PandaSQL: Parallel Randomized Triangle Enumeration with SQL Queries

Abir Farouzi
ISAE-ENSMA
Poitiers, France

Ladjel Bellatreche
ISAE-ENSMA
Poitiers, France

Carlos Ordonez
University of Houston
USA

Gopal Pandurangan
University of Houston
USA

Mimoun Malki
Ecole Supérieure en Informatique
Sidi Bel Abbes, Algeria

ABSTRACT

Triangles are an important pattern in large-scale graph analysis for their practical use in many real-life applications. However, with the expansion of networks, maintaining a balanced computational load is challenging especially for problems like triangle computations because of skewed vertices. On the other hand, there is a huge amount of data in database management systems (DBMSs) that can be modeled and analyzed as graphs. With these motivations in mind, we developed PandaSQL, a novel approach using *SQL queries* to enumerate all the triangles in a given graph based on *Randomized Triangle Enumeration Algorithm.* Our approach is elegant, abstract, and short compared to traditional languages like C++ or Python. Moreover, our partitioning queries ensures a perfect load balancing. Thus, the triangle enumeration is independent, local, and parallel.

CCS CONCEPTS

• **Information systems → Relational parallel and distributed DBMSs**; **Structured Query Language**.

KEYWORDS

DBMS, Parallelism, Triangle Enumeration, Graphs, Query Language.

ACM Reference Format:
Abir Farouzi, Ladjel Bellatreche, Carlos Ordonez, Gopal Pandurangan, and Mimoun Malki. 2020. PandaSQL: Parallel Randomized Triangle Enumeration with SQL Queries. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20), October 19–23, 2020, Virtual Event, Ireland.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3340531.3417429

First author is also affiliated to Ecole Supérieure en Informatique, Sidi Bel Abbes, Algeria.

1 INTRODUCTION

Larger networks are being generated day after day as the world is more interconnected than before. Examples include real-world networks such as social, transportation, Web, and biological networks. Hence, graph problems including *triangle enumeration* problem are becoming even more difficult to solve since the current networks present larger size, more complex structure with different degree distributions, and the existence of complex topological structures like cycles or cliques. Taking this into account, our work focuses on enumerating triangles on large graphs inside DBMS using SQL queries inspired by the Randomized Triangle Enumeration Algorithm [7].

Analyzing graph in DBMS received scant intentions, this can be explained by the fact that relational DBMSs don't define graph concepts. Nevertheless, some work like [2, 8, 9] add an additional layer to DBMS that can deal with graph concepts while the processing remains inside DBMS. Another, more practical reason, not to use relational DBMSs for graph analysis is the emergence of powerful parallel graph engines in the Hadoop big data ecosystem like Neo4j and Spark GraphX. However, these systems are standalone requiring significant efforts to link and integrate information about the graph. Unlike DBMSs that enable direct and transparent integration using SPJA (selection, projection, join,aggregation) operations. In addition, most of graph data are stored in non-graph databases,thus they are not explicitly stored as graphs. Moreover, analyzing large networks in relational databases outside DBMSs using traditional programming languages or graph-based systems, users need to export data to external files which is considerably slow, thereby going through performance bottleneck while losing data management capabilities provided by DBMSs including query processing, security, concurrency control, and fault tolerance.

In this perspective, our approach PandaSQL[1] based on SQL queries to enumerate graph triangles presents an inside DBMS solution that exploits all the benefits and functionalities of DBMSs while performing graph analysis. Indeed, our approach feats the distributed DBMS partitioning strategy to partition the graph represented as edge table in such a way that eliminates the data movement between cluster hosts during the triangle computations. Hence, ensuring that the triangle enumeration step remains local,

[1]PandaSQL can be found at: https://forge.lias-lab.fr/projects/sql4triangle. A beginner video demonstration via an intuitive GUI is available at: https://youtu.be/pwcYkOUV8_s and an expert video with detailed queries is available at: https://youtu.be/78Dd0rnMR4Q

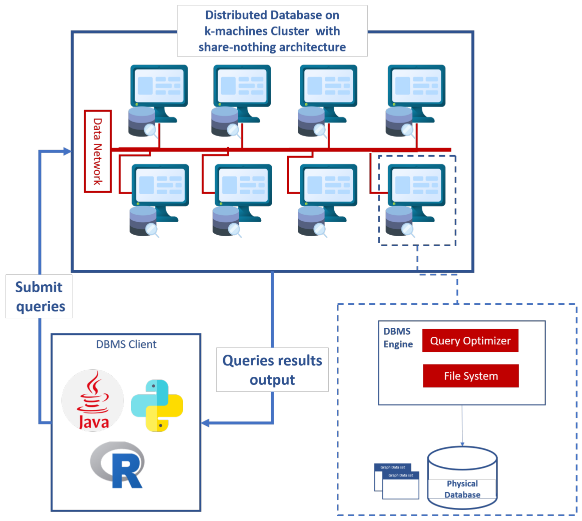


Figure 1: System Architecture

parallel, and independent from the other hosts. In this demonstration, we opt for columnar DBMS giving its fast query processing, however, our queries should work perfectly on row DBMS as well.

2 SYSTEM DESCRIPTION

This work is a continuation of previous research efforts targeting to implement different graph analysis algorithms using queries, among others triangle counting problem [1]. Besides, optimizing linear recursive queries, which has triangles as a particular case of the transitive closure of the input graph [5, 6].

In the following, we start by presenting the global architecture of the system, then we define the graph in database perspectives and we give an overview of our approach to enumerate the triangles in a given graph G. Finally, we present the different query optimizations leading to the best performances.

2.1 System Architecture

Our system consists of k-machines model; a popular theoretical model for large-scale distributed graph computation [4], where k is the size of the cluster, built over share-nothing architecture. Each machine can communicate directly with the others by message passing (no shared memory).

In order to achieve a perfect load balancing, the number of machines k must be chosen as shown in equation (1):

$$k = p^3/p \in \mathbb{N} \tag{1}$$

Fig. 1. depicts an overview of our system. Initially, a client (Python, Java,..) submits the query to the distributed DBMS via its corresponding database connector API. The host receiving the query becomes an Initiator node while the others remain Executor nodes. The initiator node picks an execution plan and shares it with the executors then all the hosts perform the query (Data can be exchanged between hosts by message passing depending on the query and data distribution). Finally, the result of the query is output to the client for presentation.

2.2 Graph Definitions

In our system, a given graph G is stored as an adjacency list in an edge table $E\_s(i, j)$ with primary key (i, j) representing source vertex i and destination vertex j. Each tuple of table E_s defines the existence of an edge. If the graph is directed, the edge table E_s holds the set of directed edges. Otherwise, for each tuple (i, j) inserted in the edge table E_s, (j, i) is also inserted.

2.3 Randomized Triangle Enumeration

The high-level idea behind our approach is to partition the set of vertices of the input graph in such a way that eliminates data movement during the task of triangle enumeration, by ensuring a perfect balancing of the workload. Full details can be found in [3].

PandaSQL consists of two fundamental steps: (i) graph partitioning and (ii) local triangle enumeration.

2.4 Graph Partitioning

In this step, the vertices set of the input graph is randomly partitioned into $k^{1/3}$ subset of $n/k^{1/3}$ each, then each of the k machines examines the edges between pairs of subsets of one of the $(k^{1/3})^3 = k$ possible triplets of subsets of vertexes. In order to achieve this partitioning, our approach consists of the following sub-steps:

Initially, the input graph is imported to the edge table $E\_s(i, j)$ and a small table called $Triplet(machine, color1, color2, color3)$ is created, it holds all the k possible triplets of color subsets, where each triplet is attributed to one unique machine among the k machines. The table $Triplet$ is replicated on each machine in order to accelerate the local join. Then, each vertex of the input graph picks independently and uniquely at random one color from $k^{1/3}$ distinct colors. Each vertex and its picked color are stored in a table called $V\_s(i, color)$ that allows to send each edge to a random proxy machine. This is done by a double join between table V_s and edge table E_s. The resulting output is stored in proxy table $E\_s\_proxy(i, j, i\_color, j\_color)$. Sending edges to proxies is the most important step in our approach because all the partitioning step depends on it. Finally, each machine collects the edges that may form a triangle according to the color triplet assigned to it from the proxies. As result, the table $E\_s\_local(machine, i, j, i\_color, j\_color)$ is created by performing join and union operations between E_s_proxy and $Triplet$. The E_s_local table is partitioned on the cluster hosts so that all the edges belonging to the same machine are stored on one single machine among the k hosts, which eliminate having multiple copies of edges replicated over the cluster.

2.5 Local Triangle Enumeration

The previous step ensures having all possible formed triangles hosted, according to their end-vertices, on their corresponding machines. This partitioning ensures having a balanced data distribution and workload. Therefore, the current step becomes local, independent, and parallel. Indeed, local triangle enumeration is performed by double local self-join on the table E_s_local since all required edges to form triangles on each machine are available on it (no data movement). Fig. 2. illustrates an example of our approach on cluster of 8 hosts.

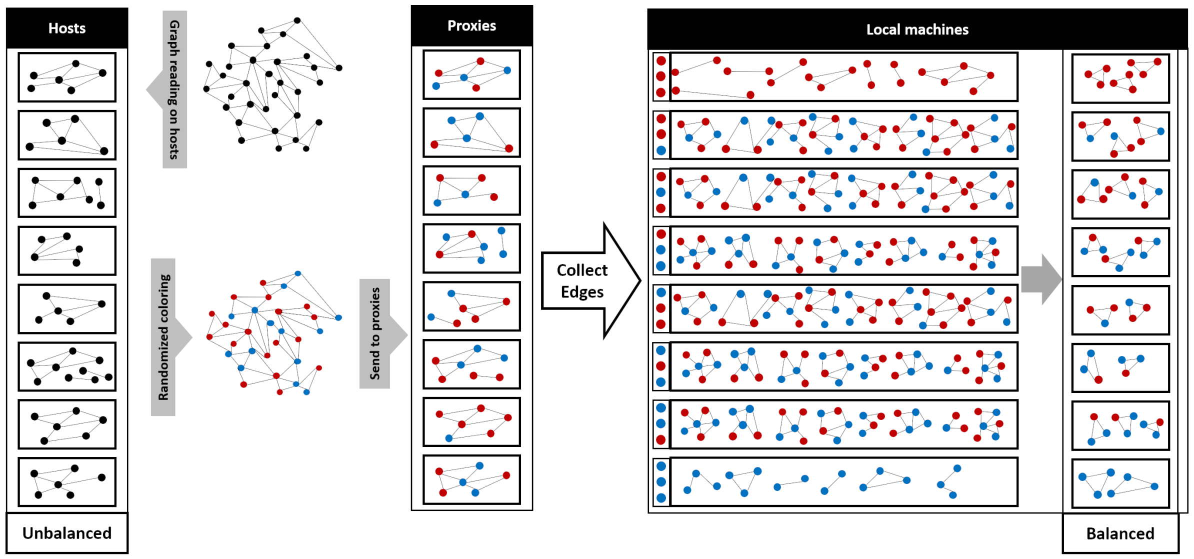


Figure 2: Randomized Triangle Enumeration on Cluster of 8 hosts

2.6 Optimization

As we chose columnar DBMS for our demonstration, we emphasize that setting up the following optimisations can speed up the query running time and help us to reach the desired workload balancing. These optimizations are applied during the DDL and they are less portable.

2.6.1 Compression: Data compression is a technique used by columnar DBMS to reduce the storage required to save the contents. We used run-length encoding (RLE) which replaces a sequence of identical values in a column with a set of pairs. Each pair represent the number of contiguous occurrences for a given value <value, occurrence>.

2.6.2 Small table replication: replicating small tables on each host of the cluster represents a good optimization step. It ensures the availability of required data on each host, thus, exclude having data movement during join operations. We replicate the table $Triplet$ since it has only 8 tuples. This speeds up the execution time of the join between $Triplet$ and E_s_proxy in the edge collecting by local machine sub-step.

2.6.3 Projections: are optimized collections of table columns that provide physical storage for data. They can contain some or all of the columns of one or more tables. Projections in columnar DBMS are equivalent of indexes in row DBMS. Columnar DBMS creates automatically projection for each created table, however, the definition of projections can be done manually to satisfy some requirements. In the previous section, projections were specifically defined for two main reasons: (i) to replicate the Triplet table on each host of the cluster and (ii) to send each edge to it corresponding machine in the edge collecting from proxies sub-step.

3 SYSTEM DEMONSTRATION

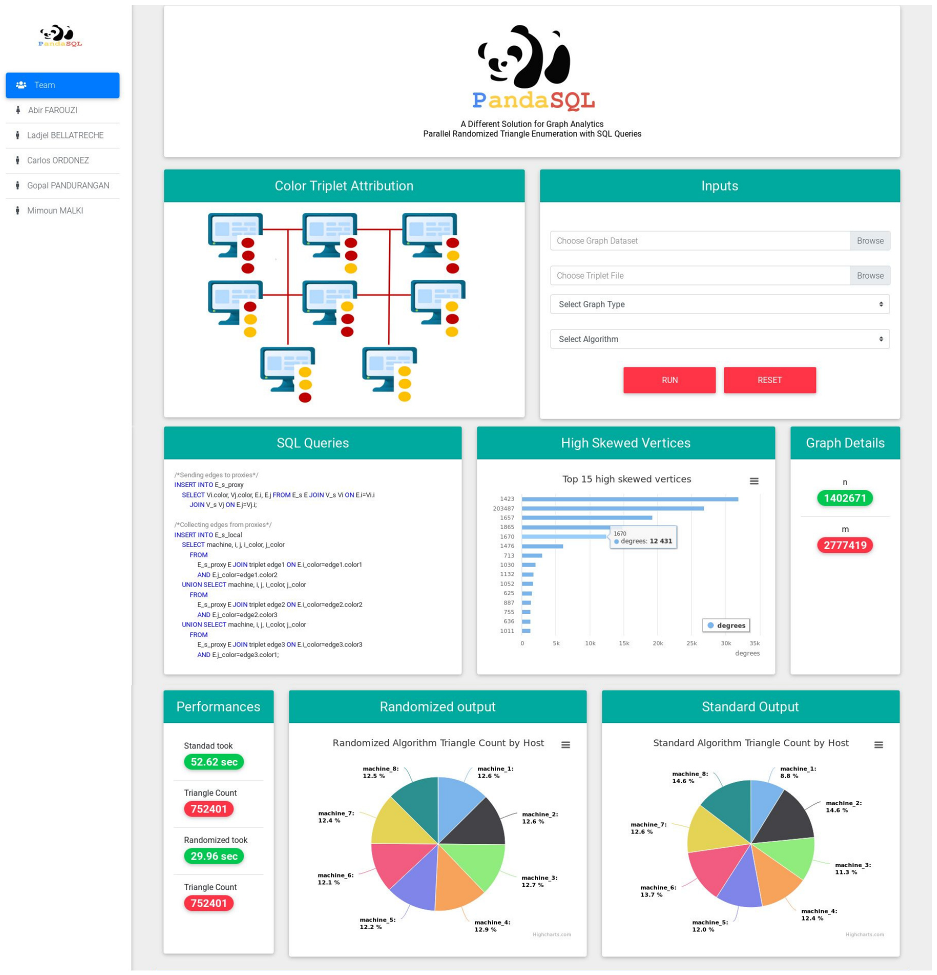
3.1 Main Points

As explained in the introduction, enumerating triangles is fundamental to solve more complex graph problems, but the triangles themselves may not be interesting to the end user, especially if there is a large number of triangles (e.g. many groups of 3 people). Therefore, the goal of our system demonstration will be to illustrate how a clever randomized distributed algorithm can work on top of a traditional DBMS, evaluating SQL. We hope our system can open the possibility of programming other randomized algorithms with queries.

Our main goals, going from logical database aspects to physical and processing aspects, are the following: (1) an overview of the parallel computation model, contrasting it with respect to the number of machines (processing nodes); (2) highlighting advantages of SQL over C++ or Java (slower, but more scalable and getting benefit from DBMS features); (3) understanding SQL table definitions; (4) explaining how color triplets are assigned to machines; (5) understanding edge distribution imbalance when there exist highly skewed vertices; (6) explaining why a distributed join does not scale to a large number of machines; (7) explaining why a traditional SQL queries may hit a bottleneck with skewed vertices; (8) explaining why a random assignment of edges re-balances workload to solve the triangle enumeration.

3.2 Demonstration Overview

Our scenario is directed to both beginner and expert researchers who want to learn how randomized algorithms can work in SQL. As shown in Figure 3, in an intuitive GUI the user will choose a graph to analyze, where this graph is stored as a list of edges (i, j). The system will display m (# of edges) and n (# of vertices). Then the system will display "top" skewed vertices in descending order of degree (say 15 or less). Then, to solve triangle enumeration, the user will choose between the traditional SQL queries or the new randomized queries. We will provide medium-sized graphs that



```sql
/*Sending edges to proxies*/
INSERT INTO E_s_proxy
  SELECT Vi.color, Vj.color, E.i, E.j FROM E_s E JOIN V_s Vi ON E.i=Vi.i
    JOIN V_s Vj ON E.j=Vj.i;

/*Collecting edges from proxies*/
INSERT INTO E_s_local
  SELECT machine, i, j, i_color, j_color
    FROM
      E_s_proxy E JOIN triplet edge1 ON E.i_color=edge1.color1
      AND E.j_color=edge1.color2
  UNION SELECT machine, i, j, i_color, j_color
    FROM
      E_s_proxy E JOIN triplet edge2 ON E.i_color=edge2.color2
      AND E.j_color=edge2.color3
  UNION SELECT machine, i, j, i_color, j_color
    FROM
      E_s_proxy E JOIN triplet edge3 ON E.i_color=edge3.color3
      AND E.j_color=edge3.color1;
```

Figure 3: PandaSQL Main GUI

can be analyzed in less than one minute (n=10k). After evaluating the queries the user will visualize dynamically created pie charts showing edge distribution among the machines (i.e. workload). For the traditional solution, the user will be able to see skewed vertices across machines that will slow down join computation (larger intermediate results). In contrast, for the randomized solution, the user will see edges involving skewed vertices are shuffled and then the number of edges per vertex on each machine is approximately the same. Moreover, the user will understand why the distributed join with the randomized edges assignment is faster.

For experienced users, who understand query processing and DBMS internals, we will provide a more technical overview of our randomized solution. We will explain: why the number of machines must be a cube of the number of colors (i.e. in our demo 2 colors, hence 8 machines), potential speedup as k grows (8,27,64, and so on), efficient access to edges in tables (by index lookup or binary search), how tables are physically partitioned by chosen columns (critical for balanced workload), the actual balanced edge distribution after edges are sent to the proxy machine, why the two randomized edges join produce intermediate tables that are uniformly partitioned as well, how joins can indeed work locally, and finally why triangle enumeration does not require edge redistribution at the end. To round up this scenario, we will explain why columnar DBMSs have a performance edge over row DBMSs for graph analytics.

REFERENCES

[1] S. T. Al-Amin, C. Ordonez, and L. Bellatreche. 2018. Big Data Analytics: Exploring Graphs with Optimized SQL Queries. In *DEXA Workshops*. 88–100.

[2] J. Fan, A. G. S. Raj, and J. M. Patel. 2015. The Case Against Specialized Graph Analytics Engines.. In *CIDR*.

[3] A. Farouzi, L. Bellatreche, C. Ordonez, G. Pandurangan, and M. Malki. 2020. A Scalable Randomized Algorithm for Triangle Enumeration on Graph based on SQL Queries. In *DaWaK*.

[4] H. Klauck, D. Nanongkai, G. Pandurangan, and P. Robinson. 2015. Distributed Computation of Large-scale Graph Problems. In *ACM-SIAM SODA*. 391–410.

[5] C. Ordonez, , W. Cabrera, and A. Gurram. 2017. Comparing columnar, row and array DBMSs to process recursive queries on graphs. *Information Systems* 63 (2017), 66–79.

[6] C. Ordonez. 2010. Optimization of Linear Recursive Queries in SQL. *IEEE TKDE* 22, 2 (2010), 264–277.

[7] G. Pandurangan, P. Robinson, and M. Scquizzato. 2018. On the Distributed Complexity of Large-Scale Graph Computations. In *SPAA*. 405–414.

[8] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. T. Xie. 2015. SQLgraph: An efficient relational-based property graph store. In *ACM SIGMOD*. 1887–1901.

[9] Y. Tian, E. K.Xu, W. Zhao, M. H. Pirahesh, S. Tong, W. Sun, T. Kolanko, M. S. H. Apu, and H. Peng. 2020. IBM Db2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2. ACM, 345–359.