteral;
import org.deri.iris.api.basics.IPredicate;
import org.deri.iris.api.basics.IRule;
import org.deri.iris.api.basics.ITuple;
import org.deri.iris.api.terms.ITerm;
import org.deri.iris.api.terms.IVariable;
import org.deri.iris.factory.Factory;
import org.deri.iris.storage.IRelation;

public class PartialEvaluation {

  private Map<IRule, Set<ITerm>> partialRules;

  // partial Rules with rule and recipients
  public Map<IRule, Set<ITerm>> getPartialRules() {
    return partialRules;
  }

  private void addPartialRule(IRule rule, Set<ITerm>
      recipients) {
    partialRules.put(rule, recipients);
  }

  // partial Rule for broadcasting
  private Set<IRule> broadcastRules;

```

```java
public Set<IRule> getBroadcastRules() {
  return broadcastRules;
}

private void addBroadcastRule(IRule rule) {
  broadcastRules.add(rule);
}

/**
 * evalutate rule with given facts. (Partial evaluation)
 *
 * @param facts
 * @param rule
 *            obtain partial rules with getter!
 */
public PartialEvaluation(Map<IPredicate, IRelation> facts,
     IRule rule) {
  partialRules = new HashMap<IRule, Set<ITerm>>();
  broadcastRules = new HashSet<IRule>();
  for (ILiteral literal : rule.getBody()) { // for every
       Literal in the
                        // body
    IRelation values = facts.get(literal.getAtom().
        getPredicate()); // get
                                    // facts
                                    // for
                                    // that
                                    // exact
                                    // literal
    if (values != null) { // if empty stop
      for (int i = 0; i < values.size(); i++) {
        IRule interRule = changeRule(values.get(i),
            literal, rule);
        if (interRule != null) {
          interRule = checkRule(facts, interRule);
          if (interRule != null) {
            if (!rule.equals(interRule)) {
              Set<ITerm> recipients =
                  getRecipientsFromRule(interRule);
              if (recipients.isEmpty()) {
                addBroadcastRule(interRule);
              } else {
                addPartialRule(interRule, recipients);
              }
            }
          }
        }
      }
    }
  }
}

public static Set<ITerm> getRecipientsFromRule(IRule rule)
     {
```

```java
    Set<ITerm> recipients = new HashSet<ITerm>();
    if (rule.getBody().isEmpty()) {
      recipients.addAll(checkSegmentForRecipients(rule.getHead()));
    } else {
      recipients.addAll(checkSegmentForRecipients(rule.getBody()));
    }
    return recipients;
  }

  public static Set<ITerm> checkSegmentForRecipients(List<ILiteral> segment) {
    Set<ITerm> recipients = new HashSet<ITerm>();
    for (ILiteral literal : segment) {
      for (ITerm term : literal.getAtom().getTuple()) {
        if (!(term instanceof IVariable)) {
          recipients.add(term);
        }
      }
    }
    return recipients;
  }

  private static IRule checkRule(Map<IPredicate, IRelation> facts, IRule rule) {
    List<ILiteral> changedBody = new LinkedList<ILiteral>();
    // check rules for facts from and remove parts that are true( aka
    // represented in the facts)
    for (int i = 0; i < rule.getBody().size(); i++) {
      IAtom atomToCheck = rule.getBody().get(i).getAtom();
      boolean builtIn = isBuiltIn(atomToCheck.getPredicate());
      if (builtIn) {
        // check for Build in Predicates: so far implemented ::
        // NOT_EQUAL
        if (checkBuiltIn(changedBody, atomToCheck) == true) {
          return null;
        }
      } else { // atomToCheck is no built-in predicate, continue checking
          // facts
        IRelation edb = facts.get(atomToCheck.getPredicate());
        if (edb != null && edb.size() != 0) { // entries in edb, check!
          boolean anyEDBTrue = false;
          for (int j = 0; j < edb.size(); j++) {
            boolean edbTrue = true;
            for (int k = 0; k < atomToCheck.getTuple().size(); k++) {
```

```
              if (atomToCheck
                  .getTuple()
                  .get(k)
                  .getValue()
                  .toString()
                  .equals(edb.get(j).get(k).getValue()
                      .toString())) {
                edbTrue &= true;
              } else {
                edbTrue &= false;
                break;
              }
            }
            if (edbTrue) {
              anyEDBTrue = true;// true => can be removed
              break;
            }
          }
          if (!anyEDBTrue) { // no EDB value verifies, add for others
                      // Vertices to Check.
            changedBody.add(Factory.BASIC.createLiteral(true,
                atomToCheck));
          }

        } else { // no entries!
          if (atomToCheck.getTuple().getAllVariables().size() == 0) {
            // p(d):-t(a). still possible! -> check constant =|= ID -> broadcast (not yet implemented), else return null
            return null; // filled false.
          } else { // add partial for others to check.
            changedBody.add(Factory.BASIC.createLiteral(true,
                atomToCheck));
          }
        }
      }
    }
    return Factory.BASIC.createRule(rule.getHead(), changedBody);
  }

  private static IRule changeRule(ITuple fact, ILiteral literal, IRule rule) {
    Map<IVariable, ITerm> varMap = new HashMap<IVariable, ITerm>(); // map
                                    // for
                                    // (Variable,Constant) for
                                    // replacement
    // add every Variable to the Map and save the string
```

```
        that matches given
    // the facts.
    for (int i = 0; i < literal.getAtom().getTuple().size();
        i++) { // for
                                      // every
                                      // Element
                                      // in
                                      // the
                                      // Tuple
      ITerm term = literal.getAtom().getTuple().get(i);
      if (term instanceof IVariable) { // check if it is a
          IVariable
        varMap.put((IVariable) term, fact.get(i)); // and
            add it to the
                                  // map
      } else { // term is a constant
        if (!term.equals(fact.get(i))) {
          return null; // the constant in the rule is not
              equal to the
                   // constant in fact, abort!
        }
      }
    }// Map finished, change rule now!
    List<ILiteral> changedHead = changeSegment(varMap, rule.
        getHead());
    List<ILiteral> changedBody = changeSegment(varMap, rule.
        getBody());
    return Factory.BASIC.createRule(changedHead, changedBody
        );
  }

  /**
   * Replaces the variables in literals with the appropriate
       constants from
   * the varMap
   *
   * @param varMap
   * @param literals
   * @return new literals with substitutions
   */
  private static List<ILiteral> changeSegment(Map<IVariable,
      ITerm> varMap,
      List<ILiteral> literals) {
    List<ILiteral> newLiterals = new LinkedList<ILiteral>();

    for (int i = 0; i < literals.size(); i++) {
      ILiteral literal = literals.get(i);
      ITuple tuple = literal.getAtom().getTuple();
      List<ITerm> newTerms = new LinkedList<ITerm>();
      for (int j = 0; j < tuple.size(); j++) {
        ITerm newTerm = varMap.get(tuple.get(j));
        if (newTerm == null) {
          newTerms.add(tuple.get(j));
        } else {
```

```java
          newTerms.add(newTerm);
        }
      }
      ILiteral newLiteral = Factory.BASIC.createLiteral(true, literal
          .getAtom().getPredicate(), Factory.BASIC
          .createTuple(newTerms));
      newLiterals.add(newLiteral);
    }
    return newLiterals;
  }

  /**
   * checks if atomToCheck is a verifiable with a built-in function
   *
   * @param changedBody
   * @param atomToCheck
   * @return true, if verified; false, if others need to verify it.
   */
  private static boolean checkBuiltIn(List<ILiteral> changedBody,
      IAtom atomToCheck) {

    if (atomToCheck.getPredicate().getPredicateSymbol().equals("NOT_EQUAL")) {
      if (atomToCheck
          .getTuple()
          .get(0)
          .getValue()
          .toString()
          .equals(atomToCheck.getTuple().get(1).getValue().toString())) {
        return true; // both equal, drop.
      } else {
        if (atomToCheck.getTuple().getAllVariables().size() != 0) {
          // still variables to be filled => add
          changedBody.add(Factory.BASIC.createLiteral(true,
              atomToCheck));
        }
      }
    }
    return false;
  }

  private static boolean isBuiltIn(IPredicate pred) {
    return pred.getPredicateSymbol().equals("NOT_EQUAL");
  }
}
```

Appendix Q:
pdop.iris.SimplePDoPRelation.java

```java
package pdop.iris;

import java.util.List;

import org.deri.iris.api.basics.ITuple;
import org.deri.iris.storage.IRelation;
import org.deri.iris.utils.UniqueList;

public class SimplePDoPRelation implements IRelation {

  /*
   * SimpleRelation from Iris with explicit public constructor
   */

  /**
   * Constructor. For performance reasons where the user of the class can
   * enforce uniqueness (or does not require it), uniqueness enforcement can
   * be turned off.
   *
   * @param forceUniqueness
   *            true, if this object should enforce uniqueness.
   */
  public SimplePDoPRelation() {
    mTuples = new UniqueList<ITuple>();
  }

  public boolean add(ITuple tuple) {
    assert mTuples.isEmpty() || (mTuples.get(0).size() == tuple.size());

    return mTuples.add(tuple);
  }

  public boolean addAll(IRelation relation) {
    boolean added = false;

    for (int i = 0; i < relation.size(); ++i)
      if (add(relation.get(i)))
        added = true;

    return added;
  }

  public ITuple get(int index) {
    return mTuples.get(index);
  }
```

```
  public int size() {
    return mTuples.size();
  }

  public boolean contains(ITuple tuple) {
    return mTuples.contains(tuple);
  }

  @Override
  public String toString() {
    return mTuples.toString();
  }

  /** The array list (or unique list) of tuples. */
  private final List<ITuple> mTuples;

}
```

Bibliography

[1] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[2] Inc LogicBlox. Datalog for Enterprise Applications: from Industrial Applications to Research. `http://www.logicblox.com/presentations/arefdatalog20.pdf`, March 2010. accessed on: September 20, 2012.

[3] Daniel Ritter and Till Westmann. Business Network Reconstruction Using Datalog. In Barceló and Pichler [22], pages 148–152.

[4] Marianne Shaw, Paraschos Koutris, Bill Howe, and Dan Suciu. Optimizing Large-Scale Semi-Naïve Datalog Evaluation in Hadoop. In Barceló and Pichler [22], pages 165–176.

[5] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[6] Yingyi Bu and Bill Howe. HaLoop. `http://code.google.com/p/haloop/`, 2012. accessed on: September 24, 2012.

[7] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[8] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[9] Apache Software Foundation. Apache Incubator Giraph. `http://incubator.apache.org/giraph/`, February 2012. accessed on: September 24, 2012.

[10] Peter Alvaro, William Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell C Sears. Dedalus: Datalog in Time and Space. Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley, Dec 2009.

[11] François Bry, Norbert Eisinger, Thomas Eiter, Tim Furche, Georg Gottlob, Clemens Ley, Benedikt Linse, Reinhard Pichler, and Fang Wei. Foundations of Rule-Based Query Answering. In *Reasoning Web, Third International Summer School 2007*, volume 4636 of *LNCS*. Springer-Verlag, 2007.

[12] Ravel. GoldenOrb. `http://goldenorbos.org/`, 2012. accessed on: October 15, 2012.

[13] Jimmy Lin and Michael Schatz. Design patterns for efficient graph algorithms in MapReduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, pages 78–85, New York, NY, USA, 2010. ACM.

[14] Apache Software Foundation. Apache Hadoop. `http://hadoop.apache.org/`, September 2012. accessed on: October 15, 2012.

[15] Apache Software Foundation. Apache ZooKeeper. `http://zookeeper.apache.org/`, September 2012. accessed on: October 15, 2012.

[16] Apache Software Foundation. Apache Incubator Giraph Wiki. `https://cwiki.apache.org/confluence/display/GIRAPH/Index`, February 2012. accessed on: September 24, 2012.

[17] Apache Software Foundation. Apache Incubator Giraph API. `http://incubator.apache.org/giraph/apidocs/index.html`, February 2012. accessed on: September 24, 2012.

[18] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997.

[19] Barry Bishop, Florian Fischer, Uwe Keller, Nathalie Steinmetz, Christoph 'Gigi' Fuchs, Matthias Pressnig, Darko Anicic, Cristina Feier, Holger Lausen, and Richard Pöttler. IRIS Reasoner. `http://iris-reasoner.org/`, February 2010. accessed on: October 15, 2012.

[20] Avanade® Research Insights. Global Survey: Is Big Data Producing Big Returns? `http://www.avanade.com/Documents/Research%20and%20Insights/avanade-big-data-executive-summary-2012.pdf`, June 2012. accessed on: October 11, 2012.

[21] Yingyi Bu, Vinayak R. Borkar, Michael J. Carey, Joshua Rosen, Neoklis Polyzotis, Tyson Condie, Markus Weimer, and Raghu Ramakrishnan. Scaling Datalog for Machine Learning on Big Data. *CoRR*, abs/1203.0160, 2012.

[22] Pablo Barceló and Reinhard Pichler, editors. *Datalog in Academia and Industry - Second International Workshop, Datalog 2.0, Vienna, Austria, September 11-13, 2012. Proceedings*, volume 7494 of *Lecture Notes in Computer Science*. Springer, 2012.