Processing Regular Path Queries on Giraph

Maurizio Nolé
DIMIE - Università della Basilicata
Via dell'Ateneo Lucano 10
Potenza,Italy
mnole@gmail.com

Carlo Sartiani
DIMIE - Università della Basilicata
Via dell'Ateneo Lucano 10
Potenza,Italy
sartiani@gmail.com

ABSTRACT

In the last few years social networks have reached an ubiquitous diffusion. Facebook, LinkedIn, and Twitter now have billions of users, that daily interact together and establish new connections. Users and interactions among them can be naturally represented as *data graphs*, whose vertices denote users and whose edges are labelled with information about the different interactions.

In this paper we sketch a novel approach for processing regular path queries on very large graphs. Our approach exploits Brzozowski's derivation of regular expressions to allow for a vertex-centric, message-passing-based evaluation of path queries on top of Apache Giraph.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

General Terms

Algorithms, Performance

Keywords

Graph Query Processing, Distributed Computing

1. INTRODUCTION

In the last few years social networks have reached an ubiquitous diffusion. Facebook, LinkedIn, and Twitter now have billions of users, that daily interact together and establish new connections. Users and their interactions can be naturally represented as *data graphs*, whose vertices denote users and whose edges are labelled with information about the different interactions. The problem of managing, querying, and mining graph databases, hence, is becoming more and more important; similar problems emerge in many different application fields where data have a graph structure, e.g., traffic analysis, crime detection, the Semantic Web.

Data graphs have attracted a significant research interest since the mid 90's. In particular, several query languages based on regular expressions [8, 9, 6] have been proposed and a few data graph query processors have been designed [5, 10].

Today data graphs can be very large and easily exceed 100 millions vertices. To manage this kind of graphs, Google designed a novel class of graph processing systems, based on the Bulk Synchronous Parallel Model by Valiant [11], where graphs are automatically partitioned across the nodes of a computing cluster, and algorithms are expressed through *vertex-centric* functions, i.e., functions that are executed by each vertex in the graph. Systems in this class (e.g., Google Pregel [7] and Apache Giraph [1]) exhibit very good scalability properties for many graph algorithms, but have not been designed for querying graphs.

Our Contribution. In this paper we sketch a novel approach for evaluating path queries on very large graphs. Our approach exploits Brzozowski's derivation of regular expressions [4] to allow for a vertex-centric, message-passing-based evaluation of path queries on top of Giraph (see Section 3.1). In particular, when each vertex receives a query q, it derives q according to the symbols labelling outgoing edges and propagates the derivative of q to its neighbours. To avoid network flooding, only outgoing edges labelled with symbols in the *first set* of q are considered.[1]

2. DATA MODEL AND QUERY LANGUAGE

2.1 Data Model

Following [6], we model a *data graph* as an edge-labelled graph, as shown below.

Definition 2.1 (Data Graph) *Given a finite alphabet Σ and a (possibly) infinite value domain $\mathcal{D}$, a data graph G over Σ and $\mathcal{D}$ is a triple $G = (V, E, \rho)$, where:*

- *V is a finite set of vertices;*
- *$E \subseteq V \times \Sigma \times V$ is a set of labelled, directed edges (v_i, a, v_j);*
- *$\rho : V \to \mathcal{D}$ is a mapping from vertices to values.*

Given a vertex v, we will indicate with $in(v)$ and $out(v)$ the set of incoming and outgoing edges, respectively. More formally:

[1]The first set of a regular expression r is the set of symbols that appear in the first position of words matching r.

(c) 2014, Copyright is with the authors. Published in the Workshop Proceedings of the EDBT/ICDT 2014 Joint Conference (March 28, 2014, Athens, Greece) on CEUR-WS.org (ISSN 1613-0073). Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

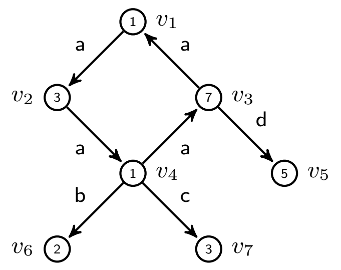

Figure 1: A graph.

- $in(v) = \{(v', a, v) \in E \mid v' \in V \wedge a \in \Sigma\}$;
- $out(v) = \{(v, a, v') \in E \mid v' \in V \wedge a \in \Sigma\}$.

We will also indicate with $id(v)$ the unique identifier of v. We assume that sequences of outgoing (incoming) edges of a vertex are unordered, as it is often the case in graph databases.

2.2 Query Language

Many query languages for graph data have been presented in the past [8, 9, 6]. We focus our attention here on GXPath, a language recently proposed by Libkin et al. in [6]. GXPath is based on the idea of using regular expressions to specify patterns that must be matched by paths in the input graph. Given a query q, the result of its evaluation over a graph G is always a set of vertex pairs (v, v') such that v and v' are connected by a path p in G matching the query q. GXPath extends other path languages like RPQs or NREs with the introduction of the *complement* operator, data tests on the values stored into vertices, as well as counters, which generalize the Kleene star, and it can be considered as an adaptation of XPath [2] to data graphs.

Among the various fragments of GXPath, we focus here on a navigational, path-positive fragment with counting, but without complement, intersection, and nested conditions, as described by the following grammar.

$$\alpha \quad ::= \quad \epsilon \mid \_ \mid a \mid \alpha + \alpha \mid \alpha \cdot \alpha \mid \alpha^{m,n}$$

Given a graph $G = (V, E, \rho)$, the semantics of our fragment of GXPath can be defined as follows.

$$\begin{array}{rcl}
[\![\epsilon]\!]_G & = & \{(u,u) \mid u \in V\} \\
[\![\_]\!]_G & = & \{(u,v) \mid \exists a \in \Sigma.(u,a,v) \in E\} \\
[\![a]\!]_G & = & \{(u,v) \mid (u,a,v) \in E\} \\
[\![\alpha_1 + \alpha_2]\!]_G & = & [\![\alpha_1]\!]_G \cup [\![\alpha_2]\!]_G \\
[\![\alpha_1 \cdot \alpha_2]\!]_G & = & [\![\alpha_1]\!]_G \circ [\![\alpha_2]\!]_G \\
[\![\alpha^{m,n}]\!]_G & = & \cup_{i=m}^{n} [\![\alpha]\!]_G^i
\end{array}$$

where $\circ$ is the symbol for the concatenation of binary relations and R^i denotes the concatenation of R with itself i times. Here, ϵ denotes the empty word, $\_$ matches any symbol, $\alpha_1 \cdot \alpha_2$ and $\alpha_1 + \alpha_2$ are the standard concatenation and union operators, and $\alpha^{m,n}$ denotes the repetition of α from m to n times ($m \in \mathbb{N}$, $n \in \mathbb{N} \cup \{*\}$, $m \leqslant n$).

Example 2.2 Consider the graph depicted in Figure 1.

Consider now the following query: $a^{2,3} \cdot (b+d)$. This query returns all vertex pairs (u, v) connected by the following paths: *aab*, *aaab*, *aad*, *aaad*. The result of this query is $\{(v_1, v_6), (v_1, v_5), (v_2, v_5), (v_5, v_6)\}$.

■

3. PROCESSING PATH QUERIES

3.1 Brzozowski's derivatives

Brzozowski's derivatives [3] represent an alternative way to check if a word w belongs to the language generated by a given regular expression r. The idea is to iterate over w and to rewrite r according to the last read symbol, hence computing a *derivative*; if the derivative generated after the last symbol of w has been read is ϵ, then the check is successful.

Brzozowski's derivatives can be extended to regular path expressions on data graphs in the following way.

Definition 3.1 (Derivative) *α' is a derivative of α in a graph $G = (V, E)$ according to $a \in \sigma$ iff $\bigcup_{(u,a,v)\in E}\{(u,v)\} \circ [\![\alpha']\!]_G = [\![\alpha]\!]_G$.*

Definition 3.2 (Empty expression) *$\emptyset$ denotes the* empty *regular expression, that is, $[\![\emptyset]\!]_G =_{def} \emptyset$*

Proposition 3.3 (Empty expression properties) *$\emptyset$ satisfies the following properties:*

$$\begin{array}{ccccc}
\alpha + \emptyset & = & \emptyset + \alpha & = & \alpha \\
\alpha \cdot \emptyset & = & \emptyset \cdot \alpha & = & \emptyset
\end{array}$$

Notation 3.4 (m^-, $* - 1$) *In the following definitions, we use m^- to denote $max(m-1, 0)$, and assume that $* - 1 = *$.*

Definition 3.5 *$N(\alpha)$ is a predicate on regular expressions, defined as follows:*

$$\begin{array}{lcllcl}
N(\emptyset) & = & false & N(\epsilon) & = & true \\
N(a) & = & false & N(\_) & = & false \\
N(\alpha_1 + \alpha_2) & = & N(\alpha_1) \vee N(\alpha_2) & & & \\
N(\alpha_1 \cdot \alpha_2) & = & N(\alpha_1) \wedge N(\alpha_2) & & & \\
N(\alpha^{m,n}) & = & N(\alpha) & & &
\end{array}$$

Definition 3.6 *$first(\alpha)$ is a function on regular expressions, defined as follows:*

$$\begin{array}{lcl}
first(\emptyset) & = & \emptyset \\
first(\epsilon) & = & \emptyset \\
first(a) & = & \{a\} \\
first(\_) & = & \Sigma \\
first(\alpha_1 + \alpha_2) & = & first(\alpha_1) \cup first(\alpha_2) \\
first(\alpha_1 \cdot \alpha_2) & = & \begin{cases} first(\alpha_1) \cup first(\alpha_2) & \textit{if } N(\alpha_1) \\ first(\alpha_1) & \textit{otherwise} \end{cases} \\
first(\alpha^{m,n}) & = & first(\alpha)
\end{array}$$

Definition 3.7 (Derivation) *$d_a(\alpha)$, where α is a regular path query and $a \in \Sigma$, is defined in Figure 2.*

It is easy to see that $d_a(\alpha)$ is a derivative of α according to a in G.

3.2 Evaluation Algorithm

Brzozowski's derivatives can be used to implement path query processing on top of Giraph (or similar systems). In Giraph a computation consists of several *supersteps*, representing global synchronization points. Each superstep comprises a *master* computation, performed by a special node ("the master") at the beginning of the superstep, and by a

$$
\begin{array}{lll}
d_a(\epsilon) & =_{def} & \emptyset \\
d_a(\emptyset) & =_{def} & \emptyset \\
d_a(b) & =_{def} & \begin{cases} \epsilon & \text{if } a = b \text{ or } b = \_ \\ \emptyset & \text{otherwise} \end{cases} \\
d_a(\alpha_1 + \alpha_2) & =_{def} & d_a(\alpha_1) + d_a(\alpha_2) \\
d_a(\alpha_1 \cdot \alpha_2) & =_{def} & \begin{cases} d_a(\alpha_1)\cdot \alpha_2 + d_a(\alpha_2) & \text{if N}(\alpha_1) \\ d_a(\alpha_1)\cdot \alpha_2 & \text{otherwise} \end{cases} \\
d_a(\alpha^{m,n}) & =_{def} & \begin{cases} \emptyset & \text{if } n = 0 \\ d_a(\alpha) \cdot \alpha^{m^-,n-1} & \text{if (not N}(\alpha) \text{ and } n > 0) \text{ or } (m = 0 \text{ and } n = *) \\ d_a(\alpha) \cdot \alpha^{m^-,n-1} \;\; + \;\; d_a(\alpha^{m^-,n-1}) & \text{otherwise} \end{cases}
\end{array}
$$

Figure 2: Derivation function.

vertex-centric computation, where each graph vertex processes incoming messages from other vertices, sends messages to other vertices, and executes a given algorithm. Vertices communicate together through messages, that are delivered at the beginning of the next superstep; the communication between the master and vertices is performed through special data structures called *aggregators*, and is bidirectional. The computation halts when each vertex decides to halt and no more message is flowing in the network.

In our system each input query q is sent to each vertex by the master at the beginning of the first superstep; the master also sends the command `derive`. If q contains a subexpression of the form $\alpha^{m,*}$, the master transforms it into $\alpha^{m,|V|}$

At superstep 0, each vertex v checks for a message by the master; if the message is a pair (q, `derive`), then the vertex is instructed to start the derivation process. If $q = \epsilon$ or N(q), then v sends the pair ($id(v)$, $id(v)$) to the master through the aggregator `result`; the master will push the content of `result` on persistent store at the beginning of the next superstep. If $q \neq \emptyset$ and $q \neq \epsilon$, v starts the derivation process by looking at the symbols labelling outgoing edges. The derivation of q according to a symbol a is performed only if $a \in first(q)$, i.e., a may appear in the first position of a word generated by q. If $d_a(q) \neq \emptyset$, then v sends to the target vertex a message ($id(v)$, $d_a(q)$).

At each superstep $s > 0$, the master checks if the new results have been aggregated in `result` by graph vertices in the previous superstep; in that case, the master moves the result pairs to the persistent store and cleans the aggregator.

At each superstep $s > 0$, each vertex v checks if there are incoming messages from other vertices. If v receives a message $m = (id(v_0), q')$, then it starts analyzing q' to understand if it must be derived; in particular, if $q' = \epsilon$ or N(q'), v adds the pair $(id(v_0), id(v))$ to `result` and, if $q' = \epsilon$, it starts processing the next message. If $q' \neq \epsilon$, v starts deriving q' according to the symbols labelling outgoing edges, provided that they are in $first(q')$. If the derivative is equal to $\emptyset$, no message is sent; otherwise, given an edge (v, a, v'), v sends to v' the message $(id(v_0), d_a(q'))$.

The pseudocode of the algorithms for master and vertex computation is shown in Figures 3 and 4.

```
MasterCompute
   / - - Input: a query $q$
   / - - Input: aggregators result and command
1  if ($superstep$ == 0)
2       if ($q$ == $C[\alpha^{m,*}]$) $q = C[\alpha^{m,|V|}]$
3       $command$.aggregate(($q$, derive))
4  else List $resultList$ = $result$.getAggrValue()
5       if ($resultList \neq \{\}$)
6             add $resultList$ to persistent storage
7             $result$.clean()
```

Figure 3: Master computation.

3.3 Implementation Issues

We implemented the algorithms described in the previous section in a very preliminary research prototype called Vertigo. While implementing Vertigo, we faced several challenges. First of all, Brzozowski's derivation does not behave well on non-deterministic regular expressions, as it may generate exponentially larger derivatives;[2] this is even more evident when the regular expression contains a counting operator. To speed up the derivation process, our derivation algorithm works modulo associativity and commutativity of union. In detail, we memoize the derivation process through a small LRU cache on each Giraph worker, and systematically simplify derivatives through the following rules: $\alpha + \alpha \to \alpha$, $\alpha + \emptyset \to \alpha$, $\alpha \cdot \epsilon \to \alpha$, $\alpha \cdot \emptyset \to \emptyset$.

Second, when working on very large graphs and queries with low selectivity, query results can be quite large. Hence, their transmission to the master through aggregators can be quite expensive. To decrease this overhead, result transmission is performed at the end of each superstep by each worker through a post-superstep computation. Workers perform a preliminary duplicate elimination, while the master just appends result pairs on a file on HDFS; final duplicate elimination is performed when computation halts.

Finally, the most efficient way to evaluate queries of the form $\alpha^{m,*}$ is to evaluate α and, then, to compute the reflexive and transitive closure of $[\![\alpha]\!]_G$. This can be performed in a BSP fashion, but it would be too expensive as it requests to stop the current graph traversal. Therefore, we prefer to drop this technique in favour of a more naive one, where $*$

[2]Intuitively, a regular expression r is non-deterministic (or 1-ambiguous) if there exists at least a word w that can match r in multiple ways.

```
VertexCompute
    / - - Input: aggregators result and command
    / - - Input: current vertex $v$
if ($superstep == 0$)
    MasterMessage $m = command$.getAggrValue()
    Query $q = m$.query
    if ($q == \emptyset$) HALT()
    elseif ($q == \epsilon$)
        $result$.aggregate($(id(v), id(v))$)
        HALT()
    elseif (N($q$)) $result$.aggregate($(id(v), id(v))$)
    for each $(v, a, u) \in E$ : ($a \in first(q)$)
        Query $q' = d_a(q)$
        if ($q' \neq \emptyset$)
            if ($q' == q'_1 + q'_2$)
                if ($q'_1 \neq \emptyset$) SEND($u$, $(id(v), q'_1)$)
                if ($q'_2 \neq \emptyset$) SEND($u$, $(id(v), q'_2)$)
            else SEND($u$, $(id(v), q')$)
                HALT()
elseif ($superstep > 0$)
    for each message $m = (id(v_0), q)$
        if ($q == \epsilon$)
            $result$.aggregate($(id(v_0), id(v))$)
            skip to next message
        elseif (N($q$)) $result$.aggregate($(id(v_0), id(v))$)
        for each $(v, a, u) \in E$ : ($a \in first(q)$)
            Query $q' = d_a(q)$
            if ($q' \neq \emptyset$)
                if ($q' == q'_1 + q'_2$)
                    if ($q'_1 \neq \emptyset$)
                        SEND($u$, $(id(v_0), q'_1)$)
                    if ($q'_2 \neq \emptyset$)
                        SEND($u$, $(id(v_0), q'_2)$)
            else SEND($u$, $(id(v_0), q')$)
                HALT()
```

Figure 4: Vertex computation.

is replaced by the number of vertices in the input graph.

4. CONCLUSIONS AND FUTURE WORK

In this paper we sketched a novel algorithm for evaluating regular path queries on data graphs. This algorithm exploits Brzozowski's derivation and can be used in Giraph and any other similar system. We developed a very preliminary prototype implementation of our algorithm; in early tests on a single commodity machine this prototype easily processed queries on 200-million-edge graphs. We are currently testing our implementation on Pivotal's AWB cluster.

In a very near future we plan to extend our algorithm to support a larger fragment of GXPath comprising backward navigation, branching, and intersection.

5. ACKNOWLEDGMENTS

(Portions of) the research in this paper use results obtained from the Greenplum Analytics Workbench, made available by Greenplum, a division of GoPivotal Corporation.

6. REFERENCES

[1] Apache giraph, 2013. `http://http://giraph.apache.org`.

[2] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML Path Language (XPath) 2.0 (Second Edition). Technical report, World Wide Web Consortium, 2010. W3C Recommendation.

[3] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Inf. Comput.*, 142(2):182–206, 1998.

[4] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[5] A. Koschmieder and U. Leser. Regular path queries on large graphs. In A. Ailamaki and S. Bowers, editors, *SSDBM*, volume 7338 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2012.

[6] L. Libkin, W. Martens, and D. Vrgoc. Querying graph databases with XPath. In W.-C. Tan, G. Guerrini, B. Catania, and A. Gounaris, editors, *ICDT*, pages 129–140. ACM, 2013.

[7] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In A. K. Elmagarmid and D. Agrawal, editors, *SIGMOD Conference*, pages 135–146. ACM, 2010.

[8] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6):1235–1258, 1995.

[9] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. *J. Web Sem.*, 8(4):255–270, 2010.

[10] M. Sarwat, S. Elnikety, Y. He, and G. Kliot. Horton: Online query execution engine for large distributed graphs. In A. Kementsietsidis and M. A. V. Salles, editors, *ICDE*, pages 1289–1292. IEEE Computer Society, 2012.

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