Triangle Enumeration for Billion-Scale Graphs in RDBMS

Aly Ahmed, Keanelek Enns, and Alex Thomo
{alyahmed,keanelekenns,thomo}@uvic.ca

University of Victoria, BC, Canada

Abstract. Triangle enumeration is considered a fundamental graph analytics problem with many applications including detecting fake accounts, spam detection, and community searches. Real world graph data sets are growing to unprecedented levels and many of the existing algorithms fail to process them or take a very long time to produce results. Many organizations invest in new hardware and new services in order to be able keep up with the data growth and often neglect the well established and widely used relational database management systems (RDBMSs). In this paper we present a carefully engineered RDBMS solution to the problem of triangle enumeration for very large graphs. We show that RDBMSs are suitable tools for enumerating billions of triangles in billion-scale networks on a consumer grade server. Also, we compare our RDBMS solution's performance to a native graph database and show that our RDBMS solution outperforms by order of magnitude.

1 Introduction

The problem of triangle listing or triangle enumeration can be stated as follows: Given an undirected graph $G = (V, E)$ with no parallel edges or self loops, output all tuples (a, b, c) such that nodes $a, b, c \in V$ are pairwise connected in G (i.e., they form a triangle). Some algorithms only need to touch triangles or count them, but in this paper, we require that algorithms store them explicitly (hence the term triangle listing).

Triangle enumeration is considered a fundamental graph analytics problem that constitutes a large portion of the computational work required for problems such as calculating a graph's global clustering coefficient (in which the number of triangles each node is involved in is required) [20], finding k-truss decomposition, and calculating the transitivity ratio of a graph. Some of the indirect applications of triangle enumeration include identifying social networks, determining a community's age [11], performing community searches [5, 18], and detecting fake accounts, malicious pages, or instances of web spamming [16, 4].

Triangle enumeration is a nontrivial problem. The optimal worst case for any algorithm's time complexity is $O(|E|^{3/2})$ [16]. Triangle enumeration does not scale very well with large datasets as any node can participate in a triangle with any other node in a graph so long as they have an edge between them, making it difficult to break the problem into smaller pieces. The sizes of datasets

continue to grow, and massive datasets are becoming more common in business and research. In particular, data in real-world networks is growing to unprecedented levels [10]. For example, Walmart is estimated to create 2.5 petabytes of consumer data every hour [13], Facebook processes tens of billions of likes and messages every day, Google receives 1.2 trillion search requests every year, and Internet of Things (IoT) data is expected to exceed 175 zettabytes by 2025 [12]. Many algorithms either enumerate triangles for graphs that can fit in memory [c.f. [19, 22]], are I.O. intensive [c.f. [9, 6]] or use distributed systems such as MapReduce [c.f [17, 20]].

Relational Database Management Systems (RDBMSs) have been vital tools in storing and manipulating data for many decades [7]. They are commonplace in most businesses, and they are familiar to technical and non-technical users alike. This paper aims to show that they are also useful for analyzing large graphs, specifically in the area of triangle enumeration. The paper also shows how a single machine can use simple partitioning techniques to enumerate billions of triangles efficiently.

In recent times, Graph Databases (GDBs) have grown in popularity. Many businesses may be considering moving to GDBs if they often analyze large graphs. Using dedicated graph databases for graph processing is presumed to provide better performance and scalability over relational databases (c.f. [3]); however, graph databases still have a long way to reach the level of maturity of RDMBSs. Using an RDBMS to implement graph algorithms is, in many situations, more efficient. However, computing graph algorithms using SQL queries is challenging and requires novel thinking. As such, there is active research on the use of novel methods to compute graph analytics on RDBMSs (c.f. [1, 14, 2, 8]). These works have shown that RDBMSs often provide higher efficiency over graph databases for specific analytics tasks.

The contributions of this paper are as follows:

1. We engineer triangle enumeration algorithms in SQL using partitioning and coloring
2. We suggest a modification of the PTE CD [16] algorithm, which we name Source Node Partitioning, which allows us to scale efficiently
3. We compare the performance of a popular open source GDB and a commonly used RDBMS.
4. We give a comparison of the performance of each algorithm on several graphs including billion-scale graph.

2 Triangle Enumeration in RDBMS

The following section introduces state of the art triangle enumeration algorithms and techniques. The basic method used for triangle enumeration, known as the Compact Forward algorithm [21], is explained first. Adaptations created to handle larger graphs that cannot fit into a single machine's memory are then discussed. These adaptations are known as Triangle Type Partitioning [15], and

Pre-partitioned Triangle Enumeration [16]. Finally, a further adaptation contributed by this paper that aims to reduce the complexity and the number of queries generated, named Source Node Partitioning, is explained. The SQL implementation of each algorithm developed is illustrated so that readers may recreate these results in an RDBMS of their choice.

2.1 Compact Forward

The compact forward algorithm constitutes the main portion of work done to enumerate triangles in all of the algorithms that follow. Its pseudo code is shown as algorithm 1. It consists of two main parts: an edge iterator and an orientation technique.

Algorithm 1: Compact Forward

Input : Undirected graph G = (V, E)
Output: All triangles of G
//Orientation of G
for $(u, v) \in E$ **do**
  if $deg(u) > deg(v)$ **or** $(deg(u) = deg(v)$ **and** $u > v)$ **then**
    Replace (u, v) with (v, u) in E;
//List triangles of G
for $(u, v) \in E$ **do**
  for $w \in N(u) \cap N(v)$ **do**
    Output triangle (u, v, w);

Edge Iterator. For any edge $(u, v) \in E$, we can find the triangles associated with it by considering the intersection of the neighbors of u and v (denoted $N(u)$ and $N(v)$ respectively). That is, if (u, v) exists, we can check to see if both (u, w) and (v, w) exist for all such nodes $w \in V$. By performing this operation on all edges, we are guaranteed to enumerate all triangles in the graph. Unfortunately, we might count many duplicates depending on how the undirected graph G is represented.

Orientation Technique. In order to eliminate duplicates, we direct the graph G with a total ordering. Define the degree (or number of neighbouring nodes) of a node v to be $deg(v)$ and assume nodes are represented by unique integer identifiers (i.e. $V \subset \mathbb{N}$). Define a total ordering of the nodes in V, denoted $\rightarrow$, as follows: For all nodes $u, v \in V$, we say $u \rightarrow v$ if and only if $deg(u) < deg(v)$ OR ($deg(u) = deg(v)$ AND $u < v$). We then arrange all the edges in the graph according to this total order. The resulting graph is a Directed Acyclic Graph or DAG. But how does this help?

Consider a triangle (a, b, c), such that $a \rightarrow b, b \rightarrow c$, and $a \rightarrow c$. When iterating over edge (a, b), we discover (b, c) and (a, c) in the neighbours of a and b. However, when we iterate over edge (b, c), we will not find (c, a) due to the total ordering properties (similarly for edge (a, c), we will not find (c, b)). Thus each triangle is counted exactly once. From now on, when we refer to triangle (a, b, c), we assume the total ordering applies from left to right.

Notice that a total ordering could have been defined on the node identifiers alone assuming they are unique. The reason for involving node degrees in the total ordering is to prevent any given node from having a large list of outgoing neighbours to search through. In the given total ordering, nodes of high degree will have fewer outgoing neighbours and nodes of low degree, though their neighbourhood primarily consists of outgoing neighbours, have few neighbours by definition.

The SQL queries for orienting G are lengthy, yet simple to implement, so we omit them. Algorithm 2 shows the edge iteration component written in SQL and assumes the edge list E has already been oriented.

Algorithm 2: Compact Forward Edge Iteration in SQL

```
SELECT g1.fromNode AS A, g1.toNode AS B, g2.toNode as C
FROM E g1, E g2
WHERE g1.toNode = g2.fromNode
AND EXISTS (
 SELECT 1 FROM E
 WHERE fromNode = g1.fromNode AND toNode = g2.toNode);
```

2.2 Triangle Type Partitioning

Suppose G is a large graph that does not fit in a single machine's memory, then the edge list E must be partitioned into smaller lists in order to fit. Even though an RDBMS is designed to handle data that does not fit in main memory, when it has to deal with smaller chunks of the data at a time, the RDBMS can use more efficient join algorithms, such as one-pass joins. However, a problem arises when trying to list triangles in the edge partitions. Consider triangles that have edges in more than one partition, they are certainly missed. As such, we cannot expect the RDBMS query optimizer to be able to automatically find ways to break up the data into chunks to facilitate better query evaluation algorithms. Therefore, we focus here in ways to intelligently partition the data and create independent subtasks.

The triangle type partitioning (TTP) algorithm [16] was designed to resolve this issue. Originally it was designed as a Map Reduce algorithm to be run on distributed systems, but in this paper, it is used on a single machine.

The first step is to colour the nodes using a function $f : V \to \{0, 1, ..., \rho - 1\}$. This allows every triangle to be classified into the following three types:

- Type 1: All three nodes have the same colour.
- Type 2: Exactly two nodes have the same colour.
- Type 3: Each node has a distinct colour.

A visualization of the triangle types can be seen in figure 1.

Let E_{ij} be the set of edges that have endpoints coloured i or j. There are $\binom{\rho}{2}$ such sets. Let E_{ijk} be the set of edges that have endpoints coloured i, j, or k. There are $\binom{\rho}{3}$ such sets as seen in figure 2.

No edge in a type 3 triangle has endpoints with the same colour. Knowing this, we can transform the set E_{ijk} into $E'_{ijk} = E_{ijk} - \{(u,v)|f(u) = f(v), (u,v) \in E_{ijk}\}$ where we remove all such edges. This vastly improves performance when enumerating type 3 triangles by reducing the size of the edge lists. On the other hand, each set E_{ij} can contain type 1 and type 2 triangles.

In order to enumerate all triangles, algorithm 2 is run on all $\binom{\rho}{2} + \binom{\rho}{3}$ edge sets (replace E with E_{ij} or E'_{ijk} for all i,j, and k). This can be seen in algorithm 3, which shows the structure for partitioning and generating the SQL queries.

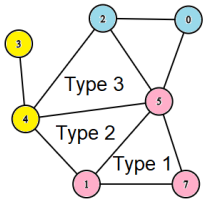


Fig. 1: An example of each type of triangle in a node coloured graph

Algorithm 3: Triangle Type Partitioning

```
for i = 0 to ρ − 2 do
    for j = i + 1 to ρ − 1 do
        Generate E_ij from E
        Run algorithm 2 on E_ij
for i = 0 to ρ − 3 do
    for j = i + 1 to ρ − 2 do
        for k = j + 1 to ρ − 1 do
            Generate E'_ijk from E
            Run algorithm 2 on E'_ijk
Eliminate duplicates from generated triangles
```

Consider the sets E_{02} and E_{12} from figure 2. The type 1 triangle of colour 2 is counted twice. In fact, this is true of all type 1 triangles.

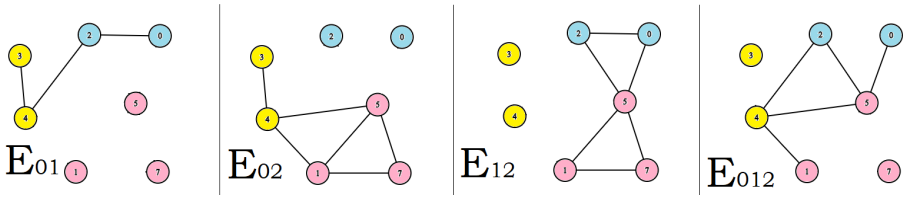


Fig. 2: A visualization of the edge sets from the graph in figure 2 when $\rho = 3$: yellow = 0, blue = 1, pink = 2

Thus we count every type 1 triangle $\rho - 1$ times, and we count type 2 and type 3 triangles exactly once. The duplicate triangles can then be eliminated.

2.3 Prepartitioned Triangle Enumeration - Colour Direction

Triangle type partitioning [16] is an effective method for enumerating triangles in massive graphs that do not fit into memory, yet it is also possible to en-

sure each triangle is counted exactly once as well as greatly reduce the size of the edge sets to be searched through in the CF algorithm. This is the goal of the Prepartitioned Triangle Enumeration - Colour Direction (PTE CD) method, which expands on the TTP algorithm. Again, this algorithm was proposed in [16] for a Map Reduce setting. Here we adapt it for an RDBMS. Consider how the TTP algorithm pays no attention to the direction of edges when partitioning the edge set, indeed figures 2 and 3 do not display the direction of the edges because that information is irrelevant to the algorithm. PTE CD, however, takes advantage of the direction of each edge in the oriented graph.

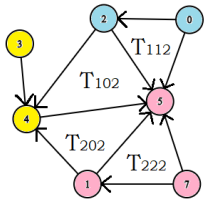


Fig. 3: The graph of figure 1 with edge directions shown. The labels indicate which triangle set each triangle belongs to.

Let T_{ijk} be the set of triangles (a, b, c) where $f(a) = i, f(b) = j$, and $f(c) = k$ (ie. ijk is a permutation of the colours with replacement). For example, $T_{001} \neq T_{010}$ due to the total ordering on the edges in G. This is made more clear in Figure 3. There are ρ^3 such triangle sets.

Suppose we are trying to find a triangle $(a, b, c) \in T_{ijk}$ and we reach edge (a, b). We know edge (b, c) is in E_{jk} and edge (a, c) is in E_{ik}, which may or may not all be the same edge set from our previous definitions, but more specifically, we know (b, c) goes from colour j to k, and (a, c) goes from colour i to k.

Let E_{ij} be redefined in the following way:

$$E_{ij} = \{(u, v) | f(u) = i, f(v) = j, (u, v) \in E\}$$

There are ρ^2 such edge sets. This new definition allows the algorithm to more precisely choose which edge sets to search through and thereby reduces the total work.

The pseudo code in Algorithm 4 shows the implementation of the PTE CD algorithm. It counts triangles of all types exactly once and improves the performance greatly.

One issue is that it increases the number of edge sets from $\binom{\rho}{2}$ to ρ^2 and the number of enumeration tasks from $\binom{\rho}{2} + \binom{\rho}{3}$ to ρ^3. However, it reduces the number of colours needed to be effective compared to TTP, and since ρ is often relatively small, this does not create a major issue in performance.

Algorithm 4: PTE CD

```
for i = 0 to ρ − 1 do
    for j = 0 to ρ − 1 do
        Generate E_ij from E
for i = 0 to ρ − 1 do
    for j = 0 to ρ − 1 do
        for k = 0 to ρ − 1 do
            Run algorithm 2 with E_ij as g1, E_jk as g2, and E_ik in the
              EXISTS clause.
```

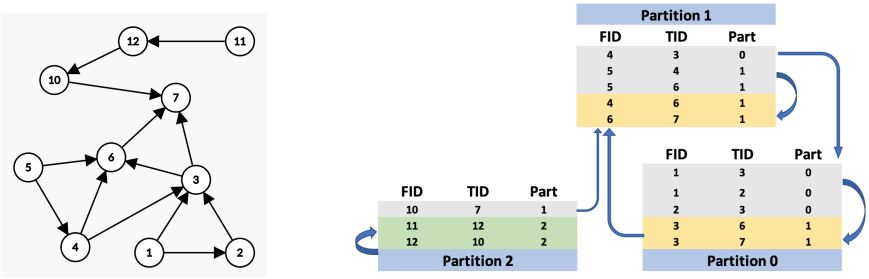
2.4 Source Node Partitioning (SNP)

In this section, we propose another algorithm, called Source Node Partitioning (SNP), that partitions the graph into ρ partitions and generates ρ^2 enumeration tasks. SNP is conceptually simpler than PTE CD. Similar to PTE CD, SNP enumerates each triangle only once and exhibits comparable performance to PTE CD for medium datasets and even better for larger datasets.

The main idea is to partition an oriented input graph G into ρ parts where all the edges (u, v) whose source nodes u are landed in the same partition i based on partitioning function $f(u)$. Each partition i will have a tuple (u, v, j) where j equal the partition number for node v.

$$E_i = \{(u, v, j) | f(u) = i, f(v) = j, (u, v) \in E\},\ 0 \leq i, j \leq \rho - 1$$

Now, in order to find a triangle (u, v, k) we only need to check if there is a shared node k between the neighbours of nodes u and v. This is achieved by joining in SQL the tables for E_i and E_j (assuming $f(u) = i$ and $f(v) = j$).



(a) Input Graph (b) Partitioning into tables

Fig. 4: SNP partitioning

For example, given the graph in Figure 4 (a), and $\rho = 3$, we partition the edges of the graph into the partition tables given in Figure 4 (b). Along with the FID and TID attributes, which give the source and target of each edge, we also have a third attribute, called Part, which stores the partition number, of the target node.

To check for instance, if edge $(3, 6)$ is part of a triangle, we need to find the intersection of the neighbors of node 3 with the neighbors of node 6. These neighbors can be extracted from the set of edges $(6, \_)$, which exist in Partition 1 (as indicated by column Part) and the set of edges $(3, \_)$, which exist in the current partition, Partition 0.

Partition oriented graph G based on the source node could be done by several ways for instance, we could use interval partitioning where each partition i will

have a unique range of nodes ID or use module function to distribute the source nodes over partitions as shown in Algorithm 5.

In terms of pseudo code we create the partition tables using the queries given in Algorithm 5. Then, we enumerate triangles using Algorithm 6. In a nutshell, it joins two tables, for partitions E_i and E_j, respectively, and limits the scope of the join in the table for E_i to only those nodes that have $Part = j$.

Algorithm 5: SNP: Graph Partitioning

Input: Oriented Graph $G = (V, E)$, number of partitions ρ
Output: A set of ρ partition tables.
for $i = 0$ **to** $\rho - 1$ **do**
    INSERT INTO E_i
      SELECT FID, TID, TID % ρ AS Part
      FROM E
      WHERE FID % $\rho = i$

Algorithm 6: SNP (enumerate triangles)

Input: A set of ρ partition tables E_i.
for $i = 0$ **to** $\rho - 1$ **do**
    for $j = 0$ **to** $\rho - 1$ **do**
        SELECT G1.fid AS A, G1.tid AS B, G2.tid as C
        FROM E_i AS G1, E_j AS G2
        WHERE G1.tid = G2.fid AND G1.part = j AND
            EXISTS
              (SELECT 1 FROM E_i AS G3
               WHERE G3.fid = G1.fid AND G3.tid =G2.tid)

Theorem 1. *SNP Partitioned graph can only distribute a triangle* (u, v, k) *over a maximum of 2 partitions.*

Proof. To prove it by contradiction let us assume there is a triangle (u, v, k) whose edges exist in three partitions which means neighbors of u and v in an edge (u, v) must exist in three partitions. However, SNP will not separate the edges with the same source node across different partitions, therefore neighbors of u and v will only exist in a maximum of 2 partitions. □

3 Experimental Results

3.1 Setup Configuration

We executed the experiments on a cloud based virtual server running Windows Server 2019 with 4 vCores and 16 GB of RAM.

As RDBMSs we used the latest versions of a commercial database (which we anonymously call CD) As graph database, we used the latest version of a graph database (which we anonymously call GD). We refrain from using the real names of these databases for obvious reasons.

3.2 Datasets

We used four datasets from Stanford's Data collection and four datasets from The Laboratory for Web Algorithmics including a one-billion-edge graph.

The datasets are Web-Google, Pokec, Live-Journal and Orkut (from `http://snap.stanford.edu`), and Hollywood 2009, Hollywood 2011, UK 2005 and IT 2004 (from `http://law.di.unimi.it/webdata`). Table 1 shows statistics about the datasets used.

Table 1: Graph Datasets

Dataset	**# Nodes**	**# Edges**	**# Triangles**
Web-Google	875,713	5,105,039	13,391,903
Pokec	1,632,803	30,622,564	32,557,458
Live Journal	4,847,571	68,993,773	177,820,130
Orkut	3,072,441	117,185,083	627,584,181
Hollywood 2009	1,139,905	113,891,327	4,916,374,555
Hollywood 2011	2,180,759	228,985,632	7,073,951,555
UK 2005	39,459,921	936,364,282	21,779,347,099
IT 2004	41,291,594	1,150,725,436	47,249,138,589

3.3 Results

Our experiments began with a comparison of the performance of a popular open source graph database and a well known, commonly used RDBMS. The initial idea was to compare the time it took for each database system to compute the clustering coefficient of each graph. However, after running various experiments, it was determined that the graph database likely used approximation algorithms. This theory was then supported when attempting to use the graph database's triangle listing algorithm which took substantially longer than the clustering coefficient calculation, which does not make logical sense if no approximation algorithms are involved because triangle listing is a subset of the computations required for calculating the clustering coefficient. Moreover, the triangle listing algorithm seemed to have a bug at the time of experimentation, and only listed around a tenth of the triangles in the graph.

In order to ensure the measurements for the graph database were as fair as possible, we caused the output to be written to a file instead of writing to standard out (which can often take longer than a computation itself), and we

Fig. 5: A logarithmic runtime comparison between a graph database and RDBMS when enumerating triangles on the four smallest datasets: google, pokec, livejournal, and orkut

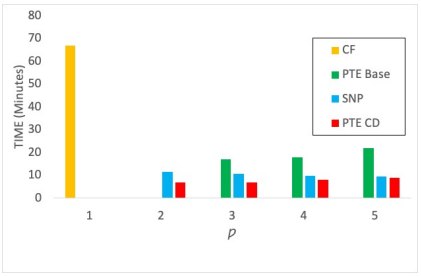


Fig. 6: A comparison of runtimes of the four algorithms discussed in section 3 on the orkut dataset. ρ is the number of node partitions of the graph, not the number of edge sets created by the algorithms.

scaled the triangle listing time linearly to match the approximate amount of time it would take to enumerate all of the triangles rather than a fraction of them.

Figure 5 shows a comparison between the projected runtime of the graph database and the actual runtime of the RDBMS baseline, which is the compact forward algorithm run on the entire edge set at once. Note that the time scale is logarithmic and we can see that the RDBMS greatly outperforms the graph database when the graph database is required to list all triangles explicitly.

After seeing the drastic performance difference between the two databases in just the baseline case, we did not see value in pursuing any further comparisons with other methods or larger datasets.

Figure 6 compares the performance of all the algorithms discussed in section 2 on the orkut dataset. Smaller datasets did not see much improvement when partitioned, as their edge sets were already quite small. No entry has been given for the PTE BASE method (which is another name for TTP) when $\rho = 2$. This is because the algorithm no longer benefits from the partitioning in this case, and the results would be the same as the baseline CF results.

Note that the yellow bar is the same datapoint that was used in figure 5 for the RDBMS baseline on the orkut dataset. Clearly the partitioning algorithms made great improvements over the single table compact forward algorithm. From this we conclude that partitioning can be helpful even when the dataset is able to fit into a machine's memory (as is the case with the orkut dataset). However, as noted before, the benefits become less notable the smaller the dataset becomes and in some cases the performance decrease slightly as the cost of partitioning tables and query planning overcome the savings. As we moved to graphs with billions of triangles, PTE Base reached its performance bottleneck; eliminating duplicate triangles takes a considerable amount of time and resources to perform, therefore, we decided not to test it with other datasets.

We turn our attention to the hollywood datasets for 2009 and 2011 which have over 8 and 11 times more triangles than orkut respectively. Figure 7 shows that PTE CD slightly outperforms the SNP method. Notice, however, that the performance gap begins to close with more node partitions or colours. At first inspection, this seems to be due to the growth rates in the number of queries generated by each method. SNP generates ρ^2 queries compared to PTE CD's ρ^3 queries, which conceivably increases compiling and execution time as well as clutters the SQL script files.

However, the difference is not as clear cut as it may seem. SNP only creates ρ edge sets (or edge partitions), whereas PTE CD creates ρ^2 edge sets. If the desired edge set size is the same relative to the size of a machine's memory (e.g. we want edge sets to be a third the size of a machine's available memory), then PTE CD can use a smaller value for ρ than SNP can. Therefore PTE CD may actually use less queries in practice.

For example, suppose a graph has an edge list with 1.2 million entries, and we desire to use a machine that can fit 300,000 edges, then our edge sets should have about 100,000 edges in them (if we want the database to perform an all-in-memory join). SNP will use $\rho = 12$ to meet this requirement, whereas PTE CD will use $\rho = 4$. Notice that $4^3 = 64 < 12^2 = 144$ in this instance, and PTE CD actually has fewer queries for similar-sized edge sets. Nevertheless, in a very large graph, fewer queries might not be as desired. SQL might need in worst-case scenario memory size of $|E_i| * |E_j| * |E_k|$ to find triangles; hence more scoped queries would be more efficient. Also, the partitioning function used in SNP impacts the performance; figure 9 shows a noticeable difference in performance when we used the modulo function, and this could be explained as it is most likely to distribute high degree nodes across all partitions relatively than distribute them using interval partitioning.

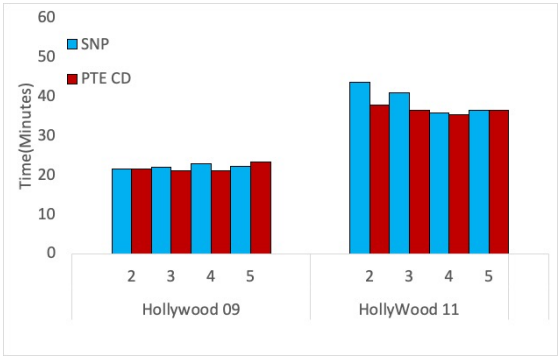


Fig. 7: Runtimes of SNP (blue) and PTE CD (red) algorithms on the hollywood-2009 dataset ($\sim$ 5 billion triangles) and hollywood-2011 dataset ($\sim$ 7 billion triangles). The x axis shows the value of ρ or the number of node partitions.

It is also worth mentioning that the distance between squares increases quickly. In the previous example, PTE CD creates 16 edge sets rather than the ideal 12. Suppose we need to divide the edge list of the graph into 37 edge sets. Then PTE CD must use $\rho = 7$ and create 49 edge sets, which may cause performance issues because the edge sets are too small compared to the optimal value and the time to shuffle the data will increase. Figure 8 shows a decrease in the performance of PTE CD as the number of triangles increase.

It would seem that using the same value of ρ for both methods is in inappropriate comparison, and this would explain the reduction in the performance gap as ρ increases, since SNP performs better for higher values of ρ.

In each case, it may take some testing and experimentation to determining the ideal value for ρ for a given dataset.

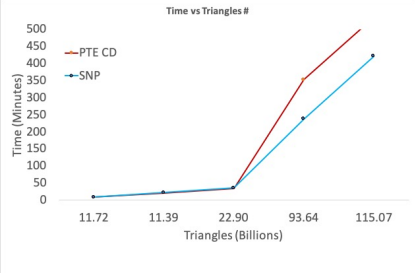


Fig. 8: A comparison of running time between SNP and PTE CD as the number of triangles increased. PTE CD takes more time as partitioning takes longer than SNP.

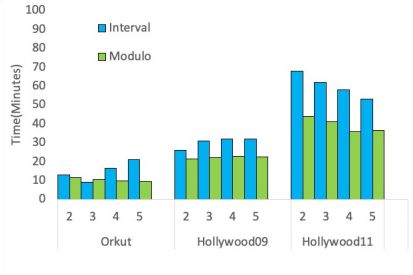


Fig. 9: SNP: A noticeable difference in running time when partitioning the graph using Interval Partitioning vs using Modulo function.

3.4 Experiments on Billion-Scale Networks

We show our experiments on two vast datasets, namely UK-2005 and IT-2004, the latter with more than a billion edges. They represent the web network of UK and Italy in 2005 and 2004, respectively. The precise number of nodes and edges is given in Table 1. We ran both the SNP algorithm with $\rho = 20$ and PTE CD algorithm with $\rho = 8$. Both algorithms ran in a single machine and enumerated all triangles in a very reasonable time regarding the graph's size and the number of triangles found in each data set; IT 2004 and UK 2005 contain 47.2 and 21.7 billion triangles, respectively.

We compare the running time of the two algorithms in Figure 10. The result indicates that the SNP algorithm shows better performance than PTE CD. The reason being that SNP scales linearly with ρ in partitioning the tables and quadratically in the number of queries when enumeratimg triangles, however

PTE CD scales quadratic to ρ for partitioning tables and cubic to the number of queries performed.

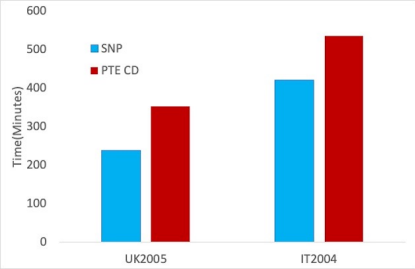


Fig. 10: Results of Triangles Enumeration on very large data sets, IT 2004: 1.15 billion edge graph and UK 2005:0.93 billion edge, Using SNP and PTE CD.

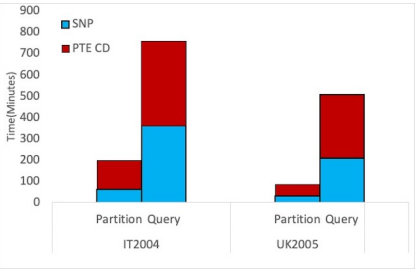


Fig. 11: The time for SNP to build the partitions and run queries is significantly less when datasets get significantly larger.

Figure 11 shows the running time differences to partition the datasets and to enumerate triangles between SNP and PTE CD algorithms. PTE CD takes significantly more time to create ρ^2 tables and execute ρ^3 queries.

4 Conclusion And Future Work

We implemented Triangle Enumeration algorithms, such as CF, PTE Base, and PTE CD, using RDBMSs. We proposed the SNP algorithm that partitions the graph into ρ partitions and generates ρ^2 enumeration tasks. SNP exhibits comparable performance to PTE CD for medium datasets and even better for larger datasets. We experimented with billion scale graphs and enumerated tens of billions of triangles, showing that RDBMSs can perform better than GBDs by multiple orders of magnitude and process massive datasets in a consumer-grade server. One possible direction for future research is to improve performance on RDBMSs by compressing the edge list using variable byte encoding (VBE).

References

[1] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. “Database mining: A performance perspective”. In: *IEEE TKDE* 5.6 (1993), pp. 914–925.

[2] Aly Ahmed and Alex Thomo. “PageRank for Billion-Scale Networks in RDBMS”. In: *INCOS*. 2020, pp. 89–100.

[3] Renzo Angles and Claudio Gutierrez. “Survey of graph database models”. In: *ACM CSUR* 40.1 (2008), pp. 1–39.

[4] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. "PATRIC: A parallel algorithm for counting triangles in massive networks". In: *CIKM*. 2013, pp. 529–538.
[5] Jonathan W Berry et al. "Tolerating the community detection resolution limit with edge weighting". In: *Physical Review E* 83.5 (2011), p. 056119.
[6] Shumo Chu and James Cheng. "Triangle listing in massive networks". In: *ACM TKDD* 6.4 (2012), pp. 1–32.
[7] Edgar F Codd. "A relational model of data for large shared data banks". In: *Software pioneers*. Springer, 2002, pp. 263–294.
[8] "Computing source-to-target shortest paths for complex networks in RDBMS". In: *JCSS* 89 (2017), pp. 114–129.
[9] Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. "Massive graph triangulation". In: *SIGMOD*. 2013, pp. 325–336.
[10] Hosagrahar V Jagadish et al. "Big data and its technical challenges". In: *CACM* 57.7 (2014), pp. 86–94.
[11] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. "Mining Social-Network Graphs". In: *Mining of Massive Datasets*. 2nd ed. Cambridge University Press, 2014, pp. 325–383. DOI: 10.1017/CBO9781139924801.011.
[12] Sandeep Mahanthappa and BR Chandavarkar. "Data Formats and Its Research Challenges in IoT: A Survey". In: *Evolutionary Computing and Mobile Sustainable Networks*. Springer, 2020, pp. 503–515.
[13] Andrew McAfee et al. "Big data: the management revolution". In: *Harvard business review* 90.10 (2012), pp. 60–68.
[14] Carlos Ordonez and Edward Omiecinski. "Efficient disk-based K-means clustering for relational databases". In: *IEEE TKDE* 16.8 (2004), pp. 909–921.
[15] Ha-Myung Park and Chin-Wan Chung. "An Efficient MapReduce Algorithm for Counting Triangles in a Very Large Graph". In: *CIKM*. 2013.
[16] Ha-Myung Park, Sung-Hyon Myaeng, and U. Kang. "PTE: Enumerating Trillion Triangles On Distributed Systems". In: *KDD*. 2016.
[17] Ha-Myung Park et al. "Mapreduce triangle enumeration with guarantees". In: *CIKM*. 2014, pp. 1739–1748.
[18] Filippo Radicchi et al. "Defining and identifying communities in networks". In: *Proceedings of the National Acad Sciences* 101.9 (2004), pp. 2658–2663.
[19] Thomas Schank. "Algorithmic aspects of triangle-based network analysis". In: (2007).
[20] Siddharth Suri and Sergei Vassilvitskii. "Counting triangles and the curse of the last reducer". In: *WWW*. 2011.
[21] Michael Yu et al. *AOT: Pushing the Efficiency Boundary of Main-memory Triangle Listing*. 2020. arXiv: 2006.11494 [cs.DB].
[22] Yang Zhang and Srinivasan Parthasarathy. "Extracting analyzing and visualizing triangle k-core motifs within networks". In: *ICDE*. 2012, pp. 1049–1060.