Big Data Analytics: Exploring Graphs with Optimized SQL Queries

Sikder Tahsin Al-Amin[1(✉)], Carlos Ordonez[1], and Ladjel Bellatreche[2]

[1] University of Houston, Houston, USA
sal-amin2@uh.edu
[2] LIAS/ISAE-ENSMA, Poitiers, France

Abstract. Nowadays there is an abundance of tools and systems to analyze large graphs. In general, the goal is to summarize the graph and discover interesting patterns hidden in the graph. On the other hand, there is a lot of data stored on DBMSs that can be potentially analyzed as graphs. External graph data sets can be quickly loaded. It is feasible to load data quickly and that SQL can help prepare graph data sets from raw data. In this paper, we show SQL queries on a graph stored in relational form as triples can reveal many interesting properties and patterns on the graph in a more flexible manner and efficient than existing systems. We explain many interesting statistics on the graph can be derived with queries combining joins and aggregations. On the other hand, linearly recursive queries can summarize interesting patterns including reachability, paths, and connected components. We experimentally show exploratory queries can be efficiently evaluated based on the input edges and it performs better than Spark. We also show that skewed degree vertices, cycles and cliques are the main reason exploratory queries become slow.

Keywords: Graph · Parallel DBMS · SQL

1 Introduction

Within big data analytics graph problems are particularly difficult given the size of data sets, the complex structure of the graph (density, shape) and the mathematical nature of computations (i.e. graph algorithms). In graphs analytics, the goal is to obtain insight and understanding of complex relationships that are present in the graph. Large-scale graphs have been widely applied in many emerging areas. Graphs can be represented in terms of database perspective. However, processing large graphs in large scale distributed system has not received much attention in DBMS using relational queries. Recent works on graphs offer vertex-centric query interface to express many graph queries [5], compares columnar, row and array DBMSs for recursive queries [13]. There are some libraries [9,17] available for large graph processing. Also, it has been established that columnar DBMS perform better than array DBMS or GraphX for certain kind of graph queries [4].

© Springer Nature Switzerland AG 2018
M. Elloumi et al. (Eds.): DEXA 2018 Workshops, CCIS 903, pp. 88–100, 2018.
https://doi.org/10.1007/978-3-319-99133-7_7

Relational database systems remain the most common technology to store transactions and analytical data, due to optimized I/O, robustness and security control. Even though, the common understanding is that RDBMSs cannot handle demanding graph problems, because relational queries are not sufficient to express important graphs algorithms and a poor performance of database engines in the context of graph analytics. Consequently, several graph databases and graphs analytics systems have emerged, targeting large data sets, especially under the Hadoop/MapReduce platform. First, we analyze graphs stored on a DBMS. Using SQL query, we can reveal many interesting properties from the graph like *indegree* and *outdegree*, highly connected components, potential paths, isolated vertices, triangles and so on. Then we focus on the evaluation of queries with linear recursion, which solve a broad class of difficult problems. Using recursion, we can answer questions like reachability, detecting cycles, adjacency matrix multiplication, paths and so on. We perform our experiments on columnar parallel DBMS with shared nothing architecture. While our queries can work in any DBMS, it is experimentally that proven that columnar and array DBMSs present performance substantially better than row DBMSs for graphs analysis [13]. Also, parallel database systems have a significant performance advantage over Hadoop MR in executing a variety of data-intensive analysis benchmarks [14].

2 Definitions

This is a reference section which introduces definition of a graph from mathematical and database perspective, recursive queries and our problem definition. Each subsection can be skipped by a reader familiar with the material.

2.1 Graph

Let $G = (V, E)$ be a directed graph with $n = |V|$ vertices and $m = |E|$ edges. An edge in E links two vertices in V and has a direction. Our definition allows the presence of cycles and cliques in graphs. A cycle is a path which starts and ends at the same vertex. A clique is a complete subgraph of G. The adjacency matrix of G is a $n \times n$ matrix such that the cell i, j holds a 1 when exists an edge from vertex i to vertex j.

From database perspective, graph G is stored in relation (table) E as a list of edges (adjacency list). Let, relation 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. A row from relation E represents the existence of an edge. In summary, from a mathematical point of view E is a sparse matrix and from a database perspective, E is a long and narrow table having one edge per row. For example, if we have city names as graphs, then i and j will be city names and v can be the distance or travel cost between them. However, if we have phone calls as a graph, then E will have edges per person pair, not one edge per phone call or message.

2.2 Recursive Queries

Recursive queries are used here to analyze graphs. Let, relation E is the input for a recursive query using columns i and j to join E with itself multiple times. Let R be the resulting relation returned by a recursive query, defined as $R(d, i, j, v)$ with primary key (d, i, j), where d represents recursion depth, i and j identify an edge at some recursion depth and v represents the numeric value. We include v in both E and R to have consistent definitions and queries. We study queries of the form: $R = R \cup (R \bowtie E)$, where the result of $R \bowtie E$ gets added to R itself.

2.3 Problem Definition

In this paper, we study how to express the computation of several graph algorithms with queries and how we can optimize those queries. We solve from simpler to harder graph problems with relational queries.

3 Exploring Graphs

In this section, we discuss the main contribution of our work. First we discuss how we can optimize relational queries and then use the optimized queries to solve from simpler to harder graph problems.

3.1 Optimizing Recursive Queries

Pushing GROUP-BY and Duplicate Elimination: First, we review our previous optimization technique to optimize recursive queries [13]. This optimization corresponds to the classical graph problem (reachability). $E \bowtie E$ may produce duplicate vertex pairs. We can compute a GROUP BY aggregation on E, grouping rows by edge with the grouping key i, j. A byproduct of a GROUP BY aggregation is that duplicates get eliminated in each intermediate table. Therefore, a SELECT DISTINCT query represents a simpler case of a GROUP BY query.

In recursive queries, the fundamental question is to know if it is convenient to wait until the end of recursion to evaluate the GROUP BY or it is better to evaluate the GROUP BY during recursion. Considering that a GROUP BY aggregation is a generalization of the π operator extended with aggregation functions, this optimization is equivalent to pushing π through the query tree, like a traditional SPJ query. In relational algebra terms, the unoptimized query is $S = \pi_{d,i,j,sum(v)}(R)$. And the optimized query is given below. Therefore, it is unnecessary to perform a GROUP BY or DISTINCT in the final S. Hence, the GROUP BY we used in queries discussed later, they all follow the optimized version.

$$S = \pi_{1,i,j,sum(v)}(R_1) \cup \pi_{2,i,j,sum(v)}(R_2) \ \cup ... \tag{1}$$

Partitioning and Maintaining Duplicate Table E: While performing self joins ($E \bowtie E$), it is more feasible to maintain a duplicate copy of E (E_d). As we are self joining on $E.j = E.i$, we can maintain a duplicate version of E and partition one E by i column and other E by j column to optimize the join computation. Partitioning capability of a DBMS divides one large table into smaller pieces based on values in one or more columns. Partitions can improve parallelism during query execution and enable some other optimizations. The graph should be partitioned in such a way that uneven data distribution and costly data movement across the network is avoided. The latter is possible when the parallel join occurs locally on each worker node. Partitioning provides opportunities for parallelism during query processing. There is a distinction between partitioning at the table level and segmenting a projection. Table partitioning segregates data on each node for fast data purges and query performance. And segmenting distributes projection data across multiple nodes in a cluster. Hence, the join query will have better performance.

Here, we perform the partitioning by vertex. All the neighbors of a vertex are stored on the same machine. Our assumption is that there is not one high degree vertex but there are a few high degree vertices. This number is greater than the number of machines in the cluster. If high degree vertices are less than the number of machines, queries will be slow.

Encoding: This optimization method is limited to columnar DBMS only. There are many encoding types in columnar databases. The default encoding is ideal for sorted, many-valued columns such as primary keys. It is also suitable for general purpose applications for which no other encoding or compression scheme is applicable. Encoding options in DBMS include run length encoding (RLE), which replaces sequences (runs) of identical values in a column with a set of pairs, where each pair represents the number of contiguous occurrences for a given value: (occurrences, value). RLE is generally applicable to a column with low-cardinality, and where identical values are contiguous. However, the storage for RLE and AUTO encoding of CHAR/VARCHAR and BINARY/VARBINARY is always the same.

3.2 Building Graph Summaries with Queries

Finding Indegree and Outdegree: The *outdegree* of a vertex v is the number of outgoing edges of v and the *indegree* of v is the number of incoming edges of v. We can get the *indegree* and *outdegree* of each vertices from a graph. This can help to answer queries like which is the most visited page on the web or who is the most popular person in a social network. The following two SQL queries find the *indegree* and *outdegree* respectively. These queries are $\mathbf{1} * E$ and $E * \mathbf{1}$ meaning matrix-vector multiplication.

```
SELECT j AS nodes, COUNT(j) AS indegree
FROM E
GROUP BY j;
```

```
SELECT i AS nodes, COUNT(i) as outdegree
FROM E
GROUP BY i;
```

Using these, we can also detect the source-only or destination-only vertices. That is, the vertices with 0 as *indegree* are source-only and vertices with 0 as *outdegree* are destination-only vertices. However, the above queries only select the vertices who have *indegree* or *outdegree*. To get source-only or destination-only vertices we have to find *indegree* and *outdegree* for all vertices.

Counting and Enumerating Triangles: As triangles are subsets of cliques, it is fundamental to understand connectivity. For instance, triangle count in a network is used to compute transitivity, an important property for understanding graph evolution over time. Triangles are also used for various other tasks completed for real-life networks, including community discovery, link prediction, and spam filtering. It is possible to detect and count how many triangles are presented in the graph using relational queries. We can detect the number of triangles by performing join operations on E ($E \bowtie E \bowtie E$) where each join uses $E.j = E.i$. Here, we are performing a self-join twice. It is a very costly operation and makes the queries slower. As discussed above, we can optimize the query performance by maintaining a duplicate version of E and partition them based on vertices. Then, if E_D is the duplicate version of E, the join operation will be $E \bowtie E_D \bowtie E$.

Exploring Paths:

(a) Potential Number of Paths: We can get the potential number of paths from a graph. We can use the information from *indegree* and *outdegree* discussed above. The number of paths (P) passing through a certain vertex is the multiplication of its *indegree* and *outdegree*. The Eq. 2 gives all the paths that can be generated from a graph including cycles. However, if we use GROUP BY or DISTINCT to eliminate duplicate vertex pairs, the number of paths generated will be much less than the one generated by this equation. And paths may also represent connections, for instance, chemical compounds, or subparts of a part, that is, not necessarily distances/cost.

$$\begin{aligned} potential\ P = \sum (indegree(\forall i) \times outdegree(\forall i)) \\ \text{where } indegree_i = outdegree_i \end{aligned} \tag{2}$$

(b) Top K Connectivity Vertices: Highly connected vertices mean the vertices to which most edges are connected. In other words, these vertices are responsible to generate more paths. Therefore, we can get them from *indegree* and *outdegree*. The highest connected vertex will be $max(indegree(i) \times outdegree(i))$. And the vertices whose both *indegree* and *outdegree* are 0 are isolated vertices in the graph. We can also get top K highest or lowest connected vertices in such way.

(c) Exploring Paths: We can find all the vertices starting from vertex u or check if there exists a path between vertex u and v. To find all the reachable vertices starting from u, we can first filter E on $E.i = u$ and store the vertices on another table/relation P. Then to find the reachable vertices, we can perform join $P \bowtie E$ on $P.j = E.i$. This will give reachable vertices of path length 3. To get all the reachable vertices with different path lengths we can repeatedly set P to $P \cup (P \bowtie E)$ on the same Join condition as mentioned before. The union operation will keep only the identical vertex pairs. We can also check if there is a connection between two vertices u and v with this method until a row with $P.j = v$ arrives. As we are filtering rows at first, P is much smaller than E, this bound to have benefits in query time. Also, we can apply the pushing GROUP BY optimization as discussed earlier and eliminate duplicates at each round.

Using this method, we can also find the vertices where distance is below/above a certain value. We only populate P where the value of v column is below/above a threshold value.

Finding Connected Components: A weakly connected component of a directed graph G is a subgraph G' such that for any vertices $u, v \in G$, exists an un-directed path between them. We can compute with the SPJA query for matrix-vector multiplication (join between two tables and aggregation). This is an improvement of HCC, an iterative algorithm proposed in [7]. In contrast with this, we avoid the second join, necessary to find the minimum value for each entry of the new and the previous vector. We propose inserting an artificial self loop in every vertex; by setting $E(i, i) = 1$, for every i. At each iteration, the resulting vector will compute $S_d \leftarrow \pi_{j:min(E.v*S.v)}(E \bowtie_{j=i} S_{d-1})$. The algorithm stops when the current vector is equal to the second. The relational query is presented below.

```
INSERT INTO S1
SELECT E.i, min (S0.v*1)v
FROM E JOIN S0 on S0.i=E.j
GROUP BY E.i;
```

Adjacency Matrix Multiplication: In our work, we multiply adjaceny matrix to get the transitive closure of a graph using recursive queries. The standard and most widely used algorithm to evaluate a recursive query comes from deductive databases and it is called Seminaive [2,3]. Let R_k represent a partial output relation (table) obtained from $k-1$ self-joins with E (input relation) as operand k times, up to a given maximum recursion depth k: $R_k = E \bowtie E ... \bowtie E$. Here, each join uses $E.j = E.i$ and is a matrix-matrix multiplication. The general form of recursive join is $R_{d+1} = R_d \bowtie_{R_d.j=E.i} E$, where the join condition $R_d.j = E.i$ links a source vertex with a destination vertex if there are two edges connected by an intermediate vertex. Assuming graph E mentioned in Sect. 2 as input, the π computes $d = d + 1$, $i = R_d.i$, $j = E.j$ and $v = R_d.v + E.v$ at each iteration.

$$R_{d+1} = \pi_{d,i,j,v}(R_d \bowtie_{R_d.j=E.i} E) \quad (3)$$

The final result is the union of all the partial results: $R = R_1 \cup R_2 \cup ... \cup R_k$ for recursive depth k. This query evaluation stops when R_d becomes empty at some iteration because there are no rows to satisfy the condition then. Applying pushing GROUP BY optimization as discussed earlier helps this query to perform faster as we are eliminating duplicates at each round.

4 Experimental Evaluation

In this section, we provide an overview of how we conducted our experiments and our findings. First, we discuss how we set up the experimental parameters and then we discuss our experimental outcome. We perform comparisons of our queries with Spark in parallel machines and present the results.

4.1 Experimental Setup

DBMS Software and Hardware: We conducted experiments on an eight node cluster each with Intel Pentium(R) Quadcore CPU running at 1.60 GHz, 8 GB RAM, 1 TB disk, 224 KB L1 cache, 2 MB L2 cache and Linux Ubuntu 14.04 operating system. So for parallel computation, total RAM size is 64 GB and total disk space is 8 TB and 32 cores. We used Vertica [8], a columnar DBMS supporting ANSI SQL to execute the queries. However, our queries are standard SPJ queries and will work on other RDBMSs too. We used Python as the host language to generate SQL queries and submit the queries to the databases/systems as it is faster than JDBC. We compared our results with Spark-GraphX which is used for graph-parallel computation in Spark.

Data Sets: We used both synthetic and real graph data sets summarized in Table 1. For synthetic graph data sets, we generated graphs with varying complexity. We generated graphs with varying clique sizes using uniform distribution where clique sizes increase either linearly (cliqueLinear) or geometrically (cliqueGeometric). For cliqueLinear data set, we generate the cliques with sizes $2, 3, 4...$ and for cliqueGeometric data set we generated the graphs with sizes $2, 4$,8.. and so on. For real data sets we used from the Stanford SNAP repository. Both real data sets have a significant number of cliques and medium diameter. All the time measurements are taken as the average of running each query five times and excluding the maximum and minimum value.

4.2 Parallel Graph Summarization

In DBMS, we calculate the *indegree* for each vertex using the relational query mentioned above. Spark-GraphX includes in its library an implementation of *indegree* as graph operators. It also includes an implementation to find the max *indegree*. Table 2 shows that Spark takes a lot of time to calculate the *indegree* for all data sets. As the relational query in DBMS is very simple and does not require any join or costly operations, it is very fast and executes within seconds.

Table 1. Data sets.

Name	Type	n	m	Density
tree10m	Synthetic	10M	10M	Sparse
wiki-vote	Real	8k	103.6k	Sparse
cliqueLinear	Synthetic	48.5k	10M	Dense
cliqueGeometric	Synthetic	2047	1.5M	Dense
web-Google	Real	875k	5.1M	Dense

For counting triangles, we perform experiments both with and without optimization. As we are performing self-join while counting triangles using relational queries, we can accelerate the performance with the optimizations discussed above. Hence we used maintaining duplicate E while partitioning both tables based on columns and encode using RLE. Spark-GraphX includes in its library an implementation of counting triangles. Table 3 shows the results of counting triangles using relational queries in DBMS with and without optimization and in Spark. Spark loses to DBMS whether we perform the optimization or not. However, with our optimized method, it takes almost half the time in DBMS than the normal method. Partitioning and maintaining duplicate E speeds up the performance of join operation and the queries execute faster.

In Table 4, we see the results of what happens when we eliminate duplicate results from each step of recursive queries. We try to find all the reachable vertices from a particular vertex up to path length 6. Using GROUP BY at each recursive step eliminates the duplicates and thus accelerates the performance. However, for tree10m data set, it performs better when we do not eliminate the duplicates. Here, the total number of paths generated in both methods are same for this data set, so doing an extra GROUP BY operation in each recursive step is taking time. As for other methods, if we perform the optimization with GROUP BY, it works better. The number of paths in each recursive step grows when there are many duplicate paths. We stop the program each time after 10 min and put "Stop" on the table. We see the other methods could not finish calculating all the paths when there is no GROUP BY optimization. Also, storage requirements can grow exponentially for dense graphs. However, Spark performs better in this cases. Path reachability requires a union operation at each iteration. As Spark computes the union operation in main memory, it has a significant advantage over DBMS to perform this kind of operations.

We also perform finding the connected components in both DBMS and Spark. For DBMS, we used recursive relational queries as mentioned in the previous section. For Spark-Graphx, it includes in its library an implementation of Connected Components similar to HCC, propagating minimum vertex-ids through the graph. From Table 5, we see the time it takes for both systems. Dataset tree10m takes most time for both systems as it does not have any connected component. So the program iteratively checks for connected components until there are no vertices left. However, for all data sets, DBMS performs better than Spark.

Table 2. Time to compute indegree (in seconds).

Data set	DBMS	Spark
tree10m	1.4	8.4
wiki-vote	0.1	6.1
cliqueLinear	0.1	7.5
cliqueGeometric	0.2	8.5
webGoogle	0.6	20.4

Table 3. Time to enumerate triangles (in seconds).

Data set	DBMS (no optimization)	DBMS (with optimization)	Spark
tree10m	6.5	3.1	59.2
wiki-vote	0.9	0.4	16.4
cliqueLinear	5.1	2.3	27.9
cliqueGeometric	41.8	22.9	51.7
webGoogle	7.0	2.1	142.7

Table 4. Path reachability for path length 6: pushing GROUP BY (in seconds).

Data set	Not pushing GROUP BY	Total paths	pushing GROUP BY	Total paths	Spark
tree10m	11.7	126	48.9	126	2.8
wiki-vote	Stop	-	2.3	2316	2.0
cliqueLinear	Stop	-	3.4	145	1.4
cliqueGeometric	Stop	-	4.0	1025	1.7
webGoogle	39.4	2898192	12.3	7083	2.3

Table 5. Time to compute connected components (in seconds).

Data set	DBMS	Spark
tree10m	45.1	649.7
wiki-vote	4.3	12.9
cliqueLinear	2.2	21.7
cliqueGeometric	2.5	22.4
webGoogle	30.2	60.3

For adjacency matrix multiplication, we performed all the optimization techniques. As it is a matrix-matrix multiplication, the result will also be a matrix. Table 6 shows the results of performing the multiplication using relational queries in DBMS and Spark-GraphX. We can accelerate the performance using optimizations discussed above. When we multiply consecutively, the time difference between two methods become more significant. Using GROUP BY eliminates duplicates from next multiplication and matrix size becomes smaller. Also, partitioning based on columns improves the join performance. For Spark-GraphX, we slightly modify the implementation of transitive closure in its library to perform the multiplication as many times as we need. DBMS performs better than Spark-GraphX in every cases. We stop the program each time after 10 min.

Table 7 shows output for some queries discussed above. For each data set, it shows the maximum *indegree* and expected number of paths to be generated. Fig. 1 shows the maximum number of *indegree* and *outdegree* generated per recursion depth of recursive queries while calculating the adjacency matrix multiplication.

Table 6. Adjacency matrix multiplication time (in seconds).

Data set	Multiply once			Multiply 2 times		
	Without optimization	With optimization	Spark	Without optimization	With optimization	Spark
tree10m	6.7	7.2	181.1	12.2	18.4	408.3
wiki-vote	2.3	1.9	25.9	68.5	8.9	68.3
cliqueLinear	31.7	3.9	73.8	Stop	7.6	138.2
cliqueGeometric	358.1	18.2	486.4	Stop	36.3	Stop
webGoogle	26.1	23.2	535.2	286.9	120.5	Stop

Table 7. Exploring graphs.

Data set	Edges	Max *Indegree*	Expected paths
tree10m	10M	1	10M
wiki-vote	103.6k	457	4.54M
cliqueLinear	10M	311	2333.79M
cliqueGeometric	1.5M	1024	1224.34M
webGoogle	5.1M	6326	60.68M

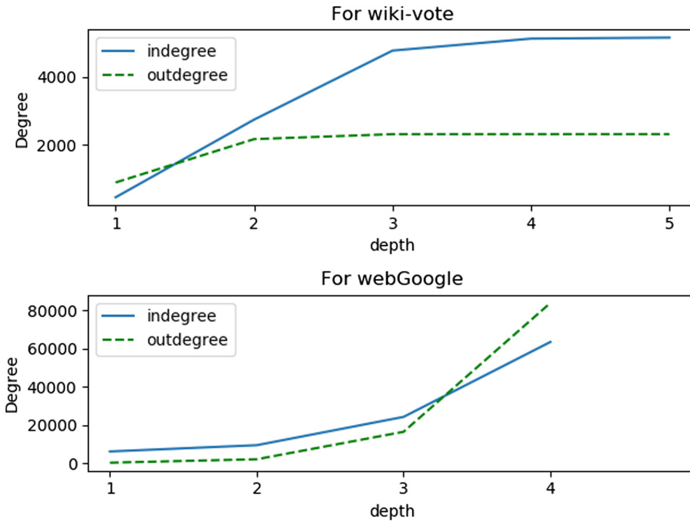

Fig. 1. Maximum number of *indegree* and *outdegree* per adjacency matrix multiplication for webGoogle and wiki-vote data sets

5 Related Work

Research on recursive queries and parallel computation is extensive, especially in the context of deductive databases [1,3,18,21,23] or adapting deductive database techniques to relational databases [11,12,22]. There is significant theoretical work on recursive query computation in the Datalog language [10,16]. More recently, [18] proposed a hybrid approach to query graphs with a language similar to SPARQL, maintaining data structures with graph topology in main memory and storing graph edges as relational tables like we do. This work considers reachability queries (i.e. transitive closure), but not their optimization on modern DBMSs. There is research [13] on comparing row, columnar and array DBMSs for higher data volume that shows columnar DBMS has advantage over row and array DBMSs in many cases.

Although numerous applications are related to graphs, querying from large graphs using relational queries stored on a DBMS has not received much attention. In [24], the authors revisited the issue of how RDBMS can support graph processing at the SQL level. To support graph processing, they proposed new relational algebra operations. Another recent work have focused on retrieving paths using a path query language from a network graph where they present the Nepal query language [6]. In [20], they presented a formalization of graph pattern matching for Gremlin queries. Gremlin [15] is a graph traversal language and machine, provides a common platform for supporting any graph computing system. The Boost Graph Library (BGL) [17] provides some general purpose graph classes. The graph algorithms in BGL [17] includes most popular graph searching and shortest path algorithms. Stanford Network Analysis Platform

(SNAP) [9] is a general purpose, high-performance system, and graph mining library that easily scales to massive networks with billions of nodes and edges. Other recent works on graphs offer vertex-centric query interface to express many graph queries [5] and study compression techniques for indexing regular path queries in graph languages [19].

6 Conclusions

We can perform many queries in a very short time on a graph stored on a DBMS. Large-scale graphs can be quickly loaded and analyzed in parallel systems within a very short time. We represented the graph in terms of database perspective and stored them as a form of triplets. SQL queries are good enough for common graph problems like reachability, vertex degree, triangles, isolated vertices, expected paths from the graph, adjacency matrix multiplication, and so on. The queries reveal many interesting properties and patterns from the graph. We proposed several optimization methods on the queries. Optimizing the query performance helps to execute the queries faster than usual. Our experimental results show that our queries perform better than Spark-GraphX in most cases. However, there are some graph problems that cannot be solved with SQL queries like detecting planarity, traveling salesman problem, finding all cliques and hierarchical graph summarization.

For future work we have the following: detecting shortest cycles from a graph, counting maximal cliques meaning the highest number of vertices connected each other, showing actual paths when possible (for low number of paths), parallel speed up meaning how the queries perform when we vary the number of machines and discovering more complex patterns beyond paths. Moreover, we also plan to optimize the algorithms in Spark.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases : The Logical Level, Facsimile edn. Pearson Education POD, Boston (1994)
2. Agrawal, R., Dar, S., Jagadish, H.: Direct and transitive closure algorithms: design and performance evaluation. ACM TODS **15**(3), 427–458 (1990)
3. Bancilhon, F., Ramakrishnan, R.: An amateur's introduction to recursive query processing strategies. In: Proceedings of ACM SIGMOD Conference, pp. 16–52 (1986)
4. Cabrera, W., Ordonez, C.: Scalable parallel graph algorithms with matrix–vector multiplication evaluated with queries. Distrib. Parallel Databases **35**(3–4), 335–362 (2017)
5. Jindal, A., Rawlani, P., Wu, E., Madden, S., Deshpande, A., Stonebraker, M.: VERTEXICA: your relational friend for graph analytics!. Proc. VLDB Endow. **7**(13), 1669–1672 (2014)
6. Johnson, T., Kanza, Y., Lakshmanan, L.V.S., Shkapenyuk, V.: Nepal: a path query language for communication networks. In: Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics, NDA 2016, pp. 6:1–6:8 (2016)

7. Kang, U., Tsourakakis, C.E., Faloutsos, C.: PEGASUS: a peta-scale graph mining system implementation and observations. In: Proceedings of the 2009 Ninth IEEE International Conference on Data Mining, ICDM 2009, pp. 229–238 (2009)
8. Lamb, A., et al.: The Vertica analytic database: C-store 7 years later. Proc. VLDB Endow. **5**, 1790–1801 (2012)
9. Leskovec, J., Krevl, A.: SNAP datasets: stanford large network dataset collection, June 2014. http://snap.stanford.edu/data
10. Libkin, L., Wong, L.: Incremental recomputation of recursive queries with nested sets and aggregate functions. In: Cluet, S., Hull, R. (eds.) DBPL 1997. LNCS, vol. 1369, pp. 222–238. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-64823-2_13
11. Mumick, I., Finkelstein, S., Pirahesh, H., Ramakrishnan, R.: Magic Conditions. ACM TODS **21**(1), 107–155 (1996)
12. Mumick, I., Pirahesh, H.: Implementation of magic-sets in a relational database system. In: ACM SIGMOD, pp. 103–114 (1994)
13. Ordonez, C., Cabrera, W., Gurram, A.: Comparing columnar, row and array DBMSs to process recursive queries on graphs. Inf. Syst. **63**, 66–79 (2016)
14. Pavlo, A., et al.: A comparison of approaches to large-scale data analysis. In: Proceedings of ACM SIGMOD Conference, pp. 165–178 (2009)
15. Rodriguez, M.A.: The Gremlin graph traversal machine and language (invited talk). In: Proceedings of the 15th Symposium on Database Programming Languages, DBPL 2015, pp. 1–10 (2015)
16. Seshadri, S., Naughton, J.: On the expected size of recursive Datalog queries. In: Proceedings of ACM PODS Conference, pp. 268–279 (1991)
17. Siek, J., Lee, L.Q., Lumsdaine, A.: Boost c++ libraries. https://www.boost.org/
18. Sakr, S., Elnikety, S., He, Y.: Hybrid query execution engine for large attributed graphs. Inf. Syst. **41**, 45–73 (2014)
19. Tetzel, F., Voigt, H., Paradies, M., Lehner, W.: An analysis of the feasibility of graph compression techniques for indexing regular path queries. In: Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES 2017, pp. 11:1–11:6 (2017)
20. Thakkar, H., Punjani, D., Auer, S., Vidal, M.-E.: Towards an integrated graph algebra for graph pattern matching with Gremlin. In: Benslimane, D., Damiani, E., Grosky, W.I., Hameurlain, A., Sheth, A., Wagner, R.R. (eds.) DEXA 2017 Part I. LNCS, vol. 10438, pp. 81–91. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-64468-4_6
21. Ullman, J.: Implementation of logical query languages for databases. ACM Trans. Database Syst. **10**(3), 289–321 (1985)
22. Valduriez, P., Boral, H.: Evaluation of recursive queries using join indices. In: Expert Database Systems, pp. 271–293 (1986)
23. Youn, C., Kim, H., Henschen, L., Han, J.: Classification and compilation of linear recursive queries in deductive databases. IEEE TKDE **4**(1), 52–67 (1992)
24. Zhao, K., Yu, J.X.: All-in-one: graph processing in RDBMSs revisited. In: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD, pp. 1165–1180 (2017)