GALGO: Scalable Graph Analytics with a Parallel DBMS

Wellington Cabrera*
University of Houston
Houston, USA

Xiantian Zhou
University of Houston
Houston, USA

Ladjel Bellatreche
LIAS/ISAE-ENSMA
Poitiers, France

Carlos Ordonez
University of Houston
Houston, USA

ABSTRACT

We present GALGO, a system for large scale graph analytics. GALGO provides complex graph analytics in a parallel cluster, exploiting a parallel database system as a computation engine. In this demonstration we show that fundamental graph algorithms including all pairs shortest path, single source shortest path, PageRank, triangle counting, connected components and reachability can be solved completely with queries, dynamically generated by our system. Our system presents performance that is very competitive to state-of-the-art graph systems. Furthermore, our out-of-core graph computation can process graphs larger than available main memory, without compromising performance.

KEYWORDS

Graph; Query Processing; DBMS; Parallel; SQL

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. Fueled by the practical importance of graphs, several systems for large graph analysis have emerged. The common wisdom is that demanding graph problems should 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 of an excessive message passing across the cluster [9]. In a recent work, Jindal et al. [6] proposed another graph analytics system using columnar DBMS technology, keeping the vertex-centric approach. Another recent work is EmptyHeaded [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, such as CombBLAS [3]. Highly optimized linear algebra packages (LAPACK, ScaLAPACK) work only when the input matrices fit in RAM.

*Also with Teradata.

CIKM '22, October 17–21, 2022, Atlanta, GA, USA
2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The popular wisdom considers DBMS inadequate for graph processing. In fact, due to inflexibility of the relational model, graph algorithms are hard to program and most of the solutions are inefficient. On the other hand, recently columnar DBMSs have 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 graphs stored in a DBMS. Our system extends previous research on recursive and iterative queries applied to graphs algorithms, as described in [4, 7, 8, 10]. 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. The user is able to run demanding graph algorithms in parallel, remaining isolated from the details of complex C++ programming.

Our contributions can be summarized as follows: (1) We present a powerful graph analytics system that uses parallel DBMSs as a computation engine, implementing complex graphs algorithms with queries (combining joins, aggregations and filters). (2) We improve query performance for graph analytics, by ensuring join with local matching, linear time join, and benefitting from data compression. (3) Our system 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 columnar storage is the DBMS technology that yields the best performance to compute several graph algorithms, being competitive with state-of-the-art graph systems such as Spark-GraphX.

2 SYSTEM DESCRIPTION

2.1 System Overview

Our system exploits parallel clusters [7] 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, the user sent requests using our API. These requests are translated to appropriate SQL (AQL in case of array DBMS).

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 hundredsi of millions users, a typical user is connected to just a few hundreds contacts. Other common examples are links in web pages and flights connecting airports. Therefore, it is reasonable to represent the adjacency matrix E in a sparse storage format, which saves 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 (especially E) are represented as a set of tuples (i, j, v) such that $v \neq 0$, where i and j represent source/destination vertices, and v represents the value (weight) of the edge. Thus, the space complexity of G is determined by $m = |E|$.

2.3 Linear Algebra with Queries

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 semirings [4, 5, 8]. Taking advantage of the sparsity of the graph data and a careful data partitioning, join-aggregation queries execute in parallel with promising performance.

2.4 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 $O(n^2)$.

2.5 System Architecture

Graph algorithms are called from a light-weight client by using our API and computed in the DBMS cluster. Moreover, the data analyst can retrieve results and summaries to a local object; therefore, the analyst can use iteratively Python, R, or Java 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) Retrieve partial, total or summarized results from the database to the local client. (for instance `diag(E)` will retrieve the diagonal of the adjacency matrix, or `rowSums(E)` a summation of its rows).

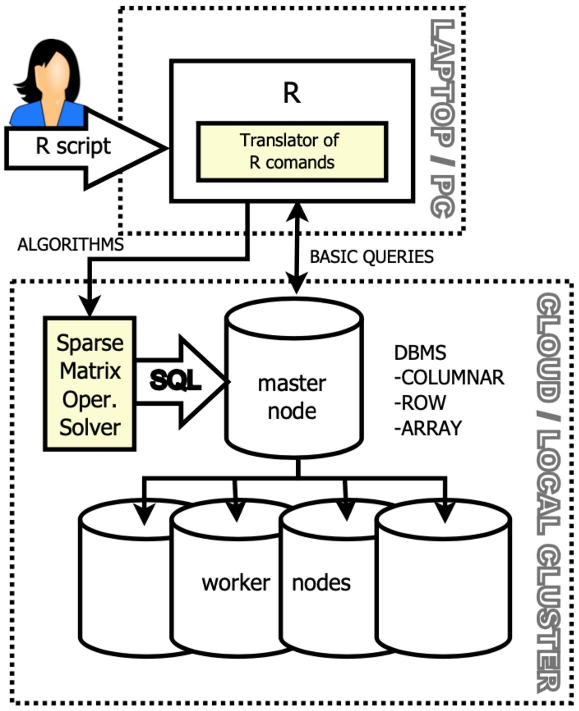

Figure 1: System Architecture. The user sends requests to the DBMS using a data science language (Python or R). Our library requests computations to our Sparse Matrix Operations Solver, which generates SQL queries. Such queries are evaluated by the DBMS.

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 2.6. 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 functions to solve graph algorithms.

2.6 Graph Algorithms Computed with 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 [8] 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 :

$$R_d = \pi_{i,j:sum(E.v*R)}(E \bowtie_{E.i=R_{(d-1)}.j} R_{d-1}) \quad (1)$$

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: $S_d = \pi_{E.i:sum(E.v*S_{d-1}.v)}(E \bowtie_{j=i} S_{d-1})$.

Algorithm 1 is a *pattern* to solve several graph problems with iterative matrix-vector multiplication, computed by relational queries[4]. The algorithm keeps iterating while the value Δ is greater than a small value ϵ. Both Δ and ϵ depend on the specific graph problem. For instance, in PageRank the value Δ is computed as the maximum difference between page rank value of a vertex in the last two iterations, and ϵ is a small value (default or user-defined). This algorithmic pattern works for relational and array DBMSs. 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 one column.
3. The output of the query is inserted in an empty table. We do not do updates. Note $|S_d| \leq n$

```
Data: Table $E$, Table $S_0$ , optional vertexId $s$, $\epsilon$
Result: $S_d$
$d \leftarrow 0$; $\Delta \leftarrow \infty$;
while $\Delta > \epsilon$ do
    $d \leftarrow d + 1$ ;
    $S_d \leftarrow$ query to compute $E \times S_{d-1}$ ;
    compute $\Delta$ ;
end
return $S_d$ ;
```

Algorithm 1: Graph algorithms via matrix-vector multiplication computed with relational queries

2.7 Optimizations in a Parallel DBMS

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

1. Local parallel joins: Rows that satisfy the join condition are always in the same cluster 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 method is key to avoid costly data transfer between nodes.

2. Presorted tables: 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, taking advantage of a merge join algorithm. This is critical for very large graphs. Presorted data is inherent of the array-based data organization.

3. Data Compression: The storage layout in the columnar DBMS uses light weight data compression, which achieves significantly less I/O.

Table 1: Available graph algorithms.

Query	Graph algorithm
Aggregation	In-degree/Out-degree
Aggregation	Graph Density
Aggregation	Degree Distribution
Aggr/Addition	Laplacian Matrix
Join	Transition Matrix
Iterative Matrix × Vec	Single Src Shortest Path
Iterative Matrix × Vec	Reachability
Iterative Matrix × Vec	Connected Components
Iterative Matrix × Vec	PageRank
Recursive Queries	Triangle Counting
Recursive Queries	Triangle Enumeration
Recursive Queries	All pairs Shortest Path
Recursive Queries	Transitive Closure

2.8 Available Graph Algorithms

Table 1 presents the algorithms provided by GALGO, classified by the type of operation in the DBMS. Note that the supported queries 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.

Running Graph Algorithms in the Cluster. Computing graph algorithms just require an API call. Furthermore, a user familiar with SQL may access results from any client, as the resuls of the algorithms are stored in regular tables. This means better interactivity and usability, compared with tools as Giraph or Spark/GraphX.

Interactive Graph Analysis. A useful feature of our system is the ability of access partial results, which are available for all the graph algorithms. As the algorithms are computed in an iterative until specific termination conditions, each iteration generates a partial result in the database, which are inmediately available for exploration. Partial results are available in SQL as well as the supported languages.

3 SYSTEM DEMONSTRATION

3.1 Goals

The objectives of our demonstration are: (1) Showing that graph algorithms can be computed by queries in the parallel cluster. Moreover, mo math libraries are required. (2) Comparing the performance of graphs algorithms in the DBMS vs Spark-GraphX. (3) Understanding how table partitioning (storing the graph edges)

Table 2: PageRank execution time (seconds). Parallel cluster: 4 Nodes with 4GB RAM.

Graph data set	$\|E\|$	Columnar DBMS	Array DBMS	Spark GraphX
web-Google	5M	18	143	58
soc-pokec	30M	72	380	153
LiveJournal	69M	99	1037	477
wikipedia-en	378M	507	stop	crash
WebCommonsc	620M	2764	stop	crash

impacts performance. (4) Solving popular graph problems. (5) Experiencing the easy access to complex graph analytics in a cluster, with straightforward function calls.

3.2 Setup

The DBMSs used in the demonstration are: (a) columnar: Vertica; (b) row: Postgres. Moreover, Spark 2.0 with GraphX will be available during the demonstration session for comparison purposes. Notice Postgres can run in one node only. A set of scripts in the supported programming languages would be available, to make calls to the cluster from a laptop. The database will contain graph data sets from the Stanford SNAP repository. We plan to show analytic tasks on sparse graphs of various sizes (m denotes the number of edges): small, web-Google (m = 5M); medium, soc-LiveJournal1 (m = 68M); large, wikipedia-en (m=378M); extra-large (m =620M). These data sets will be preloaded, but if requested we can demonstrate how to load (import into the DBMS) more data sets.

3.3 Demonstration Scenarios

User is encouraged to run performance test and comparisons between column DBMS, array DBMS and Spark-GraphX. As Postgres runs in only one node, it is meaningless 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 in-database graph algorithms The audience may trigger the computation of in-database algorithms using pre-defined scripts in Python, R or Java. Nevertheless

The audience may try a set of scripts in Python, R and Java to compute any of the graph algorithms provided. The user may get results via the API, or the results can be queried directly in the database system.

Looking under the hood: Users interested in technical aspects may browse the history of queries generated by our system, and query plans. This history is accessible from the R environment by functions from our library: queryHistory() and planHistory().

Experiencing flexible and interactive analytics:

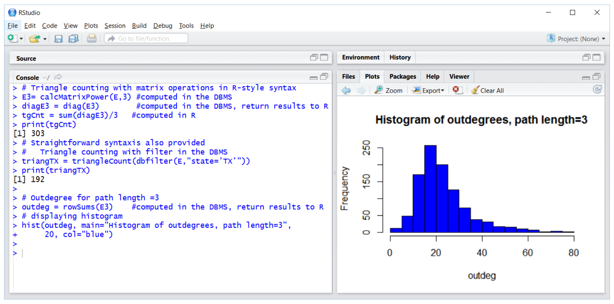
```
> # Triangle counting with matrix operations in R-style syntax
> E3= calcMatrixPower(E,3) #computed in the DBMS
> diagE3 = diag(E3)        #computed in the DBMS, return results to R
> tgCnt = sum(diagE3)/3   #computed in R
> print(tgCnt)
[1] 303
> # Straightforward syntaxis also provided
> #   Triangle counting with filter in the DBMS
> triangTX = triangleCount(dbfilter(E,"state='TX'"))
> print(triangTX)
[1] 192
>
> # Outdegree for path length =3
> outdeg = rowSums(E3)    #computed in the DBMS, return results to R
> # displaying histogram
> hist(outdeg, main="Histogram of outdegrees, path length=3",
+      20, col="blue")
>
> 
```

Figure 2: Graph analytics using R as a front-end. Demonstration of triangle counting and visualization of a graph outdegree

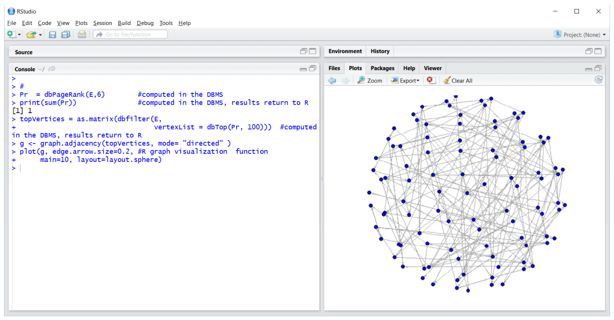
```
>
> #
> Pr  = dbPageRank(E,6)        #computed in the DBMS
> print(sum(Pr))               #computed in the DBMS, results return to R
[1] 1
> topVertices = as.matrix(dbfilter(E,
+                                vertexList = dbTop(Pr, 100)))  #computed
in the DBMS, results return to R
> g <- graph.adjacency(topVertices, mode= "directed" )
> plot(g, edge.arrow.size=0.2, #R graph visualization  function
+      main=10, layout=layout.sphere)
> 
```

Figure 3: PageRank computation, and visualization of the subgraph comprised by the top 100 vertices.

The audience is encouraged to computed analytics on the graph datasets, as vertex counting, edges counting, in-degree and outdegree by vertices. This operations can be done to a complete data set, or only to interesting vertices, based on custom conditions. SQL-savvy users may also try filters, projection an joins with the results, directly in the DBMS.

REFERENCES

[1] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16. ACM, 2016.

[2] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst. *Templates for the solution of algebraic eigenvalue problems: a practical guide*, volume 11. Siam, 2000.

[3] A. Buluc and J. R. Gilbert. The combinatorial blas: Design, implementation, and applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, Nov. 2011.

[4] W. Cabrera and C. Ordonez. Unified algorithm to solve several graph problems with relational queries. In *Proc. AMW Workshop*, 2016.

[5] W. Cabrera and C. Ordonez. Scalable parallel graph algorithms with matrix-vector multiplication evaluated with queries. *Distributed and Parallel Databases*, 35(3-4):335–362, 2017.

[6] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker. Vertexica: your relational friend for graph analytics! *Proceedings of the VLDB Endowment*, 7(13):1669–1672, 2014.

[7] C. Ordonez, S. T. Al-Amin, and X. Zhou. A simple low cost parallel architecture for big data analytics. In *IEEE International Conference on Big Data,*, pages 2827–2832, 2020.

[8] C. Ordonez, W. Cabrera, and A. Gurram. Comparing columnar, row and array dbmss to process recursive queries on graphs. *Information Systems*, 2016.

[9] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "think like a vertex" to "think like a graph". *Proc. VLDB Endow.*, 7(3):193–204, Nov. 2013.

[10] X. Zhou and C. Ordonez. Computing complex graph properties with SQL queries. In *2019 IEEE International Conference on Big Data*, pages 4808–4816, 2019.