GALGO: Scalable Graph Analytics Engine Exploiting Parallel DBMSs.

Wellington Cabrera  
University of Houston  
Houston, TX 77204, USA

Carlos Ordenez  
University of Houston  
Houston, TX 77204, USA

ABSTRACT
We present GALGO, a system for large scale graph analytics. GALGO promotes easy access to complex graph analytics in a parallel cluster, exploiting parallel database systems as a computation engine for R. The current DBMS technologies are supported: columnar, row and array. In this demonstration we show that fundamental graph algorithms including all pairs shortest path (APSP), single source shortest path (SSSP), PageRank, triangle counting, connected components and reachability are solved completely with queries dynamically generated by our system. Our system presents performance that is very competitive to a state-of-the-art graph system. Furthermore, our out-of-core graph computation can process graphs larger than the system RAM, without compromising performance. GALGO allows the user to execute iteratively complex graph algorithms and simple graph analytics.

1. INTRODUCTION
In an increasingly interconnected world, graph data sets become larger and more complex to analyze, especially in domains such as telecommunication, transportation and social networks, where graph data is growing at fast pace. Because this situation, graph analysis is usually computed in parallel clusters, using specialized graph systems. On the other hand, the R programming language has a growing popularity, especially in data science and statistics communities, due to its intuitive syntax, a rich library of tools, community support, and free availability. While R is rich on analytical functions, R is not scalable: it works in one thread/one node. Thus, it is not suitable to analyze graph data sets larger than the system RAM, as occurs in several real-life problems: transportation networks, the world wide web, and social networks. On the other hand, recently column DBMSs have been shown to provide orders of magnitude time improvement in analytic query processing, preserving the scalability of row-based parallel DBMSs.

With that background, we present GALGO (Graph Analytics and aLGOrithms), a novel system for data stored in DBMSs, that uses the R environment as a front end. We support a representative set of DBMS technologies: the classical row-store, the state-of-the art column-store, and the less explored array-store. Graph algorithms are solved completely with database queries, as we will explain in detail. Given the integration with the R language, users can compute graph analytics with R syntax, without having to understand internal DBMS aspects.

Due to the practical importance of graphs, several systems for large graph analysis have emerged. The common understanding is that demanding graph problems are to be solved either in large Hadoop clusters or with optimized C++ programs. Pregel and its open source successor Giraph are well studied systems for large graph analytics, running on top of Google DFS and Hadoop DFS. More recently, Apache incorporated GraphX, a graph library for Spark users. Pregel, Giraph and GraphX are based on the vertex-centric approach. In this approach the information of a vertex needs to be propagated to its neighbors in iterations called Supersteps. Recently this approach has been questioned, mainly because an excessive message passing across the cluster [7]. In a recent work, Jinjal et al. [5] have proposed another graph analytics system using columnar DBMS technology, keeping the vertex-centric approach. Another recent work is Emp- tgyHeaded [1], a main memory system for graph processing in multiple threads, running in one large-memory node. Other systems solve graph algorithms with matrix multiplications in distributed memory clusters, as CombBLAS [3]. Multiplication of large matrices is challenging; highly optimized linear algebra packages (LAPACK, ScALAPACK) work only when the input matrices fit in RAM. In contrast, our system can handle datasets larger than RAM because computes matrix multiplication with database queries (secondary memory algorithms) which have a smaller memory footprint.

We summarize the contributions of this work as follows:
1) We present a graph analytics system that uses parallel DBMSs as a computation engine, running complex graphs algorithms based on database queries. Our system is based on our previous research on recursive and iterative queries applied to graphs algorithms, as described in [6, 4].
2) We improve query performance in the context of graph analytics, by ensuring join with local matching, merge join, and benefit from data compression.
3) We show that the combination R+DBMS opens new opportunities for interactive and flexible graph analytics for large problems, which are commonly solved with batch-oriented, more rigid tools.
4) We show that column store is the DBMS technology that presents the best performance to compute several graph algorithms, competitive with state-of-the-art systems as Spark-/GraphX.

2. SYSTEM DESCRIPTION

2.1 System Overview

Our system exploits parallel clusters as a computation engine to run a set of graph analytics and algorithms. Graphs are stored in a distributed manner in a cluster with shared nothing architecture. From a light-weight computer running R, the user sends requests using R syntax. This requests are translated to appropriate SQL (AQL in case of array DBMS).

2.1.1 System Fundamentals

It is well known that many graph algorithms can be solved via matrix-matrix and vector-matrix multiplication. Most of the real life graphs are sparse, and can be stored either: (1) as a table of edges in relational DBMS; (2) as an adjacency matrix in Array DBMS. As explained in our previous work, we use regular join-aggregation queries to perform vector-matrix and matrix-matrix multiplication under several semi-rings [4, 6]. Taking advantage of the sparsity of the graph data and a careful data partitioning, join-aggregation queries execute in parallel with promising performance.

2.2 Definitions

Let \( G = (V, E) \) be a directed graph, where \( V \) is a set of vertices and \( E \) is a set of edges, considered as an ordered pairs of vertices. Let \( n = |V| \) vertices and \( m = |E| \) edges. An edge \((i, j)\) in \( E \) links two vertices in \( V \) and has a direction. Undirected graphs, a particular case, are easily represented by including two edges, one for each direction. Notice our definition allows the existence of cycles and cliques in graphs, which make graph algorithms slower.

In general, real-life graphs are sparse. Even though popular social networks have hundred millions users, a typical user may be connected with just a few hundreds contacts. Other common examples are links in web pages and flights linking airports. Therefore, it is reasonable to represent the adjacency matrix \( G \) in a sparse representation, which saves both computing, memory and storage resources. Several methods to represent sparse matrices are well known, and the interested reader may check [2]. In our work, sparse matrices (specially \( E \)) are represented as a set of tuples \((i, j, v)\) such that \( v \neq 0 \), where \( i \) and \( j \) represent row/column, and \( v \) represents the value of the entry. Thus, the space complexity will be determined by \( m = |E| \).

2.3 Graph Storage

The input of the system consists of a table containing the edges and their attributes, and an optional table containing vertex attributes. In social network analysis, vertex attributes may be hometown, company, and other personal information. After uploading the dataset, we project the edge table to get \( E \). Table \( E(i, j, v) \) stores the adjacency matrix of \( G \) in sparse representation. The numerical attribute \( v \) is some value, representing distance in a road network, cost in distribution network, or any required weight. \( E \) has primary key \( (i, j) \). This storage layout is equivalent to store a combination of the adjacency and weights matrices of the graph in sparse matrix representation. Since we use sparse matrix representation (entries with zero values are not stored), the space required for table \( E \) is \( O(m) \), much smaller than \( O(n^2) \).

2.4 System Architecture

The front-end is a light weight node, running the R environment. Graph algorithms are called from the R environment using R syntax and computed in the DBMS cluster. Moreover, the data analyst can retrieve results and summaries to an R object; therefore, the analyst can use the full R language to present visualizations and further analysis. We developed an R library, Translator of R Commands, which provides the interface between R and the database cluster. Running in the R runtime, the library has the following duties: a) To allow the user to invoke graph analytics and algorithm, adding to the R environment a set of functions. b) Generate requests to the Sparse Matrix Operation Solver (SMOS), after the user calls the computation of an algorithm. c) Overload several R functions, in such a way that the user may apply regular R commands to database views (for instance \( \text{diag}(E) \) will retrieve the diagonal of the adjacency matrix, or \( \text{rowSums}(E) \) a summation of its rows). Note that massive data transfer from the DBMS to the client is avoided; the only data sent back to the client node are the results.

The demanding I/O and numerical computations required for graph algorithms is performed by a parallel database cluster. The cluster may run on a cloud service or on-site, as presented in Fig 1. The DBMS running in the cluster can be any of these types: a) column-store; b) row-store;
and c) array-store. Note that the DBMS is not a mere data repository. Actually, GALGO solves graph algorithms in the DBMS with dynamically generated SQL queries.

We have named Sparse Matrix Operation Solver (SMOS) the subsystem that controls the computation in the DBMS cluster, where the principal operations are matrix-vector multiplication and matrix-matrix multiplication. Of course, a DBMS does not support such matrix operations; these operations are solved via database queries, as we elaborate in section 2.5. For that reason, a main duty of SMOS is the generation of dynamic queries to compute matrix-vector products and matrix powers. For columnar and row DBMS, SMOS generates standard SQL. For SciDB, the array-store DBMS, SMOS generates queries in Array Query Language, a specialized language for array queries. We emphasize the user is isolated from such code generation, she just calls R functions to solve graph algorithms.

2.5 Graph Algorithms with Database Queries

The computation of graph algorithms in a DBMS is conceptualized on the foundation of linear algebra. Matrix powers and matrix-vector multiplication are operations that solve many important graph algorithms. Our previous research [6] showed that matrix powers can be computed in a DBMS with linear recursive queries. We implemented linear recursive queries by iteration of SPJA queries. Let \( R_d \) a table with the partial output at a recursion depth \( d \), initialized as \( R_1 = E \). In each recursive step, \( R_d \) is computed as:

\[
R_d = \pi_{i,j} \sum_{v \in u}(E \bowtie E_{d-1})(E \bowtie E_{d-1}) R_{d-1}
\] (1)

On the other hand, algorithms solved by iterative matrix vector multiplication can be expressed as an iteration of SPJA queries [4]. Considering a vector \( S \) and a matrix \( E \) stored in relational tables \( S(i,v) \) and \( E(i,j,v) \), the matrix-vector product can be clearly computed with a relational query as:

\[
\pi_{i,j} \sum_{v \in u}(E \bowtie E_{d-1})(E \bowtie E_{d-1}) (E \bowtie S)
\]

Algorithm 1 is a pattern to solve several graph problems with iterative matrix-vector multiplication, computed by relational queries. The algorithm keeps iterating while the value \( \Delta \) is greater than a small value \( \epsilon \). Both \( \Delta \) and \( \epsilon \) depend on the specific graph problem. For instance, in PageRank the value \( \Delta \) is computed as the maximum difference between page rank value of a vertex in the last two iterations, and \( \epsilon \) is a small value (default or user-defined). This algorithmic pattern works for relational databases and array databases. Moreover, we keep the query as simple as possible, as follows:

1. The query joins two tables.
2. The query performs one aggregation, grouping by 1 column.
3. The output of the query is inserted in an empty table.
We do not do updates. Note \( |S_d| \leq n \).

**Algorithm 1:** Graph Algorithms with relational queries

<table>
<thead>
<tr>
<th>Data: Table ( E, S_0 ), optional vertexId ( s ), ( \epsilon )</th>
<th>Result: ( S_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d \leftarrow 0 ), ( \Delta \leftarrow \infty )</td>
<td>( S_d \leftarrow ) query to compute ( E \times S_{d-1} ), compute ( \Delta )</td>
</tr>
<tr>
<td>while ( \Delta &gt; \epsilon ) do</td>
<td>end return ( S_n )</td>
</tr>
<tr>
<td>( d \leftarrow d + 1 )</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Algorithms provided

<table>
<thead>
<tr>
<th>Query</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggreg</td>
<td>In-degree/Out-degree</td>
</tr>
<tr>
<td>Aggreg</td>
<td>Graph Density</td>
</tr>
<tr>
<td>Aggreg</td>
<td>Degree Distribution</td>
</tr>
<tr>
<td>Agg/Addition</td>
<td>Laplacian Matrix</td>
</tr>
<tr>
<td>Join</td>
<td>Transition Matrix</td>
</tr>
<tr>
<td>Iter. Matrix ( \times ) Vec</td>
<td>Single Src Shortest Path</td>
</tr>
<tr>
<td>Iter. Matrix ( \times ) Vec</td>
<td>Reachability</td>
</tr>
<tr>
<td>Iter. Matrix ( \times ) Vec</td>
<td>Connected Components</td>
</tr>
<tr>
<td>Power of Matrix</td>
<td>Triangle Counting</td>
</tr>
<tr>
<td>Power of Matrix</td>
<td>Triangle Enumeration</td>
</tr>
<tr>
<td>Power of Matrix</td>
<td>All pairs Shortest Path</td>
</tr>
<tr>
<td>Power of Matrix</td>
<td>Transitive Closure</td>
</tr>
</tbody>
</table>

2.6 Optimizations in Parallel DBMS

In parallel graph processing, execution performance is improved by an adequate data distribution through the computing nodes. An even data distribution is necessary to avoid bottlenecks. Moreover, we apply the following strategies for efficient query execution:

1. Local match for parallel joins: Rows that satisfy the join condition are always in the same computing node. This can be accomplished partitioning the data by a hash function applied to the joining columns (i.e. \( i \) joining \( j \), as in Eq.1). This is key to avoid costly data transfer between nodes.
2. Presorted data: The join between \( E \) and \( S \) can achieve a linear time complexity when the tables are presorted by the columns participating in the join condition. The algorithm is a merge join. This is critical for very large graphs. Presorted data is inherent of the array-based data organization.
3. Data Compression: The storage layout in columnar DBMS promotes light weight data compression, which is a feature of the DBMS that promotes less I/O.

2.7 Analytics and Algorithms provided

Table 1 presents the algorithms provided by GALGO, classified by the type of operation in the DBMS. Note that the analytics and algorithms can be run for the complete graph or filtering by vertex or edges. For instance, the user may filter a social network graph by the hometown attribute, to count triangles that satisfies such condition. Likewise, the user may filter a road network to find the shortest path avoiding toll roads.

3. SYSTEM DEMONSTRATION

The objectives of our demonstration are:

1. Showing matrix multiplications is fundamental to solve common graphs problems in a DBMS.
2. Comparing the performance of graph algorithms in columnar and array DBMS.
4. Understanding how to solve popular and relevant graphs problems using GALGO.
5. Experiencing the easy access to complex graph analytics in a cluster, with straightforward R syntax. Understanding the functions of our translator library for R and the Sparse Matrix Operations Solver.
6. Showing the flexible and Iterative analytical environment by the integration R+DBMS. Learning how R can be used to perform further analysis.
Table 2: PageRank execution time (seconds). 4 Nodes x 4GB RAM
Columnar DBMS vs Array DBMS vs GraphX

<table>
<thead>
<tr>
<th>Graph data set</th>
<th>Column DBMS</th>
<th>Array DBMS</th>
<th>Spark GraphX</th>
</tr>
</thead>
<tbody>
<tr>
<td>web-Google</td>
<td>5M</td>
<td>30</td>
<td>143</td>
</tr>
<tr>
<td>soc-pokec</td>
<td>30M</td>
<td>101</td>
<td>395</td>
</tr>
<tr>
<td>LiveJournal1</td>
<td>69M</td>
<td>261</td>
<td>1037</td>
</tr>
<tr>
<td>wikipedia-en</td>
<td>378M</td>
<td>695</td>
<td>stop</td>
</tr>
</tbody>
</table>

Setup

The DBMSs used in the demonstration are: a) column-store: Vertica 7.1; b) row-store: Postgres 8.4; c) array-store: SciDB 15.7. Moreover, Spark 2.0 with GraphX will be available for comparison purposes. We show that GALGO works either with the DBMS running in our institution cluster, or in the cloud (Amazon AWS). Notice Postgres can run in one node only. A laptop will run R locally. The database will contain graph data sets from the Stanford repository. We plan to show analytic tasks on sparse graphs of three sizes: small, wiki-votes (m=100K); medium, web-Google (m = 5M); large, soc-LiveJournal1 (m = 68M); extra-large, wikipedia-en (m=378M). These data sets will be preloaded, but we will also demonstrate how to start the analysis from the data loading.

Demonstration Scenarios

User is encouraged to run performance test and comparisons between column DBMS, array DBMS and Spark-GraphX. As Postgres one-node only, it is useless to compare it to parallel systems. The demonstration will include the following scenarios:

Performance Comparisons: The audience can try any of the algorithms and data sets, and make comparisons between parallel DBMS technologies and Spark-GraphX. Table 2 presents performance comparisons for one of the algorithms: PageRank. The audience will note that columnar DBMS presents superior performance in graphs analytics.

Impact of appropriate partitioning: The user will compare the execution time of several algorithms both with an arbitrary partitioning and with an even data distribution.

Computing algorithms in the DBMS by R scripts: To run a computation based on matrix-vector multiplication (see Table 1) the user just call the corresponding R function in our library. In the next step the user will try a fundamental graph algorithm: triangle counting. In this demonstration the user will call the function in our library calc(E^3), to send to the cluster a command to compute E^3. Then, the user will be invited to re-run the computation, to get “selective” triangle counting, for instance, triangles in the social network consistent in vertex which home town is a particular state (i.e Texas). Then, the user will be invited to try any of the provided algorithms in a social network data set.

Looking under the hood: Users interested in technical aspects may browse the query history and query plans, which are accessible from the R environment by functions from our library: query.history() and plan.history().

Experience flexible and iterative analytics: Using R as a front end, the user sends a request to compute E^3 in the cluster, as well as a few matrix operations, to obtain the number of triangles. As illustrated in Figure 2, the user may also compute the graph out-degree. The result is returned in an R object for further analysis or visualization. Notice these demanding matrix multiplications are automatically translated into efficient SQL queries, without user intervention. Figure 3 illustrates the computation of page rank on a large graph. The user filters the top 100 vertices, and presents a visualization of the filtered graph.

4. REFERENCES