

# A Scalable Randomized Algorithm for Triangle Enumeration on Graphs based on SQL Queries

Abir Farouzi<sup>1,3</sup>, Ladjel Bellatreche<sup>1</sup>, Carlos Ordonez<sup>2</sup>, Gopal Pandurangan<sup>2\*</sup>,  
and Mimoun Malki<sup>3</sup>

<sup>1</sup> ISAE/ENSMA, France

(abir.farouzi, bellatreche)@ensma.fr

<sup>2</sup> University of Houston, USA

carlos@central.uh.edu, gopal@cs.uh.edu

<sup>3</sup> ESI-SBA, Algeria

m.malki@esi-sba.dz

**Abstract.** Triangle enumeration is a fundamental problem in large-scale graph analysis. For instance, triangles are used to solve practical problems like community detection and spam filtering. On the other hand, there is a large amount of data stored on database management systems (DBMSs), which can be modeled and analyzed as graphs. Alternatively, graph data can be quickly loaded into a DBMS. Our paper shows how to adapt and optimize a randomized distributed triangle enumeration algorithm with SQL queries, which is a significantly different approach from programming graph algorithms in traditional languages such as Python or C++. We choose a parallel columnar DBMS given its fast query processing, but our solution should work for a row DBMS as well. Our randomized solution provides a balanced workload for parallel query processing, being robust to the existence of skewed degree vertices. We experimentally prove our solution ensures a balanced data distribution, and hence workload, among machines. The key idea behind the algorithm is to evenly partition all possible triplets of vertices among machines, sending edges that may form a triangle to a proxy machine; this edge redistribution eliminates shuffling edges during join computation and therefore triangle enumeration becomes local and fully parallel. In summary, our algorithm exhibits linear speedup with large graphs, including graphs that have high skewness in vertex degree distributions.

**Keywords:** Inside DBMS Solution · Parallel Triangle Enumeration · Graph · Columnar DBMS.

## 1 Introduction

Large graphs are becoming pervasive as the world is more interconnected than before. Examples include real-world networks such as social, transportation,

---

\* G. Pandurangan was supported, in part, by NSF grants IIS-1633720. CCF-1717075, and CCF-1540512.

Web, and biological networks. One of the fundamental graph problems is triangle enumeration, which has attracted much interest because of its numerous practical applications, including the analysis of social processes in networks [26], dense subgraph mining [25], joins in databases [16], etc. The interested reader may refer to [7, 5] for additional applications. Triangle detection and triangle counting are also well-studied problems, and potentially significantly easier than triangle enumeration. However, we emphasize that for many applications, including all the aforementioned ones, triangle detection, and triangle counting are not enough, and a complete enumeration of all the triangles is required.

Graphs can be modeled in terms of a database perspective. DBMSs provide a different angle of graph processing since they offer an easily load/import data into them. Nonetheless, processing large graphs in DBMSs did not receive much interest as long as DBMSs did not define graph concepts. Some recent studies offer support for the vertex-centric query interface to express graph queries like Pregel and its open-source successor Giraph [15]. Other works such as [18, 6] study and compare different graph problems on row, array and columnar DBMSs with Spark GraphX. These works showed that DBMSs are faster.

In our work, we present an adaption of a parallel randomized algorithm [19] to solve the triangle enumeration problem. We prove that our approach guarantees the load balancing between the processors. We study how to express this algorithm using *standard* SQL queries that can be executed on any DBMS. We discuss various possible optimizations in order to obtain the optimum execution time while using columnar DBMS to execute the algorithm.

Our paper is organized as follows. Section 2 states an overview of related works. Preliminary concepts and notations including graph, triangle enumeration problem, columnar DBMS and parallel computational model are described in section 3. In section 4, we present the standard algorithm and its limitations; then we detailed our proposed randomized algorithm while discussing its complexity and load balancing. We introduce our experimental findings in section 5 and we conclude in section 6 with general remarks and potential future works.

## 2 Related work

In this section, we summarize the most relevant, state of the art, triangle enumeration works. We start by highlighting numerous applications related to graph processing using relation queries on DBMSs. Then, we present an overview of triangle enumeration with a brief description of recent studies.

Processing graphs in DBMSs have been studied in recent years. The work of [28] revisited graph processing support in the RDBMS at SQL level. [18] studied the optimization of recursive queries on two complementary graph problems: transitive closure and adjacency matrix multiplication. The authors experimentally proved that the columnar DBMS is the fastest with tuned query optimization. [6] studied how to solve four important graph problems: reachability, single-source shortest path, weakly connected components and page rank using relational queries on columnar DBMS, array DBMS and spark’s GraphX on share-nothing architecture. Other works like [23, 10] stored graphs in relational tables with schema optimized for graph queries by adding a specific layer

supporting graph processing on RDBMS. Other interesting work of processing graph in DBMSs can be found at [17, 2, 9]. Moreover, there exist powerful parallel graph engines in the Hadoop big data ecosystem like Neo4j and Spark GraphX, but they require significant effort to link and integrate information about the graph and, in general, they provide query languages inferior as compared to SQL. In contrast, our system can readily exploit diverse relational tables, which can be easily loaded and integrated with the graph edge table. Our SQL solution provides good time performance, but it does not intend to be the fastest compared to Hadoop systems. On the other hand, our goal is to provide perfect load balancing, which ensures scalability as more parallel processing nodes are available. A detailed benchmark comparison is left as future work.

The foundational algorithms for enumerating triangles are the node iterator [21] and the edge iterator [12] suitable for one host execution. Nonetheless, with the expansion of graph size, they become less efficient, and one host processing on the main memory is infeasible. Some works like MGT [11] and Trigon [8] use one host processing but with better I/O techniques which reduce the overhead caused by the I/O access. Other works focus on paralleling the processing and present multi-core solutions like [14, 22]; the first presented a load balance guarantee and the second proposed a cache-friendly algorithm supporting dynamic parallelism without tuning parameters. Many distributed works have been also introduced, [4] proposed MPI-based distributed memory parallel algorithm based on node iterator for counting triangles in massive networks that can be easily adapted for triangle enumeration. [27] is another approach based on distributing the processing over a cluster while reducing messages during run-time. On the other hand, many solutions have been explicitly addressed in the MapReduce framework by [1, 24, 20, 29]. These solutions paralleled the processing through two rounds of MapReduce where the first focuses on finding all the wedges and the second checks whether there is an edge connecting each wedge endpoints. However, those solutions are time-costly because of the large amount of intermediate data exchanged between the hosts during processing.

The work of [19] presented a randomized distributed algorithm for triangle enumeration and counting in the  $k$ -machine model, a popular theoretical model for large-scale distributed graph computation [13]. In this model,  $k \geq 2$  machines jointly perform computations on graphs with  $n$  nodes (typically,  $n \gg k$ ). The input graph is assumed to be initially partitioned among the  $k$  machines in a balanced fashion. Communication is point-to-point by message passing (no shared memory), and the goal is to minimize the number of communication rounds of the computation. The work of [19] presented a distributed algorithm that enumerates all the triangles of a graph in  $\tilde{O}(m/k^{5/3} + n/k^{4/3})$  rounds (the  $\tilde{O}$  notation hides a *polylog*( $n$ ) multiplicative and additive factor), where  $n$  and  $m$  are the number of nodes and edges of the input graph respectively. It also showed that this algorithm is essentially optimal with respect to the number of communication rounds by presenting a lower bound that showed that there exist graphs with  $m$  edges where any distributed algorithm requires  $\tilde{\Omega}(m/k^{5/3})$  rounds. The current work builds on the algorithm of [19] and shows how to

modify and efficiently implement this algorithm on a DBMS with queries which is significantly different from a fairly straightforward MPI implementation of the algorithm in a traditional language such as C++ or Python.

### 3 Preliminaries

This is a reference section that introduces definitions of a graph from mathematics and database perspectives, our problem definition, an overview of distributed processing using columnar DBMS, and a description of the computational model.

#### 3.1 Graph

Let  $G = (V, E)$  be an undirected unweighted graph with  $V$  non-empty set of vertices (nodes) and  $E$  a possibly empty set of edges. We denote  $n = |V|$  and  $m = |E|$ . Each edge  $e \in E$  links between two vertices  $u, v$  and defines a direction (from  $u$  to  $v$  and  $v$  to  $u$ ). We denote for each  $u \in V$ ,  $N(u) = \{v \in V : (u, v) \in E\}$  the set of neighbors of a vertex  $u$ . Thereby, the degree of  $u$  is defined as  $deg(u) = |N(u)|$ .

By this definition, we allow the presence of cliques and cycles. A clique defines a complete sub-graph of  $G$ . A cycle is a path that starts and ends on the same vertex. A cycle of length  $l$  is called  $l$ -cycle; hence a 3-cycle refers to a triangle.

Mathematically, a graph  $G$  can be represented by an adjacency matrix of  $n \times n$  (see Fig.1 (b)), where the cell  $i, j$  holds 1 if there is an edge combining vertex  $i$  to vertex  $j$ . In database perspective, a graph  $G$  is stored as adjacency list in an edge table  $E(i, j)$  with primary key  $(i, j)$  representing source vertex  $i$  and destination vertex  $j$  (see Fig.1 (c)). An entry in table  $E$  defines existence of an edge. Fig. 1 (a) depicts an undirected graph, (b) shows its adjacency matrix representation and (c) its adjacency list representation.

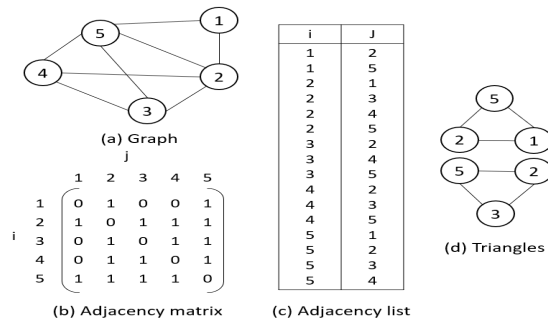


Fig. 1: Graph representations

#### 3.2 Triangles properties

**Definition 1.** Given a graph, a connected triple  $(u, v, w)$  at vertex  $v$  is a path of length 2 for which  $v$  is at the center. If  $(w, u) \in E : (u, v, w)$  is a closed triple called triangle otherwise it is open triple named wedge or open triad. A triangle contains three closed triples:  $(u, v, w)$ ,  $(v, w, u)$  and  $(w, u, v)$ .

By definition 1, we allow the enumeration of triangles on both directed and undirected graph as triangle is a cycle of 3 edges.

Besides triangle enumeration problem, the problem of finding *open triads* has many applications, e.g., in social networks, where vertices represent people, edges represent a friendship relation, and open triads can be used to recommend friends.

**Definition 2.** A triangle denoted as  $\Delta_{(u,v,w)}$ , is the smallest clique in graph  $G$  composed of three distinct vertices  $u, v$  and  $w$ . The triangle formed by these vertices include the existence of three edges  $(u, v)$ ,  $(v, w)$  and  $(w, u)$ . The set  $\Delta(G)$  includes all the triangles  $\Delta_{(u,v,w)}$  of the graph  $G$ . Fig. 1 (d) represents an example of triangles in an undirected graph (Fig1. (a)). Notice that the two examples of produced triangles are in lexicographical order. This eliminates the listing of the triangle multiple times.

**Definition 3.** Two triangles  $\Delta_1$  and  $\Delta_2$  may belong to the same clique. Fig. 1 (d) shows that both enumerated triangles  $\Delta_{(1,2,5)}$  and  $\Delta_{(2,3,5)}$  belong to the same clique formed of vertices  $\{2,3,4,5\}$ .

For a given graph  $G$ , triangle enumeration problem is to list all the unique triangles present in the set  $\Delta(G)$  which is expensive compared to counting because enumeration tests the possibility of each triplet of edges to form a triangle. Therefore, using the results of the enumeration task, one can easily obtain the count of triangles in the graph. In contrast, just counting does not necessarily give the list of resulting triangles [3].

Notably, in practice, most graphs are sparse and therefore  $m = O(n)$ . However, detecting embedded triangles is computationally hard with time  $O(n^3)$ .

### 3.3 DBMS storage

In order to enumerate triangles, standard SQL queries can be employed based on SPJ operations (selection, projection and join). These operations can be useful in simplifying and understanding the problem by formulating the solution using relational algebra ( $\sigma$ ,  $\pi$  and  $\bowtie$ ) then translating it into SQL queries that can be executed in parallel on any distributed DBMS. To handle a graph in a database system, the best storage definition is a list of edges as edge table  $E(i, j)$  where it is assumed the edge goes from  $i$  to  $j$ . If the graph is undirected, we include two edges for each pair of connected vertices. Otherwise, only the directed edge is inserted. In this manner, we can always get a triangle vertices in order  $\{u, v, w\}$  instead of  $\{v, w, u\}$  or  $\{w, u, v\}$  with  $u < v < w$ .

Fig. 2 depicts physical schema for a graph in DBMS, where the table  $E_s$  represents the adjacency list of the graph, it holds all the edges. While, the table ‘User’ stocks all the information related to vertices ( $i$  and  $j$ ) of table  $E_s$ .

We opted for columnar DBMSs such as Vertica or MonetDB to execute our queries, since they present better efficiency of writing and reading data to and from disk comparing to row DBMSs. Thus, speeding up the required time to perform a query [18]. In fact, physical storage between the two types of DBMSs

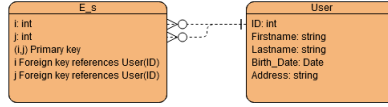


Fig. 2: A physical database model for the graph representation

varies significantly. The row DBMSs use indexes that hold the *rowid* of the physical data location. Whereas, the columnar DBMSs rely on projections, 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. For instance, Vertica allows storing data by projections in a format that optimizes query execution, and it compresses data and/or encodes projection data, allowing optimization of access and storage.

### 3.4 Parallel computation architecture

We assume a  $k$ -machines cluster ( $k$  represents the count of the hosts in the cluster) built over share-nothing architecture. Each machine can communicate directly with others. All the machines define a homogeneous setup configuration providing at least the minimum hardware and software requirements for the best performance of the columnar DBMS. Following the guidelines presented by the DBMS, each machine should provide at least 8 cores processor, 8 GB RAM and 1 TB of storage.

The number of machines  $k$  must be chosen as:  $k = p^3$  with  $p \in \mathbb{N}$ . This is important for our algorithm to achieve a perfect load balancing.

## 4 Triangle enumeration

In this section, we present our contribution to solve the triangle enumeration problem in parallel using SQL queries. Most of the queries bellow are specific to Vertica, particullary, CREATE TABLE/PROJECTION and COPY queries. For other DBMSs including row DBMSs like Oracle and SQL Server, a DBA can easily adapt them depending on their data loading and retrieval strategies.

### 4.1 Standard algorithm

Enumerating triangle in a given graph  $G$  can be done in two main iterations, the first aims to identify all the directed wedges in the input graph while the second focuses on finding whether there exists an edge connecting the endpoints of each wedge.

Basically, listing triangles using standard algorithm is performed by three nested loops, which can be translated in SQL by three self-join of table  $E$  ( $E1 \bowtie E2 \bowtie E3 \bowtie E1$ ) on  $E1.j = E2.i$ ,  $E2.j = E3.i$  and  $E3.j = E1.i$  respectively with  $E1 = E$ ,  $E2 = E$  and  $E3 = E$  ( $E1, E2$  and  $E3$  are alias table of  $E$ ). However, since only triangles defining lexicographical order ( $v_1 < v_2 < v_3$ ) are output (see Section 3.3), we can eliminate the third self-join by taking both  $E2.j = E3.i$  and  $E3.j = E1.i$  within the second self-join. As a result, the above-mentioned process can be formulated using only two self-joins on table  $E$  ( $E \bowtie E \bowtie E$ ). In SQL queries bellow, we use  $E\_dup$  which is a duplicate table of  $E$  used to speed

up the local join processing. Indeed, partitioning the first table by  $i$  and the duplicate one by  $j$  divides the two corresponding tables into small pieces based on the aforementioned columns which will make local joining on  $E.j = E.i$  faster.

```
SELECT E1.i AS v1,E1.j AS v2,E2.j AS v3
FROM
  E E1 JOIN E_dup E2 ON E1.j=E2.i
  JOIN E E3 ON E2.j=E3.i AND E3.j=E1.i
WHERE E1.i<E1.j AND E2.i<E2.j;
```

One problem can occur with this query executed in parallel. If the clique size is too large, there would be a redundancy in listed triangles caused by assigning each vertex and its neighbors to different machines to speed up local joins.

## 4.2 Randomized triangle enumeration

The high-level idea behind the randomized triangle enumeration algorithm of [19] is to randomly partition vertices into subsets of certain size and then compute the triangles within each subset in parallel. Specifically, the vertex set is partitioned into  $k^{1/3}$  random subsets (thus each subset will have  $n/k^{1/3}$  vertices), where  $k$  is the number of machines. Then each triplet of subsets of vertices (there are a total of  $(k^{1/3})^3 = k$  triplets, including repetitions) and the edges between the vertices in the subset are assigned to each one of the  $k$  machines. Each machine then computes the triangles in the subgraph induced by the subset assigned to that machine locally. Since every possible triplet is taken into account, every triangle will be counted (it is easy to remove duplicate counting, by using lexicographic order among the vertices, as described later). Hence correctness is easy to establish. The main intuition behind the randomized algorithm is that a random partition of the vertex set into equal sized subsets also essentially *balances* the number of edges assigned to a machine. This crucially reduces communication, but also the amount of work done per machine. While this is balancing is easy to see under expectation (an easy calculation using linearity of expectation shows that on average, the number of edges are balanced); however there can be a significant variance. It is shown in [19] via a probabilistic analysis, that the number of edges assigned per machine is bounded by  $\tilde{O}(\max\{m/k^{2/3}, n/k^{1/3}\})$ . We note that the randomized algorithm is always correct (i.e., of Las Vegas type), while the randomization is helpful to improve the performance.

The Fig. 3 illustrates the overview of the randomized triangle enumeration algorithm. To distinguish each partition of vertices, a color from  $k^{1/3}$  colors is assigned to it. The communication between machines is needed when creating sub-graphs or collecting edges from proxies. Otherwise, the processing is local.

## 4.3 Graph reading

The first step to enumerate triangles is to read the input graph on one host. If the graph is directed, the edge table `E_s` is built as adjacency list into the database system. Otherwise, for each tuple  $(i,j)$  inserted in the edge table `E_s`,  $(j,i)$  is also inserted. The following queries are used to read the input graph:

```
CREATE TABLE E_s(i int,j int);
COPY E_s FROM "link/to/graph_data_set";
/*if the input graph is undirected*/
INSERT INTO E_s SELECT j,i FROM E_s;
```

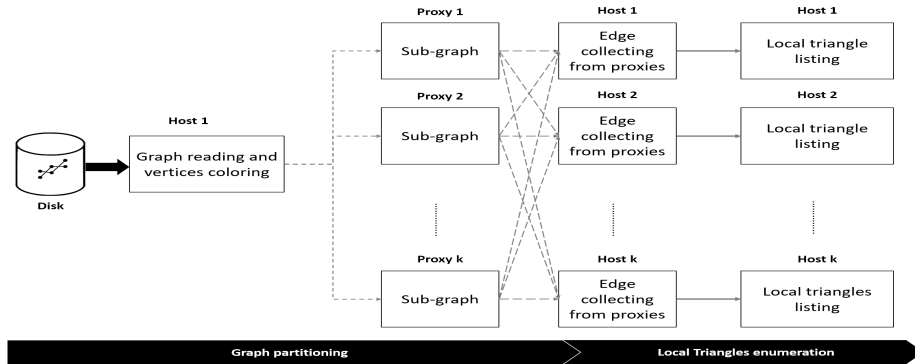


Fig. 3: Random triangle enumeration process

#### 4.4 Graph partitioning

We assume  $k$ -machine model as presented in Section 3.4. The input graph  $G$  is partitioned over the  $k$  machines using the random vertex partition model that is based on assigning each vertex and its neighbors to a random machine among the  $k$  machines [19]. Notably, for a graph  $G = (V, E)$ , a machine  $k$  hosts  $G_k = (V_k, E_k)$  a sub-graph of  $G$ , where  $V_k \subset V$  and  $\cup_k V_k = V$ ,  $G_k$  needs to be formed in a manner that for each  $v \in V_k$  and  $u \in N(v)/(u, v) \in E_k$ .

As explained previously, our partitioning strategy (proposed in [19]) aims to partition the set  $V$  of vertices of  $G$  in  $k^{1/3}$  subsets of  $n/k^{1/3}$  vertex each. Initially, the table "V\_s" in the query below ensures that each vertex  $v \in V$  picks independently and uniformly at random one color from a set of  $k^{1/3}$  distinct colors using the function `randmint` of vertica. To be noted that 1 and 2 in the query below refer to subsets of colors of  $k$ -machine model ( $k = 8$ ). A deterministic assignment of triplets of colors through the table "Triplet" in the following queries assigns each of the  $k$  possible triplets of colors formed by  $k^{1/3}$  distinct colors to one distinct machine. This can be translated by the following queries:

```

/*Each vertex picks a random color of qr=k^(1/3), for k=8, qr=2 */
CREATE TABLE V_s(i int,color int);
INSERT INTO V_s
  SELECT i,randmint(qr)+1
  FROM
    (SELECT DISTINCT i FROM E_s
     UNION
     SELECT DISTINCT j FROM E_s)V;
/*triplet_file values for k=8: (1,1,1,1)(2,1,1,2)
(3,1,2,1)(4,1,2,2)(5,2,1,1)(6,2,1,2)(7,2,2,1)(8,2,2,2)*/
CREATE TABLE Triplet(machine int,color1 int,color2 int,color3 int)
  UNSEGMENTED ALL NODES;
COPY Triplet FROM "link/to/triplet_file";

```

Then for each edge it holds, each machine designates one random machine as edge proxy for that edge and sends all its edges to the respective edge proxies, the table "E\_s\_proxy" holds all the edges with their respective edge proxies. this table is formed by coloring edge table "E\_s" end-vertices with the vertex table "V\_s" having for each vertex its picked color by using double join between the two tables. Building "E\_s\_proxy" is the most important step because all



the partitioning depends on it. Using this table, we can identify the edges end-vertices picked colors which help the next query of "E\_s\_local" building to decide which host will hold which edge according to the deterministic triplet assignment to each machine. In other words, each host collects its required edges from edge proxies to process the next step.

```

/*Sending edges to proxies*/
CREATE TABLE E_s_proxy(i_color int,j_color int,i int,j int);
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*/
CREATE TABLE E_s_local(machine int,i int,j int,i_color int,j_color int);
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 WHERE E.i<E.j
  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 WHERE E.i<E.j
  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 WHERE E.i>E.j;

```

Having  $E.i < E.j$  and  $E.i > E.j$  in last query guarantee the output of each triangle on a unique machine. For instance, a triangle  $(u, v, w)$  picking colors  $(c, b, c)$  is output on a unique machine  $m$  having triplet  $(c, b, c)$  assigned to it with  $u < v < w$ , hence triangles like  $(w, u, v)$  and  $(v, w, u)$  where  $w > u < v$  and  $v < w > u$  won't be taken in account which eliminates redundancy of enumerated triangles.

#### 4.5 Local triangle enumeration

To enumerate triangles locally on each host and in parallel, each machine examines its edges whose endpoints are in two distinct subsets among the three subsets assigned to it. This happens in two steps:

1. Each machine lists all the possible wedges that vertices have identical color and order as its triplet
2. To output triangles, each host checks whether there is an edge connecting the end-vertices of each listed wedge

The aforementioned steps are ensured through a local double self-join on table  $E$  ( $E \bowtie E \bowtie E$ ) on columns  $E.j = E.i$  on each host locally and in parallel. The queries are presented in the following:

```

SELECT E1.machine, E1.i AS v1, E1.j AS v2, E2.j AS v3
  FROM
    E_s_local E1 JOIN E_s_local E2 ON E1.machine=E2.machine AND E1.j=E2.i
    JOIN E_s_local E3 ON E2.machine=E3.machine AND E2.j=E3.i
    JOIN Triplet T on T.machine = E3.machine
  WHERE E1.i<E1.j AND E2.i<E2.j AND E1.i=E3.j
    AND E1.i_color=T.color_1 AND E1.j_color=T.color_2

```

```

AND E2.j_color=T.color_3 AND local_node_name()='node_name'
ORDER BY v1,v2,v3;

```

As explained in previous section, having  $E1.i < E1.j$  and  $E2.i < E2.j$  in the query eliminates redundancy. The last join with the table "Triplet" eliminates having triangles with vertex having same colors to be output by other machines then theirs. Fig. 4 illustrates the random partitioning and triangle enumeration in a cluster of eight machines.

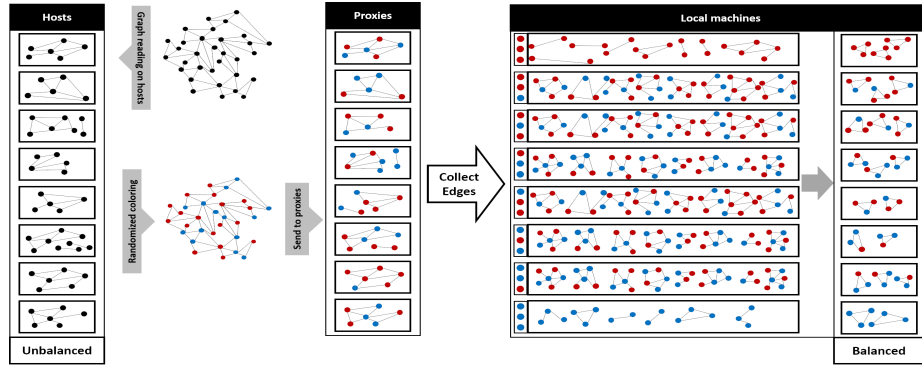


Fig. 4: Randomized triangle enumeration on 8 machines.

Furthermore, to check similarity between triangles output by randomized algorithm and those with standard algorithm, a set difference between the results of the two algorithms can be employed. The following queries are executed in Section 5 to prove the similarity of the output of both algorithms (notice that Triangle is a table containing the list of triangles resulting from randomized algorithm execution):

```

SELECT u, v, w FROM Triangle
EXCEPT
  (SELECT E1.i AS v1,E1.j AS v2,E2.j AS v3
   FROM E E1 JOIN E E2 ON E1.j=E2.i
   JOIN E E3 ON E2.j=E3.i AND E3.j=E1.i
   WHERE E1.i<E1.j AND E2.i<E2.j);

```

#### 4.6 Load balancing

Parallel computing is considered complete when all the hosts complete their processing and output the results. Therefore, reducing running time requires that all processors finish their tasks at almost the same time [4]. This is possible if all hosts acquire an equitable amount of data that they can process on.

We mentioned in data partitioning section that each vertex of the set  $V$  pick randomly and independently a color  $c$  from  $k^{1/3}$  distinct colors. This gives rise to a partition of the vertices set  $V$  into  $k^{1/3}$  subsets  $s_c$  of  $O(n/k^{1/3})$  vertex each. Each machine then receives a sub-graph  $G_k = (V_k, E_k)$  of  $G$ . As mentioned at the beginning of Section 4.2, the analysis of [19] shows that the number of edges among the subgraphs  $G_k = (V_k, E_k)$  is relatively balanced with high probability.

Hence each machine processes essentially the same number of edges which leads to a load balance. The endpoints of each  $e \in E_k$  are in two subsets  $s_c$ . This means that each triangle  $(u, v, w)$  satisfying  $u < v < w$  will be output.

#### 4.7 Time complexity of randomized algorithm

The time complexity taken by a machine is proportional to the number of edges and triangles it handles. Each machine handles a particular triplet of colors  $(c_x, c_y, c_z)$  so it handles  $O(n/k^{1/3})$  random sized subset of vertices. The worst-case number of triangles in this subset is  $O(n^3/k)$ ; however the number of edges is much lower. Indeed, as mentioned in Section 4.2 the key idea of the randomized algorithm as shown in [19], is that a random subset of the above-mentioned size (i.e.,  $O(n/k^{1/3})$ ) will have no more than  $\max\{\hat{O}(m/k^{2/3}, n/k^{1/3})\}$  edges with high probability. Hence each machine handles only so many edges with high probability. Since it is known that the number of triangles that can be listed using a set of  $\ell$  edges is  $\Omega(\ell^{3/2})$  (see e.g., [19]) the number of triangles that each machine has to handle is at most  $\max\{O(m^{3/2}/k, n^{3/2}/k^{1/2})\}$ . Since, the maximum number of (distinct) triangles in a graph of  $m$  edges is at most  $O(m^{3/2})$ , and each machine handles essentially a  $1/k$  fraction of that (when  $m^{3/2}/k > n^{3/2}/k^{1/2}$ ), we have essentially an optimal (linear) speed up (except, possibly for very sparse graphs). Indeed, we show experimentally (Section 5) that each machine handles approximately the same number of triangles, which gives load balance among the machines.

## 5 Experimental evaluation

In the following section, we present how we conduct our experiments and we outline our findings.

### 5.1 Hardware and software configuration

The experiments were conducted on 8 nodes cluster ( $k = 8$ ) each with 4 virtual cores CPU of type GeniuneIntel running at 2.4 Ghz, 48 GB of main memory, 1 TB of storage, 32 KB L1 cache, 4 MB L2 cache and Linux Ubuntu server 18.04 operating system. The total of RAM on the cluster is 384 GB and total of disk storage is 8 TB and 32 cores for processing. We used Vertica, a distributed columnar DBMS to execute our SQL queries and Python as the host language to generate and submit them to the database for its fastness comparing to JDBC.

### 5.2 Data set

Table 1 summarizes the data sets used in our experiments. We used seven real and synthetic (both directed and undirected) graph data sets. These data sets represent different sizes and structures. The aforementioned table exhibits for each data set, its nodes, edge, and expected triangle count with its type, format, and source.

### 5.3 Edge table partitioning

For the triangle enumeration problem, the input graph is mostly partitioned locally by source vertex  $i$  or destination vertex  $j$  to speedup the local join as explained in Section 4.1. Whereas the segmentation of the Vertices set  $V$  across

Table 1: data sets

data set	n	m	Triangle count	Type	Format	Skewness	Source
LiveJournal	3,997k	34,681k	177,820k	Real	Undirected	Low	SNAP
as-Skitter	1,696k	11,095k	28,769k	Real	Undirected	Low	SNAP
flickr-link	105k	2,316k	548,174k	Real	Undirected	High	KONECT
hyves	1,402k	2,777k	752k	Real	Undirected	High	KONECT
Graph500_s19	335k	15,459k	186,288k	Synthetic	Directed	High	Generated
Linear	141k	49,907k	3,300,095k	Synthetic	Directed	High	Generated
Geometric	8k	22,361k	13,078,242k	Synthetic	Directed	High	Generated

the hosts is done randomly using the random vertex partition model mentioned in Section 4.4. In Vertica DBMS, this process is performed in the DDL step through the segmentation clause definition in the projection creation query:

```
CREATE PROJECTION E_s_local_super(machine ENCODING RLE, i, j,
    i_color ENCODING RLE, j_color ENCODING RLE)
AS
SELECT machine, i,j, i_color,j_color
FROM E_s_local
ORDER BY i,j
SEGMENTED BY (machine*4294967295//k) ALL NODES OFFSET 0 KSAFE 1;
```

In fact, Vertica attributes for each machine a segment between 0 and 4 bytes that represents its hash values interval. Initially, the hash value of each tuple is calculated using the segmentation clause then according to the resulting value, the tuple is sent to the corresponding machine owning that hash value within its segment. We exploited this property to send each edge to its corresponding machine. This allowed us to perform joins locally on each machine independently from other hosts by specifying the name of the host in the WHERE clause of the triangle enumeration SQL query.

#### 5.4 Triangle enumeration

Here we analyze the performance of the randomized triangle enumeration algorithm output and compare it to the standard algorithm results.

We start by discussing our randomized algorithm results in terms of load balancing between processors and time execution on each machine. Our main purpose is to experimentally prove that the count of output triangles is almost even on all hosts. Thus, we present Fig. 5 and Fig.6 pie charts of examples of triangles count (TC) on two data sets on each host. The count of triangles output on each host is  $1/8$  of the total count which confirms our theoretical statement.

Fig. 7 represents the triangles count output on each machine for the remaining data sets. It is obvious that the distribution of output triangles is balanced over machines. Moreover, Fig. 8 presents the execution time of the randomized algorithm on each host. The lines chart reveals that all the processors finish their tasks at almost the same time on all the data sets with small-time shifts due to data movement in the preprocessing phase or the presence of skewness. For instance, data sets present a small overhead on machine 1 responsible for data read and shuffling which explains this additional processing time. Experimenting

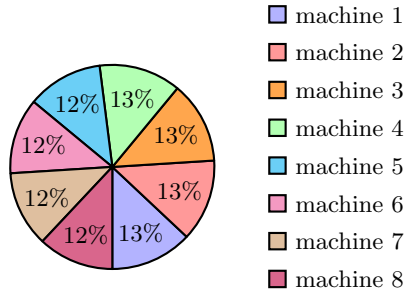


Fig. 5: as-Skitter TC by machine

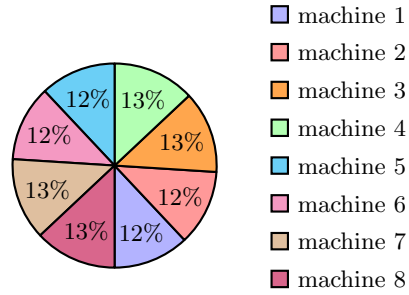


Fig. 6: Geometric TC by machine

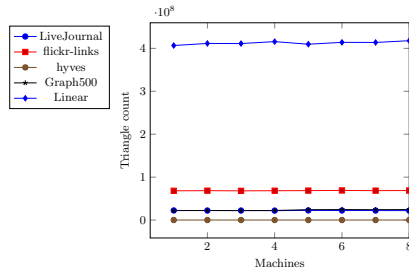


Fig. 7: Triangle count (TC)

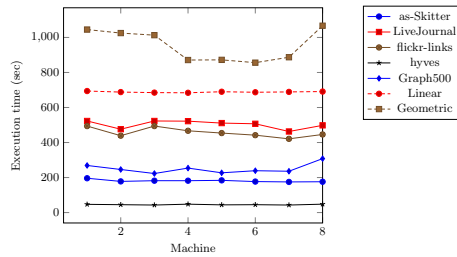


Fig. 8: Execution time

on high skewed graphs data sets, we can notice that there is some machines that take less time than the other hosts and finish first for the same data set, these machines may have fewer cliques compared to other machines.

We compare now the results between the standard algorithm and the randomized algorithm, summarized in Table 2. We notice that the triangle count is the same as the expected triangle count defined in Table 1. The load balancing is not ensured and a lot of data movement is performed to complete the join task using the standard algorithm. Thus, we added the preprocessing step as we want to ensure the load balancing in the standard algorithm, however this costs a significant overhead in execution time. The column "Rebalanced" in Table 2 exhibits the cost of such approach.

The column "Randomized" give the average time execution of randomized algorithm on the different data sets. As the data set size grows or presents high skewness, the performance of the randomized algorithm becomes better than standard algorithm such as the case for directed graph Graph500\_s19 and undirected graph flickr-link or hyves respectively. Moreover, when the skewed graph data set is becoming significantly large like Linear and Geometric, standard algorithm fails because of the data movement during the join processing causing memory issues while randomized algorithm succeeds to finish the task because the triangle listing is done locally on the subgraph stored on each machine and no data exchange is performed.

Table 2: Triangle count (in millions) and execution time (secs) results

data set	$TC_{stand}$	$TC_{rand}$	Standard	Rebalanced	Randomized
com-LiveJournal	177,820k	177,820k	480	stop	503
as-Skitter	28,769k	28,769k	90	1988	180
flickr-link	na	548,174k	stop	stop	485
hyves	752k	752k	49	604	47
Graph500_s19	186,288k	186,288k	610	stop	250
Linear	na	3,300,095k	stop	stop	365
Geometric	na	13,078,242k	stop	stop	954

Columns  $TC_{stand}$  and  $TC_{rand}$  summarize the resulting triangle count using both algorithms, we notice that both algorithms give the same triangle count. We experimentally executed the set difference SQL query presented in Section 4.5 between the two results that returns an empty set for each data set, hence, the similarity of results from both algorithms is confirmed.

## 6 Conclusions

We presented a parallel randomized algorithm for triangle enumeration on large graphs using SQL queries. Our approach using SQL queries provides elegant, shorter and abstract solution compared to traditional languages like C++ or Python. We proved that our approach scales well with the size of the graph and its complexity especially with skewed graph. Our partitioning strategy ensures balanced load distribution of data between the hosts. The experimental findings were promising. They were compatible with our theoretical statements.

For our future work, we are planning to perform a deeper study of the randomized algorithm with dense and complex graphs. As well as, running further experiments to compare the randomized algorithm with graph engine solutions for triangle enumeration. Finally, we are opting for expanding our algorithm to detect larger cliques which is another computationally challenging problem.

## References

1. Afrati, F.N., Sarma, A.D., Salihoglu, S., Ullman, J.D.: Upper and lower bounds on the cost of a MapReduce computation. *PVLDB* **6**(4), 277–288 (2013)
2. Al-Amin, S.T., Ordonez, C., Bellatreche, L.: Big data analytics: Exploring graphs with optimized SQL queries. In: *DEXA Workshops*. pp. 88–100 (2018)
3. Al Hasan, M., Dave, V.S.: Triangle counting in large networks: a review. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* **8**(2), e1226 (2018)
4. Arifuzzaman, S., Khan, M., Marathe, M.: A space-efficient parallel algorithm for counting exact triangles in massive networks. In: *HPCC*. pp. 527–534 (2015)
5. Berry, J.W., Fostvedt, L.A., Nordman, D.J., Phillips, C.A., Seshadhri, C., Wilson, A.G.: Why Do Simple Algorithms for Triangle Enumeration Work in the Real World? *Internet Mathematics* **11**(6), 555–571 (2015)
6. Cabrera, W., Ordonez, C.: Scalable parallel graph algorithms with matrix–vector multiplication evaluated with queries. *DAPD Journal* **35**(3-4), 335–362 (2017)
7. Chu, S., Cheng, J.: Triangle Listing in Massive Networks. *ACM Trans. Knowl. Discov. Data* **6**(4), 17 (2012)

8. Cui, Y., Xiao, D., Cline, D.B., Loguinov, D.: Improving I/O complexity of triangle enumeration. In: IEEE ICDM. pp. 61–70 (2017)
9. Das, S., Santra, A., Bodra, J., Chakravarthy, S.: Query processing on large graphs: Approaches to scalability and response time trade offs. *Data Knowl. Eng.* **126**, 101736 (2020). <https://doi.org/10.1016/j.datak.2019.101736>, <https://doi.org/10.1016/j.datak.2019.101736>
10. Fan, J., Raj, A.G.S., Patel, J.M.: The Case Against Specialized Graph Analytics Engines. In: CIDR (2015)
11. Hu, X., Tao, Y., Chung, C.W.: Massive graph triangulation. In: ACM SIGMOD. pp. 325–336 (2013)
12. Itai, A., Rodeh, M.: Finding a minimum circuit in a graph. *SIAM Journal on Computing* **7**(4), 413–423 (1978)
13. Klauck, H., Nanongkai, D., Pandurangan, G., Robinson, P.: Distributed Computation of Large-scale Graph Problems. In: ACM-SIAM SODA. pp. 391–410 (2015)
14. Latapy, M.: Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical computer science* **407**(1-3), 458–473 (2008)
15. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: ACM SIGMOD. pp. 135–146 (2010)
16. Ngo, H.Q., Ré, C., Rudra, A.: Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record* **42**(4), 5–16 (2013)
17. Ordonez, C.: Optimization of linear recursive queries in sql. *IEEE Transactions on Knowledge and Data Engineering* **22**(2), 264–277 (2010)
18. Ordonez, C., Cabrera, W., Gurram, A.: Comparing columnar, row and array DBMSs to process recursive queries on graphs. *Information Systems* **63**, 66–79 (2017)
19. Pandurangan, G., Robinson, P., Squizzato, M.: On the Distributed Complexity of Large-Scale Graph Computations. In: SPAA. pp. 405–414 (2018)
20. Park, H.M., Chung, C.W.: An efficient MapReduce algorithm for counting triangles in a very large graph. In: ACM CIKM. pp. 539–548 (2013)
21. Schank, T.: Algorithmic aspects of triangle-based network analysis (2007)
22. Shun, J., Tangwongsan, K.: Multicore triangle computations without tuning. In: ICDE. pp. 149–160 (2015)
23. Sun, W., Fokoue, A., Srinivas, K., Kementsietsidis, A., Hu, G., Xie, G.: SQLgraph: An efficient relational-based property graph store. In: ACM SIGMOD. pp. 1887–1901 (2015)
24. Suri, S., Vassilvitskii, S.: Counting triangles and the curse of the last reducer. In: WWW. pp. 607–614 (2011)
25. Wang, N., Zhang, J., Tan, K.L., Tung, A.K.H.: On Triangulation-based Dense Neighborhood Graph Discovery. *Proc. VLDB Endow.* **4**(2), 58–68 (2010)
26. Watts, D.J., Strogatz, S.H.: Collective dynamics of ‘small-world’ networks. *Nature* **393**, 440–442 (1998)
27. Zhang, Y., Jiang, H., Wang, F., Hua, Y., Feng, D., Xu, X.: LiteTE: Lightweight, Communication-Efficient Distributed-Memory Triangle Enumerating. *IEEE Access* **7**, 26294–26306 (2019)
28. Zhao, K., Yu, J.X.: All-in-one: Graph processing in RDBMSs revisited. In: SIGMOD. pp. 1165–1180 (2017)
29. Zhu, Y., Zhang, H., Qin, L., Cheng, H.: Efficient MapReduce algorithms for triangle listing in billion-scale graphs. *DAPD Journal* **35**(2), 149–176 (2017)