Parallel relational databases for diameter calculation of large graphs

Fabiano da Silva Fernandes
Faculty of Campo Limpo Paulista (FACCAMP)
Campinho Paulista, Sao Paulo, Brazil
Email: contato@fabiandfernandes.adm.br

Eduardo Javier Huerta Yero
Faculty of Campo Limpo Paulista (FACCAMP)
Campinho Paulista, Sao Paulo, Brazil
Email: huerta@faccamp.br

Abstract—Parallel relational databases are seldom considered as a solution for representing and processing large graphs. Current literature shows a strong body of work on graph processing using either the MapReduce model or NoSQL databases specifically designed for graphs. However, parallel relational databases have been shown to outperform MapReduce implementations in a number of cases, and there are no clear reasons to assume that graph processing should be any different. Graph databases, on the other hand, do not commonly support the parallel execution of single queries and are therefore limited to the processing power of single nodes. In this paper, we compare a parallel relational database (Greenplum), a graph database (Neo4J) and a MapReduce implementation (Hadoop) for the problem of calculating the diameter of a graph. Results show that Greenplum produces the best execution times, and that Hadoop barely outperforms Neo4J even when using a much larger set of computers.

I. INTRODUCTION

For decades relational databases stood as the dominant choice for storing, managing and retrieving large datasets, as its widespread adoption indicate. The declarative nature of the relational model allows programmers to specify what data they want to store and retrieve, leaving it to them implementation to choose the data structures and algorithms necessary to do so. Furthermore, features such as integrity constraints, ACID properties and referential integrity, which are present in many implementations of the model, release the programmer from dealing with the intricacies of creating and maintaining a consistent dataset in the face of concurrent access and hardware or software failures.

However, the recent explosion of digitally available data has exposed weaknesses in relational databases. The challenges faced by these traditional systems can be broadly classified in:

- **Volume**: The amount of data stored has grown into orders of magnitude that overwhelm current relational databases. In particular, the size of the tables and, therefore, the time required to perform the join operations needed to execute queries (the so-called join pain) has made it unfeasible to manage modern datasets using traditional tools.

- **Velocity**: Data velocity refers to the rate at which data changes over time. Modern systems typically deal with high volumes of data operations, often write-heavy, concentrated in short periods of time. Furthermore, the data model usually changes over time as well, due mainly to the fluid nature of modern businesses and the experimental nature of data acquisition and processing. Relational databases have been known to perform poorly during peak loads and are ill prepared to deal with constantly changing data models.

- **Variety**: Today’s data may be dense or sparse, data items may be interconnected or independent and may be structured, semi-structured or unstructured. This landscape is far removed from the type of problems relational databases were designed to solve. Forcing current data into a relational model often produces inefficient and counter-intuitive data models that are hard to maintain.

Relational databases have responded to some of these challenges by increasing the adoption of known techniques such as data sharding, column-oriented tables and the parallel execution of single queries. Databases supporting the latter feature are known as parallel relational databases and have been successfully deployed on highly demanding scenarios in the industry where the more traditional relational databases were not sufficient.

A host of alternatives approaches have also been developed to cope with these challenges. One of the most widely known is MapReduce, a simple and powerful programming model and implementation initially developed by Google [7]. Other approaches have been grouped under the name of NoSQL databases [11], representing their departure from the traditional relational model. NoSQL databases typically alter or completely eliminate some essential feature of the relational model (e.g. data model, ACID properties) in order to improve metrics regarded as more vital, such as performance, availability and/or fault tolerance.

In the particular case of graphs, a number of specific solutions such as GraphLab [18] and NoSQL graph databases such as Neo4J [8] have been developed. For large graphs, the recent body of work has focused on the MapReduce model [7] and to a lesser extent the Bulk Synchronous Parallel (BSP) model [27].

MapReduce has been proved to perform poorly for graph algorithms, both theoretically [20] and through practical experiments [12]. Some of the reasons behind this lack of performance are its reliance on an external data storage layer (e.g. HDFS for Hadoop), its need to materialize intermediate results on this storage layer (e.g the results from the map phase) and the stateless nature of the map and reduce phases.
that forces data to be redistributed at the beginning of every cycle.

Graph databases use their multi-node deployments for high availability and fault tolerance, but do not commonly support the parallel execution of single queries. Therefore, when processing complex queries on large graphs, their performance tends to be limited by the computing power of single nodes.

Although proven to be superior to MapReduce in a number of use cases [26], parallel relational databases are not commonly considered as a potential solution for graph problems in current literature. More interestingly, the characteristics that made parallel relational databases outperform MapReduce in [26] (i.e. better data distribution and locality, indexing capabilities, execution plan optimization) are all true for graph algorithms as well. When compared to graph databases, parallel relational databases offer a less intuitive data model, but coupled with a highly tuned parallelization mechanism for single queries.

In this paper we intend to explore this issue by calculating the exact diameter of a graph with a graph database, a MapReduce implementation and a parallel relational database and comparing their performances. We selected the diameter calculation because it requires the calculation of the distance between every pair of vertices in the graph (known as the All-Pairs Shortest Path problem), which is computationally intensive. Also, an exact solution to this problem is less likely to be affected by the particular partitioning of the graph among the parallel nodes, since every vertex has to be considered against every other vertex in the graph.

We do not intend to present a full comparison of these models in the realm of graph algorithms, or to provide innovative algorithms for graph calculation (which is typically done using approximate algorithms [13]). Instead, our goal is to present an initial set of experiments to explore the feasibility of using a parallel relational database to process large graphs instead of a MapReduce implementation or graph databases. We believe, however, that the problem of calculating the exact diameter of a graph is complex enough to expose most of the strengths and weaknesses of each solution.

Rest of the paper is organized as follows. Section II describes literature relevant to this paper. Section III briefly presents each model compared. Section IV compares each model and points out their main differences. Section V describes the algorithms used for calculating the diameter of a graph on each model. Section VI explains the design of the experiments and the datasets used. Section VII presents the results of the experiments and comments on them. Finally, Section VIII presents the conclusions of this paper.

II. RELATED WORK

A large effort has been devoted to write graph algorithms for the MapReduce model and understanding their performance characteristics. In [6] the author identify challenges and propose MapReduce based solutions for graph mining, matching and querying. In [17] several design patterns are proposed whose objective is to enhance the performance of graph algorithms in MapReduce. In [23] a class of MapReduce algorithms is introduced that is named Scalable Graph Processing Class, and its demonstrated that algorithms belonging to it exhibit good scalability when processing large graphs.

The PEGASUS graph mining library [14], built on top of Hadoop MapReduce, implements typical graph-mining tasks such as calculating graph diameter and the radius of a specific vertex, as well as finding the connected components. For the specific problem of calculating the diameter of a graph, [13] offers an approximate solution, while [19] offers the exact results, both based on MapReduce.

MapReduce has also been compared to parallel relational databases in traditional scenarios not involving graphs. In [21] and later in [26] the authors compare the performance and suitability of MapReduce and two parallel databases for problems commonly found in traditional scenarios. The papers concludes that parallel databases are best suited for querying large datasets, while MapReduce is best suited for ETL (Extract, Transform and Load) and complex analytics.

MapReduce has also been extended or enhanced to improve its performance on graph algorithms. In [1] the authors propose an extension to the MapReduce model called PACT, which is based on the idea of parallel contracts. Among the problems used to compare both models are the pairwise shortest path and the edge-triangle enumeration in large graphs.

In [22] a MapReduce implementation is described that uses the well-known Message Passing Interface (MPI) as the communication mechanism. By giving up on some of the most heralded features of standard MapReduce implementations, such as fault tolerance and data redundancy, the authors are able to provide an implementation that outperforms its standard Hadoop counterpart in several graph problems.

In [5] is described a system called Surfer, which extends MapReduce with a propagation operation designed to ease the implementation of edge-oriented tasks in graphs. Similarly, [28] and [4] propose an extension to MapReduce that introduces the capability of executing iterative computations, typically needed for graphs algorithms.

Other models have also been compared to MapReduce in the context of graph algorithms. In [15], MapReduce is compared to the PRAM (Parallel Random-Access Machine) model using the Minimum Spanning Tree (MST) of a dense graph in the comparison. In [10] the authors compare the MapReduce, BSP, PACT and GraphLab [18] (which is based on the Gather-Apply-Scatter pattern) programming models for the k-core decomposition problem in graphs. With few exceptions, all other models performed better than MapReduce.

In [25] Microsoft’s Trinity [24] and other platforms for graph computing, such as MapReduce, Pregel, Pegasus and several NoSQL based approaches are superficially compared. Finally, in [12] the authors compare MapReduce and BSP for graph algorithms. The problems chosen for the comparison were the Single Source Shortest Path (SSSP) and Relational Influence Propagation (RIP). The results show that BSP is better suited to handle graph problems, although the particular implementation chosen (Apache Giraph) presented some scalability issues for the larger graphs tested that discourage the adoption of this platform.
III. Models

In the MapReduce model the input data to be processed is divided into chunks, which are then fed to several map tasks. Each map task produces an output as a set of \{key, value\}, which are then shuffled by the MapReduce implementation such that all outputs with the same key are fed to the same reduce task. The reduce task then combines these intermediary results to produce the final output.

The power of the MapReduce model relies in letting the programmer focus on the details of the application while the MapReduce implementation deals with all problems associated with its parallel execution, such as code and data distribution, load balancing, fault tolerance and data flow.

Parallel relational databases work by harnessing the power of several database servers to execute single user queries in parallel. Traditional models for parallel and distributed databases include shared-memory, shared-disk and shared-nothing, although shared-nothing architectures have prevailed due to their simplicity, low cost and high performance and scalability.

Relational database servers execute SQL queries that consist of a set of relational operators applied to potentially large sets of data. SQL queries do not specify how they should be executed, thus giving database implementors considerable freedom in the execution plan and offering large opportunities for parallelism [9]. A SQL query can be executed as a dataflow graph, composed of a set of relational operators such that:

- the output of one operator can be the input of another, creating opportunities for pipelined parallelism
- the data to be processed can be partitioned and submitted to several dataflows whose results are merged at the end.

A graph database uses graph theory to represent discrete entities and their relationships. Entities are represented by graph nodes, while relationship among entities are represented through edges. Both nodes and edges can have associated properties to describe their characteristics, much like the Entity-Relationship model. In graph databases relationships are materialized and stored in the database at creation time instead of being calculated through join-like operations at execution time, so graph traversal is more natural and efficient.

IV. Implementations

The architectural differences between parallel relational databases, MapReduce and graph databases have been studied in current literature [9] [2] [8]. Comparing these architectures is difficult, given the number of different options available. In this work we chose the standard Hadoop/HDFS implementation of the MapReduce model, a mainstream graph database without support for parallel query execution (Neo4J) and a mainstream parallel relational database (Greenplum). We will now briefly compare these particular systems instead of the general models to which they belong.

- Parallelism support: Both Hadoop and Greenplum support parallelism and therefore are capable of partitioning the data to be processes among the available nodes. The main difference is that Hadoop materializes intermediate results on an external data storage and requires the map phase to be finished before the reduce phase begins. Greenplum exploits pipeline parallelism to immediately push intermediate results to the next processing stage. Neo4J do not support parallelism.
- Indexing: Both Neo4J and Greenplum support indexing, while Hadoop does not.
- Input parsing: Input data is parsed at creation time by both Greenplum and Neo4J, while Hadoop requires data to be parsed at every execution.
- Scheduling: Greenplum creates a distributed query plan so that each node knows exactly what to do before the query execution begins. Such compile-time scheduling contrasts with Hadoop, that dynamically allocates input blocks to map and reduce tasks at execution time. Compile-time scheduling usually produces better performance results, while execution-time scheduling is better at adapting to workload skews and changes in node performance.
- Fault tolerance: Both Neo4J and Greenplum provide transaction-level fault tolerance, meaning that if an operation within a transaction fails the whole transaction is rolled back. Hadoop provides block-level fault tolerance, so that if a task assigned to process an input block fails then that block will be reallocated to another task.

V. Diameter Calculation

The diameter of the graph is calculated using the same strategy for all approaches: a breadth-first search expanding from every vertex of the graph until it reaches every other vertex. This algorithm produces the eccentricity of each vertex, and the diameter of the graph is then determined as the maximum value of all eccentricities.

A. MapReduce algorithm

The MapReduce algorithm used in the comparison is based on the one presented in [19]. The driver program, that repeatedly calls the map() and reduce() functions until all vertices are processed, is shown in Figure 1.

```plaintext
input: Graph G
output: Diameter D
for all v \in G do
    P \leftarrow P \cup \{v, v, 0, \text{PROCESSING}\}
end for
while \exists \text{path} \in P, \text{path}.state = \text{PROCESSING} do
    map()
    reduce()
end while
D \leftarrow \text{Max}(P.distance)
```

Figure 1. MapReduce main algorithm for the exact calculation of graph diameter

This algorithm relies on a path vector \( P \) that contains all currently known paths. Each vector entry contains the path
origin vertex (path.origin), the destination vertex (path.dest),
the currently known distance (path.distance) and the path
state (path.state), which can be either PROCESSING or
PROCESSED, depending on whether all neighbors of
the destination vertex of the path have been processed. The
algorithm initializes the path vector (with paths from each
vertex to itself with zero distance and PROCESSING state)
and repeatedly calls map() and reduce() until all paths are in
state PROCESSED.

Figure 2 presents the map() function invoked by Figure 1.

```plaintext
input: Graph G, path ∈ P
output:
(path.orig, path.dest) → (path.distance, path.state)

if path.state = PROCESSED then
    Output((path.orig, path.dest) →
        (path.distance, path.state))
else
    for all v ∈ Neighbors(path.dest) do
        Output((path.orig, v) →
            (path.distance + 1, PROCESSING))
    end for
end if
```

Figure 2. Map function for the exact calculation of graph diameter

The map() function receives the complete graph G and an
entry of the path vector P called path. If path needs further
processing, the map function outputs a new entry for every
neighbor of the destination vertex path.dest with its distance
increased by 1. The reduce() function invoked by Figure 1 is
shown in Figure 3.

```plaintext
input: (path.orig, path.dest) → values =
    {(path.distance1, path.state1), ...}
output:
(path.orig, path.dest) →
    (path.distance, path.state)

path.distance ← Min(values.distance)
/*Consider PROCESSED > PROCESSING */
path.state ← Max(values.state)
Output((path.orig, path.dest) →
    (path.distance, path.state))
```

Figure 3. Reduce function for the exact calculation of graph diameter

The reduce() function receives as input all distances and
states calculated for a specific pair of vertices. The path
distance is calculated as the minimum distance of all entries
in the input.

B. Relational DB algorithm

For the relational database algorithm the graph is repres-
ented by a table G with two fields containing the origin and
destination vertices of an edge. The algorithm then produces
a second table D with the distance for every pair of vertices,
as shown in Table I.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>orig</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>dest</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>distance</td>
<td>int</td>
<td>int</td>
</tr>
</tbody>
</table>

(a) Table G

(b) Table D

The algorithm is described in Figure 4. It works by iteratively adding new entries to the result table D, such that at iteration i it contains all pairs of vertices that can be reached with i hops from every possible origin vertex. At every iteration a table Frontier is created with the vertices reachable from destination vertices in D, and then updates D with the new entries. The loop finishes when no new entries are found, at which point the eccentricity for each vertex and the diameter of the graph are calculated.

```plaintext
input: Table G
output: Table D with distances for every pair of vertices
D ← G, D.distance=1
go ← true
while go do
    Frontier ← Select D.orig, G.dest, D.distance+1
    From D, G Where D.dest = G.orig
    AND G.dest <> D.orig
    Update D with Frontier if D does not contain
    an entry with a lesser distance
    go ← (items updated > 0)
end while
Ecc ← Select orig, MAX(distance) From D
Group By orig
Diameter ← SELECT MAX(distance) From Ecc
```

Figure 4. Relational database algorithm used to calculate the diameter of a graph.

C. Graph database algorithm

The algorithm for the graph database algorithm is straight-
forward. It sequentially scans the graph and calculates the
eccentricity of every vertex, which is then used to calculate
the diameter of the graph, as shown in Figure 5.

```plaintext
diameter ← 0
for all Nodes in graph as nodeSrc do
    ecc ← 1
    for all Nodes in graph as nodeDest do
        ecc ← max(ecc, distance(nodeSrc,nodeDest))
    end for
    diameter ← max(diameter, ecc)
end for
```

Figure 5. Sequential algorithm used in graph databases.

Graph databases offer convenient ways to traverse graphs.
For instance, Neo4J provides a Traverser class to reach every
node in the graph starting at a source node according to settings
specified by the user. In the example shown in Figure 6 a `Traverser` object is created from `nodeSrc` that reaches all other nodes in the graph using a breadth-first strategy.

```java
Traverser traverser = nodeSrc.traverse(
    Order.BREADTH_FIRST,
    StopEvaluator.END_OF_GRAPH,
    ReturnableEvaluator.ALL_BUT_START_NODE,
    RelTypes.KNOWS,
    Direction.BOTH);
```

Figure 6. `Traverser` object used in the Neo4J graph database to find all paths from a single node.

VI. DESIGN OF EXPERIMENTS

Experiments were conducted using a set of real and synthetic graphs of different sizes. The tests were designed to assess how execution time and speedup were affected by changing the size of the graphs and the number of available computers.

It is worth noting that the size of the graphs used were limited by the available computing time in the test environment. Since the calculation of the exact diameter of a graph is a demanding task, for larger graphs some of the tests would exceed the maximum allotted time and be interrupted.

A. Test environment

Tests were conducted on the Grid’5000 platform [3], a platform composed by approximately 1200 computers with more than 8000 cores combined, distributed among 11 locations in France. The experiments used two of the clusters offered by the Grid’5000 platform:

- **Cluster Paradent**: from the Rennes site, it is formed by 45 computers with two Intel Xeon L5420 processors each with four 2.66 GHz cores each (360 cores total), 32GB RAM and a 350GB SATA hard drive.
- **Cluster Graphene**: from the Nancy site, it is formed by 120 computers with an Intel Xeon X3440 processor containing four 2.53 GHz cores (480 cores total), 32 GB RAM and a 350 GB SATA hard drive.

Computers used in the experiments were connected by a 1 GB Ethernet switch. They were running Linux CentOS 5.9, kernel 2.6.18-371.4.1.el5.x86 64, Java Virtual Machine 1.7.0_21, Python 2.7.6, Python-NetworkX 1.9.0, Apache Hadoop 1.2.1 and the Pivotal Greenplum Database, version 4.2.2.

B. Datasets

The real graphs used in the tests were obtained from the Stanford dataset [16] and are shown in Table II.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
<th>Diameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caheph</td>
<td>9.877</td>
<td>51.971</td>
<td>17</td>
</tr>
<tr>
<td>Calphp</td>
<td>12.008</td>
<td>237.01</td>
<td>13</td>
</tr>
<tr>
<td>Astroph</td>
<td>18.772</td>
<td>396.161</td>
<td>14</td>
</tr>
<tr>
<td>Com-amazon</td>
<td>334.863</td>
<td>925.872</td>
<td>44</td>
</tr>
</tbody>
</table>

Table II. **REAL GRAPHS USED IN THE EXPERIMENTS (STANFORD DATASET [16]).**

Synthetic graphs are divided in two groups. The first group contain graphs with a fixed number of vertices and varying number of edges, and is presented in Table III. The second group has graphs with varying number of vertices and edges and is shown in Table IV.

<table>
<thead>
<tr>
<th>Edges</th>
<th>Diameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>200.000</td>
<td>7</td>
</tr>
<tr>
<td>300.000</td>
<td>6</td>
</tr>
<tr>
<td>400.000</td>
<td>5</td>
</tr>
<tr>
<td>500.000</td>
<td>5</td>
</tr>
</tbody>
</table>

Table III. **SYNTHETIC GRAPHS WITH 20,000 VERTICES AND VARYING NUMBER OF EDGES.**

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Edges</th>
<th>Diameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.000</td>
<td>100.000</td>
<td>4</td>
</tr>
<tr>
<td>20.000</td>
<td>200.000</td>
<td>6</td>
</tr>
<tr>
<td>30.000</td>
<td>300.000</td>
<td>8</td>
</tr>
<tr>
<td>40.000</td>
<td>400.000</td>
<td>11</td>
</tr>
<tr>
<td>50.000</td>
<td>500.000</td>
<td>13</td>
</tr>
</tbody>
</table>

Table IV. **SYNTHETIC GRAPHS USED IN THE EXPERIMENTS WITH VARYING NUMBER OF VERTICES AND EDGES.**

C. Data distribution

The test graphs had to be loaded on each of the 3 platforms compared. The simplest case was Neo4J: since we only used one computer to process the graphs they were loaded from a file containing vertices and edges.

For Greenplum, the table containing the graph was sharded among all nodes in the cluster using a simple hash function aimed at guaranteeing that data about a single vertex was located on the same node. Hadoop, on the other hand, offers less control about how data is divided. The graph, represented by a file, is loaded into HDFS which then divides it into blocks that are replicated and spread among nodes in the cluster.

VII. RESULTS

A. Real graphs

The best execution times for real graphs are shown in Figure 7. These results were obtained in the Paradent cluster using all 45 computers for both Greenplum and Hadoop and a single computer for Neo4J.

![Figure 7. Execution times for real graphs](image_url)

The performance of Neo4J was better for the smaller graphs and degraded for the larger graphs. Greenplum outperformed Hadoop in all cases, and was almost 2.5 times faster for the Com-Amazon graph.
Figure 8 shows the execution times for Hadoop and Greenplum when increasing the number of computers used. The behavior is similar for all graphs: for smaller numbers of computers Hadoop produces better execution times, while Greenplum produces better results when the number of computers available is larger.

Figure 8. Execution times for real graphs with Greenplum and Hadoop when varying the number of available computers.

Figure 9 shows the relative speedup for the Hadoop and Greenplum execution times for real graphs as the number of computers grow. The relative speedup is calculated dividing the execution time obtained with N computers by the execution time of the same solution when using a single computer.

Both speedup curves have similar shapes, increasing as the number of available computers grew. However, speedup numbers for Greenplum are clearly superior, reaching nearly 30 with 45 computers for the Com-Amazon graph.

B. Synthetic graphs

All synthetic graphs were processed on cluster graphene using all 120 computers available. Figure 10 shows the execution times for the first set of graphs, all with 20,000 vertices and varying number of edges.

As the number of edges increase there are two opposing forces affecting the execution time. On one side, the larger number of edges means more processing to be done to analyze the whole graph, while at the same time the diameter of the graph decreases and therefore less iterations are needed to reach the end of the algorithms.

As expected, Neo4j had the worst results, unable to take advantage of all the computing power available. However, it is worth noticing that Neo4j produced results that were about twice the execution times obtained by Hadoop, a remarkable fact considering it only used one computer. Greenplum had the best results and seemed to be less affected by the increasing number of edges, maintaining its execution times nearly constant while those of Neo4j and Hadoop increased.

The speedup values corresponding to the execution times in Figure 10 are shown in Figure 11.

The speedup values for Greenplum were considerably better than Hadoop, staying around 80 and reaching nearly 100 for the graph with 300,000 vertices. Speedup for Hadoop remained nearly constant for all graphs.

The second group of synthetic graphs was built by varying the number of vertices and edges. Unlike the first group, the larger graphs in this group had larger diameters, which meant a larger number of iterations to calculate it. Furthermore, the work to be done at each iteration was larger as well, as a consequence of increasing the number of vertices and edges.

Figure 12 shows the execution times for graphs in the second group. To simplify presentation, graphs are named according to their number of vertices and edges, so that Graph10 represents the graph with 10,000 vertices and 100,000 edges and so on.
Again, Neo4J had the worst results but took approximately only twice as long as Hadoop, that used 120 computers for the task. Execution times for Greenplum increased only slightly for the larger graphs, and were remarkably better than Hadoop’s.

Speedup values for the results in Figure 12 are shown in Figure 13.

As expected, speedup values for Greenplum were better than Hadoop. However, Greenplum had better speedup results for the smaller graphs. As the graphs grew larger the speedup values decreased, presumably because a smaller part of the processed data could fit into memory.

C. Analysis

The tests conducted prove that both Hadoop and Greenplum, being able to exploit the computing power available, produce better execution times than Neo4J, which was confined to a single computer. This is hardly a surprise: however, it is worth noticing that the execution time of Neo4J was remarkably close to those of Hadoop, taking about twice as long even when Hadoop used 120 computers and Neo4J was executed on a single computer.

Greenplum produced the best results, with speedups at about 2/3 of the optimum value and execution times more than 10 times faster than Neo4J. Perhaps counterintuitively, Hadoop performed better than Greenplum only in tests with a smaller number of computers, while Greenplum seemed to be better at handling effectively a larger number of nodes.

There are no doubts that a graph database such as Neo4J is better at handling graphs, as they should. This fact was clearly demonstrated by the simple and intuitive algorithm and data models necessary to calculate the graph diameter and the excellent performance results obtained with a single computer.

However, the tests conducted in this paper indicate that, when better performance is required, a parallel relational database such as Greenplum should be considered as a solution before MapReduce does. Greenplum was able to produce shorter execution times while still ensuring fault tolerance and data integrity. MapReduce implementations such as Hadoop seem to be useful when cost is an issue and/or when “quick and dirty” analysis are required for which purchasing, installing and configuring a parallel database would be considered an overkill [26].

VIII. CONCLUSIONS

This paper presented a set of experiments aimed at understanding the performance characteristics of a parallel relational database, a MapReduce implementation and a graph database in the context of graph processing. Since there are a considerable number of options and variations of each model, we selected a popular representative of each model and used it in the comparison: Greenplum (parallel relational database), Hadoop (MapReduce) and Neo4J (graph database).

The problem selected was the calculation of the diameter of a graph, since we felt it was demanding enough to shed light on the strengths and weaknesses of each system. Tests conducted with both real and synthetic graphs revealed that Greenplum had the best execution times and speedup values, followed by Hadoop and Neo4J in that order. However, it is worth noticing that although Neo4J does not support the parallel execution of single queries its execution times were only about twice as large as those of Hadoop, even when 120 computers were used by the latter.

We conclude that when performance is an issue, a parallel relational database such as Greenplum is a better choice than a MapReduce solution, unless other concerns such as cost or setup time favors a cheaper and quicker solution with MapReduce. Neo4J combines simple, intuitive and easy to maintain programming and data models with excellent performance results, even with its lack of support for parallelism.

REFERENCES
