Matrix Multiplication with SQL Queries for Graph Analytics

Xiantian Zhou*
University of Houston§
USA

Carlos Ordonez
University of Houston§
USA

***Abstract*—Analyzing large data sets are challenging. Most data analytics research has proposed parallel algorithms that outside a DBMS because SQL is considered inadequate for complexity computations. R and Python are popular analysis systems that provide a vast collection of mathematical models and functions. However, they are limited by main memory and single computer. Recently, parallel DBMSs have significantly improved query processing performance. Moreover, SQL queries are elegant and efficient. This paper introduces a novel system architecture integrating a popular analysis system and parallel DBMSs, which has the matrix multiplication involving a large matrix evaluated inside a parallel DBMS and complex mathematical computations are done in R or Python. Many graph problems can be solved by matrix multiplication. In this paper, we show optimized queries which perform matrix multiplication in DBMSs to solve two fundamental graph problems, single-source reachability and transitive closure.**

***Index Terms*—Matrix Multiplication, SQL Queries, Graph Analysis**

I. Introduction

Big data analytics is a major challenge, characterized by volume, variety and velocity of data, gaining interest in data warehousing research. And Graph analytics remains one of the most computationally intensive tasks in big data analytics.DBMSs are the main data management platform, while R and Python are the most popular systems to perform data analysis due to their ample library of models, powerful data transformation operators, interpreted and interactive language. Many research progress on efficient analytic algorithms work outside a DBMS on flat files, which frequently are exported from a DBMS. However, R and Python are slow to analyze large data sets, especially when data do not fit in RAM. Moreover, exporting data sets from a DBMS is slow and redundant. Even some packages in R or Python enable the parallel processing, but they generally require re-programming existing functions and operators, which limits their impact. More importantly, they cannot compete with a DBMS in speed, parallel speedup and robustness when data size exceeds RAM. On the other hand, columnar DBMSs can largely improve SQL queries' performance involving joins and aggregations used for matrix multiplication. Many graph algorithms can be expressed as matrix multiplication or their solution can be derived or approximated from matrix multiplication. Thus, we present a system that combines popular systems such as R and Python with DBMSs. While the data fitting in RAM are processed in Python or R, the large matrix multiplication are evaluated by operators optimized for the underlying DBMS architecture.

In our system, the matrix multiplication of large graphs, which is the most computationally intensive part of many graph algorithms, is done in a DBMS instead of using R or Python libraries. We also study how to do matrix multiplication and matrix-vector multiplication using queries. Moreover, we analyze how to optimize those queries.

II. Definitions

Let $G = (V, E)$ be a directed graph with $n = |V|$ vertices, where V is a set of vertices. The adjacency matrix of G is a $n \times n$ matrix such that the cell i, j holds 1 when exists an edge from vertex i to vertex j, while 0 otherwise. From a database perspective, graph G is stored in a table E as a list of edges (adjacency list). Let table E be defined as $E(i, j, v)$ with primary key (i, j) representing the source and destination vertices and v representing a numeric value e.g. cost/distance. In this work, we solve graph analytics via matrix multiplication with E.

III. Matrix Multiplication in SQL

Our system's main idea is to compute matrix multiplication inside a parallel DBMS if there is a matrix product involving a large matrix. The structure of our system is shown in Fig.1. This idea is to provide a valuable guideline to split computations between R and a DBMS. It also helped us decide where to optimize algorithms. The DBMS is a parallel system

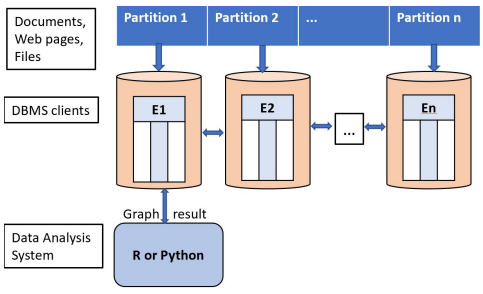

Fig. 1. An overview of our system

§Department of Computer Science, University of Houston, Houston TX 77204, USA
*Contact author:xiantianzhou@gmail.com

with N nodes under a shared-nothing architecture. The data analysis system runs on a separate server or one of the N nodes, under the assumption that any computation involving a large matrix is always evaluated inside the DBMS using queries. That is, a large matrix is not exported. Numerically intensive matrix numerical methods are solved in R or Python, but on the result matrix. As explained before, optimized SQL queries that are used to perform matrix multiplication are sent to the DBMS and the result matrix is stored in the DBMS. Then another query fetches a subset of the result matrix from the DBMS to R or Python, leaving it in RAM. A program at the client computer connects to the DBMS and it maintains a reference (table name) to each matrix in the DBMS. Then, the program has the ability to evaluate matrix multiplication with its syntax. In general, the large data set is stored with the native storage mechanism of the parallel DBMS, whereas small matrices that can fit in RAM are transferred to R for further processing.

Our system takes care of translating matrix multiplication in the data analysis systems into SQL queries for the relational DBMS. In the end, the result matrix is imported back into the data analysis system. Our queries are standard SQL queries, they can be used in any parallel DBMS.

Many graph algorithms can be expressed as matrix multiplication or their solution can be derived or approximated from matrix multiplication (Dijkstra's shortest path, minimum spanning tree, maximum flow) [3]. For example, the matrix product $E \cdot 1$, where 1 is an n-dimensional column vector, helps finding the vertices with highest connectivity (i.e.hubs in a network). The matrix product $E \cdot E \cdot E$ can be used to get all triangles in G, which has been identified as an important primitive operation to solve many more complex graph problems. The iteration $E \cdot E \ldots E$, multiplying E k times gets all paths of length k, from which we can filter the shortest/longest ones and count them. Finally, $E \cdot E \cdot ...E$ ($n-1$ times) until a partial product vanishes is a demanding computation returning $G+$, the transitive closure (reachability) of G, which gives a comprehensive picture about G connectivity [2].

A. Matrix-Vector multiplication

Graph problems such as single-source reachability can be solved by multiplying a sparse vector S iteratively by E, as shown in the following [1].

$$S_k = (E^T)^k \cdot S_0 = E^T \cdots (E^T \cdot (E^T \cdot S_0)) \qquad (1)$$

The $\cdot$ is the regular matrix multiplication and S_0 is a vector such that: $S_0[i] = 1$ when i is the source vertex, and 0 otherwise. Accordingly, the SQL queries for this problem is shown in the following:

```
/*single-source reachability*/
SELECT E.j as j, sum(E.v * S.v) as v
FROM E JOIN S AS E.i=S.i
GROUP BY E.j;
```

Optimization: For the matrix-vector multiplication, we can replicate the vector S on each working node in the DBMS system, since the size of S is $O(n)$, which is small. But we need to partition the table E across the DBMS system. Since S is copied to each node, no message passing is required when performing the query. But, if we need to iterate the query, the vector S is summarized and repeated across the DBMS system after each join.

B. Matrix-Matrix multiplication

Matrix-matrix multiplication can also solve many graph problems. The $E \cdot R$ can be expressed as a join and aggregation query as shown below.

```
SELECT DISTINCT (E.i AS i,R.j AS j)
FROM E JOIN R AS E.j=R.i
GROUP BY E.i, R.j;
```

Similarly, you can also use the operator $sum()$, $min()$, or aggregations in the queries according to different graph algorithms.

Optimization: To optimize this query, we need to do an identical partition. Implementing an identical partition allows joins to occur locally on each node, thereby helping to reduce data movement across worker nodes during query processing. We partition table E and R by the join key. In the query we presented, the edges having the same destination vertex in table E are partitioned in the same node with the corresponding vertices in R since the join condition $E.j = R.i$. Thus, no data movement is required to perform the query. If the graph algorithm needs to repeat the join query, the temporal result table needs to be re-partitioned across each node.

Based on the optimized queries, we can solve many graph algorithms that can be expressed as matrix multiplication or their solution can be derived or approximated from matrix multiplication by using the queries we presented together with $sum(), min()$ or $distinct$.

In general, we present a system that combine popular data analysis languages R and Python with DBMSs. While the small data that can fit in RAM are processed in Python or R, the large matrix multiplication is evaluated by queries optimized for the underlying DBMS architecture. SQL will remain the main DB query language. And queries are short and elegant. Moreover, columnar DBMS enables fast multiplication of sparse matrix with no RAM limitation. We can split computations between R or Python and DBMSs SQL queries, without mess the analysis system with query plans. For the future work, we will combine the C++ with Python. The matrix multiplication is processed with C++ language in a distributed system, the further and complicate analysis is performed in Python.

REFERENCES

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

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

[3] Zhou, X., Ordonez, C.: Computing complex graph properties with SQL queries. In: 2019 IEEE International Conference on Big Data. pp. 4808–4816 (2019)