Enumération Randomisée des Triangles dans des Graphes à Grande Echelle à base de SQL

Abir Farouzi[*,***], Ladjel Bellatreche[*], Carlos Ordonez[**]
Gopal Pandurangan[**], Mimoun Malki[***]

[*]ISAE-ENSMA, Poitiers, France
(abir.farouzi, bellatreche)@ensma.fr
[**] University of Houston, Texas, USA
carlos@central.uh.edu, gopal@cs.uh.edu
[***]ESI-SBA, Sidi Bel Abbès, Algérie
(a.farouzi, m.malki)@esi-sba.dz

Résumé. Chercher tous les triangles dans un graphe donné est un problème théorique classique avec un nombre important d'applications pratiques. Plusieurs algorithmes d'énumération de triangles dans des graphes à grande échelle ont été proposés et implémentés dans des environnements centralisés/distribués avec des langages comme C/Python. Dans le contexte des bases de données, il existe une grande masse de données qui peuvent être modélisées et analysées sous forme de graphes. Offrir des solutions d'énumération de triangles avec des requêtes SQL pour ces graphes représente un enjeu important pour la communauté. Dans cet article, nous proposons un algorithme distribué randomisé implémenté à l'aide de requêtes SQL pour l'énumération des triangles dans des graphes à grande échelle. Cet algorithme assure l'équilibrage de charge entre les processeurs grâce à sa stratégie de partitionnement visant à éliminer les échanges coûteux de données entre les hôtes lors de l'énumération de triangles. Nos expériences ont montré le passage à l'échelle de notre approche déployée sur un cluster de 8 hôtes et hébergeant le SGBD Vertica.

1 Introduction

Les graphes deviennent omniprésents du fait que notre monde est de plus en plus interconnecté. L'un des problèmes fondamentaux des graphes est l'énumération des triangles. Il consiste à lister tous les triangles d'un graphe. Ce problème a suscité beaucoup d'intérêt en raison de ses nombreuses applications pratiques : l'analyse des processus sociaux dans les réseaux (Watts et Strogatz, 1998), l'extraction de sous-graphes denses (Wang et al., 2010), les jointures dans les bases de données (Ngo et al., 2013). Ainsi, d'autres applications sont largement commentées dans (Chu et Cheng, 2012; Berry et al., 2015). En outre, il est important de souligner que les processus de détection et de comptage de triangles sont plus faciles que leur énumération qui est une tâche cruciale pour plusieurs applications. En effet, l'énumération teste la possibilité de former un triangle à partir des triplets d'arêtes (Farouzi et al., 2020). Donc, en utilisant le résultat de l'énumération nous pouvons facilement obtenir le nombre des triangles dans le graphe. En revanche, un simple comptage ne fournit pas forcément la liste des

triangles résultante . Dans la pratique, la plupart des graphes ayant n sommets et m arêtes sont clairsemés avec $m = O(n)$ définissant une complexité de $O(n^3)$ pour l'énumération de tous les triangles du graphe. De ce fait, l'énumération devient rapidement coûteuse avec les graphes à grande échelle.

Des solutions SQL-like ont émergées pour établir la tâche d'énumeration de triangles dans les graphes. Cela est motivé par la volonté de plusieurs chercheurs défendeurs du principe "Inside DBMSs" (Ordonez, 2013; Zhao et Yu, 2017). *Il signifie que toutes les opérations sur une base de données doivent se faire à l'intérieur du SGBD hébergeant la donnée*. Également, il existe beaucoup de données de graphes qui sont stockés sous forme autre que graphe, principalement dans des bases de données relationnelles. Dans cette optique, les auteurs dans (Ordonez et al., 2017; Cabrera et Ordonez, 2017) ont étudié quelques problèmes de graphes comme la fermeture transitive et le classement des pages (PageRank) avec des implémentations sur des SGBDs avec différents modèles de stockage (en lignes, en colonnes et en array). Les résultats de ces travaux ont été comparés avec des systèmes dédiés pour l'analyse des graphes comme *Spark GraphX* et ont montré leur efficacité. *Cela a joué en faveur des défenseurs de ce principe*.

Dans ce travail, nous présentons un algorithme randomisé parallèle (Pandurangan et al., 2018) pour résoudre le problème d'énumération de triangles suivant le paradigme "Inside DBMSs" déployé dans un environnement distribué. Plus précisément, il est implémenté à l'aide des requêtes SQL sur un cluster de 8 hôtes hébergeant le SGBD *Vertica*. Dans un tel environnement, le problème d'équilibrage de charges entre les processeurs est un enjeu crucial pour assurer un temps d'exécution optimal.

Notre article est organisé comme suit : La Section 2 présente un aperçu des travaux connexes. Les concepts et notations préliminaires sont décrits dans la Section 3. La Section 4 présente notre algorithme et ses propriétés. La Section 5 détaille nos résultats expérimentaux. La Section 6 conclut l'article.

2 Travaux similaires

Nous résumons dans cette section les applications liées au traitement des graphes sur les SGBDs ainsi que les travaux récents des algorithmes d'énumération des triangles dans les graphes à grande échelle.

Le traitement des graphes par des SGBDs a suscité l'intérêt des recherches ces dernières années. Zhao et Yu (2017) ont revu la prise en charge du traitement des graphes dans le SGBD relationnel au niveau SQL. Ordonez et al. (2017) ont étudié l'optimisation des requêtes récursives sur deux problèmes de graphe complémentaires : la fermeture transitive et la multiplication de matrices d'adjacence. Les auteurs ont prouvé expérimentalement que le SGBD en colonnes est le plus rapide avec l'optimisation ajustée de requêtes. Cabrera et Ordonez (2017) ont étudié la résolution de quatre problèmes de graphe importants : accessibilité, chemin le plus court à source unique, composants faiblement connectés et classement des pages à l'aide de requêtes relationnelles. D'autres travaux comme (Sun et al., 2015; Hassan et al., 2017) ont stocké les graphes dans des tables relationnelles avec un schéma optimisé pour les requêtes en ajoutant une couche supplémentaire prenant en charge le traitement de graphes sur un SGBD relationnel. Notons qu'il existe de puissants systèmes pour le traitement parallèle de graphes dans l'écosystème Hadoop comme Neo4j et Spark GraphX. Leur utilisation nécessite des ef-

forts considérables afin de faire correspondre un graphe à ses informations (caractérisant ses sommets). Souvent ils offrent des langages de requêtes moins expressifs que SQL.

Les algorithmes fondamentaux pour l'énumération de triangles sont "Node iterator" (Schank, 2007) et "Edge iterator" (Itai et Rodeh, 1978) convenant à être exécuté sur une seule machine. Néanmoins, avec l'expansion de la taille des graphes, ils deviennent moins efficaces et le traitement sur la mémoire principale d'une seule machine demeure irréalisable. Certains travaux comme MGT (Hu et al., 2013) et Trigon (Cui et al., 2017) améliorent ces deux algorithmes en proposant de meilleures techniques d'entrée/sortie (E/S) qui réduisent la surcharge sur la mémoire centrale tout en gardant l'exécution sur une seule machine. D'autres travaux se concentrent sur la mise en parallèle du traitement et présentent des solutions multi-cœurs comme (Latapy, 2008; Shun et Tangwongsan, 2015).

De nombreuses solutions basées sur la distribution des données ont également été introduites. Arifuzzaman et al. (2013, 2015) ont proposé un algorithme parallèle de mémoire distribuée utilisant MPI (Message Passing Interface) basé sur l'algorithme de "node iterator" pour compter et énumérer les triangles dans les réseaux massifs. Le travail de (Zhang et al., 2019) est une autre approche basée sur la distribution du traitement sur un cluster avec réduction du nombre des messages échangés pendant l'exécution. Pandurangan et al. (2018) ont présenté un algorithme distribué randomisé pour l'énumération et le comptage des triangles dans le modèle k-machine, un modèle théorique populaire pour le calcul de graphe distribué à grande échelle (Klauck et al., 2015). Dans ce modèle, les machines $k \geq 2$ effectuent conjointement des calculs sur des graphes de n sommets (généralement, $n \gg k$). Le graphe introduit est supposé initialement être partitionné entre les k machines de manière équilibrée. La communication est de point à point par passage de messages (pas de mémoire partagée), et l'objectif est de minimiser le nombre de cycles de communications du calcul. Le travail de (Pandurangan et al., 2018) a présenté un algorithme distribué qui liste tous les triangles d'un graphe. Cet algorithme est optimal en ce qui concerne le nombre de cycles de communication. Ce travail nous a inspiré pour définir notre solution.

3 Notions préliminaires

Dans cette section, nous donnons des notions fondamentales qui facilitent la lecture de cet article. Elles couvrent quatre aspects essentiels : (1) les graphes et leur projection dans le monde des bases de données, (2) la définition du problème d'énumération des triangles, (3) un aperçu du traitement distribué des SGBD en colonnes et (4) une description du modèle distribué de calcul.

3.1 Généralités sur les graphes

Définition 1. Soit $G = (V, E)$ un graphe non orienté non pondéré avec V un ensemble non vide de sommets (nœuds) et E un ensemble d'arêtes (éventuellement vide). Nous notons $n = |V|$ et $m = |E|$. Chaque arête $e \in E$ relie entre deux sommets u, v et définit une direction (de u vers v et v vers u). Nous définissons pour chaque $u \in V,\ N(u) = \{v \in V : (u, v) \in E\}$ l'ensemble des voisins du sommet u. Ainsi, le degré de u est $deg(u) = |N(u)|$.

Par cette définition, nous permettons la présence de cliques et de cycles. Une clique définit un sous-graphe complet de G. Un cycle est un chemin qui commence et se termine sur le même sommet. Un cycle de longueur l est appelé l-cycle ; donc un 3-cycle fait référence à un triangle.

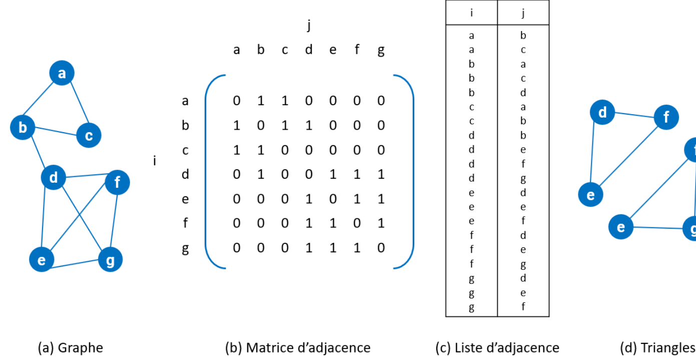

FIG. 1: Les différentes représentations d'un graphe

Mathématiquement, un graphe G peut être représenté par une matrice d'adjacence de $n \times n$ (voir Fig.1 (b)), où la cellule (i, j) contient 1 lorsqu'il existe une arête reliant le sommet i au sommet j. En termes de la base de données, un graphe G est stocké sous forme de liste d'adjacence dans une table d'arêtes $E(i, j)$ avec la clé primaire (i, j) avec i est le sommet source et j est le sommet de destination (voir Fig.1 (c)) (Cabrera et Ordonez, 2017). Chaque tuple de la table E définit l'existence d'une arête. Fig.1 (a) représente un graphe non orienté, (b) montre sa représentation en matrice d'adjacence et (c) sa représentation en liste d'adjacence.

3.2 Propriétés de triangles

Définition 2. Un triplet connecté (u, v, w) au sommet v dans un graphe (orienté ou non orienté) est un chemin de longueur 2 pour lequel v est au centre. Si $(w, u) \in E$: (u, v, w) est un triplet fermé appelé triangle, sinon c'est un triplet ouvert nommé coin ou triade ouvert. Un triangle contient trois triplets fermés : (u, v, w), (v, w, u) et (w, u, v).

Définition 3. Un triangle noté $\Delta_{(u,v,w)}$, est la plus petite clique du graphe G composée de trois sommets distincts u, v et w. Le triangle formé par ces sommets implique l'existence de trois arêtes (u, v), (v, w) et (w, u). L'ensemble $\Delta(G)$ comprend tous les triangles $\Delta_{(u,v,w)}$ du graphe G. Fig.1 (d) représente un exemple de triangle dans un graphe non orienté (Fig.1. (a)). Notez que l'ordre des deux triangles produits dans notre exemple est *lexicographique*. Cela élimine la redondance dans les triangles obtenus.

Propriété 1. Deux triangles t_1 et t_2 peuvent appartenir à la même clique.

Fig.1 (d) présente un exemple de deux triangles (d,e,f) et (e,f,g) qui appartiennent à la même clique formée de sommets {d,e,f,g}.

3.3 Modèles de Stockage dans les SGBDs

Afin d'énumérer les triangles, des requêtes SQL standard peuvent être employées en fonction des opérations SPJ (sélection, projection et jointure). Ces opérations peuvent être utiles pour simplifier et comprendre le problème en formulant la solution avec l'algèbre relationnelle (σ, π et $\bowtie$) suivi de sa traduction en requêtes SQL qui peuvent être exécutées en parallèle sur un SGBD distribué . Pour traiter un graphe dans un système de base de données, la meilleure définition de stockage est une liste d'adjacence en tant que table d'arêtes $E(i, j)$ où l'on sup-

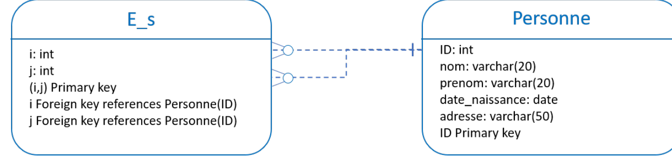

FIG. 2: Modèle physique de base de données pour la représentation du graphe

pose que l'arête possède une direction de i vers j. Si le graphe est orienté, seul l'arête orientée est insérée, sinon deux arêtes pour chaque paire de sommets connectés (dans les deux directions) sont incluses. De cette manière, nous pouvons toujours obtenir les sommets du triangle dans l'ordre lexicographique $\{u, v, w\}$ au lieu de $\{v, w, u\}$ ou $\{w, u, v\}$ avec $u < v < w$.

Fig. 2 illustre le schéma physique d'un graphe dans un SGBD, où la table E_s représente la liste d'adjacence portant toutes les arêtes, et la table «Personne» stocke toutes les informations relatives aux sommets (i et j) de la table E_s.

Dans ce travail, nous avons opté pour des SGBDs relationnels définissant un modèle de stockage en colonnes tels que Vertica ou MonetDB. Ces systèmes présentent une meilleure efficacité d'opérations de lecture/écriture par rapport aux SGBDs avec un stockage en ligne (Ordonez et al., 2017). En fait, le stockage physique entre les deux types de SGBD est considérablement différent. Les SGBD de ligne utilisent des index qui se trouvent dans l'emplacement physique des données. Alors que les SGBD en colonnes reposent sur des projections (des collections optimisées de colonnes de table qui fournissent un stockage physique des données). Ces projections peuvent contenir une partie ou la totalité des colonnes d'une ou plusieurs tables. Par exemple, Vertica permet de stocker des données par projections dans un format qui optimise l'exécution des requêtes. En addition, les SGBD en colonne sont dotés des systèmes de compression performants.

3.4 Modèle de calcul parallèle

Supposons un cluster de k-machines (k représente le nombre d'hôtes dans le cluster) construit sur une architecture *Shared-Nothing*. Chaque machine peut communiquer directement avec les autres. Toutes les machines définissent une configuration d'installation homogène fournissant au moins les exigences matérielles et logicielles minimales pour les meilleures performances du SGBD en colonnes. Le nombre de machines k doit être choisi suivant la formule : $k = p^3$ avec $p \in \mathbb{N}$. (voir la section 4.2 pour plus de détails).

4 Énumération de triangles

Dans cette section, nous présentons notre contribution dédiée à la résolution du problème d'énumération de triangles en parallèle en utilisant les requêtes SQL.

4.1 Algorithme Standard

L'énumération de triangles dans un graphe G peut se faire en deux itérations principales, la première vise à identifier tous les coins dirigés dans le graphe introduit tandis que la seconde se concentre sur la recherche de l'existence d'une arête reliant les extrémités de chaque coin.

Fondamentalement, lister les triangles en utilisant l'algorithme standard est effectué par trois boucles imbriquées, qui peuvent être traduites en SQL par trois auto-jointures de la table E ($E1 \bowtie E2 \bowtie E3 \bowtie E1$) sur $E1.j = E2.i$, $E2.j = E3.i$ et $E3.j = E1.i$ respectivement avec $E1 = E, E2 = E$ et $E3 = E$ ($E1$, $E2$ et $E3$ sont des tables d'alias de E). Cependant,

comme seuls les triangles suivant un order lexicogrphique ($v_1 < v_2 < v_3$) sont listés, nous pouvons éliminer la troisième auto-jointure en prenant à la fois $E2.j = E3.i$ et $E3.j = E1.i$ comme condition de la deuxième auto-jointure. Par conséquent, le processus mentionné ci-dessus peut être formulé en utilisant seulement deux auto-jointures sur la table E ($E \bowtie E \bowtie E$). Dans les requêtes SQL ci-dessous, nous utilisons E_dup qui est un réplica de E pour accélérer le traitement des jointures locales. En effet, le partitionnement de la première table par i et de le réplica par j divise les deux tables correspondantes en petits morceaux sur la base des colonnes susmentionnées, ce qui rendra la jointure locale avec la condition $E.j = E.i$ plus rapide.

```
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;
```

4.2 Algorithme Randomisé

L'algorithme randomisé d'énumération de triangles proposé par (Pandurangan et al., 2018) a deux étapes principales : (1) partitionner aléatoirement les sommets en fragments de taille équivalente et (2) énumérer d'une manière parallèle les triangles de chaque fragment. Plus précisément, l'ensemble de sommets est partitionné en $p = k^{1/3}$ sous-ensembles aléatoires (ainsi chaque sous-ensemble aura n/p sommets), où k est le nombre de machines. Ensuite, chaque triplet de sous-ensembles de sommets (il y a un total de ($p^3 = k$ triplets, y compris les répétitions) et les arêtes reliant les sommets de chaque sous-ensemble sont affectées à chacune des k machines. Chaque machine calcule ensuite les triangles du sous-graphe induits localement par le sous-ensemble affecté à elle. Étant donné que chaque triplet possible est pris en compte, chaque triangle sera compté (il est facile de supprimer les redondances, en utilisant l'ordre lexicographique des sommets, comme décrit dans la section 3.3). Par conséquent, l'exactitude de notre algorithme est facile à établir.

L'intuition principale derrière l'algorithme randomisé est qu'une partition aléatoire de sommets, définie en sous-ensembles de taille égale, *équilibre* le nombre d'arêtes attribuées à une machine. Cela réduit considérablement la communication, mais également la quantité de travail effectuée par chaque hôte. Bien que cet équilibre soit facile à voir sous les attentes (un calcul facile utilisant la linéarité des attentes montre que sur la moyenne, le nombre d'arêtes est équilibré) ; cependant, il peut y avoir un écart important. Il est montré dans (Pandurangan et al., 2018) via une analyse probabiliste, que le nombre d'arêtes attribuées par machine est limité par $\tilde{O}(\max\{m/k^{2/3}, n/k^{1/3}\})$. Nous notons que l'algorithme randomisé est toujours correct (c'est-à-dire de type Las Vegas), tandis que la randomisation est utile pour améliorer les performances.

4.3 Chargement du graphe

La première étape pour énumérer les triangles consiste à lire le graphe introduit sur un hôte. Si le graphe est orienté, la table d'arêtes E_s est construite à partir des arêtes orientées. Sinon, pour chaque tuple *(i,j)* inséré dans la table E_s, *(j,i)* est également inséré (voir la section 3.3). Les requêtes suivantes sont utilisées pour lire le graphe dans la base de données :

```
CREATE TABLE E_s(i int,j int);
COPY E_s FROM "link/to/graph_data_set";
/*Si le graphe est non oriente */
INSERT INTO E_s SELECT j,i FROM E_s;
```

4.4 Partitionnement du graphe

Consédirant un modèle de k-machines tel que présenté dans la section 3.4. Le graphe G est partitionné sur les k machines en utilisant le *modèle aléatoire de partition de sommets* qui est basé sur l'affectation de chaque sommet et ses voisins à une machine aléatoire parmi les k machines (Pandurangan et al., 2018). En effet, notre stratégie de partitionnement (voir section 4.2) vise à partitionner l'ensemble V de sommets de G en $p = k^{1/3}$ sous-ensembles de n/p sommets chacun. Notamment, pour un graphe $G = (V, E)$, une machine k héberge $G_k = (V_k, E_k)$ un sous-graphe de G, où $V_k \subset V$ et $\cup_k V_k = V$, G_k doit être formé de telle sorte que pour chaque $v \in V_k$ et $u \in N(v)/(u, v) \in E_k$.

D'abord, la table "V_s" dans la requête ci-après garantit que chaque sommet $v \in V$ sélectionne indépendamment et uniformément au hasard une couleur parmi un ensemble de $p = k^{1/3}$ couleurs distinctes en utilisant la fonction *randomint* de Vertica. À noter que 1 et 2 dans la requête ci-dessous se réfèrent à deux sous-ensembles de couleurs pour le modèle à 8 machines. Une affectation déterministe de triplets de couleurs dans la table "Triplet" dans les requêtes suivantes affecte un des k triplets de couleurs possibles formés par les p couleurs distinctes à une machine spécifique. Cela peut être traduit par les requêtes suivantes :

```
/*Chaque sommet choisit une couleur al atoire de qr=k^(1/3), pour k=8, qr=2 */
CREATE TABLE V_s(i int,color int);
INSERT INTO V_s
    SELECT i,randomint(qr)+1
        FROM
            (SELECT DISTINCT i FROM E_s
            UNION
            SELECT DISTINCT j FROM E_s
            )V;
/*Les valeurs dans triplet_file pour le modele 8-machines (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";
```

Ensuite, pour chaque arête qu'elle détient, chaque machine désigne une machine aléatoire comme proxy d'arêtes et envoie toutes ses arêtes aux proxys d'arêtes respectifs, contruisant ainsi la table "E_s_proxy". Cette table est construite en colorant les extrémités de chaque arête de la table "E_s" en fonction de la couleur choisis par chaque sommet dans la table "V_s", en utilisant une double jointure entre les deux tables. *La construction de "E_s_proxy" est l'étape la plus importante car tout le partitionnement en dépend*. En utilisant cette table, nous pouvons identifier les couleurs choisies par les sommets d'extrémités de chaque arêtes, ce qui permet la requête suivante, responsable de la construction de la table "E_s_local", de décider quel hôte détiendra quelle arête en prenant en compte le triplet affecté à chaque machine.

```
/*Envoi des ar tes aux proxys*/
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;
/*Collecte des ar tes    partir des proxys*/
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 AND 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 AND 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 AND E.i>E.j;
```

Avoir $E.i < E.j$ et $E.i > E.j$ dans la dernière requête garantit l'énumération de chaque triangle sur une machine unique. Par exemple, un triangle (u, v, w) avec $u < v < w$ sélectionnant les couleurs (c, b, c) est énuméré sur une machine unique M ayant le triplet (c, b, c) qui lui est affecté, donc les triangles comme (w, u, v) et (v, w, u) où $w > u < v$ et $v < w > u$ respectivement ne seront pas pris en compte, ce qui élimine la redondance des triangles énumérés.

4.5 Énumération locale des triangles

Afin d'énumérer les triangles localement, chaque machine examine ses arêtes dont les extrémités se trouvent dans deux sous-ensembles distincts parmi les trois sous-ensembles qui lui sont affectés. Cela se produit en deux étapes :

1. Chaque machine identifie tous les coins possibles dont les sommets ont les couleurs et l'ordre identiques à ceux de son triplet
2. Pour générer les triangles, chaque hôte vérifie s'il existe une arête reliant les sommets d'extrémité de chaque coin répertorié

Les étapes susmentionnées sont assurées par deux auto-jointures locales sur la table E_s_local ($E_s_local \bowtie E_s_local \bowtie E_s_local$) sur les colonnes $E_s_local.j = E_s_local.i$, sur chaque hôte localement et d'une manière parallèle entre les hôtes. Les requêtes sont présentées comme suit :

```
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;
```

Comme expliqué dans la section précédente, avoir $E1.i < E1.j$ et $E2.i < E2.j$ dans la requête élimine la redondance. La dernière jointure avec la table "Triplet" élimine la possibilité que des triangles ayant des sommets de même couleur soient émis par d'autres machines que les leurs. Fig.3 illustre le partitionnement aléatoire et l'énumération de triangles dans un cluster de 8 machines. La communication entre les machines est nécessaire lors de la création des sous-graphes ou de la collecte des arêtes à partir de proxys. Autrement, le traitement est local.

Afin de vérifier la similitude entre les triangles générés par l'algorithme randomisé et ceux avec l'algorithme standard, une différence entre les résultats des deux algorithmes peut être utilisée. Les requêtes suivantes sont exécutées dans la section 5 pour prouver la similitude des retours des deux algorithmes (notez que *Triangle* est une table contenant la liste des triangles résultants de l'exécution de l'algorithme randomisé) :

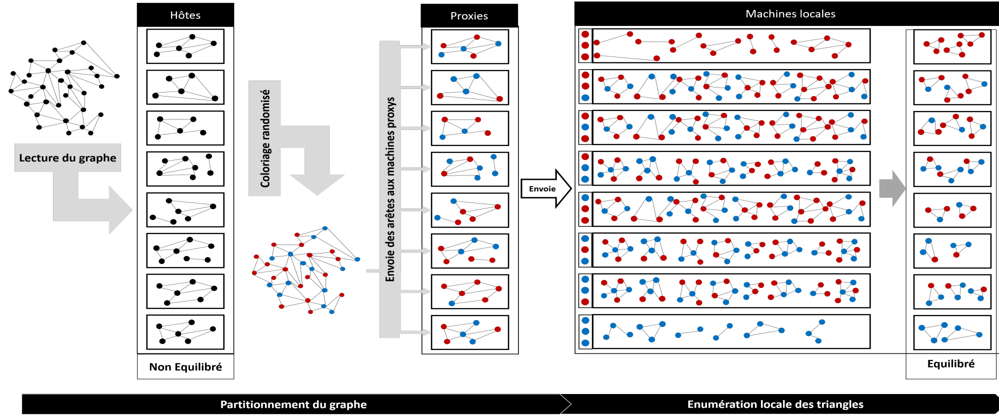

FIG. 3: Énumération randomisée des triangles sur un modèle de 8 machines.

```
SELECT v1, v2, v3 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 Équilibrage de charge de l'algorithme randomisé

Le calcul parallèle est considéré comme terminé lorsque tous les hôtes terminent leur traitement et produisent les résultats. Par conséquent, la réduction du temps d'exécution nécessite que tous les processeurs terminent leurs tâches presque en même temps (Arifuzzaman et al., 2013). Cela est possible si tous les hôtes acquièrent un volume équitable de données sur lesquelles ils peuvent lancer des traitements.

Nous avons mentionné dans la Section 4.4 que chaque sommet de l'ensemble V choisit de manière aléatoire et indépendante une couleur c parmi les $p = k^{1/3}$ couleurs distinctes. Cela donne lieu à une partition de l'ensemble des sommets V en p sous-ensembles s_p de $O(n/p)$ sommet chacun. Chaque machine reçoit alors un sous-graphe $G_k = (V_k, E_k)$ de G. Comme mentionné au début de la section 4.2, l'analyse de (Pandurangan et al., 2018) montre que le nombre d'arêtes parmi les sous-graphes G_k est relativement équilibré avec une probabilité élevée. Par conséquent, chaque machine traite essentiellement le même nombre d'arêtes qui conduit à un équilibre de charge. Les extrémités de chaque $e \in E_k$ sont dans deux sous-ensembles s_p. Cela signifie que chaque triangle (u, v, w) satisfaisant $u < v < w$ sera affiché.

4.7 Complexité temporelle de l'algorithme randomisé

La complexité temporelle prise par une machine est proportionnelle au nombre d'arêtes et de triangles qu'elle gère. Chaque machine gère un triplet particulier de couleurs (c_x, c_y, c_z) donc elle gère sous-ensembles aléatoire de sommets de taille $O(n/k^{1/3})$.

Le nombre le plus défavorable de triangles dans ce sous-ensemble est $O(n^3/k)$. Cependant, le nombre d'arêtes est beaucoup plus faible. En effet, l'idée clé de l'algorithme randomisé comme indiqué dans (Pandurangan et al., 2018), est qu'un sous-ensemble aléatoire de la taille mentionnée dans la section 4.2 (c'est-à-dire $O(n/k^{1/3})$) n'aura pas plus de $\max\{\tilde{O}(m/k^{2/3}), n/k^{1/3})\}$ arêtes avec une probabilité élevée. Par conséquent, chaque machine ne gère que

TAB. 1: Ensembles de données

Données	n	m	Triangle	Type	Orienté	Asym	Source
LiveJournal	3,997k	34,681k	177,820k	Réel	Non	Faible	SNAP
as-Skitter	1,696k	11,095k	28,769k	Réel	Non	Faible	SNAP
flickr-link	105k	2,316k	548,174k	Réel	Non	Haute	KONECT
hyves	1,402k	2,777k	752k	Réel	Non	Haute	KONECT
Graph500_s19	335k	15,459k	186,288k	Synt	Oui	Haute	Généré
Linear	141k	49,907k	3,300,095k	Synt	Oui	Haute	Généré
Geometric	8k	22,361k	13,078,242k	Synt	Oui	Haute	Généré

quelques arêtes avec une probabilité élevée. Le fait que le nombre de triangles peut être répertorié en utilisant un ensemble d'arêtes ℓ est $\Omega(\ell^{3/2})$ ((Pandurangan et al., 2018)), le nombre de triangles que chaque machine doit gérer est au maximum $\max\{\tilde{O}(m^{3/2}/k, n^{3/2}/k^{1/2})\}$.

Notre algorithme assure une accélération optimale (linéaire) (sauf, peut-être pour les graphes très clairsemés) pour ces deux raisons :

— le nombre maximal de triangles (distincts) dans un graphe de m arêtes est au plus $m^{3/2}$.

— Chaque machine gère essentiellement $m^{3/2}/k$ (lorsque $m^{3/2}/k > n^{3/2}/k^{1/2}$).

Dans la partie expérimentale, nous montrons que notre algorithme permet à chaque machine de gérer approximativement le même nombre de triangles, ce qui implique un équilibrage de charge parfait.

5 Évaluation expérimentale

Dans cette section, nous détaillons notre processus expérimental avec les différentes configurations que nous avons utilisées. Nous présentons et discutons ensuite nos résultats.

Configuration logicielle et matérielle Les expérimentations ont été menées sur un cluster de 8 hôtes ($k = 8$), chacun avec un processeur à 4 cœurs virtuels de type GeniuneIntel fonctionnant à $2,4$ Ghz, 48 Go de mémoire principale, 1 To de stockage, 32 Ko cache L1, 4 Mo cache L2 et un système d'exploitation serveur Linux Ubuntu 18.04. Le total de RAM sur le cluster est de 384 Go et le total du stockage sur disque est de 8 TB et un total de 32 cœurs de processeur pour le traitement. Nous avons utilisé le SGBD Vertica pour déployer notre solution à l'aide des requêtes SQL. Le langage Python est utilisé comme un langage hôte pour générer nos requêtes et les soumettre à la base de données vue sa rapidité par rapport à JDBC.

Ensemble de données Tableau 1 résume les ensembles de données utilisés dans nos expérimentations. Nous avons utilisé sept ensembles de données de graphes réels (Leskovec et Krevl, 2014; Kunegis, 2013) et synthétiques (Synt). Ces ensembles ont de différentes tailles, structures (cycles et cliques) et formes (orienté et non orienté). Le tableau susmentionné présente pour chaque ensemble de données : le nombre de ses sommets, ses arêtes et ses triangles avec son type, sa distribution de données nommée asymétrie(asym) et sa source.

Partitionnement de la table d'arêtes Afin d'optimiser ses requêtes, les SGBDs du modèle de stockage en colonnes définissent deux types de parallélisme : (i) le partitionnement des grandes tables en fonction d'une ou plusieurs valeurs de ses colonnes au local (sur chaque hôte individuellement) et (ii) la segmentation des projections sur le cluster (distribution des don-

nées sur l'ensemble des machines du cluster). Pour le problème d'énumération de triangles, le graphe est principalement partitionné localement par le sommet source i ou le sommet destination j pour accélérer la jointure locale (voir la section 4.1). Alors que la segmentation de l'ensemble de sommets V entre les hôtes est effectuée de manière aléatoire en utilisant le modèle aléatoire de partition de sommets mentionné dans la Section 4.4. Dans le SGBD Vertica, ce processus est effectué à l'étape DDL (Data Definition Language) via la définition de la clause de la segmentation dans la requête de création de la projection :

```
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;
```

En fait, Vertica attribue à chaque machine un segment (intervalle de valeurs de hachage) compris entre 0 et 4 octets. Initialement, la valeur de hachage de chaque tuple est calculée à l'aide de la clause de segmentation puis en fonction de la valeur résultante, le tuple est envoyé à la machine correspondante possédant sa valeur de hachage dans son segment. Nous avons exploité cette propriété pour envoyer chaque arête à sa machine correspondante. Cela nous a permis d'effectuer des jointures localement sur chaque machine indépendamment des autres hôtes en spécifiant le nom de l'hôte dans la clause *WHERE* de la requête SQL d'énumération de triangles.

Énumeration de triangles Dans ce qui suit, nous analysons les performances de l'algorithme randomisé puis nous comparons ses résultats à ceux de l'algorithme standard. Notre objectif principal est de montrer expérimentalement que le nombre de triangles générés est presque le même sur tous les hôtes. Pour cela, nous présentons par les graphiques circulaires (camembert) des Figs. 4 et 5 deux exemples de nombre de triangles (TC) générés en utilisant deux ensembles de données. Ce nombre sur chaque hôte représente 1/8 du nombre total, ce qui confirme notre affirmation théorique.

Fig. 6 représente le nombre de triangles émis sur chaque machine pour les autres ensembles de données. Il est évident que la distribution des triangles résultants est équilibrée sur les machines. De plus, Fig. 7 présente le temps d'exécution de l'algorithme randomisé sur chaque hôte. Le graphique des lignes montre que tous les processeurs terminent leurs tâches presque en même temps sur tous les ensembles de données avec des décalages de petite durée dû à la présence d'asymétrie ou au mouvement des données lors la phase de pré-traitement (lecture et partitionnement du graphe). Par exemple, les ensembles de données présentent une petite surcharge sur la machine 1 responsable de la lecture et le chargement des données. Cela explique ce temps de traitement supplémentaire. Pour les graphes ayant une forte asymétrique, certaines machines prennent moins de temps et finissent en premier, car ils hébergent moins de cliques par rapport aux autres.

Nous comparons maintenant les résultats (décrits dans le Tableau 2) entre deux algorithmes : standard et randomisé. Nous remarquons que le nombre de triangles est le même que le nombre de triangles attendu défini dans le Tableau 1. L'équilibrage de charge n'est pas assuré et de nombreux mouvements de données sont effectués pour exécuter les jointures de l'algorithme standard. Ainsi, nous avons ajouté l'étape de pré-traitement car nous voulons garantir l'équilibrage de charge sur l'algorithme standard, mais cela génère un sur-coût impor-

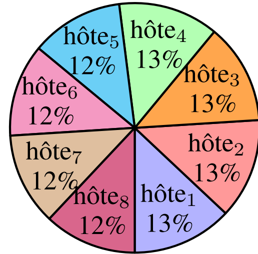

FIG. 4: TC de as-Skitter par machine

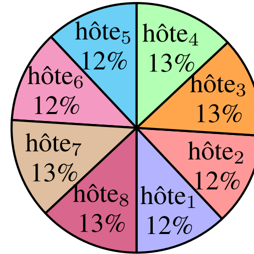

FIG. 5: TC de Geometric par machine

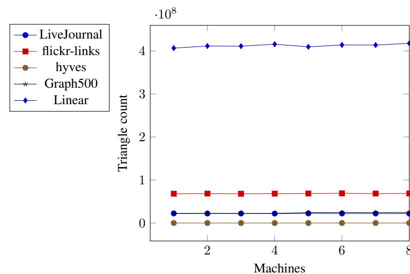

FIG. 6: Nombre de triangles (TC)

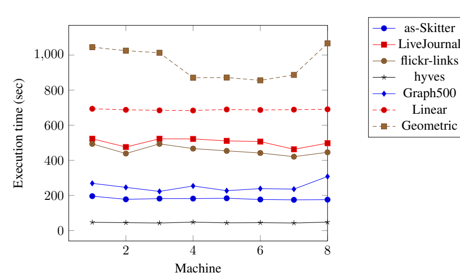

FIG. 7: Temps d'exécution

TAB. 2: Nombre de triangles (en millions) et temps d'exécution (secs)

Données	TC_{stand}	TC_{rand}	Standard	Rééquilibré	Randomisé
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

tant en temps d'exécution. La colonne "Rééquilibré" du Tableau 2 présente le coût d'une telle approche.

La colonne "Randomisé" donne la durée moyenne d'exécution de l'algorithme randomisé sur les différents ensembles de données. À mesure que la taille de l'ensemble de données augmente ou présente une asymétrie élevée, les performances de l'algorithme randomisé deviennent meilleures que celles de l'algorithme standard, comme dans le cas du graphe orienté *Graph500_s19* et du graphe non orienté *flickr-link* ou *hyves* respectivement. De plus, lorsque l'ensemble de données du graphe asymétrique devient beaucoup plus large comme *Linear* et *Géométrique*, l'algorithme standard échoue en raison des mouvements des données pendant le traitement des jointures, provoquant ainsi des problèmes de mémoire, tandis que l'algorithme randomisé réussit à terminer la tâche car les triangles sont listés localement sur le sous-graphe stockées sur chaque machine et aucun échange de données n'est effectué.

Les colonnes TC_{stand} et TC_{rand} résument le nombre de triangles résultant en utilisant les

deux algorithmes, nous remarquons que ces derniers donnent le même nombre de triangles. Nous avons expérimentalement exécuté la requête SQL de différence d'ensemble présentée dans la section 4.5 entre les deux résultats. Cette dernière a renvoyé un ensemble vide pour chaque ensemble de données, par conséquent, la similitude des résultats des deux algorithmes est confirmée.

6 Conclusions

Nous avons présenté un algorithme randomisé parallèle pour l'énumération des triangles sur les grands graphes en utilisant les requêtes SQL. Notre approche fournit une solution élégante,abstraite et plus courte par rapport aux langages traditionnels comme C ++ ou Python. Nous avons montré que notre approche évolue bien avec la taille du graphe et sa complexité, en particulier avec les graphes asymétriques. Notre stratégie de partitionnement garantit une répartition équilibrée de la charge des données entre les hôtes. Les résultats expérimentaux sont prometteurs et sont conformes à nos déclarations théoriques.

Actuellement nous étudions en profondeur l'algorithme randomisé avec des graphes denses et complexes. Nous menons également d'autres expérimentations pour le comparer avec des solutions issues des systèmes d'analyse de graphe.

Références

Arifuzzaman, S., M. Khan, et M. Marathe (2013). Patric : A parallel algorithm for counting triangles in massive networks. In *CIKM*, pp. 529–538.

Arifuzzaman, S., M. Khan, et M. Marathe (2015). A space-efficient parallel algorithm for counting exact triangles in massive networks. In *HPCC*, pp. 527–534.

Berry, J. W., L. A. Fostvedt, D. J. Nordman, C. A. Phillips, C. Seshadhri, et A. G. Wilson (2015). Why do simple algorithms for triangle enumeration work in the real world ? *Internet Mathematics 11*(6), 555–571.

Cabrera, W. et C. Ordonez (2017). Scalable parallel graph algorithms with matrix–vector multiplication evaluated with queries. *DAPD Journal 35*(3-4), 335–362.

Chu, S. et J. Cheng (2012). Triangle listing in massive networks. *ACM Trans. Knowl. Discov. Data 6*(4), 17.

Cui, Y., D. Xiao, D. B. Cline, et D. Loguinov (2017). Improving i/o complexity of triangle enumeration. In *ICDM*, pp. 61–70.

Farouzi, A., L. Bellatreche, C. Ordonez, G. Pandurangan, et M. Malki (2020). A scalable randomized algorithm for triangle enumeration on graph based on SQL queries. In *To Appear in DaWaK Conference*.

Hassan, M. S., T. Kuznetsova, H. C. Jeong, W. G. Aref, et M. Sadoghi (2017). Empowering in-memory relational database engines with native graph processing. *CoRR abs/1709.06715*.

Hu, X., Y. Tao, et C.-W. Chung (2013). Massive graph triangulation. In *ACM SIGMOD*, pp. 325–336.

Itai, A. et M. Rodeh (1978). Finding a minimum circuit in a graph. *SIAM Journal on Computing 7*(4), 413–423.

Klauck, H., D. Nanongkai, G. Pandurangan, et P. Robinson (2015). Distributed computation of large-scale graph problems. In *ACM-SIAM SODA*, pp. 391–410.

Kunegis, J. (2013). Konect : The koblenz network collection. Association for Computing Machinery.

Latapy, M. (2008). Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical computer science 407*(1-3), 458–473.

Leskovec, J. et A. Krevl (2014). SNAP Datasets : Stanford large network dataset collection. `http://snap.stanford.edu/data`.

Ngo, H. Q., C. Ré, et A. Rudra (2013). Skew strikes back : new developments in the theory of join algorithms. *SIGMOD Record 42*(4), 5–16.

Ordonez, C. (2013). Can we analyze big data inside a dbms ? In *DOLAP*, pp. 85–92.

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

Pandurangan, G., P. Robinson, et M. Scquizzato (2018). On the distributed complexity of large-scale graph computations. In *SPAA*, pp. 405–414.

Schank, T. (2007). Algorithmic aspects of triangle-based network analysis.

Shun, J. et K. Tangwongsan (2015). Multicore triangle computations without tuning. In *ICDE*, pp. 149–160.

Sun, W., A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, et G. Xie (2015). Sqlgraph : An efficient relational-based property graph store. In *ACM SIGMOD*, pp. 1887–1901.

Wang, N., J. Zhang, K.-L. Tan, et A. K. H. Tung (2010). On triangulation-based dense neighborhood graph discovery. *Proc. VLDB Endow. 4*(2), 58–68.

Watts, D. J. et S. H. Strogatz (1998). Collective dynamics of 'small-world' networks. *Nature 393*, 440–442.

Zhang, Y., H. Jiang, F. Wang, Y. Hua, D. Feng, et X. Xu (2019). Litete: Lightweight, communication-efficient distributed-memory triangle enumerating. *IEEE Access 7*, 26294–26306.

Zhao, K. et J. X. Yu (2017). All-in-one: Graph processing in rdbmss revisited. In *SIGMOD*, pp. 1165–1180.

Summary

Finding all the triangles in a given graph is a classic theoretical problem with a large number of practical applications. Several algorithms for enumerating triangles in large graphs have been proposed and implemented in centralized and distributed environments with languages like C or Python. In terms of databases, there is a large volume of data that can be modeled and analyzed in the form of graphs. Offering solutions for enumerating triangles with SQL queries for these graphs represents an important challenge for the database community. In this article, we propose a distributed randomized algorithm implemented using SQL queries to enumerate triangles in large graphs. This algorithm ensures load balancing between processors thanks to its partitioning strategy aiming to eliminate costly data exchanges between hosts when enumerating triangles. We have experimentally shown the scalability of this approach deployed on 8 machines cluster hosting Vertica DBMS.