A Distributed Implementation of GXPath

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 there has been an increasing number of application fields, like the Semantic Web, social networks, bioinformatics, astronomical databases, etc., where large graph datasets are analyzed, queried, and, more generally, manipulated. Graphs are usually queried by specifying reachability patterns through regular path expressions; this leads to the need for efficient and scalable tools for processing regular path queries on large graphs.

In this work we present a distributed implementation of GXPath and show that this implementation, built on top of Hadoop MapReduce, can scale linearly with the number of vertices and/or edges.

1. INTRODUCTION

In the last few years there has been a growing interest in querying and, more generally, analyzing huge graph datasets. These datasets arise in many real-life contexts, such as social and instant messaging networks, biology, crime detection and prevention, etc. As an example, social networks like Facebook and Twitter attract everyday more and more users from all over the world. The diffusion of these services is so wide that Facebook reported that in the second quarter of 2015 its network comprises more than 1.4 billion active users [3], while Twitter has almost 288 million users [4].

Social network interactions can be naturally modeled by using directed graphs, whose vertices denote users and edges describe user-to-user interactions: indeed, Facebook and Twitter already use graphs and graph tools to model and analyze their networks.

Given the wide diffusion of graph datasets, analyzing these graphs is becoming more and more important. This task, however, is very challenging, partly because of the flexibility of graphs and partly because of their ever increasing size. In the following example, we describe a paradigmatic application of graph analysis related to crime detection and prevention.

Example 1.1 Consider a government intelligence agency that must seek for terrorist cells. Unlike what happens for traditional criminal organizations, terrorists usually exploit Internet services by exchanging messages or uploading videos to make proselytizing campaigns, and they often leave evidences of these interactions. Security and intelligence agencies, hence, try to find these evidences, by analyzing the large amount of information exchanged on the Internet and, in particular, on social networks like Facebook or Twitter.

The structure of terrorist cells can be usually translated into well-determined reachability patterns in social network graphs. A potential way to identify these cells, therefore, is to adopt an analysis technique that inspects social network graphs and looks for those vertices that are connected by paths matching these reachability patterns; the results of these queries can be further refined by linking graph vertices to other databases, e.g., criminal records, and by manually inspecting collected results. Hence, while single users of a social network may not give rise to the suspicion of belonging to a terrorist cell, the links among them could take down the whole organization.

In Figure 1 we show the connections between the terrorists that participated to the 9-11 attacks in Washington and New York, and other people involved with terrorist activities [15]. In this figure green lines are used to denote direct connections among two Al Qaeda original suspects, while grey lines show indirect connections with other potential suspects. As it can be observed, the social network topology reveals *Mohammed Atta* emerging as the local leader, and all hijackers are connected to the two original Al Qaeda suspects by single-step or two-step paths. ■

Graph reachability patterns are usually expressed by means of Regular Path Queries (RPQs) [18]. A regular path query q is a regular expression whose semantics comprises all the pairs of vertices in the input graph that are connected by a path labeled with a string matching q. Basic RPQs contain the operators of plain regular expressions only, and they have been extended in several ways obtaining CRPQs [9], 2RPQs [12, 8, 10], 2CRPQs [8, 12], and NREs [19]. The most powerful extension of RPQ is GXPath [17], that can be considered an adaptation of XPath to graphs.

Unfortunately, traditional relational database systems are not well-suited for this kind of queries and do not scale well on large graphs. On the other hand, most existing graph database systems are inherently centralized (e.g., Neo4j [2]), which strongly limits the size of the graphs that can be

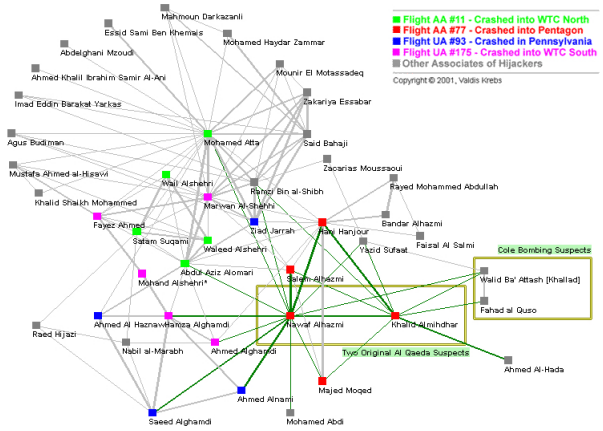


Figure 1: Connections among the terrorists of 9-11 attacks in New York and Washington.

queried, and support only small fragments of regular path query languages; distributed graph database systems like InfiniteGraph [1], instead, can deal with bigger graphs, but they still impose severe restrictions on the class of queries that can be evaluated.

Our Contribution. In this paper we describe a system for processing GXPath queries on large data graphs. In this system, built on top Hadoop MapReduce, a query is compiled into an acyclic graph of MapReduce jobs, similar in spirit to a database query plan. Intermediate results are stored in a compressed format to decrease the I/O overhead implied by MapReduce. As proved by several experiments, our system scales linearly with the number of vertices and/or edges.

Paper Outline. The rest of the paper is organized as follows. In Section 2 we describe the data model and the query language being used. In Section 3, then, we sketch our query processing technique. In Section 4, next, we present an extensive experimental evaluation of our system. In Sections 5 and 6, finally, we discuss some related works and draw our conclusions.

2. GRAPHS

2.1 Data Model

To describe a graph, we consider a model in which edges are labeled by symbols from a finite alphabet Σ and vertices can contain data values from a countably infinite set D. For the sake of simplicity, we assume that a vertex can contain a single data value.

Definition 2.1 *A* data graph *(over Σ and D) is a triple $G = \langle V, E, \rho \rangle$ where:*

- *V is a finite set of vertices;*
- *$E \subseteq V \times \Sigma \times V$ is a set of labeled edges; and*
- *$\rho : V \to D$ is a function that assigns data values to vertices.*

When we deal with purely navigational queries, we refer to a graph as $G = \langle V, E \rangle$ omitting ρ.

A path from a vertex v_1 to a vertex v_n in a graph is a sequence $\pi = v_1 a_1 v_2 \dots v_{n-1} a_{n-1} v_n$ such that each (v_i, a_i, v_{i+1}) for $i < n$ is an edge in E. We use $\lambda(\pi)$ to denote the ordered concatenation of the labels of π.

2.2 Query Language

Most navigational formalisms for querying data graphs are based on RPQs and their extensions. An *RPQ* is an expression of the form $x \xrightarrow{L} y$, where L is a regular language over Σ (typically represented by a regular expression or a NFA). Given a Σ-labeled graph $G = \langle V, E \rangle$, the answer to an RPQ as above is the set of pairs of vertices (v, v') such that there is a path π from v to v' with $\lambda(\pi) \in L$.

GXPath, proposed by Libkin et al. in [17], extends other languages like RPQs or NREs [19] with the introduction of the *complement* operator, data tests on the value stored into vertices, as well as counters, which generalize the Kleene star.

In this paper we focus our attention on the navigational fragment of GXPath, without value tests, as described by the following grammar:

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

Here, ϵ denotes the empty word, $\_$ is a wildcard matching any symbol, $\alpha_1 \cdot \alpha_2$ and $\alpha_1 + \alpha_2$ are the standard concatenation and union operators. $\alpha^{m,n}$ denotes the repetition of α from m to n times ($m \leq n, m \in \mathbb{N}, n \in \mathbb{N} \cup \{*\}$); a^-, finally, denotes the backward navigation, $\overline{\alpha}$ is the complement of α, and $[\alpha]$ is a nested condition. The Kleene star can be represented as $\alpha^{0,*}$.

Given a data graph $G = \langle V, E, \rho \rangle$, the semantics $[\![\alpha]\!]_G$ of a query α on G is a set of pairs of vertices defined as follows:

$$\begin{array}{lcl}
[\![\epsilon]\!]_G & = & \{(v, v) \mid v \in V\} \\
[\![_]\!]_G & = & \{(u, v) \mid \exists a \in \Sigma (u, a, v) \in E\} \\
[\![a]\!]_G & = & \{(u, v) \mid (u, a, v) \in E\} \\
[\![a^-]\!]_G & = & \{(u, v) \mid (v, a, u) \in E\} \\
[\![\alpha_1 \cdot \alpha_2]\!]_G & = & [\![\alpha_1]\!]_G \circ [\![\alpha_2]\!]_G \\
[\![\alpha_1 + \alpha_2]\!]_G & = & [\![\alpha_1]\!]_G \cup [\![\alpha_2]\!]_G \\
[\![\overline{\alpha}]\!]_G & = & V \times V - [\![\alpha]\!]_G \\
[\![[\alpha]]\!]_G & = & \{(v, v) \in G \mid (v, u) \in [\![\alpha]\!]_G\} \\
[\![a^{m,n}]\!]_G & = & \bigcup_{k=m}^{n} [\![r]\!]_G^k
\end{array}$$

where $\circ$ is the concatenation of binary relations and R^i denotes the concatenation of R with itself i times.

Example 2.2 Consider the graph depicted in Figure 2 and the query $a^{0,*} \cdot [b]$. This query returns all the pairs of vertices (x, y) where x and y are connected by an a-labeled path of any length and y has an outgoing b edge. In the case of the graph of Figure 2, this query returns $\{(v_4, v_4), (v_2, v_4), (v_1, v_4), (v_3, v_4)\}$.

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 $\{(v_1, v_6), (v_1, v_5), (v_2, v_5), (v_3, v_6)\}$. ■

In [17] Libkin et al. proved that the combined complexity of GXPath is polynomial.

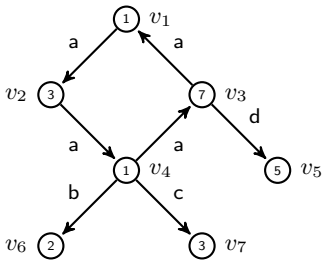


Figure 2: A graph.

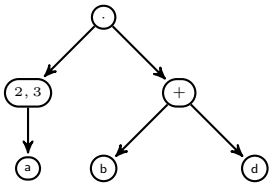


Figure 3: Abstract syntax tree of $a^{2,3} \cdot (b + d)$.

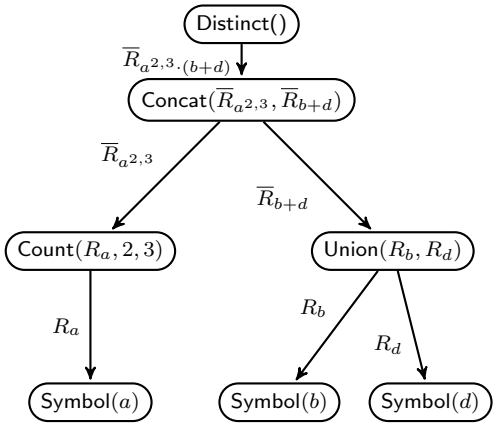


Figure 4: Query plan for $a^{2,3} \cdot (b + d)$.

3. QUERY PROCESSING

We first illustrate our query processing technique through examples and, then, present in more detail the operators used in our system.

3.1 Overview

In our system, before processing queries on an input graph $G = \langle V, E, \rho \rangle$, G must be indexed. During graph indexing, the system creates a distinct binary relation per label, containing all the edges in E having that label. These relations are stored in the distributed file system and form the actual input data that will be analyzed. In the following, we will assume that the indexing phase returns a set $\mathcal{R} = \{R_a \mid a \in \Sigma\}$, where $R_a = \{(x, y) \mid (x, a, y) \in E\}$; more generally and with a little abuse of notation, we will use $\overline{R}_\alpha$ and R_α to denote the result of the evaluation of α with and without duplicates, respectively.

When a query is submitted to the system, it is first translated into its abstract syntax tree. After a simplification phase, where common rewritings (e.g., $\alpha \cdot \epsilon = \epsilon \cdot \alpha = \alpha$) are applied, the resulting AST is transformed in an acyclic graph of MapReduce jobs; each job consumes and produces binary relations of vertices.

Example 3.1 Consider again the query $a^{2,3} \cdot (b+d)$ on the graph of Figure 2. This query can be translated in the AST of Figure 3.

In our system this AST is transformed in the query plan shown in Figure 4. Here, Symbol(a), Symbol(b), and Symbol(d) are operators that access the system catalog and return the location of binary relations R_a, R_b, R_d in the distributed file system. In the case of our input graphs, these relations have the following extensions:

$$\begin{aligned} R_a &= \{(v_1, v_2), (v_2, v_4), (v_4, v_3), (v_3, v_1)\} \\ R_b &= \{(v_4, v_6)\} \\ R_d &= \{(v_3, v_5)\} \end{aligned}$$

Count($R_a, 2, 3$), then, evaluates the bounded transitive and reflexive closure of R_a. To this aim, Count($R_a, 2, 3$) exploits a MapReduce implementation of the SemiNaive algorithm described by Bancilhon in [7]. In our example, Count($R_a, 2, 3$) returns the following binary relation: $\{(v_1, v_4), (v_2, v_3), (v_4, v_1), (v_3, v_2), (v_1, v_3), (v_2, v_1), (v_4, v_2), (v_3, v_4)\}$.

Union(R_b, R_d) accesses R_b and R_d, and returns their union.

The output of Count($R_a, 2, 3$) and Union(R_b, R_d) is passed as input to the concatenation operator Concat($\overline{R}_{a^{2,3}}, \overline{R}_{b+d}$). Concat($R, S$) takes as input two binary relations R and S, and returns their composition, by applying a standard matrix product algorithm. In our example, Concat($\overline{R}_{a^{2,3}}, \overline{R}_{b+d}$) returns the following binary relation: $\{(v_1, v_6), (v_2, v_5), (v_1, v_5), (v_3, v_6)\}$.

Finally, the output of Concat($\overline{R}_{a^{2,3}}, \overline{R}_{b+d}$) is sent to the duplicate elimination operator Distinct($\cdot$). In our system, indeed, query operators do not return sets of vertex pairs, as prescribed by GXPath set-based semantics, but collections that may contain repeated entries. Duplicates are eliminated at the end of query evaluation by relying on the Distinct($\cdot$) operator. We made this choice as early experiments showed that set-based operators were much slower than sequence-based ones.

■

Example 3.2 Consider now the query $\overline{b^- \cdot [c] \cdot a^{1,3}}$. This query first looks for all vertex pairs (x, y) such that $(x, y) \in [\![b^- \cdot [c] \cdot a^{1,3}]\!]_G$, and, then, computes the complement of $[\![b^- \cdot [c] \cdot a^{1,3}]\!]_G$ (i.e., $V \times V - [\![b^- \cdot [c] \cdot a^{1,3}]\!]_G$). In particular, in each pair (x, y) y is reachable from x by traversing backward an incoming b-labeled edge, by filtering out vertices without c-labeled outgoing edges, and by traversing a path labeled with a, aa, or aaa.

In the case of the graph of Figure 2, $[\![b^- \cdot [c] \cdot a^{1,3}]\!]_G = \{(v_6, v_3), (v_6, v_1), (v_6, v_2)\}$ and $[\![\overline{b^- \cdot [c] \cdot a^{1,3}}]\!]_G = \{(v_i, v_j) \mid i = 1, \ldots, 7,\ j = 1, \ldots, 7\} - \{(v_6, v_3), (v_6, v_1), (v_6, v_2)\}$.

To evaluate the result of this query, our system creates the query plan shown in Figure 5. Here, BackSymbol(b) accesses relation R_b, created during the indexing phase and stored in the distributed file system, and returns a relation R_{b^-} obtaining by adding a pair (v_x, v_y) for each pair $(v_y, v_x) \in R_b$. Operator Cond(R_c), instead, just accesses relation R_c and outputs a relation $\overline{R}_{[c]}$ containing a pair (v_x, v_x) for each $(v_x, v_y) \in R_c$.

The Compl($\cdot$) operator, finally, computes the complement of the binary relation passed as input. To this aim, it just

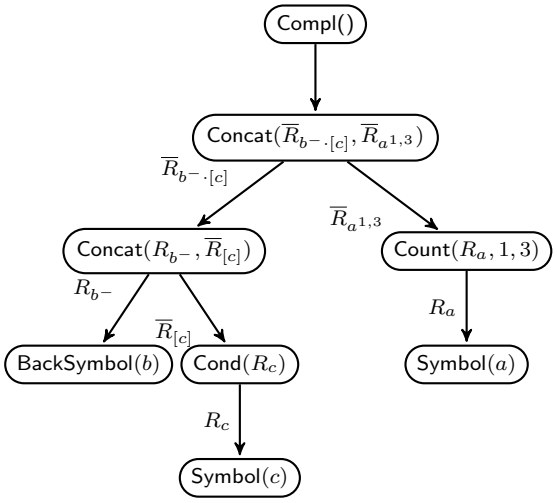


Figure 5: Query plan for $\overline{b^- \cdot [c] \cdot a^{1,3}}$.

generates all the pairs of vertices $(u, v) \in V \times V$ and, during the generation process, it discards those belonging to its input. As Compl() already suppresses duplicates by itself, the system does not introduce a Distinct() operator. ■

3.2 Indexing

As already stated, each graph must be indexed before becoming available for querying. To this aim, our system preprocesses each graph by using a MapReduce job whose Map phase returns, for each edge $(x, label, y)$, the *key-value* pair $(label, (x, y))$; during the Reduce phase, each reducer receives a pair $(label, L)$, where $L = \{(x, y) \mid (x, label, y) \in E\}$, creates a compressed file for $label$, and add each pair (x, y) to the file. As a result, the indexer creates a collection of binary relations R_a, stored as compressed files on HDFS by relying on the Snappy codec of Hadoop. During this indexing phase, the system also records in the system catalog the number of edges and vertices of the graph (for the sake of simplicity, we assume that input graphs have no *isolated* vertices, i.e., vertices without incoming and outgoing edges).

3.3 Operators

Our system exploits several basic query operators. In this section we will discuss the most prominent ones, and assume, unless otherwise noted, that each operator reads and writes compressed files.

Epsilon. Epsilon() just returns a set of pairs (u, u), where $u \in V$. This operator is implemented as a Map-only MapReduce job, where each mapper is in charge of generating pairs (u, u) for a given interval of vertices (vertices are uniquely identified by integers). By relying on the `NLineInputFormat` class of Hadoop, each mapper is forced to process exactly one vertex interval, i.e., the system creates as many mappers as intervals.

Symbol *and* BackSymbol. Symbol(a) takes as input a symbol $a \in \Sigma$, and returns the relation R_a. As input graphs are assumed to have been previously indexed, Symbol(a) just looks in the catalog for the compressed file storing R_a, and returns its location in the distributed file system. BackSymbol(a), instead, returns a relation $R_{a^-} = \{(y, x) \mid (x, a, y) \in E\}$. To this end, BackSymbol(a) is implemented as a Map-only MapReduce job, whose mappers just reads the pairs in R_a and return their inverse.

Wildcard. The Wildcard() operator implements the _ operator of GXPath. To this end, it exploits the fact the input graphs have been previously indexed, and just returns the locations in the distributed file system of the files that store relations R_a, for any $a \in \Sigma$, hence leveraging on Hadoop ability to work on multiple input files.

Concat. Concat($\overline{R}_{\alpha_1}, \overline{R}_{\alpha_2}$) takes as input two binary relations and returns their composition. This is obtained by applying a matrix multiplication algorithm. In MapReduce there are different ways to evaluate matrix multiplication [16], that differ for communication cost, computational cost, and memory management. In our system we use a variant of the Two Step MapReduce Matrix Multiplication algorithm; in this variant there is just one single step that performs the multiplication row per column, without the need of a duplicate elimination second step.

Union. Union($\overline{R}_{\alpha_1}, \overline{R}_{\alpha_2}$) implements the union between two binary relations $\overline{R}_{\alpha_1}$ and $\overline{R}_{\alpha_2}$. As MapReduce jobs can work on multiple input files and in our system duplicate elimination is performed at the end of the computation, Union($\overline{R}_{\alpha_1}, \overline{R}_{\alpha_2}$) just returns to its father operator the location of the compressed files in the distributed file system containing the pairs of $\overline{R}_{\alpha_1}$ and $\overline{R}_{\alpha_2}$, without the need to activate a MapReduce job.

Cond. Cond($\overline{R}$) takes as input a binary relation $\overline{R}$ and returns a new relation $\overline{R}'$ such that $(x, x) \in \overline{R}'$ if and only if $\exists y.(x, y) \in \overline{R}$. Cond($\overline{R}$) is implemented as a single MapReduce job, whose Mappers create a (x, x) pair for each (x, y) pair being read.

Compl. Compl($\overline{R}$) takes as input a binary relation $\overline{R}$ and returns its *complement*, i.e., $V \times V - \overline{R}$. This operator is particularly challenging as the topology of the graph cannot be used to guide the evaluation process. This operator is implemented by loading $\overline{R}$ in a persistent hash set, and by running a MapReduce job, where each Mapper creates a fragment of $V \times V$ and, for each generated pair (x, y), discards (x, y) if (x, y) is contained in the hash set. The job has no Reduce phase.

Count. Count($\overline{R}, m, n$) takes as input a binary relation $\overline{R}$, and returns the result of $\overline{R}^{m,n}$. To this end, our system actually computes the bounded transitive and reflexive closure of $\overline{R}$ by exploiting a MapReduce implementation of a variation of the Seminaive algorithm [7]. This implementation creates a chain of MapReduce jobs, hence making Count($\overline{R}, m, n$) the only operator that requires more than one MapReduce job.

Distinct. The Distinct($\overline{R}$) operator takes as input a binary relation with duplicates $\overline{R}$, and returns a binary relation R without duplicates. Duplicate elimination is implemented through a MapReduce job, whose Map phase, for each pair

$(x, y) \in \overline{R}$, returns a pair (K, V), where $K = (x, y)$ and V is empty; in the Reduce phase, then, each reducer receives a pair $((x, y), L)$, where L is an empty list, and just outputs (x, y).

4. EXPERIMENTAL EVALUATION

In this section we analyze the performance and the scalability of our query processor. As there is no standard benchmark for evaluating the performance of graph query processors, we generated a set of 15 random queries and evaluated them on three datasets comprising large synthetic power-law graphs.

4.1 Experimental Setup

We performed our experiments on a 4-node, multitenant Hadoop cluster which is part of the Cineca[1] PICO cluster. Each node features an Intel Xeon E5 2650 v2 @ 2.6GHz CPU with 16 cores, 64 GB of main memory, and 32TB of local disks. Cluster nodes run RHEL Linux 6.5 and Hadoop 2.6.0, with HDFS block size set to 128MB and up to 2GB of memory per container.

4.2 Datasets

We performed our experiments on three datasets comprising synthetic graphs generated, according to the power law, by using the *R-MAT* generator of GTGraph [5]. These datasets, that we indicate with G_1, G_2, and G_3, serve the purpose of evaluating the scalability of our tool with the number of edges, the number of vertices, and both the number of edges and vertices, respectively.

The R-MAT algorithm generates random graphs according to the power law distribution; the inputs of the algorithm are the desired number of vertices n and the requested number m of edges; we set the minimum and maximum label value to 0 and 1000, respectively. The graphs generated by R-MAT are summarized in Tables 1, 2, and 3.

Name	n	m
1	100,000,000	10,000,000,000
2	200,000,000	20,000,000,000
3	300,000,000	30,000,000,000
4	400,000,000	40,000,000,000
5	500,000,000	50,000,000,000

Table 1: G_1 graph datasets created by R-MAT.

Name	n	m
1	100,000,000	49,999,999
2	200,000,000	199,999,999
3	300,000,000	449,999,998
4	400,000,000	799,999,998
5	500,000,000	1,249,999,997

Table 2: G_2 graph datasets created by R-MAT.

Name	n	m
1	141,421,356	10,000,000,000
2	200,000,000	20,000,000,000
3	244,948,974	30,000,000,000
4	282,842,712	40,000,000,000
5	316,227,766	50,000,000,000

Table 3: G_3 graph datasets created by R-MAT.

[1]Cineca is an Italian university consortium for supercomputing services.

4.3 Queries

As there is no standard benchmark for evaluating the performance and the scalability of graph query processors, we based our experiments on a set of random GXPath queries. To create this set of queries, we implemented an extended version of the random regular expression generator described by Colazzo et al. in [11].

In our generator, operators are divided in two main classes: intermediate operators and terminal operators. The first class comprises unary and binary operators, such as $\alpha + \alpha$, $\alpha \cdot \alpha$, $[\alpha]$, $\alpha^{m,n}$, and $\overline{\alpha}$, while the second class only contains operators that may appear in a leaf of the parse tree of a query: a, a^-, ϵ, and $\_$.

The generation algorithm takes as input an expected query AST depth, as well as a distribution of probability for intermediate and terminal operators, and it works recursively. During each recursive call, it verifies whether the current AST has reached the expected depth: if so, the algorithm randomly selects a leaf operator; otherwise, it randomly chooses a unary or binary operator and performs another recursive call.

Operators are selected by using a random generator, that generates *doubles* between 0 (inclusive) and 1 (exclusive). These range values are mapped, according to the operator probability set in Table 4.(a), into GXPath operators. In a similar way, the algorithm chooses the leafs using the probability set in Table 4.(b). m and n indexes of counting operators are extracted in a random way, in the integer range between 0 and $*$. Label values are randomly selected in the interval $]0, 1000[$.

(a) Unary and binary operators.

Operator	p
$\alpha + \alpha$	0,36
$\alpha \cdot \alpha$	0,36
$[\alpha]$	0,18
$\alpha^{m,n}$	0,09
$\overline{\alpha}$	0,01

(b) Terminal operators.

Operator	p
a	0,48
a^-	0,48
ϵ	0,01
$\_$	0,01

Table 4: Probability values of unary, binary, and terminal operators.

We generated, through the above random generator, 15 queries, shown in Table 5 These queries do not contain the complement operator. Indeed, the evaluation of this operator requires to materialize intermediate results whose size can be quadratic in the size of the input graph; this leaded, in all our preliminary experiments, to the exhaustion of the available HDFS space, even with compression enabled; therefore, we decided to remove queries with complement from our query set.

4.4 Experimental Results

In our experimental evaluation we performed several kinds of experiments. As a preliminary test, we measured the indexing time for all datasets; then, we evaluated the performance and the scalability of the system by executing each query of Section 4.3 on each dataset.

4.4.1 Indexing Time

In these preliminary tests we measured the time required for preprocessing and indexing each graph in our datasets.

Query	Expression
Q_1	$(\boxed{(768^- + 838)}) \cdot ((173 + 868^-) \cdot \boxed{740^-}))$
Q_2	$(\boxed{561 \cdot 903^-})^{6,37}$
Q_3	$(((51^- + 968) \cdot [705]) + ((1 + 748^-) + (315 \cdot 446^-)))$
Q_4	$(((577 + 15) + (83 \cdot 32^-)) + \boxed{(297 \cdot 385^-)})$
Q_5	$(((205^- + 2) \cdot (803 \cdot 798^-)) + ((\epsilon + 50) + [11]))$
Q_6	$(((320 + 286) + (498 \cdot \_)) \cdot [(906 \cdot 137)])$
Q_7	$((\boxed{659^-} + (444 + 340^-)) \cdot ((736^- + 525^-) \cdot [762]))$
Q_8	$([(28 \cdot 976)] \cdot ((546 + 530^-) \cdot (788^- + 505^-)))$
Q_9	$(\boxed{(990^- + 821)} + (162^- \cdot 464))^{59,63}$
Q_{10}	$(((189^- \cdot 195^-) \cdot (930^- + 216)) + \boxed{(304 \cdot 689^-)})$
Q_{11}	$(((534 + 709) \cdot (277 + 374)) \cdot \boxed{(844^- + 426)})$
Q_{12}	$(((856 + 518^-) \cdot (668 \cdot 897)) + ((672^- + 232^-) \cdot \boxed{367^-}))$
Q_{13}	$\boxed{((831 \cdot 612) + (797^- + 509^-))}$
Q_{14}	$(((346^- + 511^-) + \boxed{833^-}) \cdot ((256^- + 251^-) + (627^- + 794)))$
Q_{15}	$\boxed{((104 + 440) + (78 + 387^-))}$

Table 5: Queries.

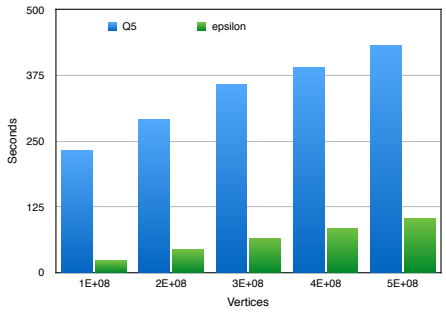


Figure 9: G_2 dataset: comparison between the execution time of query Q_5 wrt the number of vertices and the time required to generate the result of ϵ.

The results we obtained are shown in Figures 6(a), 6(b), and 6(c).

As it can be observed, the preprocessing and indexing time grows linearly with the size of the input graphs.

4.4.2 *Performance and Scalability Tests*

In these tests we evaluated the performance and the scalability of our system when processing the queries of Section 4.3 on the datasets of Section 4.2.

The results we obtained are shown in Figures 7(a), 7(b), and 7(c). As query Q_6, which contains the wildcard operator, showed a significant overhead wrt the remaining queries, we also reported in Figures 8(a), 8(b), and 8(c) these results without query Q_6. As it can be observed, the system preforms well on all queries (except for Q_6), and scales linearly with the number of edges and/or vertices.

To better understand the behavior of the system, we can analyze in more detail the results obtained so far for queries Q_5, Q_6, Q_9, and Q_{12}. Indeed, queries Q_5 and Q_6 are the only ones that include, respectively, ϵ and $\_$; query Q_9 contains counting operators with relatively high values of m and n; query Q_{12}, finally, contains all the remaining operators.

Query Q_5. Query Q_5 includes an ϵ operator inside a union at the top level of the query. This implies that the system must materialize an intermediate relation $[\![\epsilon]\!]_G = \{(u, u) \mid u \in V\}$ containing $|V|$ pairs. In turn, this implies that the selectivity of Q_5 is quite low and explains why its processing cost is higher than those of the other queries, as shown in Figs. 8(a), 8(b), and 8(c).

To understand if ϵ by itself can significantly worsen the processing time of a query, we compared in Figure 9 the query execution time with the time required for generating $[\![\epsilon]\!]_G$. As it can be observed, on large graphs ϵ impact on the system performance can be quite significant.

Query Q_6. From Figures 7(a), 7(b), and 7(c), we can observe that query Q_6 is the most expensive one in our query set, as its execution time may exceed four hours on large graphs. Indeed, to process this query the system must read the whole input graph, even if it has been previously indexed. Indexing, however, helps in decreasing Q_6 processing time, since the system can read the compressed files produced during the indexing phase.

Query Q_9. This query includes a counting operator with relatively high values of m and n (59 and 63, respectively). This implies that our evaluation algorithm must perform several iterations; however, this does not affect Q_6 processing time so badly, which confirms that our system has good scalability properties.

Query Q_{12}. This query contains all operators of the language with the exception of counting, $\_$, and ϵ. From Figures 8(a), 8(b), and 8(c) we can observe that the system behaves very well while processing this query, and scales linearly with the number of vertices and edges. This suggests that counting, $\_$, and ϵ (together with complement) are the most expensive operators to evaluate.

5. RELATED WORKS

There exist several systems for managing and querying data graphs. In this section we review a few of them.

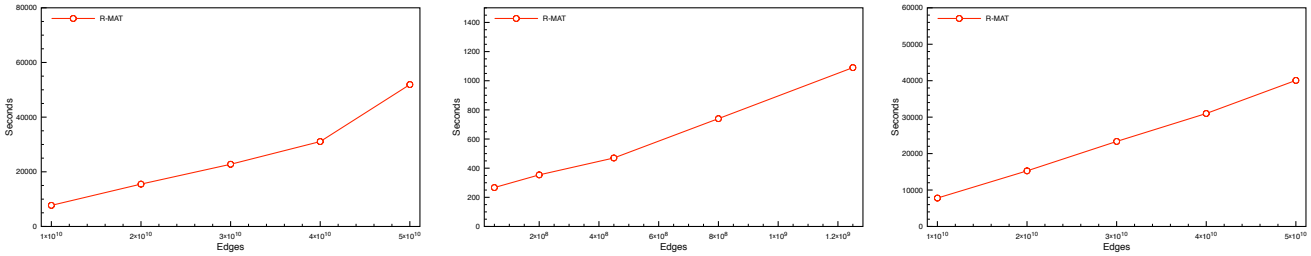
5.1 Graph Databases

Neo4j [2] is a disk-based, transactional graph database system. Neo4J allows one to use different graph query languages such as Gremlin and Cypher [13]. While very different, both Gremlin and Cypher can be used to express a very limited subset of GXPath. Neo4J is inherently centralized and it is not clear how it can scale on large or very large data graphs.

InfiniteGraph [1] is a distributed graph database system based on a distributed object store. InfiniteGraph query language allows one to express navigational queries containing forward navigation operators, unions among symbols, as well as counting and Kleene star operators on symbols only: therefore, simple queries of the form $(ab + c)^{0,*}$ cannot be easily expressed. To the best of our knowledge, there is no independent experimental evaluation of the performance of InfiniteGraph.

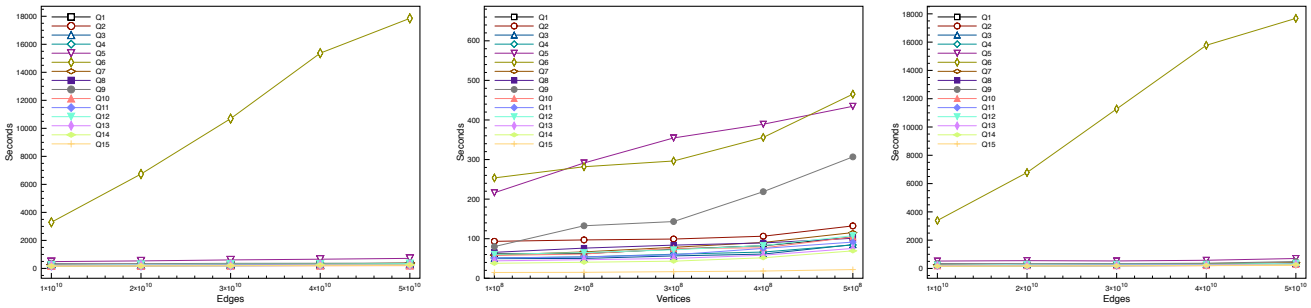
5.2 Regular Path Queries Processors

G-Path [6] is a system for processing regular path queries on graphs, based on the BSP model and built on top of Hadoop HDFS and HAMA. The query language allows the user to define regular expressions by means of twig patterns. The language allows the user to impose conditions on edge labels and vertex values, but it seems very limited wrt the language supported by our system.



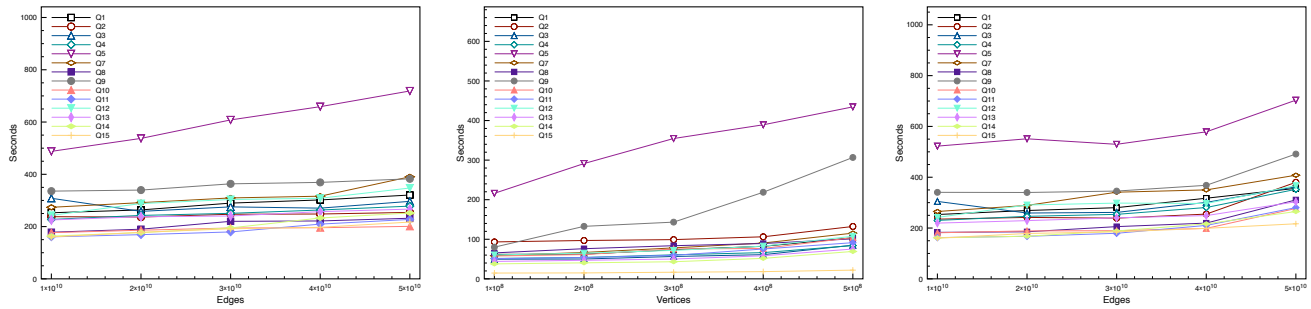
(a) G_1 results. (b) G_2 results. (c) G_3 results.

Figure 6: Indexing time.



(a) G_1 results. (b) G_2 results. (c) G_3 results.

Figure 7: Execution time (in seconds) for queries $Q_1, \ldots, Q_{15}$ wrt the number of edges.



(a) G_1 results. (b) G_2 results. (c) G_3 results.

Figure 8: Execution time (in seconds) for queries $Q_1, \ldots, Q_{15}$ (without Q_6) wrt the number of edges.

The system described by Koschmieder and Leser in [14] is a centralized query processor for evaluating RPQs on relatively small graphs. This system adopts a non standard semantics for RPQs, as it focuses on simple paths only. The evaluation algorithm is based on the idea of performing searches starting from vertices of the graph having rare labels. These rare labels are used to split the input query in smaller queries, and form the starting points of path traversal. Authors reported several experiments describing the performance of the system when evaluating queries on relatively small graphs (up to 1-2 million vertices): these experiments seem suggesting that the proposed system cannot scale linearly with the number of vertices and/or edges.

In [20] Sarwat et al. describe Horton+, a distributed query processor for reachability queries over data graphs. Horton+ query language is strongly based on RPQs and also allows the user to test for vertex values. Graphs are partitioned on a set of partition servers, which gather statistics about the partitioned graphs and cooperate in evaluating queries. An extensive experimental evaluation shows that Horton+ scales linearly with the number of vertices in sparse graphs, but there are no experiments studying the scalability in the number of edges.

6. CONCLUSIONS AND FUTURE WORK

In this work we investigated the problem of querying massive graph datasets and, in particular, we implemented a *scalable* and efficient distributed query processor for the navigational fragment of GXPath. Our query processor exploits a MapReduce infrastructure to coordinate the work of multiple machines in a cluster and to distributed the computing load among them.

To evaluate the performance and the scalability of our implementation, we performed several experiments on a very small Hadoop cluster, using synthetic graph datasets with size up to 500 million vertices and 50 billion edges; given the lack of benchmarks for graph queries, we relied on a set of randomly generated queries. These experiments showed that our system is scalable and that it can efficiently process complex queries on large graphs.

In the near future we would like to extend our system in two ways. First of all, we would like to widen the fragment of GXPath being supported by our implementation, and, in particular, to cover the value test fragment of the language. This is a challenging task, as GXPath allows for comparing values stored in vertices reachable from different paths.

In the second place, while our system is very scalable, queries containing complement operations are still too expensive to be evaluated. To overcome this limitation, we plan to introduce more sophisticated processing techniques that, looking at the part of the query following a complement operator, could limit the scope of the complement.

7. REFERENCES

[1] InfiniteGraph. `http://www.objectivity.com/products/infinitegraph/`.

[2] Neo4J. `http://neo4j.com`.

[3] Statistics on Facebook, 2015. Available at `http://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide`.

[4] Twitter statistics, 2015. Available at `http://www.beevolve.com/twitter-statistics/`.

[5] D. A. Bader and K. Madduri. GTgraph: A synthetic graph generator suite, 2006.

[6] Y. Bai, C. Wang, Y. Ning, H. Wu, and H. Wang. G-Path: flexible path pattern query on large graphs. In *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*, pages 333–336. ACM, 2013.

[7] F. Bancilhon. Naive evaluation of recursively defined relations. In *On Knowledge Base Management Systems (Islamorada)*, pages 165–178, 1985.

[8] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR 2000, Principles of Knowledge Representation and Reasoning Proceedings of the Seventh International Conference, Breckenridge, Colorado, USA, April 11-15, 2000.*, pages 176–185. Morgan Kaufmann, 2000.

[9] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Rewriting of regular expressions and regular path queries. *J. Comput. Syst. Sci.*, 64(3):443–465, 2002.

[10] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Reasoning on regular path queries. *SIGMOD Record*, 32(4):83–92, 2003.

[11] D. Colazzo, G. Ghelli, L. Pardini, and C. Sartiani. Efficient asymmetric inclusion of regular expressions with interleaving and counting for XML type-checking. *Theor. Comput. Sci.*, 492:88–116, 2013.

[12] M. P. Consens and A. O. Mendelzon. Graphlog: a visual formalism for real life recursion. In D. J. Rosenkrantz and Y. Sagiv, editors, *PODS*, pages 404–416. ACM Press, 1990.

[13] F. Holzschuher and R. Peinl. Performance of graph query languages: comparison of Cypher, Gremlin and native access in Neo4J. In G. Guerrini, editor, *Joint 2013 EDBT/ICDT Conferences, EDBT/ICDT '13, Genoa, Italy, March 22, 2013, Workshop Proceedings*, pages 195–204. ACM, 2013.

[14] 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.

[15] V. E. Krebs. Uncloaking terrorist networks. *First Monday*, 7(4), 2002.

[16] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press, 2014.

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

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

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

[20] M. Sarwat, S. Elnikety, Y. He, and M. F. Mokbel. Horton+: A distributed system for processing declarative reachability queries over partitioned graphs. *PVLDB*, 6(14):1918–1929, 2013.