Exploration des bases de données orientées graphe : Énumération des triangles dans les graphes à grande échelle

Abir Farouzi

THÈSE

pour l'obtention du Grade de

Docteur de l'Ecole Supérieure en Informatique, 08 mai 1945 (ESI-SBA)
et **de l'École Nationale Supérieure de Mécanique et d'Aérotechnique (ISAE-ENSMA)**
(Diplôme National - Arrêté du 25 mai 2016)

Secteur de Recherche : Informatique et applications

Présentée par :

Abir FAROUZI

****

Exploration des bases de données orientées graphe : Énumération des triangles dans les graphes à grande échelle

****

Soutenue le 02 décembre 2023
devant la Commission d'Examen

****

JURY

Président :		
CHIKH Azeddine	Professeur	Université de Tlemcen, Algérie
Directeurs :		
MALKI Mimoun	Professeur	ESI-Sidi Bel Abbès, Algérie
BELLATRECHE Ladjel	Professeur	ISAE-ENSMA, Poitiers, France
Examinateurs :		
BENSLIMANE Djamal	Professeur	Université Lyon 1, France
KECHADI Mohand Tahar	Professeur	University College Dublin, Ireland
SLAMA Zohra	MCA	Université de Sidi Bel-Abbes, Algérie
Invité :		
ORDONEZ Carlos	Professeur	University of Houston, USA

Without big data analytics, companies are blind and deaf, wandering out onto the web like deer on a freeway.

— Geoffrey Moore

Remerciements

Feeling gratitude and not expressing it is like wrapping a present and not giving it.

—William Arthur Ward

Á l'achèvement de cette thèse, je suis persuadée qu'elle n'aurait pas pu voir le jour sans l'inspiration et le soutien de tant de personnes. Elle n'est que le fruit de tant d'encouragement et de foi, pour cela, je voudrais remercier tous les personnes qui ont contribué de près ou de loin à sa réussite :

— Pour commencer, je tiens à remercier mes directeurs de thèse, le Prof. **Mimoun Malki** et le Prof. **Ladjel Bellatreche** pour leur encadrement, leur patience et leur soutien tout au long de ma thèse. Les échanges constructifs que j'ai eus avec vous ont indéniablement éclairé ma voie dans l'exploration de nouvelles idées. Votre attention lors des relectures a énormément contribué à l'élaboration de ce manuscrit. Grâce à vous, j'ai appris à mener des recherches avec rigueur.

— Je tiens également à exprimer ma sincère gratitude aux Professeurs **Azeddine Chikh** et **Djamal Benslimane** d'avoir consacré leur précieux temps pour être les rapporteurs de ma thèse. J'apprécie grandement vos commentaires que j'ai soigneusement pris en compte pour améliorer la qualité de mon travail. De plus, je souhaite remercier Prof. **Mohand Tahar Kechadi**, Drs **Zohra Slama** qui m'ont fait l'honneur de faire partie du jury d'examen et dont les questions ont enrichi mon étude sous de nombreux angles différents.

— Cette thèse représente une collaboration exemplaire entre trois institutions situées sur trois continents différents. En effet, elle s'inscrit dans le cadre d'une cotutelle entre l'École Nationale Supérieure en Informatique de Sidi Bel Abbès (ESI-SBA) et l'École Nationale Supérieure de Mécanique et d'Aérotechnique (ISAE-ENSMA), en collaboration avec le département d'informatique de l'Université de Houston. A cet effet, je tiens tout particulièrement à exprimer ma profonde gratitude envers le Prof. **Carlos Ordonez** et le Prof. **Gopal Pandurangan**. Grâce à vous, j'ai énormément progressé sur le plan académique et professionnel, et j'ai appris à mener des travaux de recherche selon la vision américaine. Je souhaite également adresser mes remerciements à **Xiantian Zhou**, doctorante du Prof. Ordonez. Notre collaboration sur plusieurs projets a été extrêmement enrichissante, et j'ai vraiment apprécié travailler avec toi.

— Sur le plan technique, le traitement distribué représente un pilier primordial dans cette thèse. À cet égard, je tiens à exprimer mes remerciements au Prof. **Farouk Toumani** et à Mr. **William Guyot-Lenat** du LIMOS pour m'avoir fourni un cluster performant sur la plateforme *GALACTICA*, ce qui a permis la réalisation de mes expérimentations pour ma première contribution à DAWAK en 2020. De plus, je souhaite remercier chaleureusement l'entreprise **Micro Focus** pour m'avoir attribué une licence gratuite renouvelable de Vertica, qui m'a permis d'évaluer mes algorithmes tout au long de cette thèse

— Grâce à cette thèse, j'ai eu l'opportunité d'intégrer trois laboratoires différents : (1) LabRI-SBA au sein de son équipe ISEDAW à l'ESI-SBA, (2) LIAS au sein de son équipe IDM

à l'ISAE-ENSMA, et (3) LIFAT au sein de son équipe BDTLN à l'université de Tours. À ce titre, je tiens à exprimer ma sincère gratitude à tous les membres du laboratoire LabRI-SBA, en particulier le Prof. **Sidi Mohammed Benslimane**, qui a toujours été administrativement présent pour veiller à la bonne conduite de ma thèse. Je voudrais également remercier chaleureusement l'ensemble des membres du laboratoire LIAS, avec une mention spéciale au Prof. **Emmanuel Grolleau**, directeur adjoint, pour ses qualités humaines, son aide et sa sympathie. Je tiens à adresser des remerciements spéciaux à **Mickaël Baron** pour son soutien technique tout au long de mon travail, sa disponibilité et son enthousiasme. Sincères remerciements à **Bénédicte Boinot** pour sa gentillesse, sa générosité et toute l'aide administrative apportée pour coordonner mon projet de thèse. Enfin, un grand merci à tous les membres de l'équipe BDTLN, en particulier le Prof. **Jean-Yves Antoine**, le Prof. **Thomas Devogele**, le Prof. **Arnaud Giacommetti**, le Prof. **Evelyne Moreau**, Dr. **Béatrice Markhoff**, Dr. **Patrick Marcel**, Dr. **Nicolas Labroche**, Dr. **Veronika Peralta**, et Dr. **Mohamed Taghelit**, pour m'avoir encadrée durant mes premiers pas dans ma carrière d'enseignante.

— Un immense merci tout particulier à mes parents, **Ismail** et **Houria**, d'avoir cru en moi et de m'avoir soutenu tout au long de mes études et de mes décisions. Sans vous, je n'aurais jamais atteint mes objectifs. Votre soutien inconditionnel et vos encouragements ont été essentiels pour que je puisse m'envoler et réaliser mes rêves.

— Je tiens également à exprimer ma gratitude envers mon époux, **Bilal**, pour tout son encouragement, sa patience et son amour inconditionnel. Je sais que je t'ai souvent embêté avec mes triangles, mais tu as toujours été là pour me soutenir.

— Un spécial merci à mes sœurs, **Manel** et **Houda**, à mon frère, **Ayoub**, et à ma petite nièce, **Lyna**, pour leur présence et leur affection. Je souhaite également remercier mes beaux-parents, **Mohamed** et **Rachida**, ainsi que l'ensemble de mes beaux-frères et belles-sœurs, **Sara**, **Hadjer**, **Oussama**, **Chaima**, **Anes** et **Anfel**, pour leur soutien et leur amour. Votre présence dans ma vie est précieuse et je suis reconnaissante de vous avoir à mes côtés.

— Je tiens à exprimer ma gratitude envers toutes mes amies que j'ai rencontrées durant mes années d'études et qui ont créé tous ces moments inoubliables qui m'ont permis d'avancer toujours plus loin. Particulièrement, je tiens à remercier **Fahima**, **Manel**, **Imane**, **Wissam**, **Soumia**, **Khedidja**, **Amna**, **Faten** et **Hadjira**. Votre présence et votre amitié ont été une source d'inspiration et de soutien tout au long de ce parcours.

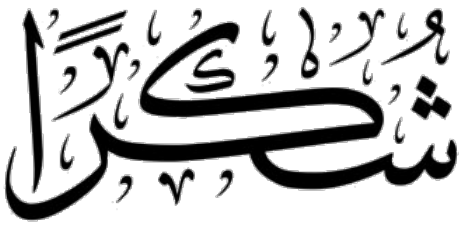


Table des matières

Liste des figures

1. *Système de Gestion de Base de Données*
2. *Random Vertex Partition*

Liste des tableaux

3. *Unité de Traitement Graphique*
4. *Message Passing Interface*
5. *Structured Query Language*

Acronymes

SGBD *Système de Gestion de Base de Données*
SGBDR *Système de Gestion de Base de Données Relationnelles*
OLAP *Online Analytical Processing*
MPP *Massively Parallel Processing*
RVP *Random Vertex Partition*
BFS *Breadth First Search*
SQL *Structured Query Language*
SSSP *single source shortest path*
CC *Connected Components*
SIMD *Single Instruction Multiple Data*
API *Interface de Programmation d'Application*
GAS *Gather-Sum-Apply-Scatter*
E/S *Entrée/Sortie*
CPU *Unité Centrale de Traitement*
GPU *Unité de Traitement Graphique*
MR *MapReduce*
MPI *Message Passing Interface*
MVC *Modèle, Vue et Contrôleur*
GPL *Licence Publique Générale*
LDD *Langage de Définition de Données*
MVC *Modèle Vue et Contrôleur*
JDBC *Java Database Connectivity interface*
BDR *Base de Données Relationnelle*
BDD *Base de Données Distribuée*
BD *Base de Données*
BDC *Bases de Données Centralisées*
SGBDD *Système de Gestion de Base de Données Distribuée*
SGBDP *Système de Gestion de Base de Données Parallèle*
UC *Unité Centrale*
SCG *Schéma Conceptuel Global*
SCL *Schéma Conceptuel Local*
SMP *Symmetric Multi-Processor (Multi-Processeur Symmétrique)*
VLSI *Very Large Scale Integration*

RLE *Run Length Encoding*

ML Machine Learning

RAM *Random Access Memory (mémoire à accès aléatoire)*

RDF *Resource Description Framework*

Introduction générale

Contexte, motivations et problématique

Exploration des graphes et extraction des triangles

Nous vivons aujourd'hui dans un monde hyperconnecté, où les liaisons entre les objets sont constamment créées, fournissant ainsi un grand volume de données. Ces données peuvent être issues de plusieurs domaine y compris les réseaux sociaux, le commerce électronique, les réseaux de transport, la biologie et bien d'autres. Beaucoup d'informations pertinentes peuvent être extraites à partir de ces connections, à partir desquelles des profits économiques, énergiques ou techniques importants peuvent être réalisés. Par conséquent, il est essentiel d'adopter une modélisation appropriée de ces données afin de les explorer, les analyser et les exploiter. Une représentation potentiellement plus efficace pourrait être réalisée à l'aide de graphes. Communément, un graphe est un ensemble de sommets et des liaisons binaires entre eux dites arêtes (voir Figure 1). Plusieurs relations dans notre quotidien peuvent être modélisée par des graphes. Á ce propos, les réseaux de transport peuvent être créés en reliant chaque deux emplacements géographiques ayant une route entre eux. Les graphes peuvent être également utilisés pour modéliser les réseaux de communication, où chaque paire de capteurs échangeant des données entre eux via un canal de communication peuvent être reliés par une arête. Ainsi, les graphes peuvent être générés simplement à partir des informations, par exemple, toutes les pages web peuvent être considérées comme des sommets, où chaque paire entre elles peut être connectée par une arêtes, si l'une des pages référence l'autre. Ce dernier exemple illustre un cas d'un graphe gigantesque (à grande échelle).

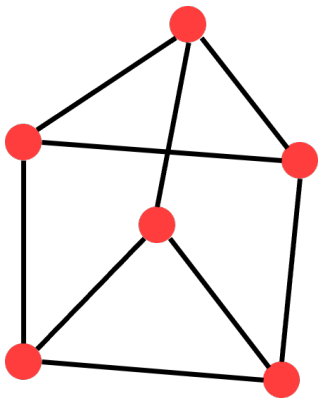


FIGURE 1 – Exemple de graphes.

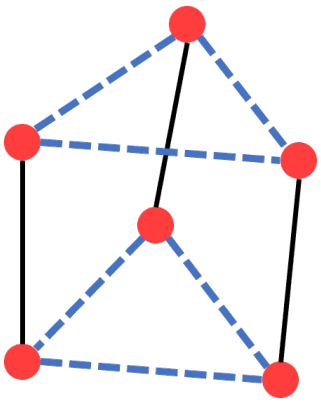


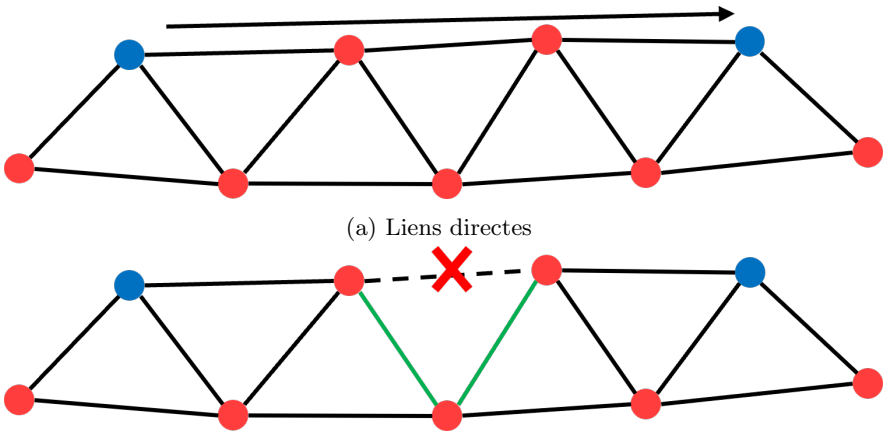
FIGURE 2 – Exemple de triangles.

De façon générale, un graphe est une structure de données issue de la théorie des graphes du domaine mathématique. Cette dernière regroupe un ensemble de traitement permettant d'interpréter les propriétés du graphe et de l'explorer. L'index d'un graphe, par exemple, permet de décrire la structure du graphe ou au moins un aspect particulier de celui-ci. L'ordre du graphe qui est le nombre de ses sommets, ou sa taille qui représente le nombre de ses arêtes peuvent servir comme de bons exemples de ses indices. La densité du graphe peut également décrire sa

structure. Elle représente le rapport entre la taille du graphe et le nombre maximal possible de ses arêtes déduit à partir de son ordre. Ainsi la densité du graphe présenté dans Figure 1 est 9/15.
De plus, chaque sommet v définit une notion de voisinage, représenté par la liste des sommets qui sont directement relié à lui. Ceci permet de définir le coefficient de clustering de chaque sommet introduit par [WS98b]. [WS98b] ont également définit le coefficient de clustering du graphe qui représente la probabilité que deux nœuds soient connectés sachant qu'ils ont un voisin en commun. Il s'agit d'un autre index pour le graphe et il est calculé en utilisant la formule suivante :

$$C = \frac{3 \times |triangles|}{|paires\_de\_voisins\_distincts|} \tag{1}$$

Le triangle est une structure dans le graphe qui intervient dans sa description, il représente un ensemble de trois sommets voisins (interconnectés par trois arêtes (voir Figure 2)). La détection de cette structure permet de déterminer d'autres structures et motifs plus complexes dans le graphe. À ce propos, considérons un réseau de métro, où quelques lignes entre les stations sont hors service à cause des travaux. Le passage par d'autres lignes dans ce cas s'avère obligatoire. Ce problème de recherche de chemin entre les stations représente un problème fondamental dans l'exploration des graphes, il s'agit du problème de la fermeture transitive. Par définition, la détection des triangles représente les deux premières itérations dans ce problème [ZFBO23]. Ainsi, la recherche des triangles dans le graphe modélisant le réseau du métro permet de trouver rapidement des chemins alternatifs. Dans Figure 3, nous présentons un cas d'un réseau, où la détection des triangles permet de trouver des chemins alternatifs dans le cas de rupture de quelques connexions.



(a) Liens directes

(b) Connexion interrompue

FIGURE 3 – Exemple d'utilisation des triangles dans la fermeture transitive.

En outre, les triangles représentent la plus petite clique[6] dans le graphe. Par conséquent, ils interviennent dans le calcul d'un autre problème important dans l'analyse des graphes. Il s'agit de la détermination de la clique maximale, où la recherche des triangles représente la première étape dans son calcul. En effet, la détermination des triangles permet de compresser le graphe, en considérant chaque triangle comme un super-sommet (supernode). Cette représentation permet de créer un nouveau graphe moins dense et plus petit, permettant de déterminer plus facilement la clique maximale dans le graphe.
Ainsi, l'énumération et le comptage de triangles représente l'étape fondamentale qui intervient dans de nombreuses applications pratiques, notamment le calcul du coefficient de clustering

6. Un sous-graphe complet où chaque couple de sommets possède une arête entre eux.

comme mentionné précédemment. Cette métrique permet de déterminer l'âge d'une communauté où une nouvelle communauté possède une faible densité de triangles[AOB18], l'analyse des processus sociaux dans les réseaux [WS98a], l'exploration de sous-graphes denses [WZTT10], les jointures dans les bases de données [NRR13], et bien d'autres.
Il est primordial de noter que l'énumération des triangles est un processus plus coûteux que leur comptage. Le comptage de triangles ne nécessite pas forcément l'énumération complète des triangles, il peut être calculé avec la formule suivante :

$$tr(A^3)/6 \tag{2}$$

Où $tr(A^3)$ représente les éléments diagonaux de la matrice d'adjacence A. En revanche, pour énumérer tous les triangles du graphe, leur comptage n'est pas suffisant et un parcours complet de celui-ci est obligatoire.

Exemples typiques d'applications des triangles

Nous retrouvons les triangles un peu partout dans le monde, ils représentent une relation forte entre trois individus, à ce propos, un triangle dans les réseaux sociaux peut signifier que les trois individus représentés par ses sommets appartiennent à la même famille ou sont des amis proches qui se connaissent en réalité. Ce genre de motifs peut être utilisé dans la recommandation, si un individu (sommet) de ce triangle s'intéresse à un article ou une opinion, il y a une forte possibilité que les autres individus (sommets) s'y intéressent également. Nous présentons dans la suite deux cas d'utilisation de triangles dans des applications réelles : (1) la détection des fraudes, et (2) les jointures dans les bases de données.

Détection des fraudes

Comme les réseaux et les graphes représentant le monde deviennent de plus en plus volumineux, ils demeurent plus vulnérables. La fraude peut être facilement émergée provoquant des dégâts. La détection des fraudes est une tâche importante bien qu'elle soit très difficile. Dans la suite, nous présentons deux exemples des fraudes : (1) dans l'e-commerce, et (2) dans les finances. Ces deux exemples ont été présentés par [QCQ+18].

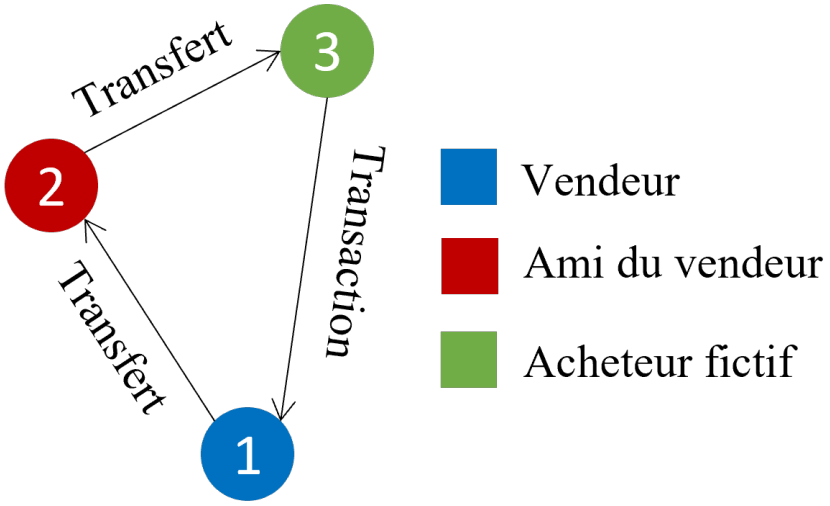


FIGURE 4 – Exemple de fraude dans l'e-commerce.

Dans Figure 4, nous illustrons un exemple de graphe d'échange entre le vendeur et l'acheteur sur une plateforme d'e-commerce. Les comptes des acheteurs et des vendeurs sont représentés

par des sommets. Ainsi, les transactions entre les comptes sont modélisées par des arêtes. Afin d'augmenter la popularité d'une marchandise dans le but d'améliorer les ventes futures, de fausses transactions sont placées pour augmenter artificiellement le nombre de transactions passées. En effet, les applications d'e-commerce se basent sur le principe de l'offre et de la demande. Lorsque la demande augmente et l'offre diminue le prix augmente. Donc pour augmenter ses bénéfices, les vendeurs augmentent le prix avec de fausses demandes. Dans le graphe, le vendeur (sommet 1) envoie de l'argent à son ami (sommet 2), ce dernier demande le produit en envoyant l'argent au vendeur à partir d'un autre compte (sommet 3) en jouant le rôle d'un acheteur. Enfin, le vendeur récupère son argent sans envoyer la marchandise. Dans une telle situation, la détection du triangle ou du cycle (1-2-3-1) permet d'empêcher cette fraude.
Dans Figure 5, nous exposons un deuxième exemple de la fraude dans les finances. Le graphe présenté modélise une opération de transfèrt où un criminel (sommet 2) peut prendre un crédit à court terme auprès d'une banque (sommet 4), en utilisant de fausse pièce d'identité. Il essaie de retirer de l'argent illégalement, en simulant un achat (arête $2 \rightarrow 3$) à l'instant t_1 à l'aide d'un marchand (sommet 3). Une fois que le commerçant reçoit le paiement (arête $4 \rightarrow 3$) de la banque (sommet 4), il essaie de renvoyer l'argent (arêtes $3 \rightarrow 1$ et $1 \rightarrow 2$) au criminel via un intermédiaire (sommet 1) aux temps t_3 et t_4, respectivement. Si le système peut détecter le triangle ($2 \rightarrow 3 \rightarrow 1 \rightarrow 2$) en temps réel, il devient possible d'arrêter une telle fraude à temps.

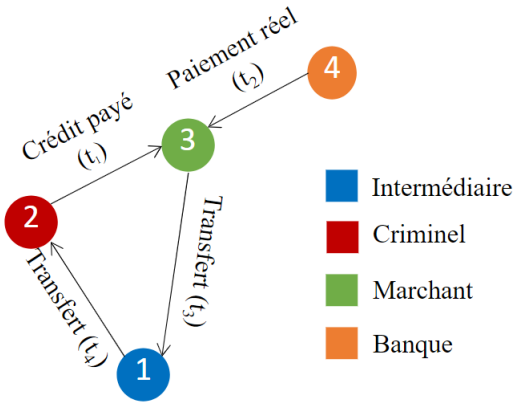


FIGURE 5 – Exemple de fraude dans les finances.

Jointure dans les bases de données

Les jointures sont parmi les opérations les plus coûteuses dans l'interrogation des bases de données relationnelles. Elles sont au centre des sujets d'évaluation algorithmique dans les systèmes de bases de données. Plusieurs algorithmes et variants de jointure existent :

(1) les boucles imbriquées,

(2) les jointures par hachage (hash join),

(3) les jointures par fusion (merge join),

(4) les bloques de boucles imbriquées,

(5) Grace Hash Join, .. etc

Afin d'améliorer les performances des jointures, les triangles peuvent être employés. En effet, [NRR14] a conduit une étude sur l'amélioration de la complexité des jointures en employant une requête dite requête de triangle. Cette dernière se présente sous la forme suivante :

$$Q_\Delta = R(A,B) \bowtie S(B,C) \bowtie T(A,C) \tag{3}$$

La requête 3 est la requête cyclique la plus simple. Elle est suffisamment riche pour illustrer la plupart des idées des nouveaux algorithmes de jointure. L'idée principale derrière le travail de [NRR14] est de faire rapprocher la complexité de deux jointures consécutives dans la requête de triangle à $O(m^{3/2})$, qui s'agit de la complexité d'énumérer tous les triangles dans un graphe avec m arêtes. En fait, la méthode traditionnelle pour évaluer les jointures, y compris la requête 3, est de déterminer le meilleur plan de jointure par paire [RGG03]. Ainsi, pour calculer la requête 3, nous déduisons trois plans d'exécution possibles, démontrés dans Figure 6.

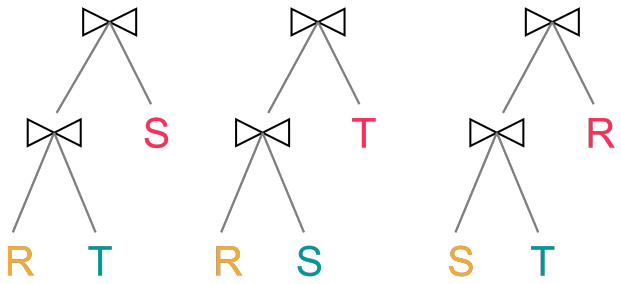


FIGURE 6 – Les trois plans de jointure par paires pour la requête 3.

La complexité temporelle de chaque plan est bornée par $\Omega(N^2)$, en supposant que chaque relation à au plus $N = 2m + 1$ tuples. Pour améliorer l'exécution de cette requête, [NRR14] propose une méthode qui permet de cerner la relation la plus lourde [7], puis effectuer un choix d'exécution permettant de réduire la complexité temporelle à $O(N)$. Soient A, B et C des instances avec les domaines $\{a_i | i \in [0, m]\}$, $\{b_i | i \in [0, m]\}$ et $\{c_i | i \in [0, m]\}$, pour les relations R, S et T respectivement.
On note a_i lourde si et seulement si

$$|\sigma_{A=a_i} R \bowtie T| \geq |Q_\Delta[a_i]| \tag{4}$$

Autrement dit, a_i est lourde lorsque sa contribution au résultat intermédiaire est plus important que sa participation au résultat final. Le calcul de la partie gauche de l'équation 4 est simple. En contrepartie, nous ne pouvons pas savoir la valeur de la partie droite qu'à la fin du calcule de Q_Δ. Néanmoins, il est évident que $Q_\Delta[a_i] \subseteq S$, donc la relation S peut être utilisée comme un proxy pour $|Q_\Delta[a_i]|$. Par conséquent, deux choix s'imposent :

(1) Calculer $\sigma_{A=a_i}(R) \bowtie \sigma_{A=a_i}(T)$ et filtrer les résultats en sondant contre S ou

(2) Considérer chaque tuple dans $(b, c) \in S$ et vérifier si $(a_i, b) \in R$ et $(a_i, c) \in T$ (calculer les triangles (a, b, c)).

L'option (1) est choisi si a_i est léger, en revanche l'option (2) est retenue dans le cas contraire.

Exploration des graphes sur les bases de données relationnelles et Python

Depuis des décennies, les bases de données relationnelles représentent la technologie de stockage de données transactionnelles la plus utilisée, très particulièrement pour les petites et moyennes organisations. En effet, les systèmes de gestion de bases de données relationnelles (SGBDR [8]) sont en constante évolution depuis les années 70s et jusqu'à présent. Les bases de données relationnelles hébergent un grand volume de données qui reflètent des relations entre des entités/objets du monde réel. Ceci peut être modélisé par des graphes, où les entités font l'objet des sommets et les relations entre elles peuvent être représentées par des arêtes. À ce titre, la relation hiérarchie entre les employés d'une entreprise dans une base de données relationnelle peut être modélisée par un graphe, où les sommets sont les employés et les arêtes représentent cette hiérarchie,

7. La relation qui présente une asymétrie et augmente le temps d'exécution de la requête Q_Δ
8. *Système de Gestion de Base de Données Relationnelles*

c-à-d. un employé est dirigé par un superviseur et dirige un ensemble de subordonnés. Nous pouvons ainsi poser des requêtes sur ce graphe telles que : "Qui est le superviseur de l'employé X ?", "Quels est la relation entre l'employé X et l'employé Y ?", "Est ce que l'employé Z est le subordonné de l'employé X ?",etc. Afin de répondre à ce type de questions, plusieurs moteurs d'analyse, librairie et SGBD graphe comme Spark Graphx, neo4j, etc. ont émergé dans le but de proposer des solutions natives dédiées exclusivement pour le traitement des graphes. Cependant, ces solutions doivent importer les données du graphes à partir des bases de données relationnelles pour effectuer des analyses, provoquant ainsi beaucoup d'E/S [9]. Cela conduit probablement à la dégradation des performances. Par conséquent, le traitement des graphes dans les SGBDR [FRP15, OCG17, CO17] présente un moyen différent (mais puissant) pour un traitement efficace de graphes. Ceci est soutenu par les raisons ci-dessous :

(1) Les SGBDR offrent un ensemble de fonctionnalités utiles, notamment poser des questions via des requêtes, ne pas se soucier des limites de la RAM [10], de la récupération après une panne de disque ou de processeur, des E/S optimisées, un contrôle de sécurité, un accès concurrent, une robustesse et une importation rapide des données. Toutes ces fonctionnalités sont disponibles dans l'OS, mais cela nécessite une programmation avancée de C++ ou des commandes compliquées avec moins de cohérence et d'intégrité (propriétés ACID),

(2) SQL est déclaratif, élégant et plus abstrait qu'un code C++ ou Java, qui est utilisé par de nombreux moteurs et bibliothèques pour traiter les graphes. Ces langages sont rapides mais très compliqués à comprendre, à manipuler et à utiliser pour reproduire les fonctionnalités du SGBDR. En effet, SQL est un langage très puissant adopté par la plupart des systèmes de Big Data, y compris les moteurs d'exploration de graphes, tels que TigerGraph et Spark SQL,

(3) De nombreuses données de graphes existent déjà ou peuvent être générées à partir de données existantes sur les bases de données relationnelles [TTP+19]. Les entités (ou une partie d'elles) peuvent être transformées en sommets de graphe et les relations entre ses entités peuvent faire l'objet d'un arc ou une arête (selon les cardinalités entre elle) [XKD15a],

(4) Le coût de la migration d'un système de base de données relationnelle vers un système de graphes est élevé en termes d'installation, de configuration et de maintenance. Il existe de nombreux graphes de *moyenne* et de *petite* taille dans plusieurs entreprises et organisations qui ne peuvent pas se permettre les coûts de migration,

(5) Dans les applications réelles, le traitement des problèmes de graphes n'est qu'une étape dans le processus de résolution d'un problème plus important qui peut inclure SQL, l'apprentissage automatique (ML [11]) et d'autres analyses. Donc, se limiter aux bases de données orientées graphes peut conduire à fragmenter le problème, provoquant ainsi beaucoup d'import/export de données entre les moteurs, car les bases de données orientées graphes sont autonomes et ne peuvent pas effectuer d'autres analyses au-delà des problèmes de graphes [TTP+19]. D'autre part, les SGBDR permettent le traitement de plusieurs problèmes dont SQL, les graphes et le ML [Ord13], ce qui permet d'éviter l'export/import de données entre les systèmes de traitement du problème.

(6) Particulièrement, les bases de données orientées colonnes présentent une meilleure performance avec le traitement des graphes, car elles sont basées sur les projections [12] au lieu des index.

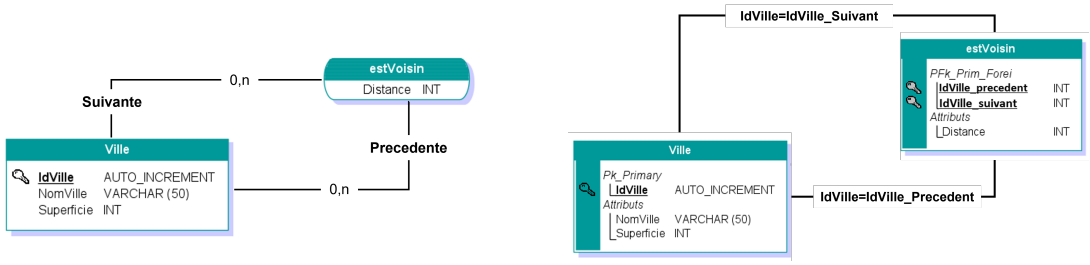
D'autre part, Python est un langage très puissant, aussi utilisé en science de données que SQL. Il offre un ensemble de librairies permettant d'analyser les données de la phase de leur acqui-

9. *Entrée/Sortie*
10. *Random Access Memory (mémoire à accès aléatoire)*
11. Machine Learning
12. Les projections sont des collections optimisées de colonnes de table qui fournissent un stockage physique pour les données.

sition jusqu'à leur visualisation. Parmi ses librairies, nous retrouvons Pandas. C'est une API [13] d'analyse et de manipulation de données open source rapide, puissante, flexible et facile à utiliser. Elle définit le *DataFrame*, qui s'agit d'une structure de données pour la représentation des ensembles de données. Le DataFrame se représente sous forme d'un ensemble de colonnes, similaire à une table dans une base de données relationnelles. Pandas définit également un ensemble d'opérations telles que la jointure, la sélection, la projection et la concaténation des DataFrames. L'utilisation de Pandas peut être aussi bénéfique que proposer une solution avec SQL. En effet, la résolution des problèmes de science de données repose principalement sur Python qui offre un ensemble de librairies, comme Pandas, Numpy et Matplotlib. Ces librairies agissent de l'acquisition des données jusqu'à leur visualisation. Donc, il est facile d'intégrer n'importe quelle solution fondée sur Pandas, dans le pipeline de la résolution d'un problème de science de données de taille importante, qui peut inclure l'exploration des graphes, le ML et bien d'autres.

Représentation du graphe dans la base de données

La génération des graphes à partir des données relationnelles permet d'analyser ces données sous un nouvel angle, tout en mettant l'accent sur les relations et leurs coûts. Par exemple, nous pouvons facilement trouver le chemin le plus court entre deux villes, ou bien les chemins alternatifs les plus optimaux en cas d'embouteillage. Dans Figure 7, nous présentons un modèle conceptuel (MCD) (voire Figure 7a) et un modèle logique (MLD) (voire Figure 7b) de données illustrant la relation de voisinage entre les villes.



(a) MCD (b) MLD

FIGURE 7 – La Relation de voisinage entre les villes.

Á partir de ce modèle, nous pouvons générer des graphes en considérant chaque entité comme sommet et chaque relation entre elles comme arête. Dans la vision des bases de données relationnelles, un graphe peut être représenté de plusieurs manières, entre autres, une liste d'adjacence ou une liste d'arêtes. Dans cette thèse, nous avons opté pour la représentation par une liste d'arêtes grâce à sa simplicité et son efficacité dans la formulation de notre solution. Figure 8 présente un modèle physique de base de données pour la représentation du graphe pour l'exemple de la relation de voisinage entre les villes.

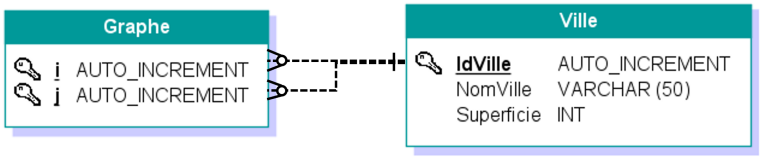


FIGURE 8 – Modèle physique de base de données pour la représentation du graphe.

Nous illustrons dans Figure 9, un extrait de la table des arêtes du graphe représentant les voisinage entre les villes dans une base de données orientée colonnes.

13. *Interface de Programmation d'Application*

i	j
1	2
1	3
2	1
2	3
3	1
3	2

FIGURE 9 – Exemple de la table des arêtes.

Cette table d'arêtes est très grande et étroite. Elle contient deux colonnes qui représentent, respectivement, le sommet de source i et le sommet de destination j et m lignes ; où m est le nombre des arêtes dans le graphe. La clé primaire de cette table est la combinaison (i, j). Dans certain cas, la table des arêtes peut définir trois colonnes, où la troisième colonne représente le poids de l'arête ; une valeur numérique qui représente une métrique entre les sommets d'extrémité comme la distance, le prix et bien d'autres.

Représentation du graphe avec Pandas

La représentation du graphe sur Pandas s'effectue par la définition d'un DataFrame de liste d'arêtes. Ce dernier est constitué de deux colonnes, où la première représente le sommet source et la deuxième le sommet de destination. La création de ce DataFrame est présentée dans la formule ci-après.

```
graph = pandas.DataFrame(index=None, columns=[[i,j]])
```

L'exemple de Figure 9 donne la représentation suivante :

```
{i: [1,1,2,2,3,3], j: [2,3,1,3,1,2]}
```

Traitement des graphes dans une base de données orienté colonnes

Pour manipuler les graphes dans une base de données orienté colonnes, une projection sera créée pour chaque colonne. Ceci permet de sauvegarder ces projections sur le disque et les utiliser directement lors de l'exécution des requêtes au lieu d'extraire les résultats à partir des colonnes de la tables. De plus, pour le traitement parallèle et distribué de données, les systèmes de gestion des bases de données orientées colonnes offrent une technique de partitionnement de données, ainsi que leur segmentation. Le partitionnement des données est une technique qui permet de diviser les données, sur le même nœud de traitement (machine) en petites partitions, et les regrouper par rapport à un critère. Cette technique assure la performance locale des jointures. Figure 10 illustre un exemple de partitionnement. En revanche, la segmentation a pour objectif de partitionner une table sur l'ensemble des machines de traitement dans le cluster, afin de minimiser principalement le transfert et la communication des données, lors de l'exécution des jointures. Figure 11 illustre un cas de segmentation des données sur un cluster de trois machines.

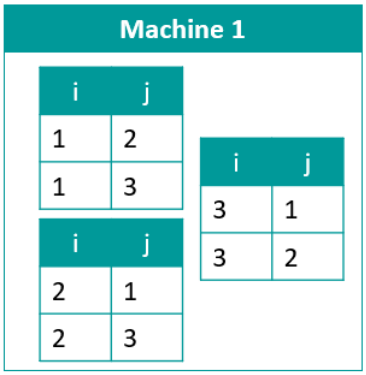


FIGURE 10 – Partitionnement.

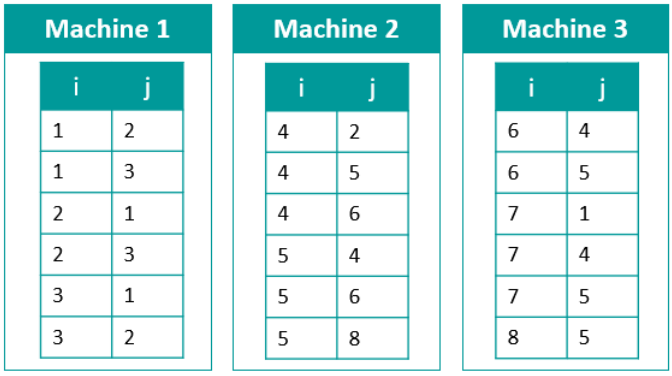


FIGURE 11 – Segmentation.

Vision de la thèse et contributions

Les bases de données relationnelles offrent plusieurs fonctionnalités qui permettent une gestion plus performante et sécurisée des données. Néanmoins, elles sont moins efficaces face au calcul intensif et distribué, comme celui de l'analyse des graphes. En effet, les SGBDR reposent sur le traitement par jointure, ce qui freine leur passage à l'échelle. Les jointures peuvent causer le goulot d'étranglement si elles ne sont pas traitées localement, surtout lorsque la taille des données deviennent très volumineuse.
Dans cette thèse, nous nous sommes focalisés sur l'analyse du problème de la détection des triangles des graphes, à grande échelle dans le perspectif des bases de données relationnelles et avec Pandas de Python. Notre objectif principal est alors d'étudier les différentes solutions existantes sur l'énumération et le comptage des triangles dans les graphes, afin de les exploiter et les comparer. Ensuite de proposer des algorithmes parallèles, qui favorise le traitement local et distribué des jointure. Puis les déployer sur une base de données relationnelle et le programmer avec la librairie Pandas sous Python. Notre algorithme résumé dans Figure 12 doit être scalable et performant, permettant ainsi le traitement efficace sur l'infrastructure de déploiement.

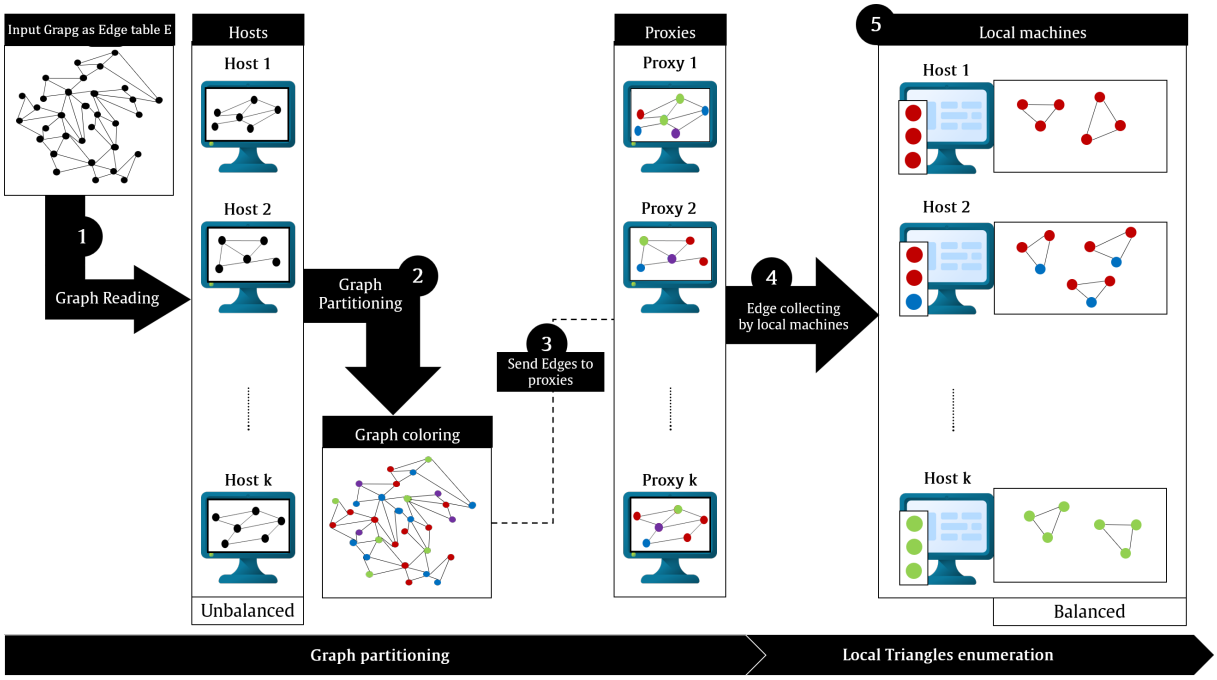


FIGURE 12 – Résumé de l'énumeration des triangles avec l'algorithme proposé.

La mise en œuvre de notre vision implique les tâches suivantes :

(1) Étude comparative des solutions existantes pour l'énumération et le comptage des triangles,
(2) Étude de l'algorithme randomisé pour la détection des triangles,
(3) Compréhension approfondie de la structure des bases de données orientées colonnes,
(4) Étude et analyse de l'expression de l'énumération distribuée des triangles avec des requêtes SQL,
(5) Préservation de l'esprit de l'algorithme distribué dans le sens où il n'y a pas de goulot d'étranglement de machine centrale ou de coordination compliquée,
(6) Adaptation des travaux antérieurs sur la fermeture transitive dans les bases de données relationnelles pour énumérer les triangles,
(7) Minimisation de la redistribution des données lors du calcul de la jointure,
(8) Optimisation du partitionnement des données afin que les jointures dans les requêtes soient calculées localement,
(9) Étude de l'équilibrage de charge et la scalabilité des requêtes SQL,
(10) Assurance de l'exactitude et la justesse des résultats obtenus.

Nos contributions principales à travers cette thèse sont :

(1) En considérant le besoin de faire des analyses approfondies sur les relations existantes dans les données résidents dans les BDR [14], nous avons proposé une solution pour l'énumération et le comptage des triangles dans les graphes en utilisant les requêtes SQL. Cette solution est basée sur l'algorithme illustré dans Figure 12. Notre approche permet un équilibrage de charge parfait entre les machines de calcul, tout en fournissant un résultat correct.
(2) PandaSQL un outil de détection des triangles dans les graphes, basé sur notre première contribution. Il est implémenté en suivant l'architecture logique MVC [15], et l'architecture physique client/serveur. Nous avons choisi Vertica comme SGBD pour exécuter nos requêtes, néanmoins tout autre SGBD peut être utilisé.
(3) Une étude de la complexité de l'énumération et du comptage des triangles sur les BDR, en analysant les plans d'exécution des requêtes. Nous avons ainsi amélioré l'exécution des requêtes en appliquant une suite de techniques d'optimisation.
(4) Une solution développée sous Python avec Pandas et MPI. Elle est fondée sur l'algorithme randomisé, et est plus adéquate aux graphes de taille petite à moyenne. Elle préserve l'équilibrage de charge et élimine la communication lors de l'énumération locale des triangles.

Organisation de la thèse

Notre manuscrit est composé de deux parties principales illustrées dans Figure 13.

Première partie : Concepts fondamentaux et état de l'art

Nous présentons à travers cette première partie, notre état de l'art ainsi que les concepts généraux qui s'articulent autour de deux chapitres. Le chapitre 1 sert comme un référentiel pour les notions de base et les généralités sur les graphes, la définition du problème des triangles ainsi que le modèle de traitement et l'infrastructure employée.
Dans le chapitre 2, nous présentons une étude bibliographique sur le traitement des graphes dans les bases de données relationnelles ainsi que les algorithmes de la détection des triangles. Dans un premier lieu, nous énumérons les travaux, les plus importants, étudiant la définition

14. *Base de Données Relationnelle*
15. *Modèle Vue et Contrôleur*

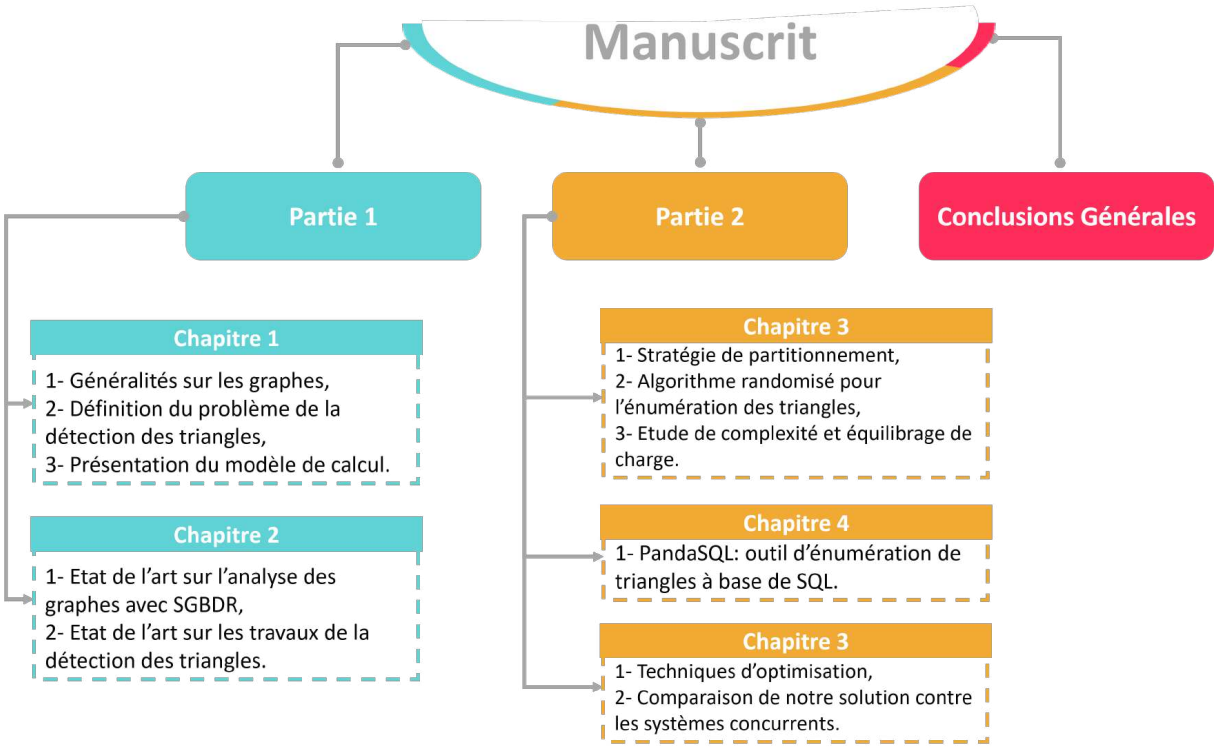


FIGURE 13 – Organisation du manuscrit

des problèmes d'analyse des graphes dans les bases de données relationnelles et offrant des solutions à base de SQL. Dans un deuxième lieu, nous étudions les algorithmes d'énumération et de comptage de triangles, principalement sur les infrastructures parallèles et distribuées. Nous nous focalisons ensuite sur l'étude de l'algorithme randomisé [PRS18b] tout en expliquant ces propriétés.

Deuxième partie : Contributions

Cette partie détaille notre contribution en trois chapitre. Le chapitre 3 présente notre première contribution qui consiste en l'implémentation de l'algorithme adapté, inspiré de l'algorithme randomisé présenté par Pandurangan et al. [PRS18b]. Notre solution permet de résoudre le problème d'énumération de triangles avec les requêtes SQL et avec Python sur un environnement distribué. Notre stratégie de partitionnement des données est détaillée dans ce chapitre, ainsi que son impact sur l'amélioration des performances. En outre, nous justifions la justesse de nos résultats et la scalabilité de notre solution, et nous étudions la complexité de nos requêtes.
Dans le chapitre 4, nous démontrons PandaSQL ; un outil basé sur l'énumération des triangles dans les graphes à grande échelle en utilisant les requêtes SQL. Nous présentons également nos choix techniques et logiques dans l'implémentation de cet outil, et nous clôturons ce chapitre avec une démonstration détaillée destinée aux débutants et aux experts dans les SGBD.
Dans le chapitre 5, nous implémentons notre solution sur le SGBD orienté colonne Vertica et avec Pandas sur Python. Nous rapportons ensuite nos résultats obtenus, tout en les comparant avec des solutions existantes. D'abord, nous présentons l'architecture matérielle et logicielle ainsi que les jeux de données employés pour valider les résultats. Nous expliquons ensuite les techniques d'optimisation utilisées pour améliorer les performances de notre système. Enfin, nous effectuons une étude expérimentale exhaustive sur notre système répartie sur trois phases. La

première phase vise à évaluer les optimisations des requêtes :

(1) avoir une table répliquée tout en effectuant une auto-jointure locale,

(2) l'impact d'éliminer les résultats redondants dans l'énumération locale des triangles.

La deuxième phase concerne une évaluation de l'équilibrage de charge et de l'accélération de l'algorithme adapté dans le contexte de bases de données relationnelles et de Python. La dernière phase compare l'algorithme adapté avec les différents solutions et systèmes existants :

(1) Une comparaison dans l'environnement des bases de données relationnelle entre l'algorithme adapté et l'algorithme classique. Le but de cette partie des expériences est de prouver la justesse et l'efficacité de nos requêtes SQL.

(2) Une comparaison dans l'environnement de Python entre notre algorithme adapté un algorithme développé avec l'API native des graphes qui s'agit de Networkx. Cette comparaison a pour objectif d'étudier l'efficacité de notre solution sur Python.

(3) Une comparaison de l'algorithme adapté contre la fonction de comptage des triangles de Spark Graphx ainsi que le système G-thinker. L'objectif de cette étude est de démonter efficience de notre solution proposée face à d'autres plateformes.

Conclusion générale

Ce chapitre conclut la thèse en fournissant un résumé et une évaluation du présent travail. Ainsi, il évoque plusieurs opportunités de travaux futurs.

Publications

Revues internationales

1. ***Abir Farouzi***, *Xiantian Zhou, Ladjel Bellatreche, Mimoun Malki, Carlos Ordonez. (2023). Balanced parallel triangle enumeration with an adaptive algorithm. Distributed and Parallel Databases. 1-39. 10.1007/s10619-023-07437-x.*

Conférences internationales

1. ***Abir Farouzi***, *Ladjel Bellatreche, Carlos Ordonez, Gopal Pandurangan, and Mimoun Malki. 2020. A Scalable Randomized Algorithm for Triangle Enumeration on Graphs Based on SQL Queries. In Proceedings of the 22nd International Conference on Big Data Analytics and Knowledge Discovery (DaWaK), pages. 141–156, Bratislava, Slovakia, September 14-17, 2020.* https://doi.org/10.1007/978-3-030-59065-9_12

2. ***Abir Farouzi***, *Ladjel Bellatreche, Carlos Ordonez, Gopal Pandurangan, and Mimoun Malki. PandaSQL : Parallel Randomized Triangle Enumeration with SQL queries. In Proceedings of the 29th ACM International Conference on Information and Knowledge Management (ACM CIKM), Galway, Irland, October 19-23,2020.* https://doi.org/10.1145/3340531.3417429

3. ***Abir Farouzi***, *Xiantian Zhou, Ladjel Bellatreche, Mimoun Malki, and Carlos Ordonez (2023). Parallel Pattern Enumeration in Large Graphs. In Proceedings at International Conference on Database and Expert Systems Applications(DEXA), pages. 408-423, Penang, Malaysia, Auguest 28-30, 2023.* https://doi.org/10.1007/978-3-031-39847-6_32

4. *Xiantian Zhou,* ***Abir Farouzi***, *Ladje Bellatreche and Carlos Ordonez (2023). Bitwise Algorithms to Compute the Transitive Closure of Graphs in Python. In Proceedings at International Conference on Database and Expert Systems Applications(DEXA), pages. 345-351, Penang, Malaysia, Auguest 28-30, 2023.* https://doi.org/10.1007/978-3-031-39847-6_27

Conférences nationales

1. ***Abir Farouzi***, *Ladjel Bellatreche, Carlos Ordonez, Gopal Pandurangan, and Mimoun Malki. Enumération Randomisée des Triangles dans des Graphes à Grande Echelle à base de SQL. In proceeding of the* 16^{th} *Edition of Business Intelligence & Big Data Conference (EDA'2020), pages. 33-46, Lyon, France, August 25-27, 2020.*

2. ***Abir Farouzi***, *Xiantian Zhou, Ladjel Bellatreche, Mimoun Malki, and Carlos Ordonez. Parallel Pattern Enumeration in Large Graphs. In Proceedings of the* 39^{th} *Days of the Data Management Conference – Principles, Technologies and Applications (BDA 2023), Montperllier, France, October 23-26, 2023.*

Première partie

Concepts fondamentaux et état de l'art

Chapitre 1

Notions de base

I've decided to get into shape, and the shape I've selected is a triangle.

—Howie Mandel

Sommaire

1.1 Introduction

Ce chapitre vise à rappeler les notions fondamentales de la théorie des graphes, en mettant particulièrement l'accent sur les motifs intégrés, et plus spécifiquement sur les triangles ainsi que leurs propriétés. Ces notions sont essentielles pour le développement des points présentés dans les chapitres ultérieurs. Il présentera également des notions relatives au partitionnement et d'organisation des données dans les différents environnements que nous allons exploiter pour développer notre solution.
Ce chapitre est principalement organisé en trois sections. Dans la première section, nous définissons les notations et nous rappelons les fondements mathématiques utilisés dans cette thèse. Ensuite, nous présentons, dans la deuxième section, les notions théoriques des graphes, ses propriétés et ses différents motifs. Nous nous concentrons particulièrement sur les triangles, qui constituent un composant fondamental dans la théorie du graphe. La troisième section portera sur l'organisation et le partitionnement des données. Nous nous focalisons sur les données relationnelles, ainsi que les données du graphe. Cette section évoquera également les différentes architectures physiques et logiques des systèmes distribués, tout en discutant leurs avantages et inconvénients. Enfin, nous présentons dans la dernière section, la représentation et la manipulation des graphes dans les bases de données relationnelles, tout en discutant la normalisation de cette représentation.

1.2 Notations et fondements mathématiques

1.2.1 Symbole

Nous avons utilisé les caractères minuscules pour désigner des nombres et des fonctions mathématiques. Ces caractères peuvent être latins ou grecs, par exemple, δ, γ, i, j, h, etc. En revanche, les caractères majuscules sont employés pour définir les ensembles et les objets qui sont différents des simples nombres et des fonctions.
Nous essayerons d'adopter cette notation tout au long de cette thèse, néanmoins, si nous constatons qu'une notation qui ne suit pas notre définition est largement utilisée dans la littérature, nous l'adoptons.

1.2.2 Nombre et complexité

Les entiers positifs y compris le zéro sont désignés par $\mathbb{N}$ et sans le zéro par $\mathbb{N}^*$. Sur le même principe, nous désignons les nombres réels comprenant le zéro par $\mathbb{R}$ et sans le zéro par $\mathbb{R}^*$. Également, pour représenter l'intervalle positif dans un ensemble mathématique, nous rajoutons un $+$ en dessous de sa notation, à ce propos, l'ensemble des réels positifs avec le zéro est désigné par $\mathbb{R}_+$ et sans le zéro par $\mathbb{R}_+^*$.
Pour étudier la complexité dans cette thèse, nous adoptons la notation de landau appelé $\mathcal{O}$, qui représente une notation commune pour étudier le comportement asymptotique des polynômes. Pour expliquer la logique derrière cette notation, soit f et g deux fonctions telles que $f, g : \mathbb{N} \to \mathbb{N}$, ceci implique f est dans $\mathcal{O}(g)$ ou de manière équivalente g est dans $\Omega(f)$, s'il existe un entier $x_0 \in \mathbb{N}$ et un nombre réel $c \in \mathbb{R}$ tels que $f(x) \leq c \cdot g(x)$ pour tout $x \geq x_0$. Autrement dit, f évolue aussi rapidement que g. Pour étudier l'évolution asymptotique équivalente, nous définissons $\Theta(g)$ tel que

$$\Theta(g) = \mathcal{O}(g) \cap \Omega(g)$$

Enfin, s'il existe un entier $x_0 \in \mathbb{N}$ pour tout $c \in \mathbb{R}_+^*$ tels que $f(x) \leq c \cdot g(x)$, f est dans $o(g)$ pour tout $x > x_0$ ou d'une façon équivalente g est dans $\omega(f)$.

1.2.3 Classes de complexité de problèmes

Nous utilisons dans cette thèse des notations de type $NP-difficile$ et $NP-complet$ pour définir la classe de complexité des problèmes étudiés. Nous expliquons brièvement dans cette section la signification de ses notations. D'abord, nous rappelons les classes de problèmes préliminaires P et NP puis, nous définissons ces classes de problèmes [AB09].

(1) La classe P : elle représente l'ensemble des problèmes de décision qui peuvent être résolus en temps polynomial.

(2) La classe NP : elle regroupe les problèmes de décision résolus en temps polynomial par une machine de Turing non déterministe. À noter qu'un problème P est toujours aussi NP. De plus, si le problème de la classes NP a une solution connue, la démonstration de l'exactitude de la solution peut toujours être réduite à une vérification P (temps polynomial). En revanche, si P et NP ne sont pas équivalents, la solution des problèmes de NP nécessite une recherche exhaustive dans le pire des cas.

(3) La classe $NP - difficile$: c'est l'ensemble des problèmes dont l'algorithme de résolution peut être réduit à un algorithme qui peut résoudre tous les autres problème NP. Notamment, il est beaucoup plus facile de montrer qu'un problème est NP que de montrer que c'est $NP - difficile$.

(4) La classe $NP - Complet$: si un problème est à la fois NP et $NP - difficile$, il est $NP - complet$.

1.3 Théorie de graphe

1.3.1 Généralités sur les graphes

Le paradigme traditionnel de l'analyse de données assume que les données sont indépendantes l'une des autres. Néanmoins, les instances de données peuvent être connectées entre elles générant ainsi des graphes. Les graphes sont un ensemble de sommets (instances) et un ensemble d'arêtes (liens) définissant les relations entre les instances. Ces sommets et ces arêtes peuvent avoir plusieurs types d'attributs : numériques, catégoriques ou même complexes comme les données de séries temporelles [ZJ14].
Formellement, un graphe $G = (V, E)$ est une structure mathématique composée d'un ensemble fini non vide de sommets (nœuds) V et un ensemble peut être vide d'arêtes $E \subseteq V \times V$ regroupant tous les liens entre les paires de sommets. Nous notons n le nombre de sommets ($n = |V|$) et m le nombre d'arêtes ($m = [E|$). Le nombre de sommets n dans le graphe définit son ordre et le nombre de ses arêtes m définit sa taille. Chaque arête $e \in E$ connecte deux sommets $\{i, j\}$ et définit une direction (de i vers j ou/et inversement). Si les paires de sommets du graphe définissent une bi-direction entre elles, le graphe est dit non-orienté autrement il est orienté. Dans un graphe orienté, nous utilisons préférablement la nomination d'arc pour tout élément reliant un couple de sommets $\{i, j\}$. Conventionnellement, l'ensemble des arcs est noté A, donc $G = (V, A)$ est un graphe orienté.
Pour tout graphe non-orienté $G = (V, E)$, nous notons :

$$m \leq \binom{n}{2} = \frac{n(n-1)}{2} \tag{1.1}$$

qui peut être écris en notation asymptotique par $m \in \mathcal{O}(n^2)$. Si l'équation 1.1 définit une égalité de $m = \binom{n}{2}$, le graphe est dit complet. Autrement dit, chaque couple de sommets dans le graphe est relié par une arête, formellement décrit par $\forall i, j \in V, (i, j) \in E$.
Les arêtes du graphe peuvent être pondérées par une fonction de poids définie par $w : E \to \mathbb{R}^+$ permettant de modéliser les interactions entre les sommets et générant ainsi un graphe pondéré

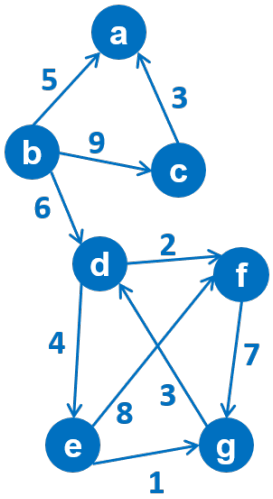
$G = (V, E, w)$. Nous notons le poids d'une arête $e = (i, j)$ par $w_{i,j} > 0$. Par convention, le poids nul représente l'absence de l'arête, et nous notons

$$w_{i,j} = 0 \implies \{i, j\} \notin E$$

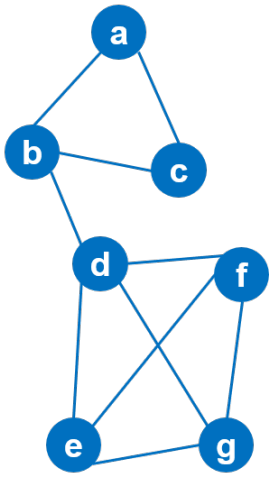
Si le poids de toutes les arêtes égal à 1, le graphe est dit non pondéré et nous notons pour ce cas particulier

$$\forall i, j \in V, w_{i,j} \in \{0, 1\}$$

Figure 1.1 présente deux types de graphe : (i) un graphe orienté pondéré $G = (V, A, w)$ dans Figure 1.1a, et (ii) un graphe non-orienté non-pondéré $G = (V, E)$ dans Figure 1.1b.



(a) Graphe Orienté Pondéré.



(b) Graphe non-Orienté non-Pondéré.

FIGURE 1.1 – Graphe

1.3.2 Représentation des graphes

1.3.2.1 Matrice d'adjacence

Un graphe $G = (V, E)$ peut être représenté par une matrice d'adjacence $\mathcal{A}$ de dimension $n \times n$. Cette matrice $\mathcal{A}$ est symétrique ($\mathcal{A} = \mathcal{A}_T$) si G est non-orienté. Chaque élément $\mathcal{A}_{i,j}$ avec $i \neq j$ représente le poids du liens entre le sommet i et le sommet j, tandis que les éléments de la diagonale $\mathcal{A}_{i,i}$ représente le poids des boucles reliant chaque sommet à lui-même. Dans un graphe simple (non-pondéré, sans boucle), la matrice d'adjacence $\mathcal{A}$ est une matrice binaire avec des zéros sur la diagonale .i.e. $\forall \mathcal{A}_{i,j} \in \{0, 1\}$. Formellement, la matrice d'adjacence $\mathcal{A}$ est définit par :

$$\mathcal{A}_{ij} = \mathcal{A}_{ji} = \begin{cases} 0 & \text{si } \{i, j\} \notin E \\ 1 & \text{sinon.} \end{cases} \tag{1.2}$$

Pour chaque graphe, il existe une et une seule matrice d'adjacence et inversement, cette dernière ne peut appartenir à aucun d'autre graphe que la sienne. La représentation par la matrice d'adjacence convient aux graphes denses, dont le nombre de 0 dans la matrice est très faible par rapport au nombre de 1. Dans Figure 1.2, nous représentons la matrice d'adjacence du graphe de Figure 1.1b.

i \ j	a	b	c	d	e	f	g
a	0	1	1	0	0	0	0
b	1	0	1	1	0	0	0
c	1	1	0	0	0	0	0
d	0	1	0	0	1	1	1
e	0	0	0	1	0	1	1
f	0	0	0	1	1	0	1
g	0	0	0	1	1	1	0

FIGURE 1.2 – Matrice d'adjacence.

1.3.3 Matrice d'incidence

La matrice d'incidence $\mathcal{B}$ représente une manière alternative pour modéliser un graphe $G = (V, E)$, en spécifiant la relation entre les arêtes et les sommets. La matrice d'incidence $\mathcal{B}$ est une matrice binaire de dimension $n \times m$, où les lignes correspondent aux sommets et les colonnes représentent les arêtes. Elle ne permet de représenter que les graphes non-pondérés. La cellule e_1, v_1 est non-nulle si l'arête e_1 est incidente au sommet v_1.
Figure 1.3 donne la matrice d'incidence du graphe décrit dans la Figure 1.1b.

	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}
a	1	0	1	0	0	0	0	0	0	0
b	1	1	0	1	0	0	0	0	0	0
c	0	1	1	0	0	0	0	0	0	0
d	0	0	0	1	1	1	0	0	0	1
e	0	0	0	0	0	1	1	0	1	0
f	0	0	0	0	1	0	0	1	1	0
g	0	0	0	0	0	0	1	1	0	1

e_1 = {a,b} e_6 = {d,e}
e_2 = {b,c} e_7 = {e,g}
e_3 = {a,c} e_8 = {f,g}
e_4 = {b,d} e_9 = {e,f}
e_5 = {d,f} e_{10} = {d,g}

FIGURE 1.3 – Matrice d'incidence.

1.3.3.1 Liste d'adjacence

La représentation du graphe $G = (V, E)$ par une liste d'adjacence consiste à créer un vecteur pour tous les sommets $i \in V$. Chaque sommet i ensuite pointera sur la liste de tous ses voisins. Cette représentation est particulièrement privilégiée avec les graphes *creux*[1]. Elle permet de les représenter d'une manière plus compacte, diminuant ainsi le stockage nécessaire pour manipuler ces graphes.
Figure 1.4 présente la liste d'adjacence du graphe de Figure 1.1b.

1. Un graphe creux est un graphe dans lequel $|E| \ll |V|^2$

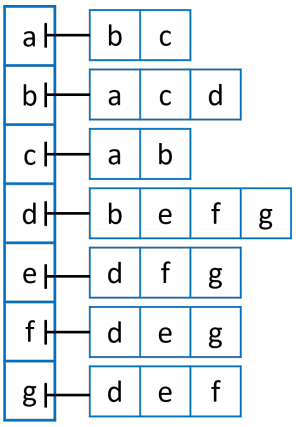


FIGURE 1.4 – Liste d'adjacence.

1.3.4 Propriétés du graphe

1.3.4.1 Voisinage et degré

Dans un graphe non orienté, une arête $e = (i, j)$ est incidente aux deux sommets i et j et les sommets i et j sont adjacents et voisins. Ainsi, nous posons

$$\forall i \in V,\ N(i) = \{j \in V : (i, j) \in E\}$$

l'ensemble des voisins du sommet i. En revanche, un arc $a = (i, j)$ est incident au sommet j, où j est le sommet de destination.
Dans un graphe non-orienté, le degré d'un sommet $i \in V$ est le nombre d'arêtes incidentes à ce sommet et est représenté par $d(i)$ ou d_i. Il s'agit du nombre des voisins du sommet i et nous écrivons :

$$d_i = |N(i)|$$

À ce propos, dans le graphe de Figure 1.1b, $d_f = 3$.
Dans un graphe orienté,le degré d'un sommet $i \in V$ est la somme de son degré intérieur et son degré extérieur. Nous notons

$$d_i = d_i^{in} + d_i^{out}$$

Le degré intérieur du sommet $i \in V$ représente le nombre d'arcs entrants au sommet, définit par :

$$d_i^{in} = |\{j \in V : \exists (j, i) \in A\}|$$

Ainsi le degré extérieur est le nombre d'arcs sortants du sommet, définit par :

$$d_i^{out} = |\{j \in V : \exists (i, j) \in A\}|$$

Dans Figure 1.1a, $d_f^{in} = 2$ et $d_f^{out} = 1$ alors $d_f = 2 + 1 = 3$.

1.3.4.2 Poids du sommet

Nous définissons également le poids w_i d'un sommet i ; la somme des poids des arêtes incidentes au sommet i donné par :

$$w_i = \sum_{j \in V} w_{i,j}$$

Notons que dans le cas d'un graphe non-pondéré, le poids de chaque sommet est le même que son degré vu que le poids de chaque arête incidente à chaque sommet égale à 1.

1.3.4.3 Chemin, cycle et connectivité

Un chemin entre le sommet i et le sommet j est une séquence d'arêtes reliant i à j. Ainsi nous définissons la notion du cycle, un chemin reliant chaque sommet à lui-même, autrement dit, c'est un chemin qui commence au sommet i et se termine au même sommet i. Dans le graphe de Figure 1.1b, le triplet $\{d, e, g\}$ représente un cycle. Un cas particulier se distingue lorsque ce chemin est constitué d'une seule arête, dans ce cas le chemin est appelé une boucle. Un graphe est dit simple si et seulement s'il ne définit aucune boucle (voir les graphes de Figure 1.1).
Un graphe $G = (V, E)$ est connecté s'il existe un chemin entre n'importe quelle paire de nœuds. Tous les graphes mentionnés dans cette thèse sont des graphes connectés. Ceci implique $m \in \Omega(n)$ ce qui nous permet d'écrire $\mathcal{O}(m)$ au lieu de $\mathcal{O}(n + m)$.

1.3.4.4 Fermeture transitive

Dans un graphe orienté non-pondéré $G = (V, A)$, la fermeture transitive représente un graphe $G^* = (V, A^*)$ où A^* contient toutes les paires de sommets (i, j), si et seulement si j est accessible à partir de i. Autrement dit, pour chaque couple (i, j) tels que $\{i, j\} \in V$, la matrice de la fermeture transitive est formée par le facteur d'accessibilité. La matrice de la fermeture transitive du graphe de Figure 1.1a est donnée dans Figure 1.5.

i \ j	a	b	c	d	e	f	g
a	1	0	0	0	0	0	0
b	1	1	1	1	0	0	0
c	1	0	1	0	0	0	0
d	0	0	0	1	1	1	1
e	0	0	0	1	1	1	1
f	0	0	0	1	1	1	1
g	0	0	0	1	1	1	1

FIGURE 1.5 – Matrice de la fermeture transitive.

La fermeture transitive permet de répondre aux questions relatives à l'existence de chemins entre chaque couple de sommets (i, j) dans le graphe G, où le graphe $G^* = (V, A^*)$ est souvent appelé le graphe de dépendance.
Le calcul de la fermeture transitive se révèle très utile dans plusieurs applications. Par exemple, dans un système de livraison, où un graphe est associé à chaque opération de livraison, les sommets de ce graphe représentent les points de livraison et les arcs, les chemins entre ces points. La fermeture transitive de ce graphe donne ainsi tous les chemins nécessaires qui mènent directement ou indirectement d'un point i à un point j. Cette information est utile pour minimiser le coût de déplacement entre les différents points de livraison.

1.3.4.5 Sous-graphe et clique

Un graphe $H = (V', E')$ est un sous-graphe de $G = (V, E)$ si $V' \subseteq V$ et $E' \subseteq E$. Inversement, le graphe G est dit un super-graphe de H. Plus précisément,$\forall i, j \in V', (i, j) \in E' \iff (i, j) \in E$ .i.e. deux sommets i et j sont adjacents dans H si et seulement s'ils sont adjacents dans G. Pour

le graphe $G = (V, E)$ et l'ensemble $\mathcal{P}(G)$ de ses sous-graphes, nous définissons $\mathcal{V} : \mathcal{P}(G) \to V$ et $\mathcal{E} : \mathcal{P}(G) \to E$ tels que $\mathcal{V}(G') = V'$ et $\mathcal{E}(G') = E'$ pour tout $G' = (V', E')$ dans $\mathcal{P}(G)$.
Un sous-graphe est nommé clique ou complet s'il existe une arête entre tous les paires de ses sommets. Un exemple peut être extrait de Figure 1.1b, où le sous-graphe constitué des sommets $\{d, e, f, g\}$ représente une clique (un sous-graphe complet).

1.3.5 Triangles

Dans cette section, nous définissons brièvement les triangles et leur utilisation dans le calcul des autres propriétés du graphe. Le problème du calcul des triangles est discuté et détaillé dans le troisième chapitre.

Definition 1.1 *Un triplet connecté (i, j, h) au sommet j dans un graphe $G = (V, E)$ est un chemin de longueur 2 pour lequel j est au centre. Si $(h, i) \in E$: (i, j, h) est un triplet fermé appelé triangle, sinon c'est un triplet ouvert nommé coin ou triade ouvert.*

Definition 1.2 *Un triangle Δ_{ijh} est un sous-graphe complet dans G formé par les trois sommets i, j et h tels que : $\mathcal{V}(\Delta_{ijh}) = \{i, j, h\}$ et $\mathcal{E}(\Delta_{ijh}) = \{\{i, j\}, \{j, h\}, \{h, i\}\}$. L'ensemble $\Delta(G)$ comprend tous les triangles $\Delta_{(ijh)}$ du graphe G ainsi Δ_j représente tous les triangles dont j est un sommet.*

Definition 1.3 *Le nombre des triangles du sommet j est le nombre des arêtes entre ses voisins. Il est défini par :*

$$\delta(j) = |\mathcal{E}(G[N(j)])|$$

Definition 1.4 *Comme chaque triangle est défini par trois sommets, il est compté pour chacun de ses sommets dans le compte total des triangles, c-à-d. il est compté trois fois. Donc par intuition, nous définissons le compte des triangles par la formule suivante :*

$$\delta(G) = \frac{1}{3} \sum_{j \in V} \delta(j)$$

Propriété 1.1 : Deux triangles Δ_{ijh} et Δ_{xyz} peuvent appartenir à la même clique (voir exemple de Figure 1.6, les deux triangles (d, e, f) et (e, f, g) appartiennent à la même clique formée de sommets $\{d, e, f, g\}$ de Figure 1.1b).

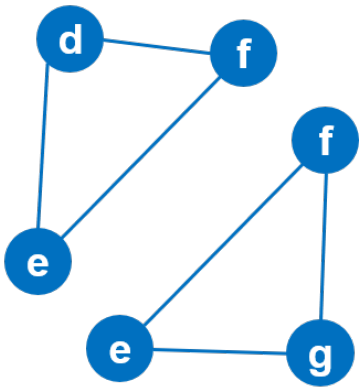


FIGURE 1.6 – Exemple de triangles dans le graphe de Figure 1.1b.

Comme les triangles apparaissent en abondance dans les graphes et les réseaux, de nouvelles métriques ont été émergées pour caractériser et analyser ces graphes. Entre autres, le ratio de transitivité (ou simplement la transitivité) d'un graphe et le coefficient de clustering.

Definition 1.5 *La transitivité d'un graphe est définie par le rapport entre le nombre des triangles et le nombres de tous les triplets (fermés et ouverts i.e. un chemin dont la longueur est égale à deux). Nous notons les triades ouverts par $\Lambda(G)$ (équation 1.3), les triangles par $\Delta(G)$ et la transitivité par $\gamma(G)$ (équation 1.4).*

$$\Lambda(G) = \sum_{j \in V} |\Lambda_j| = \sum_{j \in V} \binom{d(j)}{2} \tag{1.3}$$

$$\gamma(G) = \frac{|\Delta(G)|}{|\Lambda(G)| + |\Delta(G)|} \tag{1.4}$$

Definition 1.6 *Le coefficient de clustering représente la tendance au regroupement des sommets dans le graphe. Lorsqu'il est appliqué sur un sommet, il est appelé un coefficient de clustering local. Il est défini par le rapport entre les voisins du sommets et ses voisins qui sont voisins entre eux. Nous notons $C(j)$ (équation 1.5) le coefficient de clustering local pour le sommet j et $C(G)$ (équation 1.6) le coefficient de clustering global du graphe.*

$$C(j) = \frac{|(i,h) : (i,h) \in E \wedge i,h \in N(j)|}{|N(j)|(|N(j)|-1)/2} = \frac{\Delta_j}{\binom{d(j)}{2}} \tag{1.5}$$

Propriété 1.2 : La moyenne de tous les coefficients de clustering locaux représente le coefficient de clustering du graphe. Elle représente le rapport entre le nombre de tous les triangles avec répétitions (les trois instances) et le nombre de tous les triades ouverts.

$$C(G) = \frac{1}{n} \sum_{v \in V} C(v) = \frac{3 \times |\Delta(G)|}{|\Lambda(G)|} \tag{1.6}$$

1.4 Modèle distribué de stockage de données et partitionnement

Le graphe est une structure de données qui évolue rapidement dans le temps, c'est pourquoi son stockage dans une seule machine devient de plus en plus compliqué lorsque sa taille évolue. En même temps, le traitement sur une infrastructure distribuée favorise le stockage mais complique le traitement de ces structures de données. Dans cette section, nous nous intéressons aux différents modèles de stockages de données et aux architectures permettant le traitement sur ses modèles. Nous traitons particulièrement les BDD [2], tout en détaillant ses différentes architectures et types. Nous clôturons cette section par une comparaison entre les BDD et quelques remarques générales.

1.4.1 Systèmes distribués et bases de données distribuées

Un système distribué consiste d'un certain nombre d'unités de traitement physiquement distribuées et interconnectées où chaque unité fonctionne indépendamment des autres en utilisant son stockage privé. Ces unités peuvent être hétérogène et les interconnexions peut être différentes. Ces unités n'ont pas accès à l'état de l'autre, qu'elles ne peuvent apprendre qu'en échangeant des messages explicites entre elles, ce qui entraînent un coût de communication. Nous définissons alors un programme distribué comme un ensemble de processus qui s'exécute simultanément sur un système distribué et communiquent entre eux via la transmission des messages explicites. Chaque processus peut accéder à un ensemble de variables qui sont disjointes des variables qui peuvent être modifiées par tout autre processus [AO09].

2. *Base de Données Distribuée*

Nous définissons une BDD comme une collection de nombreuses bases de données logiquement interdépendantes situées aux nœuds d'un système distribué. Ainsi, un SGBDD [3] est un système logiciel qui permet de gérer la base de données distribuée, tout en rendant sa distribution transparente pour l'utilisateur. L'existence d'un système distribué est le principal facteur pour la construction d'une base de données distribuée. Autrement dit, un système distribué est la plateforme pour construire une base de données distribuée. De ce fait, lorsque les données sont distribuées, leur gestion et leur accès de manière logiquement intégrée nécessitent une analyse particulière de la part du SGBDD, afin d'appliquer les meilleures stratégies d'optimisation pour réduire les coûts de communication et de traitement distribué [RH10, ÖV20]. Il existe une relation d'interdépendance entre les données du SGBDD, où chaque partition de la base de données peut résider sur un site [4]. Cette relation peut prendre différentes formes selon le type de la base de données (relationnelle ou NoSQL). Par exemple, pour une base de données relationnelle, les relations ou les partitions de données peuvent être stockées sur différents sites. Par conséquent, les jointures et les unions sont utilisées pour recueillir les données nécessaires sur les différents sites pour répondre aux requêtes exprimées en SQL. Un autre exemple intéressant pourra être une base de données NoSQL orientée graphe, où les données sont les sommets du graphe et les arêtes entre eux. Dans ce cas, chaque partition de sommets avec ses arêtes incidentes est stockée sur un site.
[ÖV20] ont distingué deux types de distribution : (1) la distribution géographique sur plusieurs sites, et (2) l'emplacement unique sur un seul site sur plusieurs nœuds de traitement. Le premier type de distribution consiste en une collection de sites éloignés interconnctés par un réseau étendu, qui se caractérise par de longues latences de messages et des taux d'erreur très élevés. En revanche, dans le deuxième type de distribution, l'ensemble des unités de traitement sont situées à proximité immédiate permettant une rapidité d'échange avec une latence presque négligeable et un taux d'erreur très faible. Généralement, le deuxième type de distribution est caractérisé par un cluster de machines dans un centre de données, il est communément connu sous le nom de SGBDP [5]. Dans cette thèse, nous utilisons indifféremment SGBDD et SGBDP pour désigner une base de données parallèle (une distribution sur un emplacement unique), ainsi la BDD représente une base de données parallèle distribuée sur un emplacement unique également.
Pour résumer, le SGBDD est *logiquement intégré* où l'utilisateur a une vision unifiée de la base de données, quoiqu'en réalité sa structure est *physiquement distribuée* sur plusieurs sites/nœuds (machines).

1.4.2 Traitement distribué : avantages et défis

Plusieurs travaux ont discuté les avantages et les défis des SGBDD depuis leur apparition. T. Özsu et P. Valduriez [ÖV91, ÖV20] ont mentionné quatre avantages des SGBDD :

— **Gestion transparente des données :** un système transparent permet de séparer la couche sémantique et la couche physique, où les détails de l'implémentation ne sont pas accessibles par l'utilisateur. Un SGBDD transparent offre un haut niveau de support pour le développement des systèmes complexes. Il permet d'optimiser l'exécution des requêtes sans l'intervention de l'utilisateur, où il choisit le meilleur plan d'exécution en prenant en considération la fragmentation des données, leur réplication et leur emplacement. Afin d'atteindre cet objectif, ce système doit être capable de traiter ces indépendances : (1) l'indépendance des données qui fait référence à l'immunité des applications utilisateurs aux changements de définition et d'organisation des données, y compris les changements logiques comme le schéma de données ou les changements physiques comme les structures de stockage, (2) la transparence des réseaux qui signifie la protection contre les détails

3. *Système de Gestion de Base de Données Distribuée*
4. Un site représente une unité de traitement dans un système distribué.
5. *Système de Gestion de Base de Données Parallèle*

opérationnels du réseau de communication, où l'exécution d'une commande est indépendante de l'emplacement des données et du système sur lequel elle est exécutée, de telle sorte qu'une identification unique de chaque objet dans la BDD induit à une transparence renforcée du réseau, (3) la transparence de la fragmentation qui s'agit de traiter chaque fragment de la BDD comme un objet afin d'améliorer la performance, la disponibilité et la fiabilité, et ne pas rendre l'utilisateur conscient de cette fragmentation en laissant le système gérer l'exécution de la requête, et (4) la transparence de la réplication qui se traduit par le fait de distribuer les données de manière répliquée sur les machines d'un réseau, où le système prend la responsabilité de gérer les réplicas lors de l'exécution d'une requête, pour des raisons de disponibilité, de performance et de fiabilité.

— **Fiabilité via les transactions distribuées :** les SGBDD ont pour avantage d'éliminer les points de défaillance uniques, grâce à leur stratégie de réplication des données. Si un nœud de la BDD ou une liaison de réseau tombe en panne, les données seront toujours accessibles sur un autre nœud. Grâce au support pour les transactions distribuées des SGBDD, l'exécution simultanée des transactions utilisateur ne violera pas la cohérence de la base de données. Chaque utilisateur pense que sa requête est la seule à s'exécuter sur la base de données (une transparence de la concurrence), même en cas de défaillance du système (une transparence de défaillance), chaque transaction reste correcte, c'est-à-dire elle obéit aux règles d'intégrité spécifiées sur la BDD.

— **Performance améliorée :** les SGBDD fragmentent les données en les permettant d'être placées à proximité de leur lieu d'utilisation. Ceci permet de réduire les conflits des services de l'UC [6] et des E/S. Ainsi, il minimise les délais d'accès à distance qui sont généralement impliqués dans les réseaux étendus. De plus, le parallélisme inhérent des systèmes distribués est exploité pour le parallélisme inter-requête et intra-requête, qui peut être traduit par le parallélisme dans l'exécution de plusieurs requêtes en même temps et la décortication de la requête en sous-requêtes exécutées simultanément.

— **Scalabilité :** l'expansion de l'environnement distribué se fait généralement en ajoutant la puissance de traitement et de stockage à la configuration distribuée. Ceci permet de s'adapter à la taille croissante et la charge de travail évolutive de la BDD. Cette expansion ne permet pas d'atteindre une croissance linéaire de la puissance, à cause du surcoût de la distribution (coût de la communication et des E/S). Cependant, des améliorations significatives peuvent s'appliquer, entre autres, la mise à l'échelle (également appelée mise à l'échelle horizontale) qui fait référence à l'ajout de serveurs supplémentaires, appelés serveurs à mise à l'échelle de manière lâche, pour évoluer presque à l'infini. En facilitant l'ajout de nouveaux composants de serveurs de base de données, un SGBDD peut fournir une évolutivité horizontale.

[Sey98, Ray09] ont discuté d'autres avantages, qui s'agit du partage de données et de la réduction des coûts d'exploitation. Les auteurs ont expliqué que grâce aux SGBDD, le partage des données est devenu plus facile en utilisant les techniques de réplication et de fragmentation. Ainsi, les utilisateurs peuvent accéder aux données rapidement sur les sites/nœuds à proximité, économisant ainsi le temps et l'argent.
En revanche, les auteurs [ÖV91, Sey98, Ray09, ÖV20] ont discuté les défis du SGBDD, qui se résument comme suivant :

— **Complexité de gestion :** elle s'agit de la difficulté de la gestion des partitions et des réplicas en optimisant le temps d'exécution des requêtes. Autrement dit, c'est la complexité de trouver la bonne localisation pour les données à travers les sites/nœuds, ainsi que trouver le bon plan d'exécution de la requête en considérant la localité des données et les coûts d'E/S, et de traitement. De plus, elle inclue le contrôle de concurrence dans la BDD afin d'assurer l'intégrité de la base de données.

6. *Unité Centrale*

— **Maintenance :** la distribution des données augmente les coûts de la maintenance, vue que les données sont partitionnées sur plusieurs sites/nœuds.

— **Communication :** les SGBDD utilisent les réseaux de communication pour échanger les données entre les sites/nœuds. Ceci influence le temps d'exécution des requêtes exécutées sur des sites distants en augmentant le coût de communication.

— **Sécurité :** quelques problèmes de sécurité peuvent émerger à cause du réseau de communication utilisé par les SGBDD pour communiquer entre les sites/nœuds. Par conséquent, des mesures de sécurité doivent être mis en place pour assurer la fiabilité et l'exactitude des données échangées.

— **Stockage :** la réplication des données dans les SGBDD nécessite une augmentation proportionnelle entre volume de stockage et la taille des données. Autrement dit, la génération des reliquats implique l'évolution de la taille des données, ce qui nécessite l'augmentation du volume de stockage.

1.4.3 Architectures physiques des bases de données distribuées

L'architecture de la BDD désigne la définition de ses composants et l'interaction entre ces derniers. [ÖV20] a proposé trois dimensions pour définir l'architecture de la BDD :

— **Autonomie :** il s'agit de la distribution du contrôle. Elle indique dans quelle mesure les SGBD individuels peuvent fonctionner de manière indépendante. Autrement dit, elle permet aux composants du système d'échanger les informations, d'exécuter les transactions et de les modifier.

— **Distribution :** c'est la manière dont les données sont distribuées physiquement sur l'ensemble des sites/nœuds.

— **Hétérogénéité :** elle regroupe la diversité des configurations matérielles, les différences dans les protocoles réseaux et la gestion des données. Cette dimension n'est pas considérée dans notre thèse, car nous utilisons une configuration matérielle homogène. De même, la gestion des données et les protocoles de réseau sont identiques sur l'ensemble des nœuds du cluster.

En se basant sur ces dimensions, nous pouvons définir jusqu'à 18 architectures physiques. Néanmoins, la majorité de ces architectures sont insignifiantes. Nous nous intéressons particulièrement à deux architectures qui sont utilisées dans le cadre de cette thèse : (1) l'architecture client/serveur et (2) l'architecture pair-à-pair.

1.4.3.1 Client/serveur

Cette architecture a pris de l'ampleur avec le début des années 1990s. C'est une extension de la base de données centralisée, où les fonctions légères comme les applications sont exécutées sur le client, tandis que les fonctions d'optimisation et de gestion de données sont exécutées sur le serveur. Il pourra y exister plusieurs clients qui communiquent via un réseau avec le même serveur [DR93]. En effet, le serveur est le responsable du traitement et d'optimisation des requêtes, de la gestion des transactions et du stockage. En revanche, le client possède un module de client SGBD qui gère les données cachées au niveau du client. Autrement dit, le client envoie les requêtes SQL au serveur sans les traiter, et ce dernier effectue la majorité du travail, y compris l'optimisation et l'exécution [ÖV20]. Une illustration de cette architecture est présentée dans Figure 1.7.
Il existe deux types principaux dans cette architecture :

— **Multi-clients/Un-serveur [KB96] :** c'est l'extension la plus proche aux bases de données centralisées, où la base de données est stockée sur une seule machine (serveur) qui héberge également le logiciel pour la gérer, avec une seule différence où les clients sont responsables du cache des données, grâce aux protocoles de cohérence de cache.

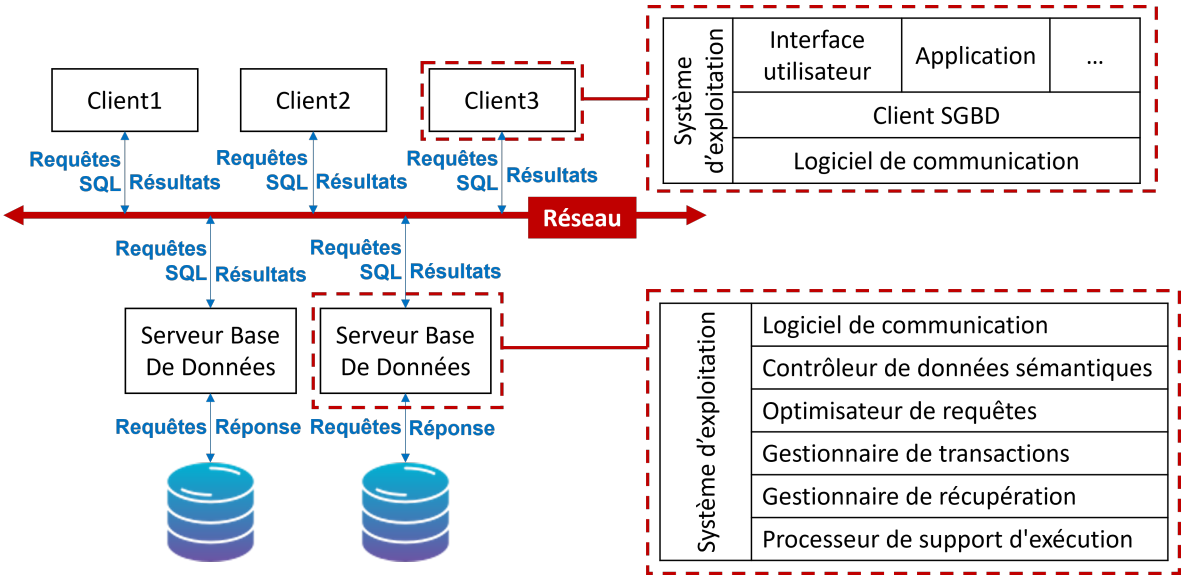


FIGURE 1.7 – Architecture client/serveur dans les SGBD (Édite de [ÖV20]).

— **Multi-clients/Multiserveurs [OHE96] :** c'est une manière plus sophistiquée pour représenter cette architecture, où la base de données est hébergée sur un cluster (plusieurs machines). Il existe deux possibilités pour accéder aux serveurs : (1) soit chaque client gère sa façon d'y accéder, (2) soit chaque client définit un seul serveur comme *serveur principal*, qui communique avec les autres serveurs pour extraire les résultats. Dans le deuxième cas, une transparence de gestion de données est présente, où le client (l'utilisateur) ne prend pas en charge la gestion d'accès aux différents serveurs mais le système qui le fera. En effet, les données sont partitionnées et peut être répliquées sur l'ensemble des serveurs qui communiquent entre eux d'une manière transparente pour répondre aux requêtes de l'utilisateur. Cette approche est utilisée dans les SGBDP pour paralléliser le traitement.

1.4.3.2 Pair-à-pair

L'architecture pair-à-pair (connue aussi sous l'acronyme P2P) est conçue pour le partage des ressources informatiques, y compris les données, le stockage et les cycles de UC via un échange direct par un passage de messages, sans devoir passer par un support ou une autorité centralisée. L'avantage principal de cette architecture est sa capacité de s'adapter aux pannes tout en gardant une performance acceptable [ATS04]. Dans les SGBDD, cette architecture suit une conception ascendante dans les BDD, où un SCG [7] est créé pour l'ensemble de la base de données et un SCL [8] est créé pour chaque site. En effet, l'entrée est une base de données (centralisée) avec sa propre définition de schéma SCG. Cette base de données est partitionnée et allouée aux sites du SGBDD qui possèdent leur propre schéma SCL. L'utilisateur formule ses requêtes selon le SCG en ignorant l'emplacement des données. Ces requêtes sont ensuite partitionnées en sous-requêtes locales par le SGBDD compatibles avec les SCL, qui sont exécutés sur l'ensemble des sites communiquant entre eux [DVS05, ÖV20]. Dans les SGBDP avec une configuration homogène, cette architecture peut être adaptée à un cluster avec plusieurs nœuds (machines), avec un SCG appliqué sur l'ensemble des nœuds de la BDD. Ces derniers communiquent et échangent les données entre eux pour répondre aux requêtes des utilisateurs.
Figure 1.8 résume le fonctionnement de l'architecture pair-à-pair. Le module *processeur utilisateur* et le module *processeur de données* sont présents sur chaque site/nœud de la BDD.

7. *Schéma Conceptuel Global*
8. *Schéma Conceptuel Local*

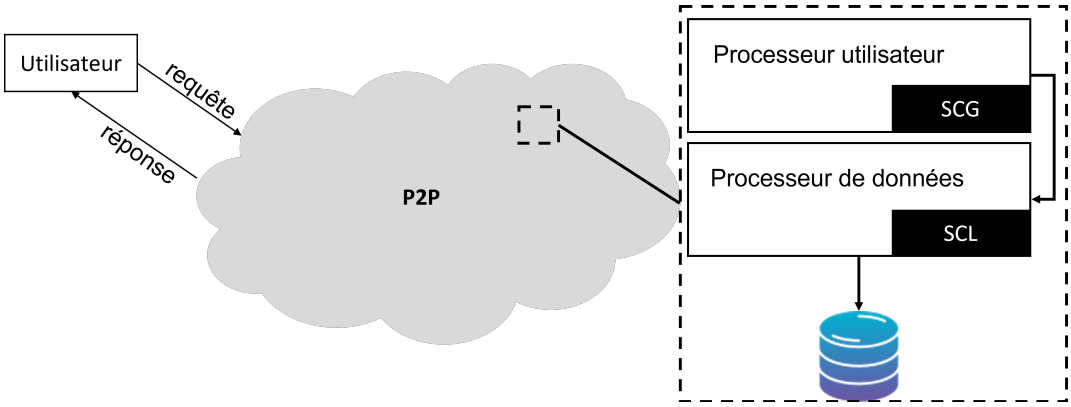


FIGURE 1.8 – Architecture pair-à-pair dans les SGBD (Extraite de [DVS05, ÖV20]).

1.4.4 Architectures logiques des bases de données distribuées

Il existe plusieurs architectures logiques pour organiser et traiter les données dans les SGBD, nous allons résumer ci-dessous, quelques architectures que nous jugeons importantes et qui justifient notre choix d'architecture utilisée dans cette thèse.

1.4.4.1 Architecture à mémoire partagée (shared-memory)

Cette architecture, aussi connue sous le nom d'architecture à tous partagés (shared-everything), permet à n'importe quel processeur d'accéder à tout module de mémoire ou unité de disque à l'aide d'un bus d'interconnexion [BCV93]. L'avantage majeur de cette architecture est la simplicité de la programmation, où le parallélisme des inter-requêtes est facile alors que le parallélisme intra-requêtes nécessite un peu de travail mais il reste plutôt simple. Également, vu que tous les processeurs peuvent accéder à tous les données en mémoire, l'équilibrage de charge est facilement atteignable.
Il existe deux types dans cette architecture illustrée dans Figure 1.9, ceci dépend du partitionnement de la mémoire physique :

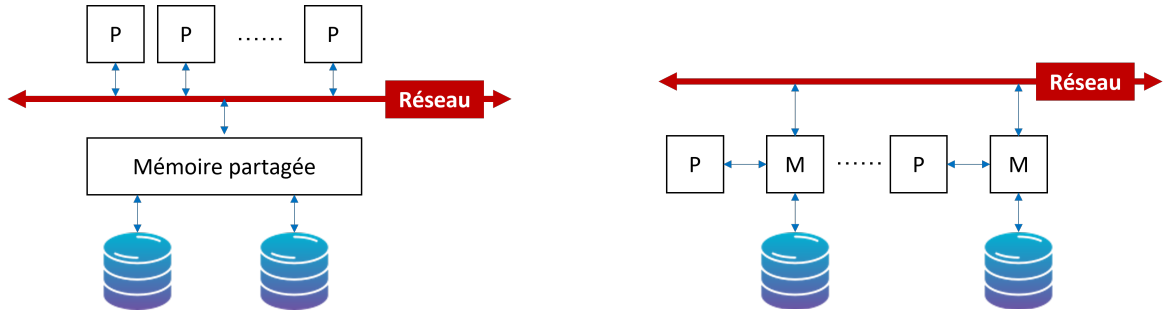
— **Accès uniforme à la mémoire (UMA) [PMC⁺90, ÖV20] :** cette architecture est apparue dans les années 1960s avec l'arrivée des mainframes et elle portait le nom de SMP [9]. Dans cette architecture, la mémoire physique est accessible par tous les processeurs dans un temps constant. Néanmoins, cette architecture souffre des coûts élevés et de la scalabilité limité. En effet, les processeurs sont devenus de plus en plus rapide ce qui conduit à des conflits au niveau de la mémoire principale et par conséquent, les performances sont dégradées. Ainsi, le passage à l'échelle est déconseillé pour cette stratégie et il est limité à dix processeurs au maximum. De plus, la mémoire est partagée par tous les processeurs, ce qui permet à une erreur dans celle-ci d'affecter tous les processeurs.

— **Accès non-uniforme à la mémoire (NUMA) [Lam13] :** cette stratégie est fondée sur le principe que chaque processeur possède son propre mémoire locale qu'il peut accéder efficacement et que l'accès à la mémoire virtuellement partagée est différent selon que celle-ci soit locale ou distante. L'avantage de NUMA est de fournir un modèle de programmation basé sur une mémoire partagée avec tous ses bénéfices. Cette stratégie est la plus utilisée aujourd'hui car elle permet une scalabilité.

1.4.4.2 Architecture à disque partagé

L'architecture à disque partagé, illustrée dans Figure 1.10, se distingue par la possibilité d'accéder à n'importe quel unité de disque via une interconnexion et copier les pages mémoires dans la

9. *Symmetric Multi-Processor (Multi-Processeur Symmétrique)*



(a) UMA (b) NUMA

FIGURE 1.9 – Architecture à mémoire partagée (édité de [ÖV20]).

mémoire non partagée de chaque processeur, où chaque couple processeur-mémoire est sous le contrôle de son propre système d'exploitation. Précisément, chaque processeur accède aux pages de la BD [10] sur le disque partagé et les copie dans le cache de sa propre mémoire. Cependant, ceci peut provoquer un blocage lorsque plusieurs processeurs demandent l'accès à la même page dans un mode de mise à jour conflictuel. Pour cela, un gestionnaire de verrouillage distribué est utilisé pour gérer l'accès conflictuel aux différentes pages sur le disque partagé. Plusieurs BD ont proposé une implémentation efficace du gestionnaire de verrouillage distribué pour implémenter cette architecture, y compris Oracle, IBM, Microsoft et Sybase [ÖV20].

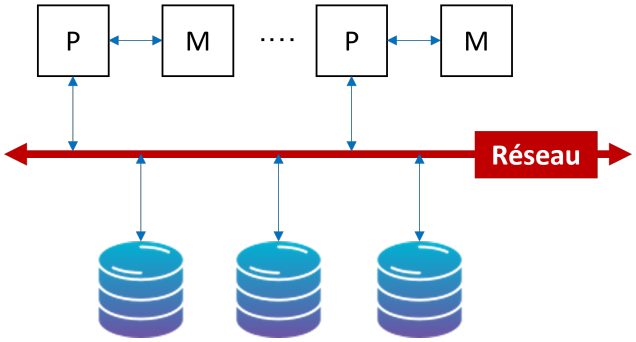


FIGURE 1.10 – Architecture à disque partagé (édité de [ÖV20]).

Selon [BCV93], cette architecture offre deux avantages intéressants pour les BD : (i) elle permet une meilleure scalabilité et disponibilité que les architectures à mémoire partagée tout en offrant un équilibrage de charge parfait vu que tous les processeurs peuvent accéder à n'importe quelle donnée sur les disques partagés, ainsi (ii) les SGBD conçus pour les systèmes centralisés peuvent facilement évoluer en cette architecture contrairement à l'architecture à mémoire partagée. En revanche, elle doit prendre en charge le coût de la cohérence lorsqu'il y a plusieurs mise à jour à réaliser sur les disques partagés, ainsi que la gestion de l'interconnexion lorsqu'elle est saturée par suite des transferts excessifs des pages des différents disques.

1.4.4.3 Architecture sans partage

C'est une architecture distribuée qui ne permet pas l'accès partagé à la mémoire ou au stockage. Autrement dit, elle ne fournit aucune abstraction de partage de matériel, laissant la coordination des différents nœuds entièrement entre les mains du SGBDD. Elle est aussi connue sous le nom de MPP [11] pour le traitement parallèle massif. Chaque unité processeur-mémoire-disque qui représente une machine, est indépendante des autres machines et est sous le contrôle de sa

10. *Base de Données*
11. *Massively Parallel Processing*

propre copie du système d'exploitation. La communication entre les machines est établie via une interconnexion de réseau à haut débit en utilisant une interface de passage de messages. Le but de cette architecture est d'éviter le conflit entre les machines lors de la lecture/écriture de/sur la mémoire ou du/sur le disque [HSH07]. Cette architecture est largement utilisée dans la pratique, car elle permet un rapport efficace de coût/performance et peut facilement passer à l'échelle pour des configurations très large constituées de million de nœuds. Dans les SGBDD, chaque machine représente un site local avec sa propre partition de BDD et son SGBD (dans le cas de BDD hétérogène), ou un nœud dans le cas de SGBDP (dans le cas de BDD homogène) [PMC+90, BCV93].
Cette architecture favorise le passage à l'échelle en ajoutant des nœuds d'une manière incrémentale, offrant ainsi un extensibilité flexible. Néanmoins, avec l'ajout de chaque nœud, une réorganisation des données et un repartitionnement doit être effectué afin d'assurer l'équilibrage de charge entre les processeurs. De plus, cette architecture nécessite une réplication de données pour garantir une tolérance aux pannes, sinon un nœud défaillant rendra les données sur son disque indisponible. Plusieurs SGBDD ont opté pour cette architecture pour implémenter leur logiciel, à ce propos, nous retrouvons la BD de Teradata, IBM, Microsoft, Sybase, des vendeurs de BD orienté colonne comme MonnetBD et Vertica et enfin tous les SGBD NoSQL et Big Data [ÖV20].
Figure 1.11 représente un aperçu de l'architecture sans partage.

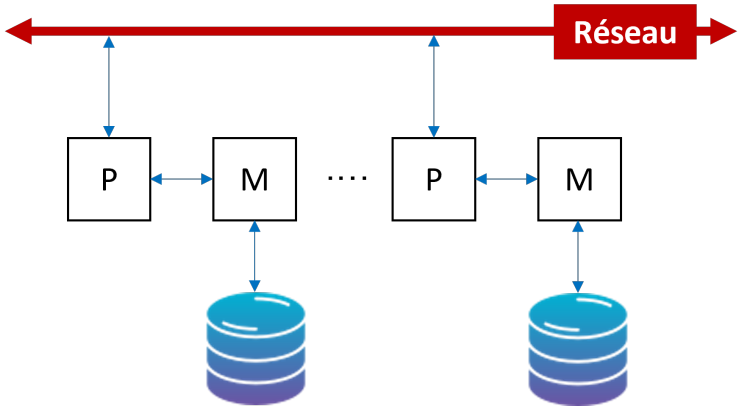


FIGURE 1.11 – Architecture sans partage (édité de [ÖV20]).

1.4.5 Traitement parallèle des données

La communauté des bases de données a toujours travaillé durement pour fournir des systèmes robustes de gestion de données. Particulièrement avec l'explosion des données dans l'ère du Big Data, le traitement distribué et parallèle des données a pris d'ampleur, pour cela des techniques et des méthodes de traitement parallèle pour stocker, gérer et récupérer les données ont vu le jour. Ces techniques se résument principalement dans des modèles de fragmentation des données qui garantissent l'intégrité, la cohérence et la sécurité des données stockées et des transactions offrant une performance raisonnable.

1.4.5.1 Définition du partitionnement des données

Selon le dictionnaire Larousse, créer des partitions est sa division en plusieurs parties, autrement dit, il s'agit de l'action de diviser une unité en plusieurs parties. Dans le contexte des BD, ceci signifie la division des données pour former des partitions, c-à-d. plusieurs ensembles de données. Les premières recherches sur le partitionnement des données dans les BD ont été conduites par Jeffrey Hoffer and Denis Severance [HS75] en 1975, où ils ont présenté le partitionnement des données comme une technique pour réduire le temps d'exécution des requêtes. Globalement,

ils ont défini deux types de partitionnement qui distribuent les données sur des sous-fichiers. Ainsi, ils ont prouvé que le temps de réponse des requêtes est réduit. Ces deux techniques de partitionnement sont fondées sur le fait de partitionner les relations au niveau des attributs ou des enregistrements, définissant ainsi implicitement les premières notions sur la fragmentation horizontale et verticale des données. Ces deux termes ont été adopté par la communauté des BD pour différencier le partitionnement par attribut (le partitionnement vertical) et le partitionnement par enregistrement (le partitionnement horizontal). Ils ont formalisé leurs algorithmes et les ont testés sur les bases de données centralisées. Leur définition pour le partitionnement est définie comme suivant :

Definition 1.7 *Étant donné un ensemble d'attributs $\mathcal{T}$ de la relation R, considérons une collection de sous-fichier $\mathcal{S} = \{(\mathcal{T}_i, R_i)\}$ avec $\mathcal{T}_i \subseteq \mathcal{T}$ et $R_i \subseteq R$, spécifiant les attributs et entités représentés dans un sous-fichier i. S regroupe les données sous deux formes de base :*

1. *Par attributs avec $R_i = R$ et $\cup_{i=1}^{n} \mathcal{T}_i = \mathcal{T}$ pour $i = 1..n$.*
2. *Par enregistrement avec $\mathcal{T}_i = \mathcal{T}$ et $\cup_{i=1}^{n} R_i = R$ pour $i = 1..n$.*

Le cluster $\mathcal{S}$ avec $\mathcal{T}_i \cap \mathcal{T}_j = \emptyset$ ($R_i \cap R_j = \emptyset$) pour tout $i \neq j$, est appelé une partition d'attributs (d'enregistrements) pour la BD.

La fragmentation a été explicitement introduit en 1977, comme la création des sous-relations utilisées comme unités de distribution entre les nœuds d'un SGBDD [RG77]. Ensuite, en 1978, les termes de partitionnement horizontal et partitionnement vertical ont été établis pour définir les stratégies de partitionnement et de placement des données dans les SGBDD [LGS+79].
En regroupant les données dans des ensembles unis, l'objectif majeur du partitionnement de données est de réduire les E/S dans les BDC [12] [BKS00], et de minimiser le trafic de données dans les BDD, qui représente un défi capital pour les SGBDD [BW09].
Les BDD exploitent le parallélisme afin de fournir des systèmes performants et hautement disponibles, pour cela ils définissent trois types de parallélismes [ÖV20] :

(i) le parallélisme inter-requêtes qui permet l'exécution parallèle de plusieurs requêtes,

(ii) le parallélisme intra-requête permettant l'exécution parallèle d'opérations indépendantes dans une seule requête, et

(iii) le parallélisme intra-opération, dans lequel la même opération est exécutée en plusieurs sous-opérations.

Ce parallélisme est atteint grâce à une stratégie appelée *désagrégation* (declustering en anglais), qui permet de répartir les données entre les nœuds de traitement afin de garantir l'équilibrage de charge. Le terme de désagrégation (declustering) a été initialement introduit pour résoudre le problème d'allocation de données sur les multidisques en 1986 [FLC86], puis il a été adopté par les SGBDP pour désigner le partitionnement horizontal des données sur l'ensemble des nœuds de traitement.

1.4.5.2 Modèle de fragmentation des données

La fragmentation [13] est importante pour assurer la performance des SGBDD, en revanche, elle peut soulever plusieurs défis, y compris de bien partitionner les données pour éviter une importante communication de données lors de l'exécution des jointures parallèles et pour que tous les nœuds de traitement effectue la même charge de travail, ainsi que minimiser le trafic des données pour ne pas saturer la bande passante du réseau, qui ralentira la réponse du SGBDD.
Selon [ÖV20], il est important que la BDD ne subit pas des modifications sémantiques durant la fragmentation. Autrement dit, la fragmentation ne doit pas permettre la perte des données

12. *Bases de Données Centralisées*

13. Le terme fragmentation et le terme partitionnement sont utilisés indifféremment dans cette thèse pour désigner l'action de diviser les données sur les nœuds de traitement.

et que lors de la reconstruction de le BDD, toutes ses données doivent être présentes. Pour cela, la fragmentation doit respecter les propriétés définies ci-dessous :

— *Complétude :* cette propriété assure que chaque données dans la relation globale R se trouve dans un ou plusieurs fragments de R. Par exemple, si R est fragmentée en n fragments $F_R = \{R_1, R_2, .., R_n\}$, chaque données de R doit être dans un ou plusieurs R_i.

— *Reconstruction :* qui s'agit de l'existence d'un opérateur ∇ qui permet de regrouper les fragments de la relation globale R tels que : $\nabla R_i = R$ pour $i \in [1, n]$. Cet opérateur dépend du type de la fragmentation effectuée.

— *Disjonction :* qui permet l'appartenance d'un élément de données d_i à un seul fragment des fragments de la relation globale R. À ce propos, si d_i est dans R_i, il ne peut pas être dans R_j avec $i \neq j$.

Nous distinguons trois types de fragmentation :

Fragmentation horizontale [CO82, CNW83] : En utilisant l'opération de sélection (σ), la fragmentation horizontale permet de partitionner les données au niveau des enregistrements. Ainsi, chaque fragment contiendra un sous-ensemble de tuples de la relation globale R. Il existe deux versions de la fragmentation horizontale :

(i) *La fragmentation horizontale primaire :* elle est effectuée à l'aide de l'opération de sélection sur des prédicats définis sur une relation source R, où ses fragments horizontaux sont définis par :

$$R_i = \sigma_{F_i}(R), 1 \leq i \leq n \tag{1.7}$$

avec F_i est le prédicat de fragmentation (prédicat de sélection). Par exemple, la fragmentation ci-dessous est effectuée sur la base des salaires :

$$EMP_1 = \sigma_{salaire>1500}(EMP)$$
$$EMP_2 = \sigma_{salaire\leq 1500}(EMP)$$

(ii) *La fragmentation horizontale dérivée :* elle s'agit du partitionnement d'une relation R qui résulte de la définition de prédicats sur une autre relation S [BKL98]. En effet, elle s'applique aux relations cibles dans le graphe de jointure et est exécutée sur la base de prédicats définis sur la relation source du graphe de jointure. À ce propos, définissant la relation source S et la relation cible R, les fragments horizontaux dérivés de R sont définis par :

$$R_i = R \ltimes S_i, 1 \leq i \leq n$$
$$avec\ S_i = \sigma_{F_i}(S)$$

Fragmentation verticale [NCWD84, SW85] : Elle représente le partitionnement au niveau des attributs, où l'opération de projection (π) est utilisée pour accomplir ce type de partitionnement. Plus précisément, la fragmentation verticale de la relation R produit $F_R = \{R_1, R_2, .., R_r\}$, où chaque fragment contient un sous-ensemble des attributs de R avec la clé primaire. Cette dernière est impliquée dans chaque fragment pour permettre la reconstruction de R, ainsi que renforcer l'intégrité des données. Pratiquement, ce type de fragmentation est plus complexe que la fragmentation horizontale, à cause du nombre important des alternatives possibles. Il existe, néanmoins, des techniques pour mieux encadrer ce nombre :

(i) *Grouper :* c'est un processus itératif qui commence par créer un fragment pour chaque attribut, puis regrouper certains des fragments dans un seul fragment jusqu'à ce que certains critères soient satisfaits.

(ii) *Diviser :* il s'agit de commencer par la relation globale R, puis selon le comportement des applications utilisateurs, décider du nombre de fragments ainsi que l'appartenance des attributs aux différents fragments.

Fragmentation hybride : Également appelée la fragmentation imbriquée, elle s'agit de la combinaison des deux fragmentations précédentes, c-à-d. elle se produit lorsqu'une table est partitionnée verticalement et qu'elle est ensuite partitionnée horizontalement (ou inversement). Selon [ÖV20], le nombre de niveaux d'imbrication peut être important mais il s'arrête lorsque chaque fragment n'est constitué que d'un seul tuple.

1.4.5.3 Équilibrage de charge

Les SGBDP doivent prendre en charges l'exécution parallèle des intra-transactions et les inter-transactions. Le parallélisme des intra-transactions est une forme de parallélisme dans lequel une requête complexe est décomposée en plusieurs tâches élémentaires, qui sont exécutées de manière concurrentielle sur plusieurs processeurs (nœuds de traitement). En revanche, les inter-transactions correspondent à une autre forme de parallélisme, où plusieurs requêtes différentes (multi-utilisateurs) sont exécutées simultanément et concurrentiellement sur plusieurs processeurs [Rah96]. Pour cela, une stratégie d'équilibrage de charge entre les nœuds de traitement du SGBDP doit être mise en place.

L'équilibrage de charge fait référence à l'allocation de la charge de travail, qui doit être répartie entre plusieurs nœuds de traitement dans les systèmes distribués [Rah96]. Notamment, l'allocation de la charge de travail s'agit de l'affectation des étapes de traitement à des ressources physiques ou logiques. En outre, l'objectif principale de l'équilibrage de charge est de permettre à tous les nœuds de traitement de terminer leurs calculs en même temps, ainsi d'exploiter équitablement les ressources de traitement, tout en minimisant le trafic de données entre eux, ce qui peut être considérablement pénalisant lors du traitement parallèle.

Nous nous intéressons, particulièrement dans cette thèse, à l'équilibrage de charge dans les SGBDP construit sur une architecture Share-Nothing. Pour cela, nous analysons d'abord les défis majeurs avec cette architecture, lors de l'implémentation d'une stratégie d'équilibrage de charge. Puis, nous expliquons brièvement les stratégies existantes.

D'abord, le parallélisme intra-transactions est principalement fondé sur le partitionnement des relations et la distribution des données asymétriques, ce qui peut facilement déséquilibrer la charge entre les processeurs [BAC$^+$90, DGS$^+$90, WDJ91, AvdBF$^+$92]. En effet, un mauvais modèle de partitionnement peut augmenter la communication entre les nœuds de traitement, lors de l'exécution des requêtes complexes (qui contiennent plusieurs opérations de jointure). Étant donné que le temps de communication est coûteux [PRS18b], le temps d'exécution de ces requêtes devient très important. Ensuite, le parallélisme inter-transactions provoque inévitablement un conflit de ressources, car les requêtes complexes exigent une puissance élevées de UC, une capacité de mémoire importante et une large bande passante pour le disque, ce qui peut entraîner des retards importants pour l'exécution simultanée de transactions [WFA95, BFV96].

Il existe principalement deux types de stratégie pour assurer l'équilibrage de charge dans les SGBDP [MD95, RM95, GI96] :

(1) Équilibrage de charge statique : qui est fondé sur des hypothèses préalables sur les besoins du système, comme les ressources nécessaires et l'heure d'arrivée des tâches. Il vise à associer un ensemble connu de tâches aux processeurs disponibles afin d'améliorer la performance. Il est exécuté durant l'optimisation du système distribué.

(2) Équilibrage de charge dynamique : prend en considération l'état actuel de la charge sur l'ensemble des nœuds de traitement, ce qui lui permettre de déplacer dynamiquement des tâches entre les nœuds surchargés à d'autre nœuds sous-chargés, afin de recevoir rapidement les résultats du traitement. Ce type de stratégie est appliqué lors de l'étape d'exécution des traitements.

1.4.5.4 Partitionnement des graphes

Nous avons discuté, dans le début de cette section, les différents modèles de partitionnement de données dans les bases de données relationnelles. Cependant, à cause de la complexité de sa structure, il s'avère nécessaire de discuter le partitionnement de graphe à part. Le partitionnement du graphe est un problème fondamental dans plusieurs domaine comme le traitement parallèle et distribué, l'équilibrage de charge et les circuits VLSI [14] [KHKM11]. Il s'appuie principalement sur deux approches : (1) l'approche par coupe d'arête (également appelée sommet disjoint), et (2) l'approche par coupe de sommet (également appelée arête disjoint). Formellement, l'approche par coupe d'arêtes est définis par [BS13] :

Definition 1.8 *Soit $G = (V, E)$ un graphe et $P_k = \{V_1, V_2, .., V_k\}$ un ensemble de partition de V, P_k est un partitionnement de G si :*
(1) $\forall V_i \subset P_k, i \in \{1, 2, ..k\}$ $/V_i \neq \emptyset$ (aucun élément de P_k n'est vide),
(2) $\forall V_i, V_j \subset P_k, i \neq j$ $/V_i \cap V_j = \emptyset$ (tous les éléments de P_k sont deux à deux disjointes),
(3) $\cup_{i=1}^{k} V_i = V$ (l'union de tous les éléments de P_k renvoie l'ensemble V).

Ainsi, l'approche par coupe de sommet est définie par :

Definition 1.9 *Soit $G = (V, E)$ un graphe et $P_k = \{E_1, E_2, .., E_k\}$ un ensemble de partitions de E, P_k est un partitionnement de G si :*
(1) $\forall E_i \subset P_k, i \in \{1, 2, ..k\}$ $/E_i \neq \emptyset$ (aucun élément de P_k n'est vide),
(2) $\forall E_i, E_j \subset P_k, i \neq j$ $/E_i \cap E_j = \emptyset$ (tous les éléments de P_k sont deux à deux disjointes),
(3) $\cup_{i=1}^{k} E_i = E$ (l'union de tous les éléments de P_k renvoie l'ensemble E).

Definition 1.10 *Dans l'approche par coupe d'arêtes, chaque sommet est affecté à une seule partition, mais les arêtes peuvent être répliquées entre les partitions si elles connectent des sommets frontières. En revanche, dans l'approche par coupe de sommet, chaque arête est affectée à une seule partition, mais les sommets peuvent être répliqués entre les partitions s'ils sont incidents à des arêtes qui sont allouées à différentes partitions [ÖV20].*

Dans tous les cas, le partitionnement de graphe vise trois objectifs essentiels : (1) allouer chaque sommet ou arête à des partitions de sorte que les partitions deviennent deux à deux disjointes, (2) s'assurer que les partitions sont équilibrées (pas de partitions vides), et (3) minimiser les coupes afin de minimiser la communication entre les machines auxquelles chaque partition est affectée [AR04].
Nous retrouvons plusieurs contributions dans ce problème, entre autres, METIS [KK95b, KK95a] qui est l'un des premiers algorithmes de coupe d'arêtes. Cette méthode a connu plusieurs extensions telles que les versions proposées par [ARK06, WXSW14]. Il existe également des algorithmes qui combinent les deux approches comme PowerLyra [CSC+19]. Dans cette thèse, nous utilisons la méthode proposé par [PRS18b] que nous expliquons dans Chapitre 3.

1.4.6 Système de gestion de données et d'exploration de graphes

Après avoir examiné les diverses stratégies de fragmentation de données précédemment, nous passons maintenant à leur application dans divers environnements de données pour l'exploration de graphes. Ces environnements de données peuvent être parfaitement utilisés pour représenter des graphes et les explorer. En effet, l'utilisation des requêtes récursives permet de parcourir le graphe, d'extraire ses index ou d'explorer sa structure [Ord13, AOB18].
Dans cette thèse, notre principal champ d'intérêt réside dans les SGBD orientés colonnes. Cependant, avant d'illustrer ce type de stockage, nous introduisons d'abord le stockage orienté ligne.

14. *Very Large Scale Integration*

1.4.6.1 SGBD orienté lignes

Les SGBD orienté ligne stockent les données dans des blocs d'enregistrement. Ils utilisent des index à base du *rowid* pour accéder à l'emplacement des données physiques. Plus précisément, les champs des enregistrements sont stockés dans des espaces mémoires contiguës. Ces systèmes ont pour but d'exploiter les E/S de la bande passante de différents disques, afin d'améliorer l'efficacité de la lecture et de l'écriture, ainsi que le stockage des données volumineuses sur une seule machine [DG92]. Sur les SGBD en ligne distribués et parallèles, les grandes tables sont partitionnées sur l'ensemble des machines du cluster. Ce partitionnement est effectué horizontalement selon trois stratégies possibles : (1) un partitionnement horizontale de la table selon les valeurs d'une colonnes, (2) une distribution des enregistrements de la table en tourniquet (round robin), ou (3) en utilisant une fonction d'hachage pour l'attribution de chaque enregistrement à sa machine correspondante.

1.4.6.2 SGBD orienté colonnes

Les SGBD orienté colonnes sont apparus en introduisant une amélioration remarquable d'OLAP [15]. Ils présentent une manière considérablement différente de stockage des données. Contrairement aux SGBD en ligne, les SGBD orienté colonnes stockent les colonnes dans des fichiers séparés en tant que grandes données contiguës. Ils reposent sur des projections (des collections optimisées de colonnes de table qui fournissent un stockage physique des données. Les projections peuvent contenir une partie ou la totalité des colonnes d'une ou plusieurs tables) et sont généralement construits sur des architectures de traitement massivement parallèle (MPP).
Le stockage des données par colonne repose sur l'utilisation de la compression. Ainsi, le SGBD en colonnes présente une meilleure performance d'écriture et de lecture de données vers et depuis le disque par rapport aux SGBD en ligne. Ceci accélère le temps nécessaire pour renvoyer une requête, en particulier ceux définissant les jointures et les agrégations. La compression améliore les performances sans aucun impact sur le parallélisme. Le partitionnement des données dans les SGBD orienté colonnes est effectué avec une fragmentation verticale des projections ou une fragmentation hybride.
Le SGBD orienté colonnes et le SGBD orienté lignes peuvent utiliser des langages de requêtes de base de données traditionnels SQL pour charger les données et les interroger.

1.4.6.3 SGBD orienté graphe

Les SGBD orientés graphes adoptent leur propre approche pour stocker les données du graphe. Ils utilisent une couche native de stockage permettant la gestion et le stockage optimisé des graphes. Ils utilisent la liste d'adjacence pour représenter les données connectées du graphe, où les sommets et les arcs pointent les uns vers les autres. Afin d'explorer le graphe, ces SGBD utilisent le principe de la contiguïté sans index, où toutes les jointures nécessaires sont effectivement précalculées et stockées dans la base de données de graphes, dans la liste d'adjacence. Ceci revient à effectuer des recherches avec un coût de $O(1)$.
Le traitement parallèle de ces SGBD repose sur la réplication entière des données sur l'ensemble des nœuds de traitement, ce qui permet de ralentir considérablement le système lorsque le graphe devient plus grand que la capacité mémoire. En effet, ces SGBD ne prennent pas en charge le traitement distribué et ne permettent pas le passage à l'échelle à cause de leur architecture, qui repose sur des relations complexes entre les objets (sommets et arêtes).

1.4.6.4 Comparaison entre les système de gestion de données

Après avoir présenté brièvement les différents SGBD qui permettent le traitement et l'exploration des graphes, nous élaborons une comparaison entre ces SGBD dans Table 1.1, sur la base des

15. *Online Analytical Processing*

critères suivants : (1) l'orientation, (2) le type de stockage, (3) le traitement distribué, (4) la stratégie de partitionnement, (5) le passage à l'échelle, (6) le coût de recherche.

Tableau 1.1 – Comparaison entre les systèmes de gestion de données

	SGBD en lignes	SGBD en colonnes	SGBD graphes
Orientation	Données relationnelles	Données relationnelles	Graphes
Stockage	Tables avec attributs et tuples	Tables avec attributs et tuples	Listes d'adjacence (sommets, arcs et propriétés)
Distribution	Oui	Oui	Non
Partitionnement	Fragmentation horizontale	Fragmentation verticale/hybride	Réplication de tous les données
Scalabilité	Facile	Facile	Difficile
Coût de recherche	O(m log (m)) avec index	O(m) avec les projections	O(1)

1.5 Représentation des graphes dans les bases de données relationnelles

L'étude et l'exploration des graphes nécessite leur modélisation d'une façon compatible avec les environnements de leur traitement. D'ailleurs, la partie logicielle de n'importe quel système d'information est constitué principalement des structures de données et d'un algorithme pour traiter ces données [UGF+20]. L'algorithme est conçu principalement pour traiter un type particulier de structure de données, en revanche, le choix des structures de données impacte profondément l'efficacité de la méthode de traitement de ces données. Ceci peut être reflété dans nombreux systèmes informatiques modernes, y compris la programmation orientée objets et les bases de données.

La représentation des données dans une base de données relationnelles requiert leur normalisation. Une base de données est considérée comme normalisée si toutes ses relations respectent, au moins, la troisième forme normale. Pour rappel, la première forme normale (1NF) exige l'atomicité de chaque attribut (c-à-d. il doit stocker une valeur simple et non-multiple). Une relation est en seconde forme normale (2NF) si et seulement si elle est en 1NF et que tous ses attributs non-clés dépendent uniquement de sa clé primaire. Enfin, la relation est en troisième forme normale (3NF) si elle est en 2NF et aucun de ses attributs non-clés dépend de ses autres attributs non-clés.

La représentation d'un graphe $G = (V, E)$ par sa matrice d'adjacence dans une base de données relationnelle satisfait 1NF, car chaque cellule de la matrice stocke une seule valeur. Néanmoins, elle ne respecte ni 2NF, ni 3NF. Dans la matrice d'adjacence, chaque ligne doit afficher la relation de voisinage entre le sommet de la ligne et tous les autres sommets. De plus, il existe une dépendance entre les lignes et les colonnes, lorsqu'une modification touche à une arête, elle doit forcément la changer dans les deux sens. Par exemple, en ajoutant une arête entre i et j, les deux cellules $\mathcal{A}_{ij}$ et $\mathcal{A}_{ji}$ doivent égales à 1.

De même, la représentation par la liste d'adjacence ne satisfait pas 1NF vu que chaque sommet doit pointer vers ses voisins. Par conséquent, la relation modélisant cette représentation doit définir une valeur multiple dans son second attribut. Figure 1.12 représente une relation basée sur la liste d'adjacence dans Figure 1.4.

De plus, ces représentations ne définissent pas un nombre fixe de colonnes (à l'exception de la représentation de la liste d'adjacence par une table où la deuxième colonne est de type complexe). Ceci ne permet pas de définir un modèle relationnel de données ; en ajoutant un sommet ou en le retirant, la définition de la table sera changée. La modification des définitions des tables

sommet	voisinage
a	b, c
b	a, c, d
c	a, b
d	b, e, f, g
e	d, f, g
f	d, e, g
g	d, e, f

FIGURE 1.12 – Liste d'adjacence dans la base de données.

relationnelles est une fonctionnalité coûteuse en termes de performance et de complexité, il est recommandé de l'éviter.
Avec toutes ces contraintes, la modélisation des graphes dans les bases données relationnelles doit respecter les conditions suivantes :

(1) Compatibilité avec 3NF,
(2) Minimisation du volume de stockage et évitement des cellules vides,
(3) Compatibilité avec tous les types de graphe : orienté/non-orienté et pondéré/non-pondéré,
(4) Simplification du traitement du graphe par les algorithmes de traitement de données.

La représentation la plus proche qui satisfait ces conditions est la liste d'adjacence.

$$S_L = \{v, N(v)\}$$

Le seul problème avec cette représentation est la complexité (non-atomicité) de son deuxième attribut. Pour le résoudre, il suffit de décomposer le second attribut en plusieurs lignes [BH10, CGK$^+$18, Sch19]. Ceci produira une relation de liste d'arêtes.

$$S_R = \{id, v_1, v_2, w\}$$

Où id, v_1 , et v_2, et w représentent respectivement la clé primaire de la relation, le sommet source, le sommet de destination et le poids de l'arête (cet attribut est optionnel dans le cas d'un graphe non-pondéré).
Tout attribut de cette relation caractérise une arête du graphe à l'exception de id. Ce dernier est optionnel et peut être remplacer par une concaténation de $\{v_1, v_2\}$ dans un graphe simple (sans boucle avec une seule liaison entre les couples de sommets). Nous illustrons dans Figure 1.13, la représentation du graphe de Figure 1.1b dans les bases de données relationnelles.
Cette représentation respecte les conditions suscités. D'abord, elle est en 1NF vu que tous ses attributs sont simples et atomiques, ainsi elle est en 2NF comme tous ses attributs non-clé (w) ne dépendent que de sa clé primaire $\underline{(v_1, v_2)}$. Enfin, elle est en 3NF car elle est en 2NF et tous ses attributs non-clés (w) ne dépendent pas d'autres attributs non-clés. Ensuite, cette représentation préserve la mémoire en stockant que les informations pertinentes (pas d'attributs nuls ou vides). Elle permet aussi de modéliser tous les types de graphes (orienté/non-orienté et pondéré/non-pondéré). Enfin, cette représentation simplifie tout type d'opération comme l'ajout d'une arête, sa suppression ou sa modification sans altérer la définition de la table. De plus, elle permet facilement d'effectuer des jointures et des agrégations.
Nous adoptons cette représentation pour modéliser les graphes dans la base de données relationnelle dans les chapitres qui suivent.

R

v_1	v_2	w
a	b	1
a	c	1
b	a	1
b	c	1
b	d	1
c	a	1
c	b	1
d	b	1
d	e	1
d	f	1

R (suite)

v_1	v_2	w
d	g	1
e	d	1
e	f	1
e	g	1
f	d	1
f	e	1
f	g	1
g	d	1
g	e	1
g	f	1

FIGURE 1.13 – Liste d'arêtes.

1.6 Conclusion

Les graphes sont des structures de données très puissantes qui permettent de modéliser les relations entre les objets. Notamment, l'exploration des graphes permet de déduire des connaissances et des informations pertinentes, en extrayant des motifs intégrés. Dans ce chapitre, nous avons abordé les différentes notions de base sur les graphes. Nous avons présenté les représentations d'un graphe sous forme de matrice d'adjacence, de liste d'adjacence et de liste d'arêtes. Ensuite, nous avons définis les propriétés élémentaires et les motifs intégrés d'un graphe, en focalisant sur les triangles qui représente le cœurs de notre travail.
Ce chapitre a également étudié les différents modèles de stockage de données, où nous avons expliqué ses différentes architectures, ainsi que les stratégies de fragmentation de données et de l'équilibrage de charges. Nous nous sommes particulièrement concentrés sur la représentation des graphes dans les systèmes distribués et parallèles, en discutant les avantages et les inconvénients de ces moteurs de calcul. En récapitulant, ce chapitre constitue un repère théorique pour comprendre les différents notions traitées dans cette thèse, et ouvrira les portes de ce manuscrit pour accéder aux différents autres chapitres.
Le prochain chapitre portera sur une étude bibliographique sur l'énumération des triangles dans les graphes à grande échelle. En outre, il étudiera l'état de l'art focalisant sur l'analyse des graphes dans les perspectives des bases de données relationnelles. À noter que pour le problème de détermination de triangles, nous considérons le graphe non-pondéré non-orienté, vu que le graphe non-pondéré est un cas particulier du graphe pondéré avec le poids fixé à 1 sur chaque arête. Ainsi le graphe orienté est un cas particulier du graphe non-orienté, où chaque paire de sommets définit une seule direction entre eux. Dans la suite de cette thèse, l'utilisation du graphe $G = (V, E)$ signifie qu'il est simple (sans boucle) non-orienté et non-pondéré.

Chapitre 2

Travaux connexes : Techniques d'énumération et de comptage des triangles

The world is one big data problem.

— Andrew McAfee,

Sommaire

2.1 Introduction

Nous allons, dans ce chapitre, faire un état de l'art relatif à l'analyse des problèmes de graphe dans les bases de données distribuées. Il s'agit d'un courant qui s'intéresse à l'analyse des problématiques fondamentales de graphes en se basant sur le traitement parallèle des données dans les SGBD relationnelles. Dans cette thèse, notre attention se porte sur une problématique spécifique : l'énumération et le comptage des triangles. Il s'agit d'un problème classique visant à répertorier tous les triangles présents dans un graphe donné. Une première approche pour l'énumération des triangles a été introduite en 1978 par [IR78]. Depuis cet article qui fait référence dans le domaine, de nombreuses nouvelles approches sont sans cesse proposées. La difficulté majeure réside dans le fait de minimiser la complexité temporelle et spatiale tout en maximisant la qualité des résultats, notamment avec l'évolution exponentielle de la taille des graphes.
Dans un premier temps, cette étude bibliographique offre un aperçu des moteurs et systèmes de traitement de graphes. Ensuite, nous passons en revue les travaux de recherche qui se penchent sur l'analyse de graphes à grande échelle dans les bases de données relationnelles. Enfin, nous examinons les différentes approches pour l'énumération des triangles, tant dans les systèmes centralisés que dans les systèmes parallèles. Il est important de noter que la liste des travaux de recherche que nous présentons est significative, bien qu'elle ne soit pas exhaustive, et se concentre sur ceux que nous considérons comme les plus pertinents. Ces travaux sont présentés dans l'ordre chronologique.

2.2 Les systèmes d'analyse de graphe

De nombreux systèmes et moteurs d'analyse de graphes ont émergé, tous ayant pour objectif de traiter les graphes et d'en extraire les informations pertinentes. Pegasus [KTF09] est parmi les premiers systèmes d'analyse de graphe. Il s'agit d'une librairie libre distribution pour le traitement des graphes à grande échelle implémentée en dessus de Hadoop en utilisant le framework MR [1], il assure plusieurs opérations d'exploration de graphe à savoir : (1) PageRank, (2) marche alétoire avec redémarrage (Random Walk with Restart), (3) estimation du diamètre, et (4) composants connectés (CC [2]). Ensuite, Malewicz et al. ont présenté Pregel [MAB+10], le premier framework centré-sommet (vertex-centric) développé par Google pour le traitement et l'analyse des graphes. Il repose sur le principe selon lequel un sommet peut émettre et recevoir des messages, ce qui lui permet de modifier son état. Pregel implémente plusieurs algorithmes de traitement de graphe présentés sous forme d'une API abstraite. S'inspirant du même principe que Pregel, plusieurs systèmes ont été développés, notamment Apache Giraph [CEK+15], Apache GraphLab [LBG+12], et GiraphX [TD13]. Ces systèmes peuvent être considérés comme des dérivés du projet Pregel, offrant des améliorations notables en termes de performances et de parallélisme.
Néanmoins, les systèmes centrés-sommets sont inefficaces avec les graphes asymétriques qui présentent une forte existence des sommets à faible degré et une faible existence des sommets à haut degré. Pour remédier à ce problème, le modèle GAS [3] est initialement introduit par Powergraph [GLG+12] visant à paralléliser le traitement du graphe en distribuant ses arêtes au lieu de ses sommets. Ce modèle a été réutilisé par Spark GraphX [GXD+14]. Il s'agit d'un système d'analyse de graphes intégré populaire basé sur Apache Spark. GraphX refond les optimisations spécifiques aux graphes en optimisant les jointures distribuées et en maintenant les vues matérialisées. Plusieurs algorithmes d'exploration de graphe sont implémentés sur Spark GraphX : (1) PageRank, (2) Composants connectés (CC), et (3) le comptage des triangles, et bien d'autres.
À la différence de SQL, qui représente un langage standard pour interroger les bases de données

1. *MapReduce*
2. *Connected Components*
3. *Gather-Sum-Apply-Scatter*

relationnelles, il n'existe pas de langage universel pour le traitement des graphes. Cependant, nous pouvons identifier quelques langages populaires utilisés par certains moteurs et bases de données orientés graphes. Parmi ces langages, nous citons principalement trois : (1) SPARQL [PS08], (2) Cypher [Tea20], et (3) Grimlin [Tin21]. Ces trois langages sont les plus adoptés par les systèmes d'analyse de graphe en pratique et présentent des différences significatives. Tout d'abord, (1) SPARQL est principalement conçu pour opérer sur des données graphes RDF [4]. Ensuite, (2) Cypher a été spécifiquement élaboré pour manipuler les attributs des graphes. Enfin, (3) Gremlin, en revanche, se distingue par sa nature plus impérative par rapport aux deux autres langages, et son principal objectif est la traversée des graphes plutôt que la correspondance de motifs.
Malgré l'existence de plusieurs systèmes de traitement et d'analyse des graphes, ces systèmes restent limités, comme l'ont souligné plusieurs travaux [JRW+14, BBJ+14, GXD+14]. Plus concrètement, ils n'ont pas encore atteint le niveau de flexibilité physique, de mise à l'échelle, et de simplicité logicielle qu'un système de flux de données à usage général pourrait offrir. En effet, le développement de systèmes d'extraction de graphes distribués est intrinsèquement difficile. La parallélisation des calculs, le partitionnement des données, et la gestion des communications représentent d'importants défis dans le développement d'algorithmes efficaces pour les graphes distribués [KVH18].

2.3 Analyse des graphes dans les SGBD

Dans la suite, nous présentons notre étude bibliographique en deux catégories :

(1) Les travaux qui explorent principalement les différentes approches pour l'utilisation des SGBDR dans le traitement de données massives, en mettant particulièrement l'accent sur les données de type graphe.

(2) Les travaux qui ont proposé des solutions et des adaptations efficaces aux problématiques liées aux graphes dans le contexte des bases de données relationnelles.

2.3.1 Traitement des données massives sur les SGBD

Les premières analyses des données massives dans les bases de données relationnelles ont été effectuées par [CDD+09, HRS+12] qui ont présenté MADlib ; une suite complète à libre distribution de méthodes statistiques et d'apprentissage automatique, conçue pour s'exécuter sur un SGBDR. La multiplication matricielle, qui est la base de l'analyse des graphes dans les bases de données, est un élément primitive de MADlib, résolue grâce à l'utilisation de bibliothèques spécialisées d'algèbre linéaire telles que LAPACK. MADlib a été initialement déployé et testé sur le SGBD parallèle GreenPlum. Il prend désormais en charge la majorité des bases de données relationnelles, y compris PostgreSQL, MySQL, et bien d'autres. Ensuite, les travaux présentés dans [SCP10] ont introduit une approche pour l'analyse des graphes denses dans les SGBD relationnels. Les auteurs ont proposé un framework simple et léger pour intégrer les applications des graphes dans les bases données relationnelles. Ce framework exploite une couche de réseau étroitement couplé implémentée en utilisant les procédures stockées, tirant ainsi parti des fonctionnalités efficaces des bases de données modernes. Dans ce travail, les auteurs se sont principalement focalisés sur deux problèmes fondamentaux dans l'analyse des graphes : (1) la recherche en largeur (BFS [5]) qui permet le parcours d'un graphe/arbre en commençant par un sommet suivi par ses successeurs d'une façon récursive, et (2) la détection des cliques qui permet de lister tous les sous-graphes complets dans le graphe introduit. En outre, [Ord10] a étudié l'optimisation des requêtes récursives dans SQL en traitant le problème de la clôture transitive d'un graphe. Il s'est focalisé principalement sur trois optimisations :

4. *Resource Description Framework*
5. *Breadth First Search*

(1) l'évaluation précoce des conditions de la sélection des tuples (lignes),

(2) l'élimination des lignes en double dans les tableaux intermédiaires,

(3) la définition d'un index amélioré pour accélérer le calcul des jointures.

Une année après, [GJZ+11] a introduit FEM ; un framework avec trois opérations permettant d'implémenter des tâches de recherche de graphes dans le contexte des bases de données relationnelles :

(1) ce travail a exploité la fonction de la fenêtre (*window*) et la déclaration de la fusion (*merge statement*) du langage SQL afin de simplifier l'expression des requêtes formulées pour la découverte des plus courts chemins entre les sommets du graphe et améliorer leur performance,

(2) les auteurs ont implémenté leur framework en deux étapes principales, la première permet de réduire l'espace de recherche tandis que la deuxième a pour but d'exécuter l'algorithme de Djikstra pour la recherche des chemins optimaux,

(3) un index *SegTable* a été retenu pour les segments des chemins les plus courts afin d'améliorer les résultats de la recherche et la performance des requêtes.

Par ailleurs, [Ord13] a défendu la possibilité de l'analyse des données massives (*Big Data*) dans le contexte des bases de données relationnelles en utilisant le langage d'interrogation SQL. L'auteur a montré que les SGBDR permettent de manipuler et d'analyser les grands volumes de données, y compris les données des graphes.
Les premières analyses sur les graphes dans les bases de données orientées colonnes ont été introduit par [JMCH15] qui ont utilisé Vertica comme plateforme pour le traitement centré sommets des graphes (*vertex-centrix graph analysis*). Les auteurs ont montré la puissance et la robustesse du SGBD Vertica dans le traitement des problème fondamentaux des graphes, en présentant trois exemples : les plus courts chemins (SSSP [6]), les composantes connexes (CC), et le PageRank. En se basant sur les résultats motivants de ce travail, le SGBD Vertica a été retenu comme une plateforme d'expérimentation dans tous nos contributions pour son efficacité et sa robustesse.

2.3.2 Problématiques des graphes dans les SGBD

Les travaux de recherche présentés dans cette section sont classés en deux catégories : (1) les solutions internes aux SGBD qui regroupent l'ensemble des travaux proposant la formulation des problèmes de graphe en utilisant les requêtes SQL, et (2) les solutions externes aux SGBD qui s'intéressent aux articles proposant l'implémentation d'une couche externe au SGBD permettant le traitement des graphes dans une base de données relationnelle.

2.3.2.1 Solutions internes aux SGBD

Les solutions internes aux SGBD sont les approches qui utilisent SQL pour définir les différents propriétés et algorithmes d'exploration de graphes sans modifier la structure interne du SGBDR, et sans ajouter une couche externe pour la prise en charge de la structure du graphe. Plusieurs approches se sont présentées sous cette rubrique, à savoir [ZY17, CO17, OCG17, ALT+17, AOB18, ZO19]. Zhao et al. [ZY17] ont révisé le traitement des graphes dans les bases de données relationnelles au niveau SQL. Les auteurs ont proposé quatre nouvelles opérations algébriques qui permettent de prendre en charge les algorithmes d'exploration de graphes : (1) *MM-join* qui définit la jointure entre deux matrices, (2) *MV-join* pour la jointure entre une matrice et un vecteur, (3) *anti-join* permet d'éliminer les sommets et les arêtes des graphes qui ne sont pas nécessaires pour le traitement suivant, et (4) *union-by-update* qui traite les mises à jour de valeur pour calculer quelques métriques comme le PageRank, par exemple. Dans [OCG17], Ordonez et al. ont

6. *single source shortest path*

étudié l'optimisation des requêtes récursives dans les bases de données relationnelles orientées colonnes, en se focalisant sur deux problèmes fondamentaux et complémentaires de traitement de graphes : (1) la fermeture transitive et (2) la multiplication de la matrice d'adjacence. Dans cette approche, trois opérations ont été considérées : la sélection, la projection et la jointure (SPJ), où la projection englobe l'agrégation par group by. Les expérimentations de cette étude ont été réalisées sur trois SGBD : SGBD orienté lignes (*Row DBMS*), SGBD orienté colonnes (*Columnar DBMS*) et SGBD orientée tableaux (*Array DBMS*). Les résultats ont montré que le SGBD orienté colonnes est le plus rapide avec les bonnes optimisations. En revanche, il n'y a pas de différence entre le SGBD orienté lignes et le SGBD orienté tableaux. Une autre étude a été réalisée par [CO17] en formulant quatre algorithme d'exploration de graphe en utilisant les requêtes récursives en SQL : (1) l'accessibilité à partir d'un sommet source (*Reachability from a source vertex*), (2) le plus court chemin à partir d'une source unique (*SSSP*), (3) les composants faiblement connexes (*Weakly Connected Components*), et (4) PageRank. Les requêtes formulées ont été testées sur trois SGBD :(1) SGBD orienté lignes, (2) SGBD orienté colonnes, et (3) SGBD orienté tableaux. Les résultats ont été comparés avec Spark ; un système dédié pour le traitement des données massives y compris les données graphe. Une autre fois, le SGBD orienté colonne s'est montré efficace et robuste comparant aux autres systèmes dans l'exploration des graphes complexes et asymétriques.
EmptyHeaded [ALT+17] est un moteur relationnel pour le traitement des graphes de haut niveau. Il prend en charge un langage de requêtes riche de type journal de données (*Datalog*) et atteint des performances comparables à celles des moteurs de bas niveau. EmptyHeaded introduit une nouvelle architecture de moteur de jointure, comprenant un nouvel optimiseur de requêtes et des mises en page de données qui exploitent le parallélisme de données multiples à instruction unique (SIMD [7]). Cette nouvelle architecture a été testé sur trois types de requêtes sur les graphes : (1) le modèle de graphe, (2) le PageRank et (3) les plus courts chemins à source unique (SSSP). Al-Amin et al [AOB18] ont mené une étude sur plusieurs algorithmes élémentaires d'exploration de graphe : (1) les degrés entrants/sortants (*indegree/outdegree*), (2) la connectivité Top K, (3) le comptage de triangles, (4) les composants connectés, (5) le coefficient de clustering, (6) le chemin d'accessibilité, et (7) la circonférence (*Girth*). Cette étude a été menée sur trois systèmes : SGBD orienté lignes, SGBD orienté colonnes et Spark GraphX. Les résultats ont montré l'efficacité du SGBD orienté colonnes. Ce résultat a été théoriquement étudié par [OT18] qui ont présenté une analyse de complexité temporelle et d'accélération parallèle des requêtes relationnelles pour résoudre les problèmes de graphes. En effet, ils ont exposé des résultats théoriquement intuitifs sur l'estimation des cardinalités et la complexité temporelle en considérant la taille, la forme et la densité du graphe. [ZO19] ont formulé en SQL, deux problèmes d'exploration de graphe : (1) la centralité (*Betweenness Centrality*), et (2) le diamètre du graphe. Les expérimentations qui ont été menées sur deux systèmes : SGBD orienté colonnes et Spark GraphX, montrent que le SGBD orienté colonnes est plus performant. De plus, Zhao et al. [ZSYZ19] ont présenté un travail intéressant intitulé SQL-G, qui s'agit d'un système exploitant la robustesse de SQL et les requêtes récursives pour exprimer les opérations d'analyse de graphe. Les auteurs ont mis en œuvre un système qui intègre étroitement Spark SQL et GraphX sur la plateforme Spark. Dans un premier lieu, ils ont amélioré la prise en charge des requêtes récursives par Spark SQL, ensuite ils ont proposé de nouvelles règles de transformation pour optimiser et traduire les opérations des requêtes SQL récursives en opérations effectuées par GraphX. Les résultats obtenus se sont montrés concurrents avec l'efficacité des moteurs de traitement de graphes existants. En outre, Ali et al. [AET21] ont proposé une approche d'énumération de triangles à base de requêtes SQL. Cette solution s'inspire de celle proposée par [PC13, PMK16] exécutée dans un environnement MapReduce. Plus précisément, elle [AET21] repose sur deux étapes : (1) l'attribution des couleurs aux sommets, et (2) la création des classes de solutions basées sur les couleurs. Ces classes serviront ensuite pour la détermination des triangles. Enfin,

7. *Single Instruction Multiple Data*

[FZB+23b] ont proposé une solution récursive pour l'énumération des graphlets d'ordre 3 et 4, formulée avec des requêtes SQL. Cette solution est basée sur le partitionnement randomisé et est implémenté sur le modèle $k - machines$. Les résultats de cette solution démontre qu'elle est efficace et peut concurrencer les solutions existantes.

2.3.2.2 Solutions externes aux SGBD

Les contributions principales de cette classe d'algorithmes se résument dans [FRP15, SFS+15, XKD15b, SAL+17, HKJ+18, TXZ+20]. Ces travaux présentent des frameworks permettant de prendre en charge la structure du graphe et d'exploiter les fonctionnalités des SGBDR pour exécuter les algorithmes d'exploration des graphes. En effet, Fan et al. ont présenté *GRAIL* dans [FRP15], qui s'agit d'une couche synthétique sur une base de données relationnelle permettant la conversion des requêtes des graphes en script SQL. *SQLGraph* introduit dans [SFS+15] présente un schéma qui combine le stockage des données relationnelles avec le stockage des sommets et des arêtes en JSON. Cette approche prend en charge la structure du graphe (en sommets et arêtes) en traduisant les requêtes Gremlin d'apache sur les graphes en SQL par le biais d'une couche qui assure cette translation. Xirogiannopoulos et al. ont développé le framework *GraphGene* [XKD15b] qui permet aux utilisateurs de spécifier, d'une manière déclarative, des tâches d'extraction de graphes sur des bases de données relationnelles. Il permet également l'exploration visuelle des graphes extraits, et l'écriture et l'exécution des algorithmes de graphes sous deux manières différentes : (1) sur la base de données relationnelle, ou (2) en utilisant des bibliothèques de graphes externes existantes comme NetworkX de Python.
En 2017, Steer et al. ont présenté *Cytosm* [SAL+17] ; une application middleware permettant l'exécution des requêtes de graphe sur des bases de données relationnelles sans migration de données. *Cytosm* est basé sur un schéma nommé gTop contenant une topologie abstraite des propriétés de graphe avec son mapping vers la base de données utilisée. *Cytosm* utilise gTop pour exécuter efficacement les requêtes OpenCypher [8], en exploitant les informations du schéma pour optimiser le plan de requête et en faisant correspondre les concepts de requête au backend relationnel. *GRFusion* [HKJ+18] est un système de base de données relationnelles en mémoire, basé sur le SGBD VoltDB qui permet la manipulation des graphes en améliorant, d'une manière déclarative, le langage d'interrogation SQL et les moteurs de requêtes de VoltBD, afin d'exécuter des plans de requête inter-modèles de données. Ces derniers se composent d'opérateurs relationnels et d'opérateurs de graphe nouvellement introduits. Enfin, IBM a proposé IBM Db2 Graph, une approche de requête de graphe dans le SGBD qui permet la gestion de requêtes graphes interactives et modifiables au sein de la base de données relationnelle IBM Db2, comme décrit dans [TST+19, TXZ+20]. Leur approche est mise en œuvre en tant que

2.4 Énumération et comptage des triangles

Nous organisons la présente section en deux parties : (1) les approches centralisées, et (2) les méthodes distribuées et parallèles. Dans la première catégorie, nous listons les approches qui peuvent être placées dans la mémoire principale. En revanche dans la deuxième catégorie, nous nous intéressons aux approches dont les données dépassent la taille de la mémoire principale, ce qui nécessite un partitionnement efficace antérieur du graphe.

2.4.1 Méthodes centralisées

Parmi les algorithmes fondamentaux pour l'énumération des triangles est celui proposé par Itai et Rodeh [IR78] (voir Algorithme 1). Cette méthode a été initialement proposée pour extraire un seul triangle, néanmoins elle peut être facilement étendue pour lister tous les triangles du graphe. Cette approche peut être résumée dans les étapes suivantes :

8. Le langage d'interrogation des bases de données orientées graphe le plus utilisé.

(1) créer un arbre couvrant T du graphe,

(2) vérifier pour $\forall(i,j) \in E$ si $(pred(i), j) \in E$ ou $(pred(j), i) \in E$ avec $pred(i)$ est le prédécesseur de i et $pred(j)$ est le prédécesseur de j,

(3) lister $(i, j, pred(i))$ ou $(i, j, pred(j))$ respectivement comme triangle,

(4) enfin éliminer les arêtes de T du graphe et réitérer à partir de (1) jusqu'il n'y aura plus d'arêtes dans le graphe introduit.

Cette méthode est coûteuse car elle nécessite la modification de la structure du graphe à chaque itération, donc elle ne peut être utilisée sur des applications réelles.

Algorithm 1 Algorithme d'edge iterator

```
Input: G = (V, E)
Output: count
count ← 0;
for each edge (i, j) ∈ E do
    adj_1 ← {x|x ∈ adj(i), x > i};
    adj_2 ← {x|x ∈ adj(j), x > j};
    count_e ← |intersection(adj_1, adj_2)|;
    count ← count + count_e;
end
```

Par ailleurs, NodeIterator [Sch07] représenté dans l'algorithme 2 est mécanisme intéressant permettant d'énumérer tous les pairs de sommets (u, v) qui ont le même sommet adjacent w. Plus précisément, pour chaque sommet $w : u \in adj(w)$ et $v \in adj(w)$, si $(u, v) \in E$, alors (u, v, w) est un triangle. Le nombre total des triangles renvoyés par l'algorithme est divisé par 3 vu que chaque sommet du triangles est compté trois fois. La complexité de cette approche est $\Theta(n^3)$ pour les graphes avec une distribution fortement asymétrique. Un autre algorithme itératif est EdgeIteraor [AYZ97] qui permet d'énumérer tous les triangles d'un graphe, en itérant sur les arêtes afin d'en trouver celles qui contribuent à la formation de chaque triangle. Sa complexité est de $O(m \cdot d_{max})$ avec d_{max} est le diamètre du graphe.

Algorithm 2 Algorithme du node iterator

```
Input: G = (V, E)
Output: count/3
count ← 0;
for each node j ∈ V do
    count_j ← 0;
    for each pair of distinct neighbor i and h in adj(j) do
        if (i, h) ∈ E then
            count_j ← count_j + 1;
        end
        count ← count + count_j;
    end
end
```

Ces deux algorithmes nécessitent que tous les sommets soient présents sur la même machine, car ils itèrent sur l'ensemble des sommets ou des arêtes. Cependant, avec l'expansion de la taille des graphes, leur efficacité diminue, et le traitement en mémoire principale sur une seule machine devient tout simplement irréalisable. Il existe plusieurs variantes de ces deux algorithmes qui ont été développées pour résoudre le problème de la mémoire principale, telles que [Sch07] et [Lat08]. Tout d'abord, l'algorithme proposé par [Sch07] repose sur l'ordonnancement des sommets par degré croissant, ce qui facilite le traitement du graphe en suivant un ordre particulier.En revanche, l'algorithme décrit par [Lat08] commence par trier la liste d'adjacence du graphe, puis il utilise des itérations pour calculer l'intersection entre les ensembles d'adjacence des paires de sommets.

L'utilisation de techniques efficaces d'E/S est une autre solution pour aborder les limitations de la mémoire principale. Les frameworks MGT [HTC13], Pagh [PS14], PCF [CXL19] et Trigon

[CXCL20] améliorent les algorithmes d'énumération de triangles *EdgeIterator* [IR78] et *NodeIterator* [Sch07] en proposant de meilleures techniques d'E/S qui réduisent la surcharge sur la mémoire centrale. En effet, MGT [HTC13] s'exécute par itérations, où chacune exécutant deux étapes : (1) charger en mémoire centrale les arêtes cM du E^*, où $c < 1$ est une constante à définir et E^* est l'ensemble trié des arêtes, et (2) rapporter tous les triangles dont les arêtes sont en mémoire centrale. La complexité de cette approche est $O((|E|^2/(MB) + K/B)$ avec $M \geq 2B$ est la taille de la mémoire principale, B est la taille d'un bloque en mémoire, et K est le nombre de triangle dans le graphe introduit. Pagh et al. [PS14] ont étudié la complexité d'E/S du problème d'énumération de triangle en proposant un algorithme inconscient du cache avec une meilleure optimisation d'E/S que MGT [HTC13]. Ce nouvel algorithme possède une complexité de $O(E^{3/2}/\sqrt{MB})$, ce qui représente une amélioration d'un facteur de $min(\sqrt{E/M}, \sqrt{M})$. PCF [CXL19] prend en charge le fonctionnement des 18 algorithmes d'énumération de triangles, tout en réduisant considérablement les E/S. Après avoir trouvé le meilleur ordre de traversée des sommets, l'algorithme construit une implémentation autour de celui-ci en utilisant les instructions SIMD pour l'énumération des triangles. Selon les auteurs, cette approche est 5 à 10 fois plus rapide avec un ordre de grandeur d'E/S moins que les approches précédentes. Trigon [CXCL20] est basé sur Pagh [PS14] et PCF [CXL19]. Les auteurs ont analysé les asymptotiques des E/S dans Pagh et PCF, dans le but de comprendre leurs forces et leurs faiblesses. Ils ont ensuite proposé l'approche Trigon qui élimine les redondances dans l'importation des arêtes en mémoire centrale et exploite les leçons tirées à partir de l'analyse précédente pour donner un meilleur temps d'exécution tout en équilibrant le coût d'E/S et de CPU [9].
Toutes les approches centralisées sont résumées dans Table 2.1.

Tableau 2.1 – La complexité temporelle des méthodes centralisées.

Référence	Année	Complexité Temporelle
EdgeIterator [AYZ97]	1997	$O(m \cdot d_{max})$
NodeIterator [Sch07]	2007	$\Theta(n^3)$
Schank [Sch07]	2007	$O(m^{3/2})$
Latapy [Lat08]	2008	$O(m^{3/2})$
MGT [HTC13]	2013	$O((\|E\|^2/(MB) + K/B)$
Pagh [PS14]	2014	$O(E^{3/2}/\sqrt{MB})$
PCF [CXL19]	2019	Dépend de l'algorithme choisi
Trigon [CXCL20]	2020	Dépend de l'algorithme choisi

2.4.2 Méthodes parallèles et distribuées

Le calcul parallèle pour l'énumération des triangles est considéré comme terminé lorsque toutes les machines terminent leur traitement et produisent les triangles. Par conséquent, les méthodes de cette classe d'algorithmes utilisent l'une de ces trois stratégies pour assurer le parallélisme et la distribution du traitement : (1) l'utilisation des machines multicœurs ou le GPU, (2) l'exécution sur le framework MR, ou (3) l'exploitation du MPI.

2.4.2.1 Multicœurs et GPU

Dans cette classe d'algorithmes, nous résumons les travaux qui s'intéressent au calcul des triangles dans un environnement parallèle fondé sur une mémoire partagée. Nous présentons d'abord les techniques aptes à être exécutées sur un système multicœurs, ensuite nous listons

9. *Unité Centrale de Traitement*

quelques travaux de calcul de triangles sur les GPU, et nous clôturons par les approches hybrides qui exploite les CPU pour la gestion de la mémoire et les GPU pour le calcul parallèle des triangles.

Parmi les travaux basés sur les systèmes multicœurs, nous retrouvons celui de Shun et Tangwongsan [ST15] qui ont présenté une implémentation parallèle des algorithmes de calcul de triangles, entre autres les algorithmes de comptage exacte et approximatif, dans les graphes à grande échelle. Ces algorithmes sont compatibles avec le cache et faciles à implémenter dans un langage prenant en charge le parallélisme dynamique, tel que Cilk++ ou OpenMP. Les auteurs ont proposé deux types d'algorithmes pour le comptage exacte des triangles : (1) TC-merge qui s'agit d'un algorithme basé sur le tri puis la fusion de la liste d'adjacence du graphe avec une complexité de $O(E^{3/2})$, et (2) TC-hash ; un algorithme qui utilise une table de hachage pour accélérer l'accès aux arêtes avec une complexité de $O(VlogV + E\sqrt{E})$. Ainsi, ils ont fondé leur raisonnement sur le travail de [PT12] pour proposer un algorithme parallèle pour comptage approximatif des triangles avec une complexité de $O(E + (pE)^{3/2})$ où $0 < p < 1$ est un paramètre à définir.

Wolf et al. [WDB+17] ont implémente une approche parallèle basée sur l'algèbre linéaire pour le comptage des triangles et exécutée sur un environnement doté d'une architecture multicœurs utilisant *KokkosKernels* qui s'agit d'une bibliothèque de noyaux pour l'algèbre linéaire creuse, l'algèbre linéaire dense et les noyaux de graphes.

Ancy et al. [TSA+17] ont présenté des algorithmes parallèles de comptage de triangles, ces derniers prennent en charge les caractéristiques des architectures des processeurs modernes et les caractéristiques des grands graphes clairsemés (sparse graphs), pour un traitement efficace sur des systèmes multicœurs à mémoire partagée. Leurs expérimentations ont été effectuées sur des processeurs Intel Xeon. Ils ont montré une performance efficace et ils ont démontré que leurs algorithmes peuvent être généralisés à d'autres framework en les portant sur GraphMat [10].

Zhang et al. [ZSMF18] ont conduit une étude d'optimisation sur les algorithmes de comptage de triangles dans un système parallèle à mémoire partagée. Afin d'obtenir des performances optimales pour le comptage de triangles, les auteurs ont mené des explorations préliminaires pour savoir si la prise en compte des caractéristiques du système, ainsi que les propriétés du graphe d'entrée peuvent être efficaces pour le comptage de triangles sur des systèmes multicœurs à mémoire partagée.

Nous retrouvons également sous cette classes d'algorithmes, *SuiteSparse :GraphBLAS* [Dav18] qui est une implémentation complète basée sur le multithreading en utilisant OpenMP du standard *GraphBLAS*[11], elle définit un ensemble d'opérations matricielles éparses sur une algèbre étendue de semirings en utilisant une variété presque illimitée d'opérateurs et de types. Davis [Dav18] a présenté le comptage de triangles comme exemple d'application de son implémentation. Il a montré qu'une implémentation en langage C peut être plus rapide avec certains graphes. Selon Davis [Dav18], GraphBLAS peut servir de bibliothèque efficace, permettant aux utilisateurs finaux de développer un code simple mais performant. Une nouvelle version de *SuiteSparse*, introduite dans [Dav21], prend également en charge le traitement parallèle en utilisant les GPU.

Enfin, [YQZ+21] ont introduit DPTL ; un algorithme parallèle pour lister les triangles dans les graphes dynamiques. L'idée principale de cet algorithme est de rapporter les triangles mis à jour (triangles supprimés et nouveaux triangles) résultant du lot de mis à jour des arêtes du graphe. Ceci empêche une énumération complète des triangles du graphe dynamique en mettant l'accent sur les triangles mis à jour. Par conséquent, le temps du listage des triangles est réduit. [YQZ+21] ont également proposé DPTL+. Il s'agit d'une amélioration de l'algorithme DPTL avec une meilleure complexité théorique et performance pratique.

10. Un framework d'analyse de graphes qui utilise un frontend de programmation de sommet et un backend matriciel optimisé, afin de combler l'écart de performance de productivité. Il permet de relier les frameworks d'analyse de graphes et les codes d'optimisation de leurs algorithmes.

11. Un standard qui définit les opération des matrices et des vecteurs sur une algèbre étendue de semirings.

Quant aux approches basées sur le calcul en utilisant les GPU, [CRA13] est parmi les premiers travaux qui permettent le comptage des triangles en utilisant le langage CUDA sur un système à mémoire partagée doté de GPU. L'approche de [CRA13] permet de dériver les informations d'adjacence à partir du traitement de l'arbre de recherche en largeur (parcours en largeur) du graphe d'entrée, puis de les stocker en mémoire globale. Cette démarche permet au GPU de récupérer efficacement les informations d'adjacence du graphe et d'exécuter l'algorithme de comptage de triangle. Selon [CRA13], les expérimentations ont montré une accélération de 10× par rapport aux CPU.
Ensuite, [GYM14] ont proposé une approche scalable pour l'énumération des triangles sur le GPU, fondée sur une extension de l'algorithme Merge Path [12] [GMB12, OGM+12] prenant en charge l'intersection [13] nommé *Intersect Path* qui possède deux niveaux de parallélisme : (1) le premier niveau partitionne les sommets vers les multiprocesseurs de flux sur le GPU, et (2) le deuxième niveau est chargé de paralléliser le travail entre les processeurs de flux du GPU et manipuler les différentes tailles de blocs de données.
Les travaux dans [Pol16] ont proposé une approche parallèle pour compter les triangles dans les graphes à grande échelle en utilisant le langage CUDA GPU. Cette approche est basée sur deux étapes principales : (1) le prétraitement pour le partitionnement des données du graphes parallélisé par des opérations de somme et de tri de préfixes, ensuite (2) le comptage des triangles est parallélisé en attribuant un seul thread à chaque arête, où chaque thread calcule l'intersection des listes d'adjacence de manière séquentielle. La complexité de cette approche est $O(m)$ avec $m \gg nb\_coeurs$ et nb_coeurs est le nombre de cœurs du GPU.
Ensuite, Hu et al. [HKSH17] ont introduit *TriX* ; un framework scalable pour le comptage des triangles qui comprend une stratégie de partitionnement de graphe à deux dimensions et un algorithme d'intersection basé sur la recherche binaire conçu pour les GPU. Le partitionnement à deux dimensions (verticalement puis horizontalement) offre une répartition équilibrée de la charge entre plusieurs GPU. En revanche, l'intersection basée sur la recherche binaire se charge du parallélisme entre les GPU via l'ordonnancement et l'accès coalisé à la mémoire.
Un autre travail notable est celui de [HLH18], où une approche pour compter les triangles dans les graphes contenant plus de 10 milliards d'arêtes est présentée. Cette approche garantit également un équilibrage de charge entre les unités de traitement des GPU en tamponnant les données partitionnées dans la mémoire du CPU (au lieu d'un accès directe à la mémoire secondaire) pour un approvisionnement plus rapide.
En se basant sur le même principe, les auteurs de [GZY+19] ont conçu un système à préchargement qui tamponne à l'avance la liste voisine du sommet traité dans la mémoire partagée rapide, pour éviter une latence élevée d'accès à la mémoire aléatoire globale. La technique de réorganisation des graphes basée sur les degrés a été également adoptée tout en conservant une heuristique simple pour répartir uniformément la charge de travail lors du comptage des triangles sur un environnement à base de GPU.
Hoang et al. [HJC+19] ont décrit *DistTC* ; un framework pour le comptage des triangles dans les graphes à grande échelle sur les systèmes à base de multi-GPU. Les auteurs ont adopté le modèle de proxy pour partitionner le graphe et assurer un équilibrage de charge entre les GPU. Plus précisément, les arêtes sont réparties entre les machines hôtes et des copies en cache des sommets d'extrémité des arêtes appelés *proxyvertices* sont créées. Un proxy pour un sommet sera désigné comme proxy maître qui est responsable de la valeur canonique du sommet ; tous les autres proxys sont désignés comme proxys de miroir. La cohérence des calculs est maintenue via les mises à jour des miroirs communiquées au proxy maître qui les utilisera pour déterminer la valeur canonique du sommet. Cette valeur est ensuite communiquée à tous les proxys pour une utilisation dans le calcul local. Ce modèle permet l'existence d'une arête proxy entre deux

12. Cet algorithme réaffirme l'opération de fusion de deux tableaux triés A et B à considérer comme un parcours d'une grille à deux dimensions de taille $|A| \times |B|$.
13. L'intersection est utilisée pour décrire l'intersection des chemins et dans le contexte de l'algorithme d'intersection entre deux ensembles.

sommets proxys voisins dans le graphe d'origine se présentant sur la même machine en utilisant les miroirs, ce qui minimise la communication entre les hôtes.
Quelques travaux utilisant une collaboration en CPU et GPU ont également émergé visant à exploiter les CPU pour le partitionnement des données entre les différents unités des GPU. [DFN+17, VLPP17, MDQ+18] ont présenté des approches hybrides collaboratrices entre CPU et GPU.
Date et al. [DFN+17, MDQ+18] ont introduit une analyse comparative approfondie des différents schémas de gestion de la mémoire offerts par CUDA 8 et NVLink [14], qui peuvent être exploités pour résoudre les problèmes dans lesquels la capacité de mémoire limitée du GPU pose un défi, notamment dans les plates-formes informatiques traditionnelles.
Voegele et al. [VLPP17] ont utilisé des méthodes centrée-graphes au lieu des techniques basées sur l'algèbre matricielle pour compter les triangles dans les larges graphes. Ces méthodes se concentrent sur une implémentation de la méthode de Galois Lonestar [KBCP09] sur un système multi-cœurs et les méthodes Polak [Pol16] et IrGL [PP16] sur un système doté de GPU.
Table 2.2 résume les divers travaux parallèles menés dans un environnement multi-cœurs ou basé sur les GPU.

Tableau 2.2 – Résumé de travaux parallèles sur un système multi-cœurs ou GPU.

Référence	Année	Type de système
Shun et Tangwongsan [ST15]	2015	CPU (Cilk++ et OpenMP)
Wolf et al. [WDB+17]	2017	CPU (KokkosKernels)
Ancy et al. [TSA+17]	2017	CPU (GraphMat)
Zhang et al. [ZSMF18]	2018	CPU
Davis Timothy A. [Dav18]	2018	CPU (GraphBLAS)
Chatterjee et al. [CRA13]	2013	GPU (CUDA)
Green et al. [GYM14]	2014	GPU (CUDA)
Polak et al. [Pol16]	2016	GPU (CUDA)
Hu et al. [HKSH17]	2017	GPU (CUDA)
HuHang et al. [HLH18]	2018	GPU (CUDA)
Hoang et al. [HJC+19]	2019	GPU (CUDA)
Date et al. [DFN+17]	2017	Hybride (CPU et GPU)
Voegele et al. [VLPP17]	2017	Hybride (CPU et GPU)
Mailthody et al. [MDQ+18]	2018	Hybride (CPU et GPU)

2.4.2.2 MapReduce

Plusieurs travaux [Coh09, SV11, ASSU13, PC13, ZZQC17] ont adopté le framework MapReduce (MR) pour énumérer et compter les triangles dans les grands graphes. Ce framework est fondé sur le parallélisme et la distribution des calculs sur un ensemble de machines hôtes formant un cluster. Il permet de lancer les traitements sous forme de cycles (*rounds)* de *Map et Reduce* sur des données potentiellement volumineuses, tels que les réseaux et les larges graphes dépassant 1 Téraoctets.
Cohen [Coh09] a présenté une série d'algorithmes pour l'exploration des graphes en utilisant le framework MapReduce. Ces algorithmes sont développés avec les deux jobs Map et Reduce et renvoient les mêmes résultats que les algorithmes de graphe existants. Cohen [Coh09] a choisi une stratégie de partitionnement qui permet de créer pour chaque sommet un bac qui porte toutes ses arêtes sortantes, et son degré correspondant. Parmi ces algorithmes, Cohen [Coh09] a implémenté

14. Une interconnexion GPU à haut débit permettant d'accélérer le déplacement des données entre le GPU et le CPU.

le problème d'énumération de triangle en deux phases : (1) trouver tous les triades ouverts, et (2) vérifier pour chaque triade, s'il existe une arête reliant ses extrémités. Cette solution peut entraîner des redondances, c'est pourquoi l'ordre lexicographique entre les sommets est utilisé pour éliminer les doublons. Le seul problème possible avec cette approche est l'explosion quadratique qui pourrait résulter de l'épuisement des paires d'arêtes enregistrées dans un bac. Pour remédier à ce problème, Cohen [Coh09] a adopté une approche consistant à enregistrer chaque arête sous son sommet de bas degré correspondant, évitant ainsi la création de bacs disproportionnés par rapport aux autres. Ainsi, la mise à l'échelle quadratique ne devrait pas poser de problème dans la plupart des graphes.

Suri et Vassilvitskii [SV11] ont introduit deux algorithmes : (1) un algorithme séquentiel de comptage de triangles basé sur l'algorithme de *NodeIterator*, qu'ils ont adapté et implémenté sur la plateforme MapReduce. (2) Un algorithme parallèle sur MapReduce qui offre un compromis entre la mémoire disponible sur chaque machine et la mémoire générale disponible pour l'algorithme. Ces algorithmes ont une complexité de $O(m^{3/2})$ et génèrent deux cycles de MapReduce.

Afrati et al. [ASSU13] ont étudié la complexité du parallélisme et le coût de la communication dans le calcul MapReduce (MR), tout en introduisant un modèle de problèmes qui peuvent être résolus en un seul cycle de calcul de MapReduce. Ce modèle permet de découvrir les bornes inférieures du coût de la communication en fonction du nombre maximum d'entrées qui peuvent être attribuées à un Reducer. Trois problèmes ont été étudiés dans cette étude : (1) L'identification des paires de chaînes distantes en utilisant la distance de Hamming [15] [BKR02]. (2) L'identification des triangles et d'autres motifs dans les grands graphes. (3) La multiplication matricielle.

Les auteurs de [ASSU13] ont prouvé qu'en attribuant $\frac{n^2}{2}$ de sommets en entrée à chaque Reducer, ce dernier renvoie $\frac{n^3}{6}$ triangles en sortie. Cela assure une borne inférieure de $r = \frac{n}{\sqrt{2q}}$ et une borne supérieure de $g(r) = \frac{\sqrt{2}}{3}q^{3/2}$ pour le problème de détection des triangles, où q est la taille du Reducer et r est le taux de réplication.

Park et Chung [PC13] ont introduit une nouvelle méthode de partitionnement de graphe sur un ensemble de Reducers pour traiter le problème de détermination du nombre de triangles dans les graphes à grande échelle. Cette méthode est basée sur les trois types de triangles :

1. *Type-1* les trois les sommets se présentent sur la même partition,
2. *Type-2* deux sommets sont sur une partition et le troisième sommet est sur une autre partition,
3. *Type-3* les trois sommets sont sur des différentes partitions.

En utilisant cette stratégie, Park et Chung ont pu éliminer les redondances générées par la méthode de [SV11] et surpasser les performances de la méthode développée dans [Coh09].

Zhu et al. [ZZQC17] ont présenté FTL – un algorithme pour l'énumération des triangles en utilisant le framework MapReduce. Cet algorithme a pour objectif de réduire les données intermédiaires transférées pendant l'étape de mélange (shuffling) en introduisant une structure légère de données. Il permet également de charger les données sur plusieurs cycles, ce qui allège la charge sur la mémoire et la bande passante du réseau.

Bien que le traitement des graphes avec le framework MapReduce est scalable, il génère un volume de données intermédiaires. Cela peut influencer la précision des résultats ou les performances de l'algorithme utilisé [Coh09, SV11, ASSU13, PC13, ZZQC17].

Un résumé des travaux suscités avec leurs caractéristiques est présenté dans Table 2.3.

15. Cette distance permet de calculer le degré de correspondance entre deux chaînes ou deux sommets.

TABLEAU 2.3 – Résumé de travaux parallèle sur le framework MapReduce.

Référence	Année	Nombre de cycle Map Reduce
Cohen [Coh09]	2009	2 cycles (map et reduce)
Suri et Vassilvitskii [SV11]	2011	2 cycles (map et reduce)
Afrati et al. [ASSU13]	2012	1 cycle (map et reduce)
Park et Chung [PC13]	2013	2 cycles (map et reduce)
Zhu et al. [ZZQC17]	2017	Multiples cycles en fonction de la mémoire disponible

2.4.2.3 MPI

L'interface de passage de messages (MPI) [GGH+96] est un standard pour la transmission de messages lors du calcul parallèle. L'architecture qui adopte le MPI est basée sur une communication directe et bidirectionnelle entre les différentes machines hôtes. Cela permet d'exécuter les opérations de la communication point à point ou collective de MPI.
Plusieurs travaux de recherche se sont intéressés à ce standard pour présenter une solution efficace au problème d'énumération de triangles. Parmi eux, Arifuzzaman et al. [AKM13, AKM15b, AKM15a] qui ont proposé une approche parallèle distribuée pour l'énumération et le comptage des triangles en utilisant la norme MPI. Cette approche suppose que la taille de la mémoire de chaque machine hôte est suffisamment grande pour accueillir l'intégralité du graphe cible.
Cette approche est fondée sur l'algorithme *NodeIterator* tout en considérant une phase de prétraitement permettant de partitionner équitablement le graphe introduit sur l'ensemble des nœuds de calcul. Dans [AKM13], les auteurs ont proposé une méthode de comptage approximatif de triangles basée sur le partitionnement statique du graphe. Ce partitionnement consiste principalement à diviser l'ensemble des sommets V en p sous-ensembles de données disjoints. Chaque sous-ensemble, ainsi que les voisins de ses sommets sont attribués à l'un des processeurs parmi les p processeurs (machines hôtes) du cluster.
En outre, les auteurs ont adopté l'algorithme Doulion [TKMF09] de approximation parallèle fondé sur la sparsification [16] permettant d'approximer le nombre de triangles lorsque le graphe est très large.
En revanche, l'algorithme présenté dans [AKM15b, AKM15a] permet de compter le nombre exact des triangles dans un graphe. Il est fondé sur une stratégie de partitionnement dynamique qui permet d'attribuer des tâches atomiques aux nœuds de calcul d'une manière dynamique. Concrètement, lorsqu'un nœud de calcul termine sa tâche, le coordinateur lui attribue une nouvelle tâche, assurant ainsi un équilibrage de charge dynamique entre les processeurs.
Ainsi, Azad et al. [ABG15] ont proposé un algorithme pour l'énumération et le comptage exacte des triangles en utilisant l'algèbre matricielle. Cet algorithme est inspiré de celui proposé dans [Coh09], où les lignes de la matrice d'adjacence $\mathcal{A}$ du graphe cible sont ordonnées sur la base des valeurs non nulles qu'elles contiennent. La matrice $\mathcal{A}$ est ensuite divisée en une partie triangulaire inférieure et une partie triangulaire supérieure via $\mathcal{A} = L + U$. La partie triangulaire inférieure contient les arêtes (i, j) où $i > j$, tandis que la partie triangulaire supérieure contient les arêtes (i, j) où $i < j$. La multiplication entre L et U produit l'ensemble des triades ouverts (i, j, k) où j est le sommet de plus petit degré. Enfin, le nombre de triangle est calculé en faisant une multiplication d'élément par élément avec la matrice d'adjacence via $\mathcal{C} = \mathcal{A}. * \mathcal{B}$.
Afin d'assurer le parallélisme, la matrice U est partitionnée sur l'ensemble des processeurs tandis que la matrice L est répliquée. Ceci est justifié par le fait que chaque processeur n'a besoin d'une partie de U pour calculer $\mathcal{B}$, en revanche, il peut avoir besoin de toute la matrice L dans le plus pire des cas (un graphe très dense). Azad et al. [ABG15] ont implémenté leur algorithme en

16. La sparsification est une technique d'échantillonnage de graphe dans laquelle certaines arêtes choisies au hasard sont conservées et les autres sont supprimées, puis le calcul est effectué dans le graphe fragmenté.

utilisant le langage de programmation C++ et le standard MPI.
Pandurangan et al. [PRS18b] ont étudié la complexité et le coût de communication pour le traitement distribué de deux problèmes de graphe : (1) l'énumération de triangle, et (2) le PageRank. Leur étude a été conduite sur le modèle de $k - machines$ (voir Sect. 3.3.1). Les auteurs ont prouvé que cet algorithme assure un équilibrage de charge entre les machines du modèle. *Nous nous sommes basés sur ce travail pour élaborer notre contribution* (plus de détails sont donné dans Partie II).
Acer et al. [AYR+19] ont proposé une approche hybride scalable pour le comptage des triangles en utilisant une formulation avec l'algèbre linéaire. Ils ont opté pour une stratégie de traitement fondé sur une entraide entre la mémoire distribuée et la mémoire partagée en utilisant MPI et Cilk [17] respectivement. Leur stratégie de partitionnement est composée de deux niveaux : (1) Un partitionnement cartésien bidimensionnel (2D) entre les processeurs MPI. (2) Un partitionnement en ligne unidimensionnel (1D) est utilisé dans les blocs cartésiens pour le parallélisme de mémoire partagée à l'aide du modèle de programmation Cilk. Ainsi, afin de compter le nombre des triangles dans le graphe introduit, Acer et al. [AYR+19] ont utilisé une multiplication matrice-matrice clairsemée ($L \times L$) suivie d'une multiplication d'élément par élément ($L \times L$). $*$ $L.$, où la somme des entrées dans la matrice résultante représente le nombre de triangles.
Zhang et al. [ZJW+19] ont conçu LiteTE ; un framework pour l'énumération de triangles dans les systèmes à mémoire distribuée. L'idée principale de leur approche se résume en trois étapes :

1. Diviser le graphe cible en grandes partitions et copier ces dernières sur plusieurs nœuds nommés groupe de nœuds. Ceci est possible en exploitant la grande mémoire des serveurs modernes et de la bande passante élevée des réseaux modernes.
2. Fusionner les messages pour réduire davantage le temps d'échange des messages et masquer la communication.
3. Utiliser l'algorithme FastBC [18] qui remplace les fonctions de communication de MPI et exploite efficacement la bande passante bidirectionnelle des câbles et la bande passante globale des clusters.

L'énumération des triangles sur chaque processeur est effectuée en utilisant une intersection entre l'ensemble des voisins des sommets dans la partition correspondante. Si deux sommets sont hébergés par deux partitions différentes, un message pour copier une des partitions est envoyé, et une copie de cette partition sera hébergée par le processeur correspondant.
Enfin, Yan et al. [YGRC+20] ont présenté G-Thinker. Il s'agit un moteur d'exploration de graphe basé sur les tâches. Il est capable de maintenir une utilisation complète des processeurs (CPUs) dans un cluster, même avec une limitation budgétaire de la consommation de mémoire. De plus, il offre une API centrée sur les sous-graphes qui est naturelle pour la rédaction d'algorithmes d'exploration de graphes. Il est implémenté en MPI et exécuté sur un environnement Hadoop qui permet d'assurer la gestion des tâches générées. D'après la littérature, ce système est très efficace, c'est pourquoi nous l'avons choisi pour le comparer avec notre solution (Plus de détails sont discutés dans le Chapitre 5).
L'ensemble de travaux basés sur MPI est résumé dans la Table 2.4.

2.4.3 Discussion

Tous les algorithmes présentés ci-dessus nécessitent un effort considérable et des codes compliqués en langages de programmation traditionnel (e.g., C/C++ ou Python) pour énumérer et compter les triangles. Table 2.5 regroupe pour chaque catégorie d'algorithmes, ses avantages et ses défis : Les méthodes centralisées [IR78, Sch07, AYZ97, Lat08] permettent de traiter des petits graphes dans la mémoire d'une seule machine. Elles sont rapides mais elles rencontrent des problèmes

17. Un langage de programmation étendu de C/C++ utilisé pour le calcul parallèle multithread.
18. Un algorithme proposé par Zhang et al. [ZJW+19] afin d'accélérer la communication entre les processeurs.

Tableau 2.4 – Résumé de travaux parallèle sur MPI

Référence	Année	Technique
Arifuzzaman et al. [AKM13]	2013	Partitionnement statique avec sparsification
Arifuzzaman et al. [AKM15a, AKM15b]	2015	Partitionnement dynamique
Azad et al. [ABG15]	2015	Algèbre matricielle
Pandurangan et al. [PRS18b]	2018	Partitionnement randomisé
Acer et al. [AYR+19]	2019	Traitement hybride (MPI + Cilk)
Zhang et al. [ZJW+19]	2019	Traitement à performance élevée
Yan et al. [YGRC+20]	2020	Traitement à base de tâche (MPI et Hadoop)

Tableau 2.5 – Comparaison entre les approches d'énumération de triangles.

Méthodes	Avantages	Défis
Centralisées	Traitement rapide des petits graphes	Passage à l'échelle Gestion des E/S
Multicœurs	Amélioration des performances	Processeurs performants Gestion des threads Équilibrage de charge Gestion mémoire
GPU	Accélération du temps d'exécution	Matériels performants Gestion mémoire CPU et GPU Coût
MapReduce	Données traitées en mémoire Tolérance aux erreurs	Des données intermédiaires Temps d'exécution élevé Moins flexible
MPI	Distribution des calculs Portabilité	Équilibrage de charges Qualité du réseau

liés au passage à l'échelle. Certains travaux [HTC13, PS14, CXL19, CXCL20] permettent de traiter des grands graphes indépendamment de la taille de la mémoire centrale en proposant des techniques efficaces d'échange de données entre le disque et la mémoire centrale. Ces méthodes nécessitent une gestion complexe des E/S et des codes qui peuvent être de bas niveau. Les méthodes multicœurs [ST15, WDB+17, TSA+17, ZSMF18, Dav18] ont largement amélioré les performances en utilisant la mémoire partagée et en exécutant l'algorithme d'énumération ou du comptage de triangles en parallèle à l'aide des threads. Ces méthodes exploitent des processeurs muticœurs qui doivent être performants, afin d'accélérer les tâches et les processus de l'algorithme suscité. La gestion des threads interviennent fortement dans ce modèle où un équilibrage de charge entre les threads doit être mis en place, afin de s'assurer que tous les processus se terminent simultanément. Enfin, les méthodes multicœurs nécessitent une gestion des données dans la mémoire partagée, pour partager les données équitablement entre les threads. Les techniques utilisant les GPU sont très rapides et permettent la gestion de très grands réseaux, néanmoins elles sont coûteuses vu le prix élevé des GPU. En outre, ces méthodes nécessitent une gestion efficace entre la mémoire centrale et la mémoire du GPU. Les méthodes basées sur MapReduce (MR), quant à elles, traitent les données en mémoire et permettent une tolérance aux erreurs, ce qui induit à une gestion moins compliquée de mémoire. Cependant, ces méthodes génèrent beaucoup de données intermédiaires qui peuvent impacter le temps d'exécution des tâches MapReduce (MR).

Enfin, les techniques basées sur MPI distribuent les traitements sur un ensemble de hôtes qui ne partagent pas de mémoire. Ce modèle permet la probabilité puisqu'il est défini sur plusieurs langages de programmation. Néanmoins, l'efficacité de ces techniques est fortement dépendante de l'équilibrage de charge entre les processeurs ainsi que le réseau mis en place entre les machines hôtes pour assurer l'échange de messages.

2.5 Conclusion

Nous avons présenté, à travers ce chapitre, un aperçu sur l'état de l'art d'énumération et du comptage de triangles dans les graphes. Nous avons commencé par présenter les moteurs et les systèmes dédiés aux traitements des graphes. Nous avons ensuite lister les différents travaux d'exploration des graphes dans les bases de données relationnelles. Pour cela, nous avons défini deux catégories : (1) les approches internes aux SGBD qui traduisent les algorithmes d'exploration des graphes en langage SQL, et (2) les travaux externes aux SGBD qui exploitent les données présentes sur les bases de données relationnelles en ajoutant une couche externe au SGBD, permettant le traitement du graphe. Enfin, nous avons présenté les travaux d'énumération et de comptage de triangles dans les graphes.

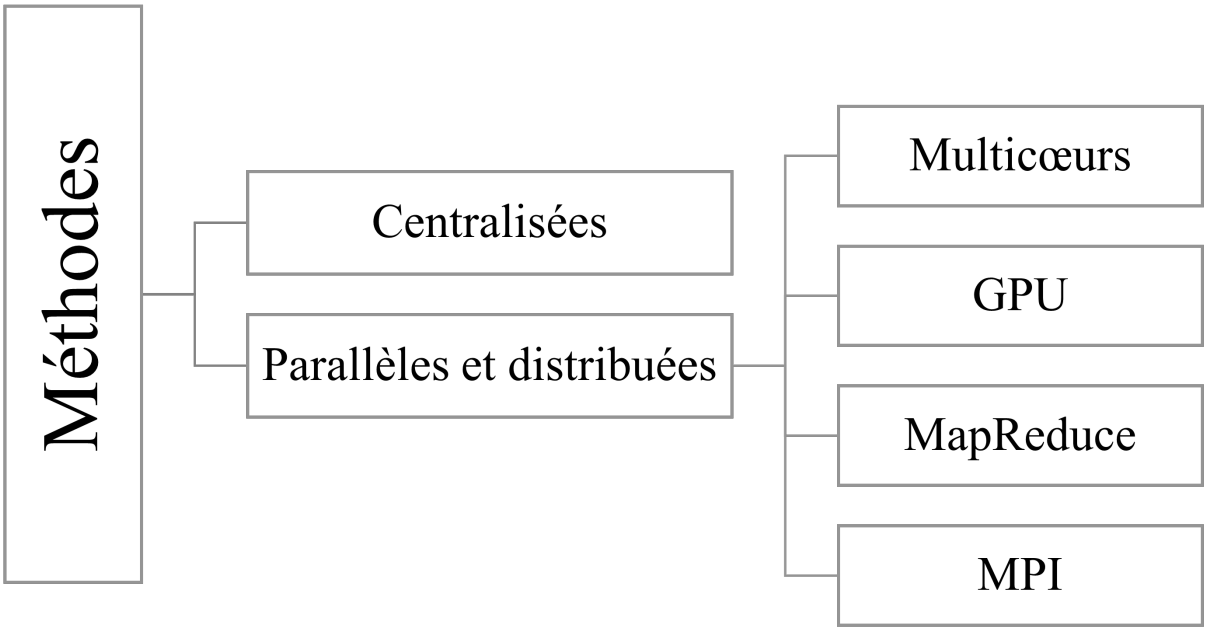


FIGURE 2.1 – Classification des travaux de calcul des triangles.

Nous nous sommes focalisés sur deux grandes familles d'algorithmes/approches : (1) les méthodes centralisées aptes à être exécutées sur une seule machine, et (2) les méthodes parallèles et distribuées qui peuvent être exécutées sur un cluster de machines. Cette dernière a été divisée en sous-classes : (i) les méthodes multicœurs et GPU qui utilisent une mémoire partagée, (ii) les méthodes basées sur MapReduce (MR), et (iii) les méthodes basées sur le passage de message MPI définies sur une architecture share-nothing. Figure 2.1 résume la classification des approches d'énumération et de comptage des triangles. Notre contribution principale consiste à exploiter les bases de données relationnelles pour explorer les graphes à grande échelle.Nous estimons donc qu'il est nécessaire de se positionner par rapport à la classification de Figure 2.1. Pour cela, nous présentons dans la Table 2.6, les avantages et les inconvénients des méthodes d'exploration de graphes avec les SGBD relationnelles et avec les langages de programmation (Lang. de prog.) traditionnels comme C/C++ ou Python (utilisés dans le cadre d'un système de gestion de graphe comme Spark GraphX ou non).

Tableau 2.6 – Comparaison entre les SGBD et les langages de programmation

Modèle	Avantages	Défis
SGBD	Code élégant, court et abstrait Sécurité Tolérance aux erreurs Gestion des concurrences Import rapide des données	Passage à l'échelle Performance Coût
Lang. de prog.	Performance Portabilité Coût	Code compliqué et très long Gestion des ressources Sécurité Tolérance aux erreurs

De manière générale, les SGBDR offrent des solutions courtes et simples à comprendre grâce au langage de requêtes standard SQL. En outres, les SGBDR sont dotés de plusieurs capacités telles que : (1) la sécurité assurée par un système de gestion des utilisateurs, de profils et de rôle, (2) la tolérance aux erreurs qui représente la capacité d'un système à remplir sa fonction en dépit des fautes [ALRL04], (3) la gestion des concurrences entre les transactions, et (4) une importation rapide des données dans la base de données. Cependant, les SGBDR particulièrement distribués souffrent d'une difficulté pour le passage à échelle, à cause de l'échange de données entre les hôtes du cluster lors de l'exécution des différentes requêtes, précisément celles contenant des jointures. Ceci peut influencer la performance et augmenter le temps d'exécution. Un autre défi qui peut être manifesté est que certains SGBD proposent des licences payantes ce qui rend leur utilisation coûteuse. Néanmoins, la majorité de ces systèmes offrent des versions à libre distribution(community) pour une utilisation limités de leurs capacités, comme il existe plusieurs SGBD sous licence GPL [19] gratuite. En revanche, les langages de programmation présentent une meilleure performance, car ils s'exécutent dans la mémoire principale. La plus part des langages les plus utilisés offrent également une probabilité entre les systèmes d'exploitation avec une utilisation gratuite. Toutefois, ces langages génèrent de longs codes compliqués à comprendre où la gestion des ressources doit être pris en charge par le développeur. Contrairement aux SGBDR, les langages de programmation ne définissent pas la notion de sécurité par défaut et les données peuvent être accessible via le code. Ainsi, ils ne permettent pas une tolérance aux fautes, une simple erreur syntaxique ou sémantique peut conduire à l'arrêt de l'exécution ou à des résultats erronés.
Dans la partie suivante, nous allons présenter notre contribution qui consiste en une solution scalable pour l'énumération des triangles dans les graphes à grande échelle en utilisant les requêtes SQL. Nous allons prouver qu'avec une bonne stratégie de partitionnement, nous pouvons garantir qu'une solution à base de SQL peut passer à l'échelle et être presque efficace qu'une solution développée en langage de programmation comme C/C++ ou Python.

19. *Licence Publique Générale*

Deuxième partie

Contributions

Chapitre 3

Algorithme adapté pour l'énumération équilibrée des triangles dans les graphes à grande échelle

I never guess. It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.

— Arthur Conan Doyle

Sommaire

3.1 Introduction

Avec l'avancement technologique et l'explosion des réseaux sociaux et des villes intelligentes, le monde est devenu fortement interconnecté. L'analyse de ces connexions nécessite leurs modélisations par des structures adéquates qui sont les graphes. L'analyse des graphes permet d'extraire des connaissances intéressantes sur la nature des relations et leur profondeur. Ceci est possible en extrayant les motifs intégrés, entre autres, les triangles. Le problème *d'énumération et de comptage des triangles* consiste à lister et à compter (respectivement) tous les triades fermés d'arêtes dans le graphe. L'énumération et le comptage des triangles interviennent dans plusieurs applications telles que : (1) l'analyse des processus sociaux dans les réseaux [WS98a], (2) l'extraction de sous-graphes denses [WZTT10], (3) les jointures dans les bases de données [NRR13], ainsi d'autres applications qui sont largement commentées dans [CC12, BFN$^+$15].
Nous avons présenté dans le chapitre précédent un état de l'art sur les travaux utilisant les requêtes SQL pour étudier et analyser les graphes. Ces travaux sont motivés par la volonté de plusieurs chercheurs défendeurs du principe "Inside DBMSs" [1] [Ord13, ZY17] de modéliser les données déjà hébergées sur les bases de données relationnelles par des graphes pour éviter leur exportation, en menant le traitement et l'analyse avec les requêtes SQL. De plus, il existe des librairies d'analyse de données dans l'environnement de Python, qui permettent l'analyse rapide et efficace des différents types de problèmes de science de données y compris l'analyse et l'exploration des graphes. Par exemple, la librairie Pandas permet d'utiliser le DataFrame. Il s'agit d'une structure de données optimisée pour modéliser les ensembles de données et d'y effectuer le traitement nécessaires, en utilisant des opérations similaires aux SQL. Ces librairies suivent le principe de "Outside DBMSs", qui consiste en l'exportation des données des SGBD, puis leur traitement avec un langage impératif comme Python.
Partant du même principe, nous exposons dans ce chapitre notre première contribution, qui se matérialise sous la forme d'un algorithme adapté, optimisé, et inspiré de l'algorithme randomisé parallèle [PRS18b]. Notre objectif est de proposer une solution au problème d'énumération et de comptage de triangles. Cette contribution suit les deux paradigmes "Inside DBMSs" (avec les requêtes SQL) et "Outside DBMSs" (en utilisant Pandas). Notre solution est déployée dans un environnement distribué.
Dans ce chapitre, nous abordons plusieurs aspects essentiels. Tout d'abord, nous définissons le problème d'énumération et de comptage de triangles. Ensuite, nous examinons le partitionnement des données dans un environnement parallèle et distribué. Nous présentons en détail l'algorithme randomisé que nous utilisons pour l'énumération et le comptage des triangles. Par la suite, nous expliquons en détail l'algorithme adapté, en mettant en lumière son utilisation des requêtes SQL et Pandas, tout en décrivant les techniques d'optimisation mises en œuvre. Une étude de complexité, d'équilibrage de charge et de justesse de notre algorithme est également fournie. Enfin, nous concluons ce chapitre en résumant les enseignements tirés de ce travail. Pour plus de clarté, nous invitons les lecteurs à se référer à l'annexe A, où nous avons résumé les notations nécessaires à la compréhension de ce chapitre.

3.2 Définition du problème d'énumération des triangles

L'énumération et le comptage des structures typologiques locales, tels que les triangles, jouent un rôle primordial dans l'analyse et l'exploration des graphes à grande échelle. En effet, l'énumération et le comptage des triangles sont parmi les problèmes fondamentaux dans le domaine de l'analyse des graphes.

Definition 3.1 *Le problème d'énumération de triangles consiste à répertorier tous les triplets de*

1. Il signifie que toutes les opérations sur une base de données doivent se faire à l'intérieur du SGBD hébergeant la donnée.

sommets ou d'arêtes qui sont mutuellement adjacents. En revanche, le comptage des triangles vise à déterminer le nombre total de ces triplets fermés.

Il est essentiel de noter que la *détection et le comptage* des triangles sont plus simples par rapport à leur *énumération*, qui revêt une importance cruciale pour de nombreuses applications. En effet, l'énumération teste la possibilité de former un triangle à partir des triplets d'arêtes ou de sommets. Ainsi 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.
Propriété 3.1 : Le problème d'énumération et de comptage de triangles est définis sur les graphes non orienté.
L'énumération de triangle ne considère pas l'orientation du graphe. Elle liste toutes les arêtes entre les triplets de sommets adjacents l'un à l'autre. D'une manière générale, ce problème est appliqué sur les graphes en ignorant leurs orientations.
Il existe plusieurs méthodes pour récupérer la liste des triangles dans un graphe. Nous avons mentionné dans la section 2.4 les algorithmes les plus utilisés (EdgeIterator [IR78], NodeIterator [Sch07], Cohen [Coh09]). Dans notre contribution, nous nous sommes focalisés sur la méthode de Cohen [Coh09]. Notre choix est justifié par le fait que cette méthode peut être facilement parallélisée sur tout système distribué ou parallèle. Nous rappelons que cette méthode d'énumération des triangles se déroule en deux étapes distinctes :

(1) Détection de tous les triades ouverts. i.e. les chemins dont la longueur est égale à deux,
(2) Pour chaque triade ouvert, l'algorithme teste l'existence d'une arête permettant de former un triangle.

Figure 3.1 illustre le fonctionnement de la méthode de Cohen [Coh09].

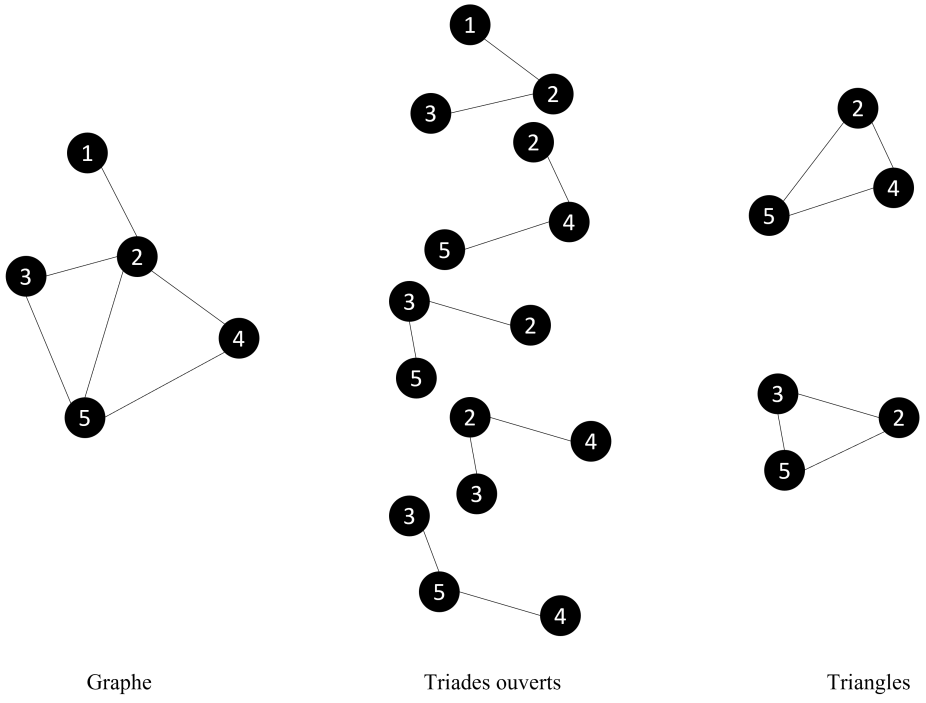


FIGURE 3.1 – Énumération de triangles avec la méthode de Cohen.

3.3 Partitionnement des données

L'objectif ultime du partitionnement des données dans l'analyse des graphes est de minimiser la communication entre les processeurs du cluster durant les tâches d'analyse, afin d'assurer une

distribution équilibrée des données du graphe. Ceci permet de garantir une répartition équitable de la charge de travail entre les processeurs.

3.3.1 Modèle de calcul parallèle

3.3.1.1 Définition du modèle

Nous définissons dans la présente section le modèle de calcul parallèle, que nous utilisions pour l'exécution de notre algorithme parallèle d'énumération et de comptage de triangles, dans les graphes à grande échelle.
Dans cette section, nous définissons le modèle de calcul parallèle que nous utilisons pour l'exécution de notre algorithme d'énumération et de comptage de triangles dans les larges graphes.
Nous avons opté pour le modèle de $k-machines$ (également connu sous le modèle Big Data) initialement introduit dans [KNPR15] et approfondi dans [FQK+15, CS16, PRS16, BIPP18, PRS18a]. D'une manière générale, le modèle consiste d'un ensemble $k \geq 2$ de machines $\{M_1, M_2, .., M_k\}$ construit sur une architecture share-nothing. En d'autres termes, chaque machine peut communiquer de manière bidirectionnelle et directe avec les autres machines par passage de message (les machines ne partagent aucune ressource de mémoire ou de traitement et n'ont pas d'autre moyen de communication à part le passage de message). Chaque machine exécute une instance de l'algorithme distribué. Figure 3.2 illustre une vue générale du modèle de $k-machines$.

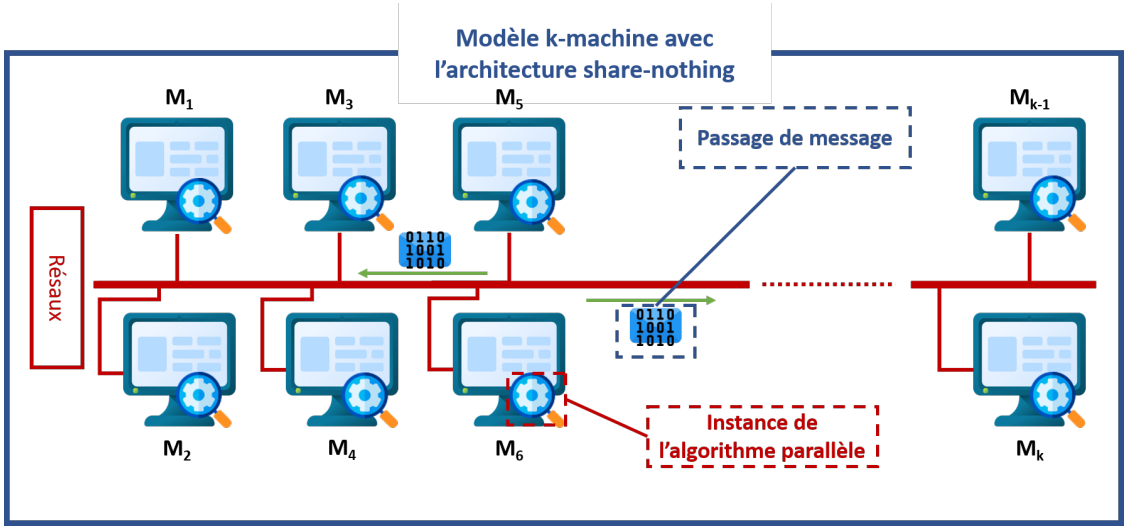


FIGURE 3.2 – Modèle de $k-machines$

Chaque connexion entre deux machines possède une bande passante (cette restriction peut être interprétée différemment. Au lieu de limiter la bande passante, nous pouvons limiter le volume de données communiqué (envoyé et reçu) par chaque machine à chaque cycle. Cette nouvelle restriction peut être facilement réécrite par le paramètre B [PRS18b]) de $B = \Theta(polylog\ n)$ bits par cycle [PRS18b]. Dans un traitement parallèle et distribué, les traitements locaux sur une machine sont supposés instantanés avec un coût presque nul, en revanche les échanges de messages entre les machines sont les opérations coûteuses. Néanmoins, pour le problème de l'énumération et du comptage des triangles dans les graphes à grande échelle, chaque machine effectue un traitement léger (ces traitements sont bornés par un polynôme éventuellement linéaire de la taille des données introduits sur cette machine) à chaque cycle [PRS18b].
En général, le modèle de $k-machines$ est un modèle adapté au traitement distribué de problèmes à grande échelle. Dans le cadre de cette thèse, nous utilisons le modèle de $k-machines$ pour aborder un problème d'analyse de graphes : l'énumération et le comptage des triangles. Plus

précisément, pour un graphe G défini par m arrêtes et n sommets où chaque sommet possède un identifiant parmi $[n]$. Nous supposons que $n \geq k$ (pratiquement $n \gg k$) afin d'éviter la trivialité.

3.3.2 Partitionnement randomisé des sommets

Pour le problème des triangles, le partitionnement basé sur la distribution équitable des données n'est pas suffisant. Ce dernier peut augmenter la fréquence d'échange de données entre les processeurs. Afin d'éliminer ce scénario, la distribution des données doit respecter la règle suivante :

"Chaque sommet et ses voisins doivent être envoyé au même processeur/machine."

Rappelons que nous utilisons un modèle $k - machines$, où les machines ne peuvent pas communiquer que via un passage de message bidirectionnel entre elles. Dans ce modèle, l'application de la règle suscitée permet de garantir un traitement local et une analyse de graphe entièrement distribuée. Un modèle de partitionnement aléatoire de sommets (RVP) est alors utilisé afin d'aboutir à ce partitionnement idéal de sommets entre les machines. Le modèle RVP est basé principalement sur l'approche par coupe d'arêtes, où chaque sommet avec ses arrêtes incidentes sont envoyés aléatoirement à une machine parmi les $k - machines$ [2].
Formellement, dans le modèle RVP, chaque sommet $i \in V$ du graphe G est attribué uniformément et aléatoirement à une machine M. La machine M est appelée la machine hôte du sommet i, et avec un léger abus de notation nous écrivons $i \in M$. Lorsque un sommet i est envoyé à une machine M, toutes ses arrêtes incidentes seront connues par sa machine hôte M (la machine hôte M peut connaître les identifiants des voisins de ses sommets hébergés et éventuellement leurs machines hôtes). Ainsi, en appliquant le modèle RVP, le nombre de sommets par machine est **équilibré**, i.e. chaque machine héberge $O(n/k)$ de sommets avec une forte probabilité. Une implémentation convenable pour le modèle RVP sera d'utiliser une fonction de hachage, qui permet d'attribuer un sommet avec ses arrêtes incidentes d'une manière déterministe à une machine. Ainsi, si une machine détermine l'identifiant du sommet, elle peut savoir sa machine hôte. Nous schématisons dans Figure 3.3, un aperçu du partitionnement des données basé sur le modèle RVP, où le graphe constitué de 11 sommets dans (a) est partitionné sur les 8 machines. Chaque sommet (en couleur rouge) et ses voisins sont envoyés à une machine parmi le modèle de $k - machines$ ($k = 8$).
Nous expliquons dans la section suivante comment ses spécifications et ses règles peuvent être satisfaites, en appliquant l'algorithme randomise pour l'énumération et le comptage des triangles.

2. Le modèle RVP est utilisé par de nombreux système tels que Pregel [MAB+10] et Giraph[sf, CEK+15] pour assurer le partitionnement du graphe cible sur l'ensemble des machines.

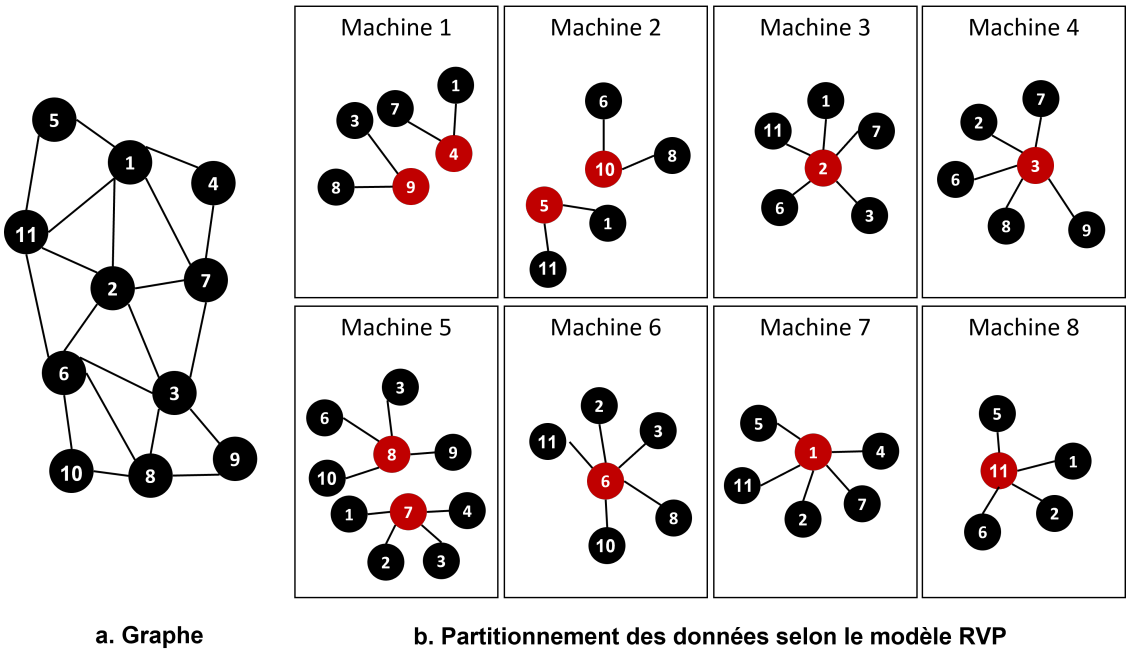


FIGURE 3.3 – Modèle RVP

3.4 Algorithme randomisé pour l'énumération des triangles

L'algorithme randomisé pour l'énumération et le comptage des triangles proposé par Pandurangan et al. [PRS18b] est une généralisation de l'algorithme *TriPartition* de Dolev et al. [DLP12] pour le modèle de congestion de clique avec quelques modifications cruciales présentées ci-après. Le principe général sur lequel se base l'algorithme randomisé est de partitionner l'ensemble des sommets V équitablement en sous-ensembles. Puis, il permet à chaque machine du modèle $k - machines$ d'examiner les arêtes, dont les extrémités sont dans deux sous-ensembles parmi les trois sous-ensembles affectés à la machine. En tout, Il existe k triplets de sous-ensembles, où chaque triplet est envoyé à une et une seule machine parmi les $k - machines$ du modèle. Nous allons maintenant décrire les différentes étapes de l'algorithme, en utilisant des exemples pour illustrer chaque étape.

3.4.1 Partitionnement du graphe

Dans l'algorithme randomisé, cette étape revêt une importance capitale, car elle élimine la nécessité de communication entre les machines pendant la phase d'énumération. Elle se compose de plusieurs sous-étapes :

3.4.1.1 Partitionnement des sommets

Dans cette étape, chaque sommet $v \in V$ du graphe G choisit indépendamment et aléatoirement une unique couleur parmi un ensemble C de $k^{1/3}$ de couleurs distinctes, en utilisant une fonction de hachage $h : V \to C$ initialement connue par toutes les machines. Figure 3.4 illustre un exemple de partitionnement des sommets du graphe représenté dans Figure 3.3 (a). Nous considérons dans cet exemple que le modèle $k - machines$ est constitué de 8 machines. Par conséquent, nous générons un ensemble C de deux couleurs : $C = \{Bleu, Rouge\}$.
En particulier, le nombre de machines dans le modèle $k - machines$ dépend du nombre de partitions de V à générer. Nous pouvons exprimer cela à l'aide de l'équation 3.1, où c représente la taille de l'ensemble C (c'est-à-dire le nombre de couleurs) :

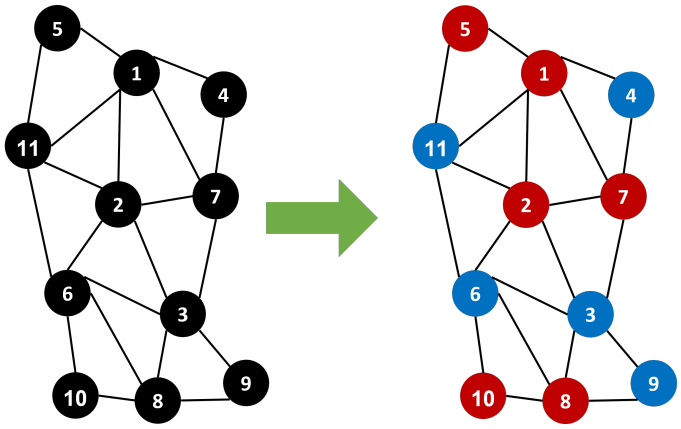


FIGURE 3.4 – Partitionnement de l'ensemble des sommets.

$$k = c^3 \ / \ c \in \mathbb{N} \tag{3.1}$$

3.4.1.2 Attribution des triplets de couleurs aux machines

Une attribution déterministe [3] des triplets de couleurs est effectuée. Elle permet d'attribuer un triplet parmi les k triplets de couleurs à chaque machine. Les triplets de couleurs sont constitués des différentes permutations des couleurs dans l'ensemble C. Figure 3.5 représente une attribution de triplet de couleurs sur un modèle constitué de 8 machines, où $C = \{Bleu, Rouge\}$.

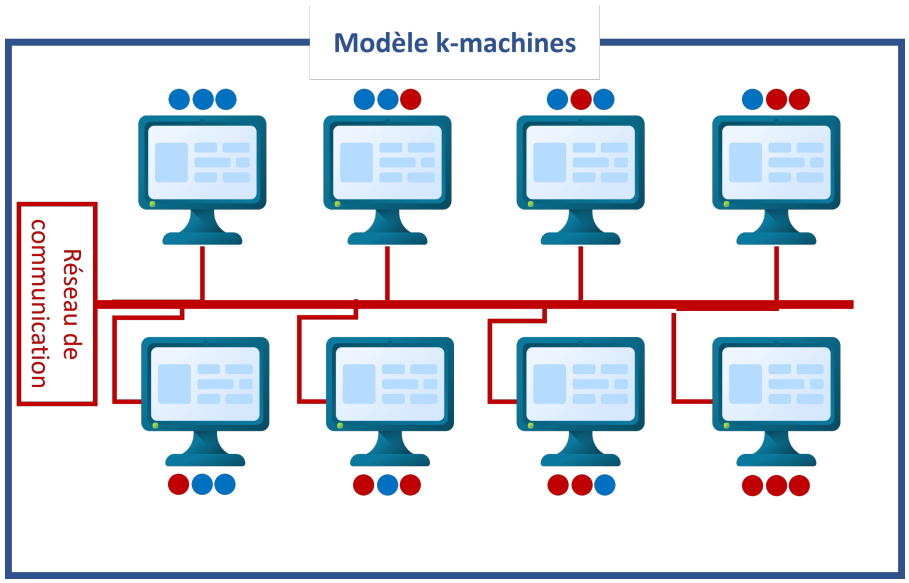


FIGURE 3.5 – Attribution des triplets de couleurs.

3.4.1.3 Répartition des arêtes

Après avoir attribué à chaque sommet une couleur aléatoire unique, les arêtes entre les différents sommets sont envoyées aléatoirement à des machines proxys. Autrement dit, pour chaque arête qu'elle détient, chaque machine désigne aléatoirement une machine parmi les $k-machines$ comme une machine proxy. Ensuite, elle envoie toutes ses arêtes aux machines proxys correspondantes. Dans la théorie de l'algorithme, il est crucial de respecter la règle d'attribution de proxy

3. Une attribution manuelle dans l'algorithme.

afin d'éviter une congestion excessive vers une seule machine : une machine qui possède un sommet v dont le degré est au moins $2k \log n$ demande à toutes les machines de la designer comme la machine proxy, pour chaque arête incidente au sommet v. Si deux machines demandent de désigner la même arête, comme ses extrémités sont hébergés par les deux machines, cette dernière est envoyée à l'une des deux machines aléatoirement. Nous représentons dans Figure 3.6 un aperçu de l'envoi des arêtes du graphe de Figure 3.3 (a) aux machines proxys. Dans cet exemple, 4 machines ont été considérées comme proxys. Nous avons choisi de distribuer les arêtes selon les couleurs de leurs extrémités afin d'éviter la congestion vers une seule machine. Il est important de noter que la règle de proxy n'est pas appliquée sur l'exemple, vu que qu'il n'y a aucun sommet dont le degré satisfait la règle ($\nexists v \in V$ tel que $deg(v) = 2 * 8 * log(11) \approx 16$).

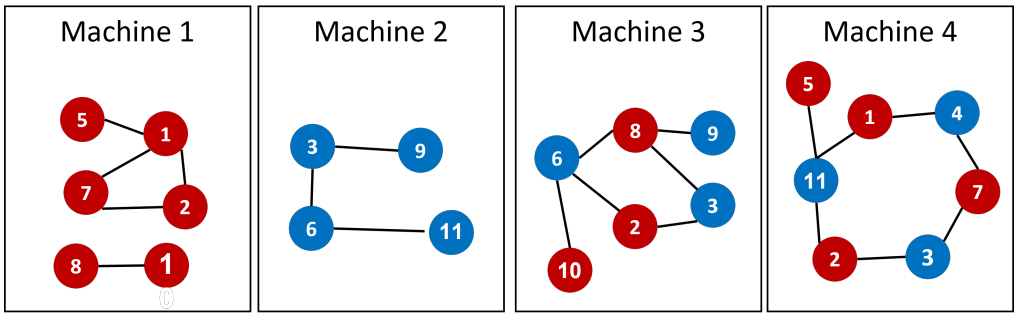


FIGURE 3.6 – Envoie des arêtes aux proxys.

3.4.1.4 Collecte des arêtes

Dans la dernière étape de la stratégie du partitionnement du graphe, chaque machine collecte ses arêtes requises des machines proxys respectives : chaque machine proxy connaît la fonction de hachage h ainsi que l'attribution déterministe des triplets de couleurs. Elle peut donc envoyer l'arête à la machine dont elle a besoin. En d'autres termes, chaque machine locale collecte les arêtes dont les extrémités se trouvent dans deux des trois ensembles de couleurs qui lui sont attribués. Une machine M avec un triplet de couleur (c_1, c_2, c_3) doit collecter toutes les arêtes dont les sommets sont dans les combinaisons de couleurs suivantes : (c_1, c_2), (c_2, c_3) ou (c_3, c_1). Figure 3.7 présente les arêtes collectées par chaque machine locale sur un modèle de 8 machines.

3.4.2 Énumération locale des triangles

Après avoir partitionné les données et équilibré la charge entre les processeurs, il ne reste qu'à énumérer tous les triangles dans chaque sous-graphe sur chaque machine, d'une manière locale et en parallèle. Pour cela, il suffit de trouver tous les triplets d'arêtes fermées qui respectent (1) un ordre lexicographique entre les sommets d'extrémité (du plus petit au plus grand identifiant) ainsi que (2) l'ordre du triplet de couleurs affecté à chaque machine. À ce propos, sur Machine 2 de Figure 3.7, nous pouvons extraire visuellement deux triangles qui respectent les deux règles suscitées. Ces triangles sont : (1,2,11) et (1,5,11). Il est clair que ces deux triangles suivent une logique lexicographique ($1 < 2 < 11$ et $1 < 5 < 11$). De plus, ils respectent l'ordre des couleurs affectées à la Machine 2 (Rouge, Rouge, Bleu). Nous représentons dans Figure 3.8 le résultat final de l'énumération locale des triangles sur chaque machine.

3.4.3 Analyse du modèle de calcul et de l'algorithme randomisé

3.4.3.1 Optimisation du modèle de calcul

Nous étudions dans cette section, l'impact de la réduction de la taille du modèle $k - machines$ sur l'exécution de l'algorithme adapté. Rappelons que le nombre de machines dans le modèle de

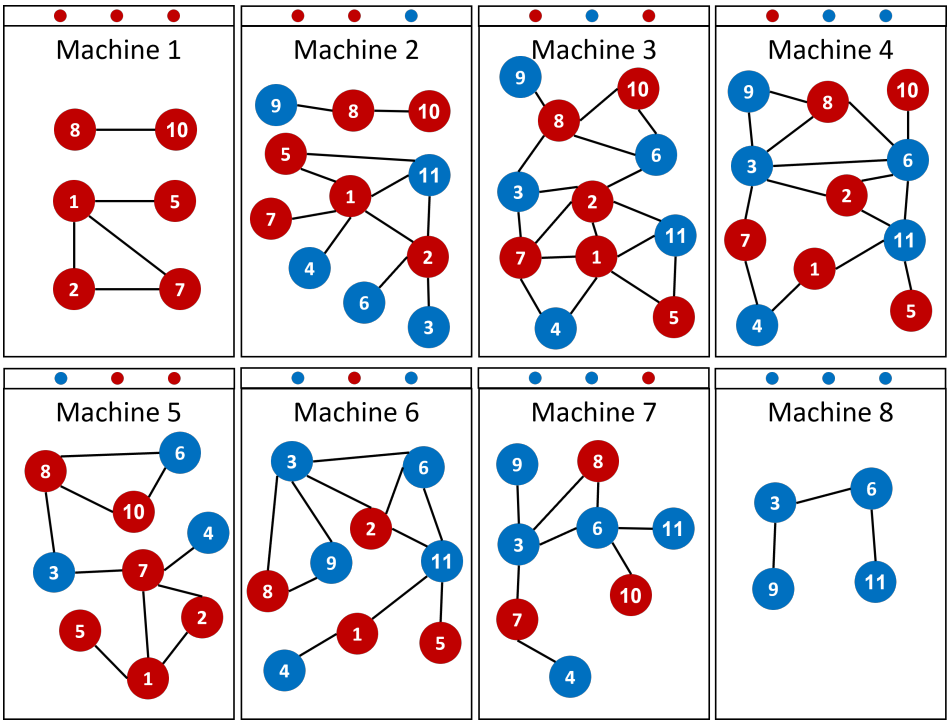


FIGURE 3.7 – Collecte des arêtes par les machines locales.

calcul doit impérativement respecter la règle suivante :

$$k = c^3/c \geq 2 \tag{3.2}$$

Ce nombre de machines peut être réduit en éliminant les répétitions dans les triplets de couleurs affectées aux machines. À titre d'exemple, le triplet de couleurs $(bleu, rouge, rouge)$ est fondamentalement le même que $(rouge, bleu, rouge)$ et $(rouge, rouge, bleu)$. Par conséquent, l'idée principale est de définir pour chaque combinaison de 3 couleurs un seul triplet de couleurs qui les définit dans tout ordre possible. Ainsi, le nombre total de machines sera réduit de (3.2) à (3.3).

$$k = c^2 + \sum_{q=2}^{c-1}(q-1)(c-q) \tag{3.3}$$

En utilisant l'équation (3.3), le nombre de machines peut être réduit de 8 à 4, de 27 à 10, de 64 à 20 et ainsi de suite. Néanmoins, opter pour une telle optimisation sur le SGBD déséquilibre la charge de travail, puisque l'algorithme adapté stocke les arêtes selon leurs directions même si le graphe n'est pas orienté. En particulier, les machines définissant les répétitions traiteront plus d'arêtes que les autres machines. Par exemple, dans un modèle à 4 machines, deux machines traiteront $\frac{3*\Delta(G)}{4}$ puisque chacune d'elles définit la répétition pour trois triplets de couleurs, alors que les deux autres machines auront $\frac{\Delta(G)}{4}$. Ainsi, les calculs locaux sur ces machines deviennent importants, et le coût de la communication augmente. Concrètement, avec $c = 2$ et $k = 4$, nous allons générer 4 triplets de couleurs (dont 2 définissent les répétitions). Sans perte de généralité, dans le cas où l'ensemble de sommets est partitionné en 2 couleurs dans un modèle à 4 machines, nous supposons que le nombre d'arêtes dont les extrémités sont dans deux sous-ensembles parmi les trois sous-ensembles de couleurs de chaque triplet est de $m/4$.
Lors d'un traitement en parallèle, la complexité temporelle de la jointure sur chaque machine dépend du nombre d'arêtes réparties sur les machines locales. Comme expliqué dans le paragraphe précédent, deux machines traitent $\frac{3*\Delta(G)}{4}$ et chacune d'elles aura trois triplets de couleurs.

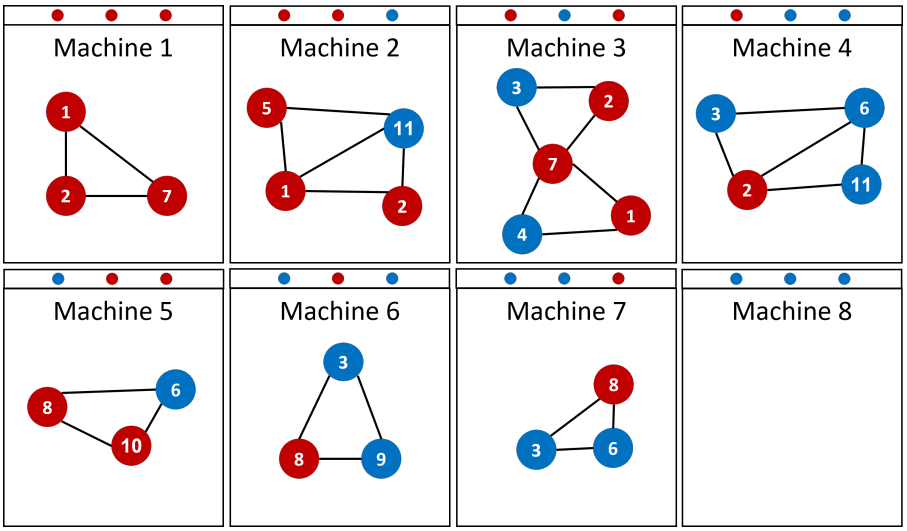


FIGURE 3.8 – Le résultat d'énumération des triangles par machines.

Supposons que le nombre des arêtes dans chaque combinaison de couleurs est de $m/4$, donc le nombre total des arêtes est de $3m/4$ pour ces machines. Cependant, le nombre d'arêtes sur la machine qui a le triplet de couleur $(rouge, rouge, rouge)$ ou $(bleu, bleu, bleu)$ est $m/4$. De même, nous pouvons obtenir le nombre maximal des arêtes sur chaque machine dans le modèle à 8 machines qui est de $3m/8$, en revanche le nombre minimal des arêtes est de $m/4$. Ainsi, la charge est déséquilibrée avec cette optimisation par rapport au modèle à 8 machines. Supposons que l'algorithme de la jointure locale est la jointure par hachage, la complexité temporelle de la jointure locale sera donc $O(3m/8)$ pour le modèle à 8 machines alors qu'elle est de $O(3m/4)$ pour le modèle à 4 machines, en supposant que le graphe a une distribution uniforme. Par conséquent, nous constatons que la complexité temporelle de la jointure locale est doublé lorsque le nombre de machines est optimisé à (3.3).
Une autre optimisation potentielle consiste à réduire le nombre de machines de (3.2) à (3.4).

$$k = c^2 \tag{3.4}$$

Cette optimisation est plus avantageuse que la précédente. Elle permet de conserver l'équilibrage de charge, car chaque machine traitera $c = k^{1/3}$ triplets de couleur donc $3m/4$ d'arêtes. Elle permet également d'accélérer les calculs pour les petits graphes, en réduisant le coût de communication entre c^2 machines au lieu des k machines. Pour les graphes à grande échelle, nous pouvons constater quelques augmentations au niveau de la complexité du calcul local puisque chaque machine du modèle défini par (3.4) traitera c fois le nombre de triangles de chaque machine dans le modèle $k-machines$ défini par (3.2). Cette étude est expérimentalement prouvée dans le chapitre 5.

3.4.3.2 Justesse de l'algorithme randomisé

Definition 3.2 *Un algorithme de Las Vegas est un algorithme probabiliste qui donne toujours un résultat correct, tout en conservant de meilleures performances temporelles grâce à son aspect aléatoire [MR95].*

L'algorithme randomisé d'énumération de triangles est un algorithme de Las Vegas. Son résultat est toujours correct quelque soit le type du graphe, sa taille ou la taille du modèle $k-machines$. Un simple raisonnement nous permet de vérifier l'exactitude des résultats. Supposons un graphe $G = (V, E)$ et un modèle $k-machines$. En appliquant l'algorithme randomisé d'énumération de triangles, nous générons un ensemble de distinctes couleurs $C = \{c_1, c_2, .., c_{k^{1/3}}\}$. Nous pouvons

ainsi créer k triplets de couleurs avec les différentes combinaisons entre les couleurs de C : (c_1, c_1, c_1), (c_1, c_1, c_2), (c_1, c_2, c_1), etc. Afin de prouver que l'algorithme randomisé d'énumération de triangles est toujours correct, nous devons prouver que :

(1) la stratégie du partitionnement est correcte,

(2) le résultat inclut tous les triangles du graphe,

(3) chaque triangle est listé une et une seule fois (pas de triangles redondants).

Propriété 3.2 : Chaque machine parmi les machines du modèle se voit attribuer un et un seul triplet de couleur (c_x, c_y, c_z) d'une manière aléatoire.
Propriété 3.3 : $\forall v \in V$ choisit aléatoirement une couleur parmi les c couleurs, ainsi $(v_1, v_2) \in E$ sera coloré avec (c_1, c_2).
Propriété 3.4 : Si (v_1, v_2, v_3) est un triangle, alors $v_1 < v_2 < v_3$ (un ordre lexicographique).
Afin de prouver (1), nous analysons la propriété 3.3. Chaque arête sera dans deux sous-ensembles de couleurs. Elle sera donc envoyée à une machine M définissant ces deux sous-ensembles de couleurs parmi son triplet de couleurs. Par exemple et sans perdre de généralité, la machine M_1 avec le triplet de couleurs (c_2, c_1, c_2) hébergera toutes les arêtes dont les extrémités ont les couleurs suivantes : (c_2, c_1), (c_1, c_2), ou (c_2, c_2). Par conséquent, la stratégie du partitionnement est correcte, car chaque machine aura que les arêtes qui peuvent former des triangles selon son triplet de couleurs, i.e. la machines M_1 hébergera que les arêtes qui peuvent former des triangles dont les sommets ont les couleurs (c_2, c_1, c_2).
Pour la preuve de (2), nous utilisons (1) et les propriétés 3.2 et 3.3. Pour une machine M avec le triplet de couleurs (c_x, c_y, c_z), si le triplet de sommets (v_1, v_2, v_3) choisit (c_x, c_y, c_z), il est envoyé à M car (1) est vérifié. À ce propos, si le triplet de sommets (v_1, v_2, v_3) choisit (c_2, c_1, c_2), il sera reconnu comme un triangle sur la machine M_1 uniquement, car c'est la seule machine détentrice du triplet (c_2, c_1, c_2). Enfin, un ordre lexicographique (du plus petit au plus grand identifiant de sommet) est considéré lors de la détection locale des triangles sur chaque machine. De ce fait, chaque triangle est énuméré une et une seule fois et (3) est vérifié.

3.4.4 Équilibrage de charge

Nous avons mentionné dans la section 3.4.1, que chaque sommet de l'ensemble V choisit aléatoirement, et indépendamment une couleur parmi $c = k^{1/3}$ distinctes couleurs. Cela donne lieu à une partition de l'ensemble de sommets V en c sous-ensembles de $O(n/c)$ sommet chacun. Chaque machine reçoit alors un sous-graphe $G_x = (V_x, E_x)$ de G. Chaque sous-graphe est composé des arêtes dont les extrémités sont dans deux ensembles parmi les trois sous-ensembles de couleurs affectés à chaque machine. L'analyse de [PRS18b] montre que le nombre d'arêtes parmi les sous-graphes G_x est relativement équilibré avec une probabilité élevée. Cette analyse consiste principalement à borner par le haut, le nombre d'arêtes attribuées à chaque machine. De ce fait, le nombre d'arêtes entre les paires de sous-ensembles d'un triplet n'est pas plus grand que le nombre d'arêtes dans G_x sur chaque machine. En revanche, en raison du partitionnement aléatoire basé sur la couleur des sommets, le nombre d'arêtes entre les paires de sous-ensembles d'un triplet est asymptotiquement équivalent au nombre d'arêtes dans le sous-graphe G_x induit par un ensemble de nœuds choisis aléatoirement du graphe G.
Nous présentons dans la suite la preuve par récursivité qui permet de démontrer que l'équilibrage de charge entre les machines est préservé.

3.4.4.1 Preuve avec $c = 2$

Supposons que nous avons un graphe avec n sommets et m arêtes et que son partitionnement se fait à l'aide de deux couleurs : (1) noir (b) et (2) blanc (w).

— **Sommets :** chaque sommet peut choisir uniformément une couleur parmi $c = \{b, w\}$. Donc il y aura $\frac{n}{2}$ sommets noirs et $\frac{n}{2}$ sommets blancs.

— **Arêtes :** chaque arête relie deux sommets colorés. Ainsi il y aura 4 possibilités pour partitionner les arêtes en se basant sur les couleurs de leurs extrémités : (b, b), (b, w), (w, b) et (w, w).

Considérant l'arête $w \to b$, chaque extrémité (sommet) possède une probabilité de $p = 1/2$ pour choisir w ou b. Sachant que ces deux évènements sont indépendants, la probabilité de produire l'arête $w \to b$ est

$$P[w \to b] = \frac{1}{2} \times \frac{1}{2} = \frac{1}{4} \tag{3.5}$$

En résumé, chaque arête aura une possibilité de $\frac{1}{4}$ pour apparaître, et nous écrivons Table 3.1 :

Tableau 3.1 – Partitionnement des arêtes par deux couleurs.

Arête	Nombre
b $\to$ b	$1/4 \times m$
b $\to$ n	$1/4 \times m$
n $\to$ b	$1/4 \times m$
n $\to$ n	$1/4 \times m$

Ainsi, il existe $2^3 = 8$ triplets de couleurs. Le nombre de triangles correspondant à chaque triplet sont relativement équivalents. Supposant que nous avons le triplet (w, w, b), le premier sommet doit être de couleur w, avec une probabilité de $\frac{1}{2}$, le deuxième sommet doit être également de couleur w avec la même probabilité. Enfin, le dernier sommet est de couleur b, avec une probabilité de $\frac{1}{2}$. En calculant la probabilité d'apparition du triangle avec les couleurs (w, w, b), nous trouvons :

$$P[\Delta_{(w,w,b)}] = \frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = \frac{1}{8} \tag{3.6}$$

En appliquant le même principe sur les autres triplets, nous recensons Table 3.2 :

Tableau 3.2 – Partitionnement des triangles.

Triplet de couleurs	Nombre de triangles
(b,b,b)	$1/8 \times T$
(b,b,w)	$1/8 \times T$
(b,w,b)	$1/8 \times T$
(b,w,w)	$1/8 \times T$
(w,b,b)	$1/8 \times T$
(w,b,w)	$1/8 \times T$
(w,w,b)	$1/8 \times T$
(w,w,w)	$1/8 \times T$

En conclusion, le nombre de triangles est équilibré entre les triplets de couleurs, où sur un modèle $k - machines$, chaque machine traite essentiellement le même nombre de triangles. Cela permet d'équilibrer la charge de travail. Dans la suite nous étudions l'équilibrage de charge, en considérant le nombre d'arêtes traitées par chaque machine sur un modèle original et un autre modèle optimisé.

Model avec $2^3 = 8$ machines

(1) Équilibrage de charge par rapport aux triangles : Chaque machine reçoit un triplet de couleurs, elle produira $\frac{1}{8} \times T$ de triangles. Ainsi, les triangles sont parfaitement équilibrés.

(2) Équilibrage de charge par rapport aux arêtes :

(i) Supposons le triplet de couleurs suivant (b, w, w). Afin d'énumérer les triangles respectant ce triplet, nous devons envoyer les arêtes dont les extrémités sont dans les couleurs suivantes : (b, w), (w, w), et (w, b). Les arêtes envoyées doivent respecter les restrictions suivantes pour éliminer la répétition :

(a) l'arête (b, w) avec $b.id < w.id$,

(b) l'arête (w, w) avec $w.id < w.id$,

(c) et l'arête (w, b) avec $w.id > b.id$.

Comme mentionné précédemment, le nombre d'arêtes de couleur (b, w) est de $\frac{1}{4} \times m$. La probabilité de $b.id < w.id$ est la même que la probabilité de $b.id > w.id$ car le partitionnement est aléatoire et uniforme. Par conséquent, le nombre d'arêtes de couleur (b, w) avec $b.id < w.id$ est de

$$P[b \to w | b.id < w.id] = \frac{m}{4} \times 0,5 = \frac{1}{8} \times m \tag{3.7}$$

De manière similaire, le nombre d'arêtes de couleur (w, w) avec $w.id < w.id$ est de

$$P[w \to w | w.id < w.id] = \frac{m}{4} \times 0,5 = \frac{1}{8} \times m \tag{3.8}$$

et le nombre d'arêtes de couleur (w, b) avec $w.id > b.id$ est de

$$P[w \to b | w.id > b.id] = \frac{m}{4} \times 0,5 = \frac{1}{8} \times m \tag{3.9}$$

Ainsi, la machine avec le triplet de couleurs (b, w, w) collecte les arêtes avec la probabilité :

$$P[\Delta_{(b,w,w)} | b.id < w.id \ \& \ w.id < w.id \ \& \ w.id > b.id] = \frac{m}{8} + \frac{m}{8} + \frac{m}{8} = \frac{1}{4} \times m \tag{3.10}$$

(ii) Supposons maintenant un autre triplet, dont toutes les couleurs sont identiques, par exemple le triplet (w, w, w). En procédant de la même manière, nous devons envoyer les arêtes dont les couleurs sont (w, w) à la machine possédant ce triplet de couleurs. Nous devons donc respecter les contraintes suivantes :

(a) l'arête (w, w) avec $w.id < w.id$,

(b) l'arête (w, w) avec $w.id < w.id$,

(c) et l'arête (w, w) avec $w.id > w.id$.

Puisque le partitionnement est aléatoire et uniforme, le nombre des arêtes dont les couleurs sont (w, w) en considérant $w.id < w.id$ et $w.id > w.id$ est :

$$P[w \to w] = \frac{1}{4} \times m \tag{3.11}$$

En résumé, le nombre des arêtes par machines selon la couleurs de leurs extrémités est présenté dans Table 3.3 :

Model avec $2^2 = 4$ machines

(1) Équilibrage de charge par rapport aux triangles : Chaque machine reçoit deux triplets de couleurs, elle produira $2 \times \frac{1}{8} \times T = \frac{1}{4} \times T$ de triangles. Ainsi, les triangles sont parfaitement équilibrés.

(2) Équilibrage de charge par rapport aux arêtes : Supposons les triplets de couleurs suivants (b, w, w) et (w, b, b). Afin d'énumérer les triangles respectant ces triplets, nous devons envoyer les arêtes dont les extrémités sont dans les couleurs suivantes : $\{(b, w), (w, w), (w, b)\}$ et $\{(w, b), (b, b), (b, w)\}$ avec les restrictions suivantes :

Tableau 3.3 – Équilibrage de charge par rapport aux arêtes.

Machine	Triplet de couleurs	Nombre d'arêtes
M_1	(b,b,b)	$1/4 \times m$
M_2	(b,b,w)	$3/8 \times m$
M_3	(b,w,b)	$3/8 \times m$
M_4	(b,w,w)	$3/8 \times m$
M_5	(w,b,b)	$3/8 \times m$
M_6	(w,b,w)	$3/8 \times m$
M_7	(w,w,b)	$3/8 \times m$
M_8	(w,w,w)	$1/4 \times m$

(a) l'arête (b, w) avec $b.id < w.id$,

(b) l'arête (w, w) avec $w.id < w.id$,

(c) l'arête (w, b) avec $w.id > b.id$.

et

(a) l'arête (w, b) avec $w.id < b.id$,

(b) l'arête (b, b) avec $b.id < b.id$,

(c) et l'arête (b, w) avec $b.id > w.id$.

En effectuant la même analyse présentée précédemment, nous résumons le nombre d'arêtes sur chaque machines dans Table 3.4 :

Tableau 3.4 – Nombre d'arêtes par machine.

Machine	Triplet de couleurs	Nombre d'arêtes	Total d'arêtes par machine
M_1	(b,b,b)	$1/4 \times m$	$1/2 \times m$
M_1	(w,w,w)	$1/4 \times m$	
M_2	(b,b,w)	$3/8 \times m$	$3/4 \times m$
M_2	(w,w,b)	$3/8 \times m$	
M_3	(b,w,b)	$3/8 \times m$	$3/4 \times m$
M_3	(w,b,w)	$3/8 \times m$	
M_4	(b,w,w)	$3/8 \times m$	$3/4 \times m$
M_4	(w,b,b)	$3/8 \times m$	

En résumé, les triangles sont parfaitement équilibrés entre les machines dans ce modèle, avec chaque machine produisant $(1/4)$ du total des triangles. Cependant, les arêtes ne sont pas équilibrées. Le nombre d'arêtes attribuées à une machine est entre $1/2 \times m$ et $3/4 \times m$.

Conclusion : dans les deux modèles ($k = c^2$ ou $k = c^3$), le nombre des triangles produits est équilibré. Cependant, les arêtes attribuées à chaque machine ne le sont pas. Nous trouvons plus d'arêtes sur les machines dont le triplet de couleurs est constitué de plus d'une couleur.

3.4.4.2 Preuve avec $c > 2$

Admettons que nous avons l'ensemble des couleurs $C = \{c_1, c_2, \cdots, c_\rho\}$, avec $c = |C|$. En appliquant l'algorithme randomisé, chaque sommet choisit aléatoirement et uniformément une couleur parmi les c couleurs.

— **Sommet :** la possibilité qu'un sommet choisisse une couleur c est de $\frac{1}{c}$.

TABLEAU 3.5 – Nombre des arêtes selon les couleurs de ses extrémités.

Arête	Nombre
$c_x \rightarrow c_y$	$1/c^2 \times m$; $x, y \in [1, \rho[$ *et* $\rho = k^{1/3}$

— **Arête :** En utilisant les c couleurs, nous générons c^2 possibilités, par exemple avec $c = 3 \rightarrow \#arete\_colore = 9$, avec $c = 4 \rightarrow \#arete\_colore = 16$, etc. D'une manière générale, nous recensons le nombre d'arête selon la couleur des extrémités dans Table 3.5.
Considérons l'arête $c_x \rightarrow c_y$, le premier sommet de couleur c_x est de probabilité $\frac{1}{c}$, et le deuxième de couleur c_y est de probabilité $\frac{1}{c}$. Étant donné que les deux évènements sont indépendants, la probabilité d'apparition de l'arête est de :

$$P[c_x \rightarrow c_y] = \frac{1}{c} \times \frac{1}{c} = \frac{1}{c^2} \tag{3.12}$$

— **Triplets de couleurs :** Selon l'algorithme, nous devons générer des triplets de couleurs en fonction des différentes permutations des c couleurs. Donc, nous devons générer c^k triplets de (c_x, c_y, c_z) avec $x, y, z \in [1, \rho]$. Ces triplets sont utilisés pour énumérer localement les triangles dans le graphe d'entrée. Rappelons la probabilité qu'un sommet choisisse une couleur c est $\frac{1}{c}$. Par conséquent, la probabilité qu'un triangle soit constitué des couleurs (c_x, c_y, c_z) est

$$P[\Delta_{(c_x,c_y,c_z)}] = \frac{1}{c} \times \frac{1}{c} \times \frac{1}{c} = \frac{1}{c^3} \tag{3.13}$$

Par conséquent, chaque triangle possède une possibilité de $\frac{1}{c^3}$ d'exister.

Modèle avec $k = c^3$ machines

(1) Équilibrage de charge par rapport aux triangles : Chaque machine reçoit un triplet de couleurs, elle produira $\frac{1}{c^3} \times T = \frac{1}{k} \times T$ de triangles. Ainsi, les triangles sont parfaitement équilibrés.

(2) Équilibrage de charge par rapport aux arêtes : Pour analyser l'équilibre des arêtes, nous devons étudier 4 cas :

(a) Le triplet de couleur définissant la même couleur comme (c_x, c_x, c_x) : Chaque machine doit recevoir les arêtes dont les couleurs d'extrémité sont (c_x, c_x) en suivant les restrictions suivantes :

— (c_x, c_x) avec $c_x.id < c_x.id$ (id du sommet source doit être inférieur à celui du sommet de destination),

— (c_x, c_x) avec $c_x.id < c_x.id$,

— (c_x, c_x) avec $c_x.id > c_x.id$.

En suivant l'analyse précédente, le nombre des arêtes avec les couleurs (c_x, c_x) est

$$P[c_x \rightarrow c_x | c_x.id < c_x.id \ \& \ c_x.id > c_x.id] = \frac{1}{c^2} \times m \tag{3.14}$$

Par conséquent, le nombre maximal des arêtes utilisées pour déterminer les triangles qui correspondent au triplet (c_x, c_x, c_x) est

$$P[\Delta_{(c_x,c_x,c_x)}] = \frac{1}{c^2} \times m \tag{3.15}$$

(b) Deux couleurs parmi les trois couleurs sont identiques dans le triplet comme (c_x, c_y, c_y) et toutes les autres variantes : Les arêtes avec les couleurs suivantes sont envoyées aux machines (c_x, c_y), (c_y, c_y) et (c_y, c_x) avec les restrictions suivantes :

— (c_x, c_y) avec $c_x.id < c_y.id$,
— (c_y, c_y) avec $c_y.id < c_y.id$,
— (c_y, c_x) avec $c_y.id > c_x.id$.

Le nombre des arêtes avec les couleurs (c_x, c_y) est

$$P[c_x \rightarrow c_y | c_x.id < c_y.id] = \frac{1}{2 \times c^2} \times m \tag{3.16}$$

Nous retrouvons la même probabilité pour les arêtes avec les couleurs (c_y, c_y) et (c_y, c_x). Donc, le nombre maximal des arêtes pour le triangle avec le triplet (c_x, c_y, c_y) est

$$P[\Delta_{(c_x,c_y,c_y)}] = \frac{1}{2 \times c^2} \times m + \frac{1}{2 \times c^2} \times m + \frac{1}{2 \times c^2} \times m = \frac{3}{2 \times c^2} \times m \tag{3.17}$$

(c) Les deux couleurs d'extrémité du triplet sont identiques, telles que (c_x, c_y, c_x) : dans ce cas, les arêtes (c_x, c_y), (c_y, c_x) et (c_x, c_x) doivent être envoyés à la même machine en suivant ces conditions :
— (c_x, c_y) avec $c_x.id < c_y.id$,
— (c_y, c_x) avec $c_y.id < c_x.id$,
— (c_x, c_x) avec $c_x.id > c_x.id$.

Le nombre d'arêtes avec les couleurs (c_x, c_y) en prenant en considération $c_x.id < c_y.id$ et $c_x.id > c_y.id$ est

$$P[c_x \rightarrow c_y] = \frac{1}{c^2} \times m \tag{3.18}$$

et le nombre d'arêtes avec les couleurs (c_x, c_x) est

$$P[c_x \rightarrow c_x | c_x.id > c_x.id] = \frac{1}{2 \times c^2} \times m \tag{3.19}$$

Nous évaluons le nombre d'arêtes sur la machine possédant le triplet (c_x, c_y, c_x) à

$$P[\Delta_{(c_x,c_y,c_x)}] = \frac{1}{c^2} \times m + \frac{1}{2 \times c^2} \times m = \frac{3}{2 \times c^2} \times m \tag{3.20}$$

(d) Les trois couleurs du triplet sont différentes, à savoir (c_x, c_y, c_z) avec $c_x \neq c_y$ et $c_y \neq c_z$: La machine possédant ce triplet collecte les arêtes suivantes : (c_x, c_y), (c_y, c_z), et (c_z, c_x), avec les conditions suivantes :
— (c_x, c_y) avec $c_x.id < c_y.id$,
— (c_y, c_z) avec $c_y.id < c_z.id$,
— (c_z, c_x) avec $c_z.id > c_x.id$.

Le nombre d'arêtes (c_x, c_y) est

$$P[c_x \rightarrow c_y | c_x.id < c_y.id] = \frac{1}{2 \times c^2} \times m \tag{3.21}$$

et le nombre d'arêtes (c_y, c_z) est

$$P[c_y \rightarrow c_z | c_y.id < c_z.id] = \frac{1}{2 \times c^2} \times m \tag{3.22}$$

enfin, le nombre d'arêtes (c_z, c_x) est

$$P[c_z \rightarrow c_x | c_z.id > c_x.id] = \frac{1}{2 \times c^2} \times m \tag{3.23}$$

Nous retrouvons ainsi le nombre des arêtes impliquées dans le triangle avec le triplet (c_x, c_y, c_z) égal à

$$P[\Delta_{(c_x,c_y,c_z)}] = \frac{1}{2 \times c^2} \times m + \frac{1}{2 \times c^2} \times m + \frac{1}{2 \times c^2} \times m = \frac{3}{2 \times c^2} \times m \tag{3.24}$$

Pour résumer, nous présentons Table 3.6 :

TABLEAU 3.6 – Nombre d'arêtes par machine sur le modèle $k = c^3$.

Triplet	Nombre d'arêtes
(c_x, c_x, c_x)	$\frac{1}{c^2} \times m$
(c_x, c_y, c_y)	$\frac{3}{2\times c^2} \times m$
(c_x, c_y, c_x)	$\frac{3}{2\times c^2} \times m$
(c_x, c_y, c_z)	$\frac{3}{2\times c^2} \times m$

Model avec $k = c^2$ machines

(1) Équilibrage de charge par rapport aux triangles : Nous savons que le nombre de triangles correspondants à chaque triplet de couleurs est $\frac{1}{c^3} \times T$. Sur un modèle de $k = c^2$, chaque machine gère c triplets de couleurs. Donc, chaque machine produira $c \times \frac{1}{c^3} \times T = \frac{1}{c^2} \times T$. Ainsi, le nombre de triangles reste parfaitement équilibré.

(2) Équilibrage de charge par rapport aux arêtes : Nous devons analyser les quatres situations précédemment présentées dans le modèle $k = c^3$.

(a) Le triplet (c_x, c_x, c_x) : En suivant le même raisonnement, le nombre des arêtes nécessaires pour déterminer les triangles correspondants à ce triplet de couleur est :

$$P[\Delta_{(c_x,c_x,c_x)}] = c \times \frac{3}{c^2} \times m = \frac{1}{c} \times m \tag{3.25}$$

(b) Le triplet (c_x, c_y, c_y) : le nombre d'arêtes nécessaires est :

$$P[\Delta_{(c_x,c_y,c_y)}] = c \times \frac{3}{2 \times c^2} \times m = \frac{3}{2 \times c} \times m \tag{3.26}$$

(c) Le triplet (c_x, c_y, c_x) : Le nombre maximal des arêtes est :

$$P[\Delta_{(c_x,c_y,c_x)}] = c \times \frac{3}{2 \times c^2} \times m = \frac{3}{2 \times c} \times m \tag{3.27}$$

(d) Le triplet (c_x, c_y, c_z) : Le nombre des arêtes nécessaires est :

$$P[\Delta_{(c_x,c_y,c_z)}] = c \times \frac{3}{2 \times c^2} \times m = \frac{3}{2 \times c} \times m \tag{3.28}$$

Nous résumons dans Table 3.7 le nombre maximal d'arêtes pour chaque machines :

TABLEAU 3.7 – Nombre d'arêtes par machine sur le modèle $k = c^2$.

Triplet	Nombre d'arêtes
(c_x, c_x, c_x)	$\frac{1}{c} \times m$
(c_x, c_y, c_y)	$\frac{3}{2\times c} \times m$
(c_x, c_y, c_x)	$\frac{3}{2\times c} \times m$
(c_x, c_y, c_z)	$\frac{3}{2\times c} \times m$

Conclusion : Les deux modèles ($k = c^2$ ou $k = c^3$) renvoient un nombre équilibré de triangles. Cependant, les arêtes attribuées à chaque machine ne sont pas équilibrées.

3.4.5 Étude de complexité

Généralement, dans le modèle $k - machines$, le calcul local au sein d'une machine est supposé être exécuté instantanément sans coût, alors que la communication entre les machines est l'opération coûteuse [PRS18b]. Cependant, pour l'algorithme randomisé d'énumération des triangles,

la complexité temporelle prise par chaque machine est proportionnelle au nombre d'arêtes et de triangles qu'elle traite. Chaque machine se voit attribuer un triplet particulier de couleurs (c_1, c_2, c_3), donc elle traite $O(n/k^{1/3})$ sous-ensembles de sommets de taille aléatoire. Le nombre de triangles dans le pire des cas dans ce sous-ensemble est $O(n^3/k)$; cependant, le nombre d'arêtes est beaucoup plus faible. En effet, l'algorithme randomisé [PRS18b] est basé essentiellement sur le principe qu'un sous-ensemble aléatoire de la taille mentionnée ci-dessus (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 à forte probabilité (la notation $\tilde{O}$ inclue un facteur multiplicatif et additif de $polylog(n)$). Par conséquent, chaque machine ne gère qu'un nombre limité d'arêtes avec une probabilité élevée. Comme on sait que le nombre de triangles qui peuvent être répertoriés en utilisant un ensemble d'arêtes ℓ est $\Omega(\ell^{3/2})$ [PRS18b], le nombre de triangles que chaque machine doit manipuler est au plus

$$\max\{\tilde{O}(m^{3/2}/k, n^{3/2}/k^{1/2})\} \tag{3.29}$$

Le nombre maximum de triangles (distincts) dans un graphe de m arêtes est au plus $O(m^{3/2})$, où chaque machine manipule essentiellement une fraction de $1/k$ de ces triangles (quand $m^{3/2}/k > n^{3/2}/k^{1/2}$). En conséquence, nous obtiendrons une accélération optimale (qui peut être linéaire), à l'exception éventuelle pour les graphes très clairsemés (quand $m^{3/2}/k < n^{3/2}/k^{1/2}$).
Selon [PRS18b], le coût de communication de l'algorithme randomisé pour l'énumération de triangles est

$$\tilde{O}(m/k^{5/3} + n/k^{4/3}) \tag{3.30}$$

cycles avec des probabilités élevées. Ainsi, si le nombre de machines k augmente et la taille du graphe d'entrée est petite, l'accélération ne sera pas garantie. Cela est dû au coût de communication et de la répartition des données entre les machines. A noter que le coût de la communication est influencé par la restriction de la bande passante et la configuration du réseau (LAN, WLAN, ..etc). Explicitement, pour les graphes à grande échelle, avoir un modèle de $k - machines$ plus grand (c-à-d $m \gg k$) peut accélérer les calculs, où chaque machine traite un nombre important de triangles. Alors que pour les graphes de petite à moyenne taille, l'augmentation de k peut entraîner une surcharge à cause du coût de la redistribution des données. Cela est du au coût de la communication délimité par la bande passante du réseau entre machines. Autrement dit, il y a une restriction sur la quantité de données que chaque machine peut communiquer à chaque cycle [PRS18b].

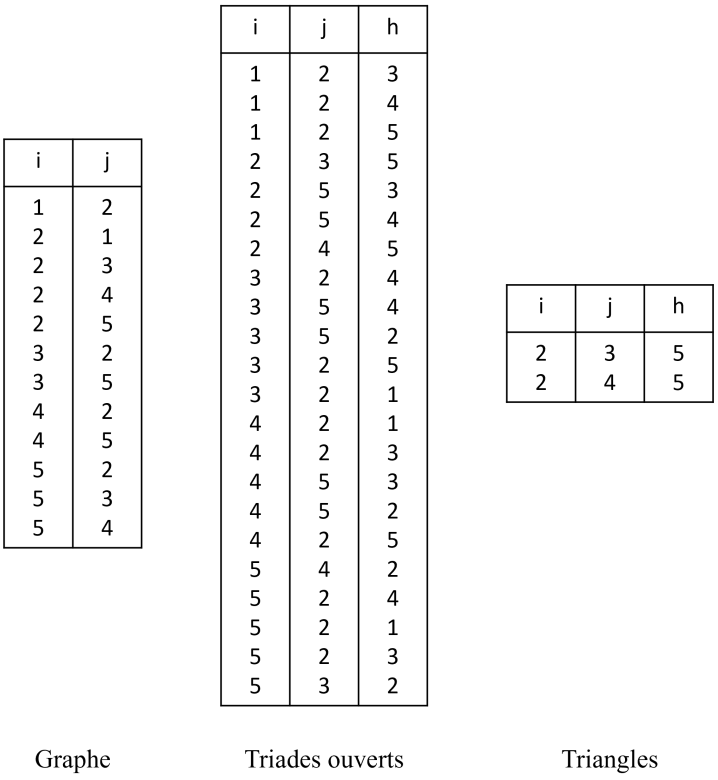
3.5 Algorithme adapté

Après avoir présenté l'algorithme randomisé d'énumération de triangles, nous nous intéressons dans cette section à notre deuxième contribution qui consiste en l'adaptation, l'implémentation, l'optimisation et l'étude de complexité de l'algorithme randomisé exécuté dans un environnement de base de données relationnelles et avec un outil d'analyse de données [FBO+20a, FZB+23a].
Tout d'abord, nous définissons le problème d'énumération de triangles dans l'environnement d'exécution, puis nous présentons un schéma conceptuel de données qui permet l'exécution de l'algorithme randomisé sur un SGBD relationnel. Ensuite, nous reprenons les différentes étapes de l'algorithme en introduisant la requête qui permet d'exécuter chacune entre elle sur le SGBD et avec Pandas de Python. Enfin, une étude de complexité et d'équilibrage de charge est élaborée afin d'examiner la performance des requêtes et du programme résultant.

3.5.1 Définition du problème d'énumération des triangles dans les bases de données relationnelles

Definition 3.3 *En s'inspirant de la méthode de Cohen [Coh09], l'énumération et le comptage des triangles dans les bases de données relationnelles s'effectue en appliquant une double auto-jointure sur la table des arêtes.*

Une application de la méthode de Cohen [Coh09] sur la liste des arêtes est donnée dans Figure 3.9. Une seule instance de chaque triangle a été donnée dans la table des résultats "Triangles".



i	j
1	2
2	1
2	3
2	4
2	5
3	2
3	5
4	2
4	5
5	2
5	3
5	4

Graphe

i	j	h
1	2	3
1	2	4
1	2	5
2	3	5
2	5	3
2	5	4
2	4	5
3	2	4
3	5	4
3	5	2
3	2	5
3	2	1
4	2	1
4	2	3
4	5	3
4	5	2
4	2	5
5	4	2
5	2	4
5	2	1
5	2	3
5	3	2

Triades ouverts

i	j	h
2	3	5
2	4	5

Triangles

FIGURE 3.9 – Énumération de triangle en SQL avec la méthode de Cohen.

3.5.2 Modèle conceptuel de données

Afin de pouvoir adapter l'algorithme randomisé d'énumération et de comptage de triangles avec un SGBD relationnel, nous avons conçu un modèle Entité/Association composé de cinq entités. Ce modèle est ensuite traduit par un modèle relationnel implémenté sur le SGBD. Cette conception conserve les propriétés du partitionnement aléatoire décrit par l'algorithme randomisé. Par conséquent, elle garantit un équilibrage de charge parfait entre les machines du modèle $k - machines$ lors de l'énumération et le comptage des triangles.
Figure 3.10 illustre ce modèle conceptuel de données :

(1) E_s : la table des arêtes ou la table d'adjacence, elle contient toutes les arêtes du graphe,

(2) V_s : contient la liste de tous les sommets avec la couleur aléatoire choisie par chaque sommet,

(3) $Triplet$: contient la liste des $k - machines$ et le triplet de couleur attribué à chaque machine,

(4) E_s_proxy : permet d'envoyer les arêtes à leurs proxys, elle porte pour chaque arête les couleurs choisies par ses extrémités,

(5) E_s_local : permet d'envoyer les arêtes à leurs machines locales correspondantes.

Un modèle relationnel est ensuite créé à l'issue de ce modèle conceptuel de donnée, il contient cinq relations décrites dans la suite :

```
E_s(i, j);
Triplet(machine, color1, color2, color3);
```

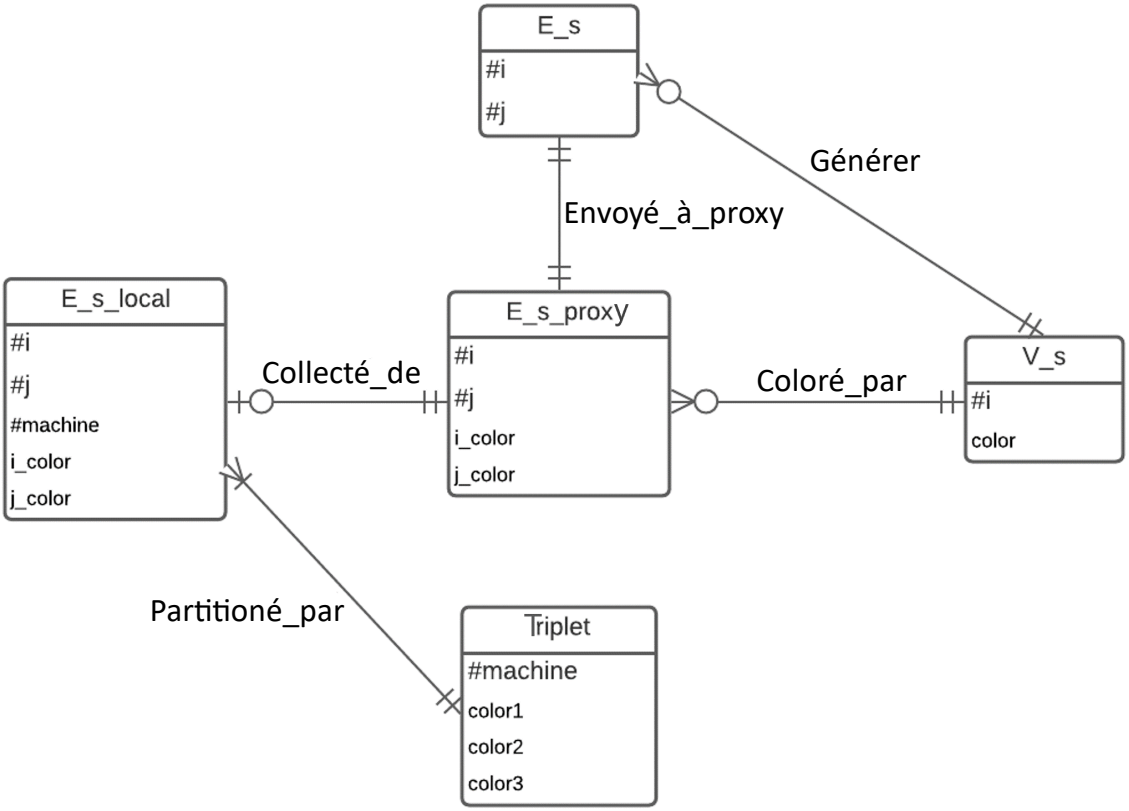


FIGURE 3.10 – Modèle conceptuel de données de l'algorithme adapté.

```
V_s(i, color);
E_s_proxy(i, j, i_color, j_color);
E_s_local (machine, i, j, i_color, j_color);
```

Les tables *E_s* et *Triplet* sont les tables d'entrée de notre algorithme. *E_s* est la table des arêtes qui contient toutes les arêtes du graphe d'entrée. La table *Triplet* contient pour chaque machine son triplet de couleurs correspondant. Comme nous l'avons expliqué dans la section précédente, nous devons affecter à chaque machine un triplet de couleurs composé des $c = k^{1/3}$ sous-ensembles de couleurs. Cette affectation est effectuée manuellement dans l'algorithme. De ce fait, le remplissage de la table *Triplet* est laissé à la charge de l'utilisateur. La table *V_s* est créée en sélectionnant tous les sommets de la table *E_s* avec leurs couleurs choisies au hasard. Les sommets source et destination de chaque arête doivent exister dans la table *V_s*. Ensuite, chaque arête est envoyée à la table *E_s_proxy* avec les couleurs correspondantes des sommets source et destination. La table *E_s_proxy* est remplie avec le résultat d'une requête de jointure entre la table *E_s* et la table *V_s*. Enfin, la table *E_s_local* récupère les arêtes de la table *E_s_proxy* selon son triplet de couleurs sur chaque machine. La table *E_s_local* se compose de : (1) la colonne *machine* pour identifier la machine locale, (2) les colonnes i et j pour les arêtes, et (3) les colonnes *i_color* et *j_color* pour les couleurs des extrémités des arêtes. L'énumération et le comptage local des triangles sur chaque machine est effectué exclusivement en exploitant la table *E_s_local*.

3.5.3 Algorithme adapté pour l'énumération des triangles

Dans cette section, nous présentons l'algorithme adapté [FBO+20a, FZB+23a] (voir Algorithme 3) inspiré de l'algorithme randomisé d'énumération et de comptage de triangles. Ensuite, nous détaillons ses deux phases : (1) le partitionnement des données, et (2) l'énumération et le comptage local des triangles, en utilisant les requêtes SQL les traduisant ainsi que les expressions avec Pandas.

Algorithm 3 Algorithme adapté d'énumération de triangle (toutes les boucles sont parallèles).

Input: $G(V, E)$; $e \in E$; c ▷ c= `nombre de couleurs.`
Output: $\Delta(G)$
Graph reading in table E ▷ E= `table d'arête.`
$machine\# \in \{1, \cdots, k\} \leftarrow (color1, color2, color3)$
for $v \in V$ **do**
  $v.color = random(c)$
end
for $proxy\_machine\# \in \{1, \cdots, k\}$ ▷ `Rééquilibrage d'arêtes sur les proxys.`
do
  $proxy\_machine\# \leftarrow e$
end
for $machine\# \in \{1, \cdots, k\}$ ▷ `Rééquilibrage de triangles sur les machines locales.`
do
  $machine\# \leftarrow Recv(e)$ ▷ `Recv = fonction pour collecter les arêtes selon le triplet de couleurs attribué à la machine.`
end
for $machine\# \in \{1, \cdots, k\}$ **do**
  for $e_1, e_2, e_3 \in E_{machine\#}$ ▷ `Énumération locale des triangles.`
  do
    if $(e_1.j = e_2.i \wedge e_2.j = e_3.i \wedge e_3.j = e_1.i)$ **then**
      if $e_1.i < e_2.i < e_3.i$ **then**
        $\Delta(G) \leftarrow \Delta(G) \cup (e_1, e_2, e_3)$
      end
    end
  end
end

3.5.3.1 Partitionnement des données

La première étape primordiale dans le partitionnement des données est le chargement du graphe d'entrée dans la base de données. Nous distinguons deux options pour cela :

(1) créer le graphe à partir de relations existantes dans la BDR,

(2) importer le graphe à partir d'un ensemble de données externe,

Plusieurs travaux [XKD15a, BK13, DVMT13] se sont intéressés par l'extraction des graphes à partir des relations entre les entités dans les BDR. Ces travaux peuvent être employés pour la lecture des graphes d'entrée. En revanche, l'import des graphes à partir des ensembles de données externes se fait à l'aide des procédures stockés dans le SGBDR utilisé. Par exemple, nous avons utilisé, dans ce travail, l'instruction COPY introduite par le SGBD utilisé (voir le chapitre 5) pour charger les données du graphe dans la base de données. Notez que le graphe introduit est non-orienté, par conséquent nous ajoutons une arête (j, i) pour chaque arête existante (i, j). La requête ci-dessous assure l'import du graphe dans la table d'arêtes E_s où chaque arête et sa direction opposée sont chargées :

SQL

```
CREATE TABLE E_s(i int,j int);
COPY E_s FROM "link/to/graph_data_set";
INSERT INTO E_s SELECT j,i FROM E_s;
```

Python Pandas

```
graph = pandas.DataFrame(index=None, columns=[[i,j]])
graph.read_csv('graph_data.csv')
```

Figure 3.11 présente un diagramme d'activité du chargement des données dans la table E_s.

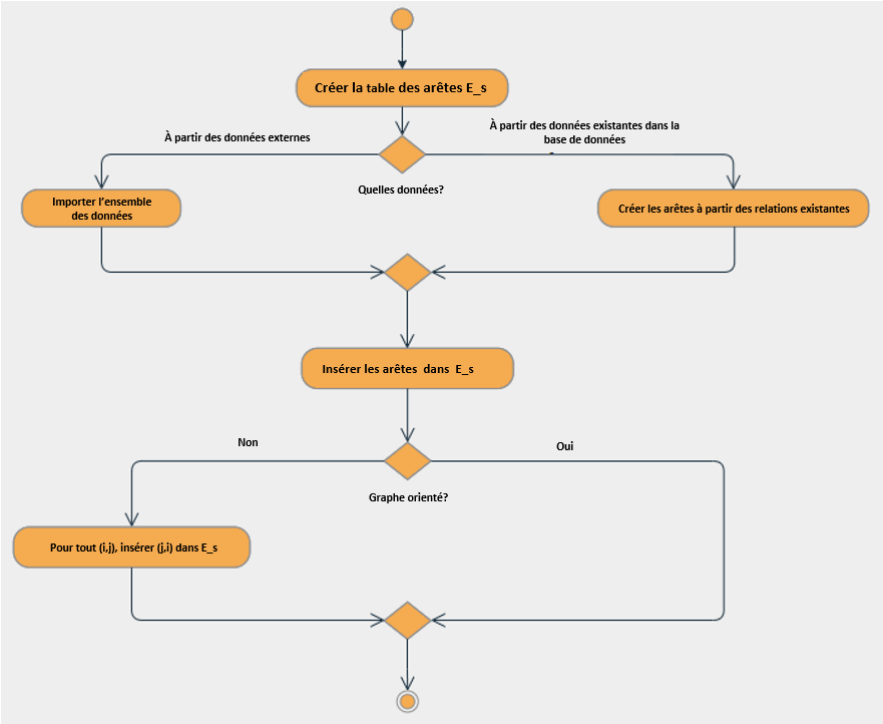


FIGURE 3.11 – Diagramme d'activité pour le chargement du graphe.

La deuxième étape est de créer la table Triplet qui permet d'attribuer aléatoirement à chaque machine un triplet de couleur unique formé des $c = k^{1/3}$ couleurs (voir Sec. 3.4.1). La requête SQL et l'expression Python suivantes représente cette affectation déterministe des triplets de couleurs :

SQL

```
/*UNSEGMENTED ALL NODE permet d'avoir une copie de la table sur chaque machine*/
CREATE TABLE Triplet(machine int,color1 int, color2 int, color3 int)
   UNSEGMENTED ALL NODES;
/*Un exemple d'affectation des triplets de couleurs sur un modele de 8 machines*/
INSERT INTO Triplet VALUES ((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))
```

Python Pandas

```
#Exemple des triplets de couleurs
t= {machine:[1,2,3,4,5,6,7,8],
    color1:[1,1,1,1,2,2,2,2],
    color2:[1,1,2,2,1,1,2,2],
    color3:[1,2,1,2,1,2,1,2]}
#Creation de la dataframe des triplets a partir des donnees de t
Triplet = pandas.DataFrame(t, index=None, columns=[[machine, color1, color2, color3]])
```

Selon l'algorithme randomisé, la stratégie du partitionnement repose sur le partitionnement de l'ensemble de sommets V en c sous-ensembles de n/c sommets chacun. Ceci est assuré par la création de la table V_s dans l'algorithme adapté. En effet, une sélection de tous les sommets du graphe est effectuée par une union entre les sommets sources et les sommets de destination de toutes les arêtes de la table E_s. Ceci permet d'insérer tous les sommets du graphe dans la table V_s. Ensuite, la fonction *randomint*[4] permet de générer aléatoirement et indépendamment une couleur unique parmi les c couleurs pour chaque sommet du graphe. Ces couleurs sont représentées par des entiers afin de faciliter le traitement. La création de la table V_s et l'insertion de ses données sont présentés par l'expression en algèbre relationnelle, en SQL et en Python.

Algèbre relationnelle

$$V\_s \leftarrow \pi_{[i,random(c)]}(\pi_i(E\_s) \cup \pi_j(E\_s) \tag{3.31}$$

SQL

```
/*Chaque sommet choisi une couleur aleatoire parmi les c couleurs*/
CREATE TABLE V_s(i int,color int);
INSERT INTO V_s
   SELECT i,randomint(c)+1
   FROM
      (SELECT DISTINCT i FROM E_s
      UNION
      SELECT DISTINCT j FROM E_s)V;
```

Python Pandas

```
#Reunir les sommets sources et de destination
V_s['i'] = pandas.concat([E_s['i'],E_s['j']], ignore_index=True)
#Eliminer les sommets redondants
V_s = V_s.drop_duplicates(keep='first')
#Attribuer pour chaque sommet une couleur aleatoire des c couleurs
V_s['color'] = V_s['i'].apply(lambda x: pandas.random.randint(1, k**(1./3.) + 1))
```

Figure 3.12 résume l'étape de création de la table V_s et la table Triplet sous forme d'un diagramme d'activité.

La coloration aléatoire des sommets du graphe est suivie par l'envoi des arêtes entre ces sommets aux machines proxys. Ceci est effectué par le biais de la table E_s_proxy. Une jointure entre la

4. Une procédure stockée définie sur le SGBD utilisé, qui permet de générer des entiers totalement aléatoires pour chaque sommet à chaque fois que la requête est exécutée.

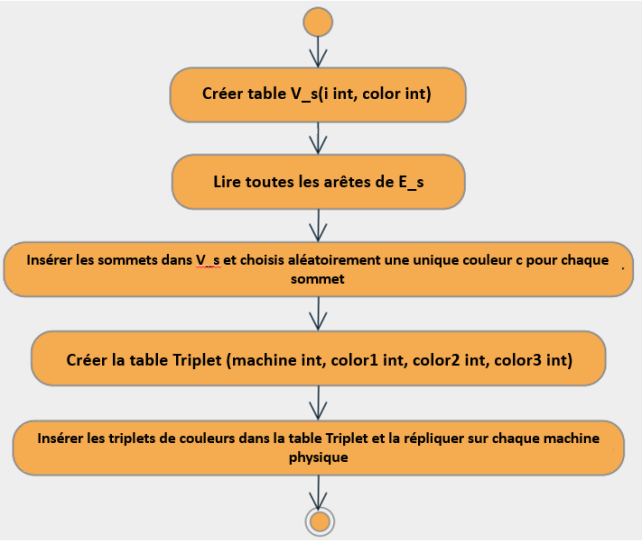


FIGURE 3.12 – Diagramme d'activité pour le partitionnement aléatoire de l'ensemble des sommets.

table E_s et la table V_s est alors exécutée afin d'associer les couleurs choisis par les sommets aux extrémités de chaque arête. Une répartition de la table E_s_proxy est effectuée par rapport aux couleurs des extrémités des arêtes, afin d'éviter la congestion vers une seule machines du modèle $k - machines$. Nous jugeons que cette étape est la plus importante dans la stratégie du partitionnement, étant donné que ce dernier est basé sur le remaniement des arêtes. Cette étape est traduite ci-dessous par une expression en algèbre relationnelle, en requête SQL et en code Python :

Algèbre relationnelle

$$E\_s\_proxy \leftarrow (E\_s \bowtie_{i=i} V\_s) \cup (E\_s \bowtie_{j=i} V\_s) \tag{3.32}$$

SQL

```
/*Envoi des aretes 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;
```

Python Pandas

```
#Associer les couleurs aux sommets sources de chaque arete
E_s_proxy = E_s.merge(V_s, on=['i'], how='inner')
#Associer les couleurs aux sommets de destination de chaque arete
E_s_proxy = E_s_proxy.merge(V_s, left_on=['j'], right_on=['i'], how='inner')
```

La dernière étape dans la stratégie du partitionnement de l'algorithme randomisé consiste à collecter des arêtes requises pour chaque machine locale. Cette collecte se fait à partir des machines proxys, tout en respectant le triplet de couleurs affecté à chaque machine locale. Dans l'algorithme adapté, la table E_s_local permet d'affecter à chaque machine ses arêtes correspondantes. Une jointure entre la table $Triplet$ et la table E_s_proxy est alors exécutée. Elle permet de vérifier si les extrémités d'une arête sont dans deux sous-ensembles de couleurs parmi les trois sous-ensembles de couleurs affectés à la machine. Enfin, une réunion des résultats permet d'envoyer toutes les arêtes à leurs machines correspondantes. L'expression en algèbre relationnelle, la requête SQL et le code Python de cette étape, sont présentés ci-dessous :

Algèbre relationnelle

$$\begin{aligned} E\_s\_local \leftarrow & (E\_s\_proxy \bowtie_{(i\_c=c\_1, j\_c=c\_2)} Triplet) \\ & \cup (E\_s\_proxy \bowtie_{(i\_c=c\_2, j\_c=c\_3)} Triplet) \\ & \cup (E\_s\_proxy \bowtie_{(i\_c=c\_3, j\_c=c\_1)} Triplet) \end{aligned} \tag{3.33}$$

SQL

```
/*Collect des aretes a 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
      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;
```

Python Pandas

```
E_s_local = pd.concat([E_s_proxy.merge(triplet, left_on=['i_color','j_color'],
    right_on=['color1','color2'], how='inner')[['machine','i','j','i_color','j_color']],
    E_s_proxy.merge(triplet, left_on=['i_color','j_color'], right_on=['color2','color3'],
    how='inner')[['machine','i','j','i_color','j_color']], E_s_proxy.merge(triplet,
    left_on=['i_color','j_color'], right_on=['color3','color1'],
    how='inner')[['machine','i','j','i_color','j_color']]], sort=False)
```

Le diagramme d'activité dans Figure 3.13 reprend les deux dernières étapes de la stratégie du partitionnement de l'algorithme adapté.

3.5.3.2 Énumération Locale des Triangles

Afin d'énumérer et compter localement les triangles dans chaque sous-graphe induit, chaque machine locale examine les triplets de sommets de son sous-graphe. Cette tâche est assurée par une double auto-jointure locale sur la table *E_s_local*, c-à-d $E1 \bowtie E2 \bowtie E3$ sur les colonnes $E1.j = E2.i$, $E2.j = E3.i$ et $E3.j = E1.i$ sur chaque machine localement et en parallèle. Notez que $E1$, $E2$ et $E3$ sont des tables alias de la table *E_s_local*. Chaque auto-jointure prendre en considération l'ordre des couleurs des extrémités de chaque triplet de sommets, qui doit correspondre au triplet de couleur affecté à la machine locale. Nous présentons ci-après l'expression de l'algèbre relationnelle, la requête SQL et le code Python de cette phase :

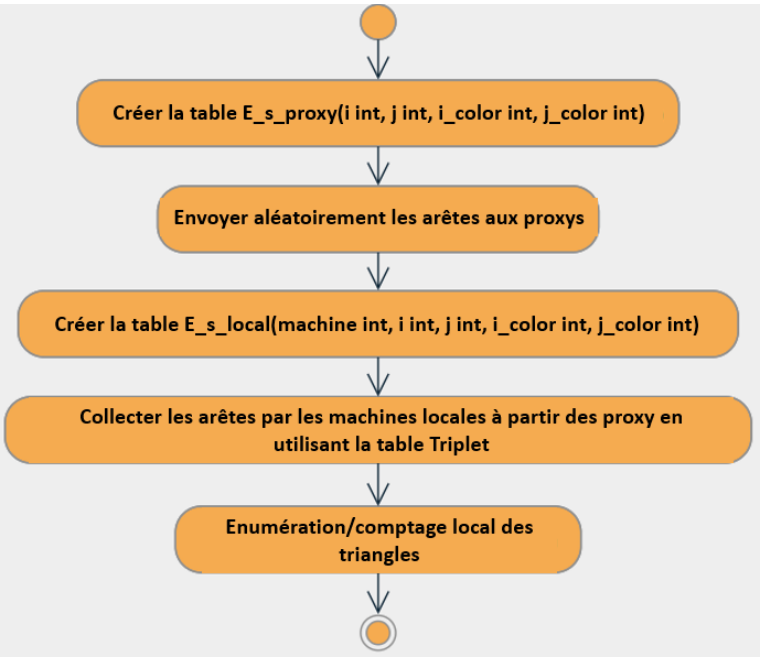


FIGURE 3.13 – Diagramme d'activité pour la collecte des arêtes et l'énumération locale des triangles.

Algèbre relationnelle

$$triangles \leftarrow \sigma_{E1.i<E1.j,E2.i<E2.j,E1.i=E3.j}((E1 \bowtie_{E1.j=E2.i} E2) \bowtie_{E2.j=E3.i} E3) \tag{3.34}$$

SQL

```sql
/*Enumeration des triangles*/
SELECT E1.machine as 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
   WHERE E1.i<E1.j AND E2.i<E2.j AND E1.i=E3.j;
/*Comptage des triangles*/
SELECT count(*)
   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
   WHERE E1.i<E1.j AND E2.i<E2.j AND E1.i=E3.j;
```

Python Pandas

```python
#Enumeration locale des triangles
triangles = E1.merge(E2, left_on=['j'], right_on=['i'], how='inner')
#v1 et v3 sont les extrimites des triades resultants de la precedante jointure
triangles = triangles.merge(E3, left_on=['v1','v3'], right_on=['j','i'], how='inner')
```

3.5.4 Exemple d'application

Nous reprenons, dans cette section, l'exemple de Figure 3.3(a), sur lequel nous exécutons l'algorithme adapté. La première étape est de charger le graphe dans la base de données. Le résultat

de l'exécution de cette requête est donné dans Figure 3.14.

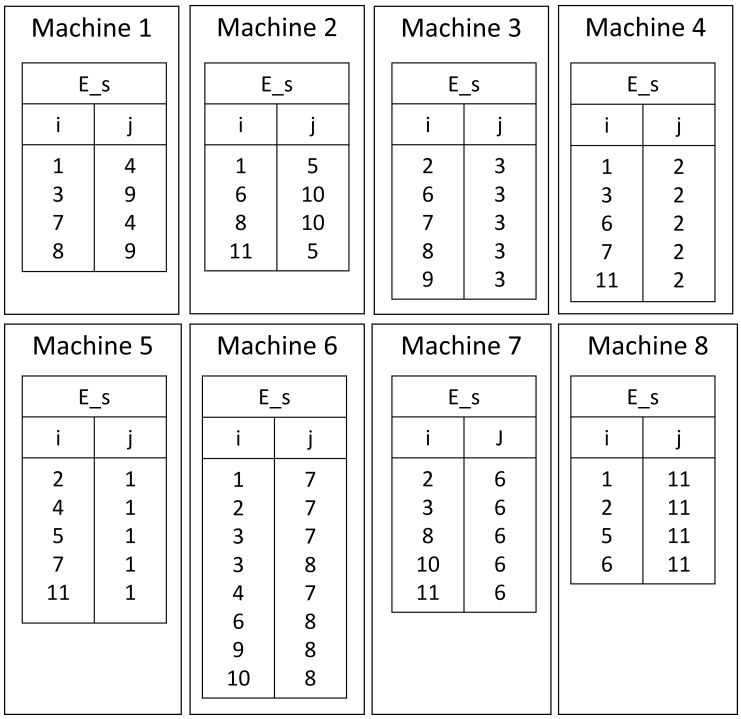


FIGURE 3.14 – Chargement du graphe dans la table *E_s*

Nous remarquons que la répartition des arêtes effectuée par le SGBD n'assure pas l'équilibrage de charge. Ainsi, une communication entre les machines est prévue afin d'échanger les triplets d'arêtes formant des triangles. De ce fait, un répartitionnement d'arêtes s'impose pour réduire les coûts de communication durant la tâche d'énumération et du comptage des triangles. Pour cela, la création de la table Triplet est de priorité. Comme mentionné précédemment, cette table doit être remplie manuellement par l'utilisateur. Nous présentons dans Table 3.8, un exemple de la table Triplet. À noter que l'attribution des triplets de couleurs est effectuée d'une manière aléatoire. L'exemple donné dans Table 3.8 ne représente qu'une seule possibilité d'affectation.

machine	color1	color2	color3
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

TABLEAU 3.8 – Affectation des triplets de couleurs aux machines.

Nous poursuivons le partitionnement par la création de la table *V_s*. Cette dernière permet de stocker les sommets avec leurs couleurs choisies aléatoirement. Le résultat de l'exécution de la requête correspondante est donné dans Figure. 3.15. Le partitionnement de la table V_s sur le modèle $k - machines$ est effectué par rapport à la colonne i. Ceci est par le biais de la stratégie du partitionnement intégrée dans le SGBD.
Les tables *V_s* et *E_s* sont utilisées pour remplir la table *E_s_proxy*, qui représente l'envoi des arêtes aux machines proxys. En effet, comme expliqué dans la section précédente, la table *E_s_proxy* est le résultat de la jointure entre la table *E_s* et la table *V_s*. Figure 3.16 illustre le

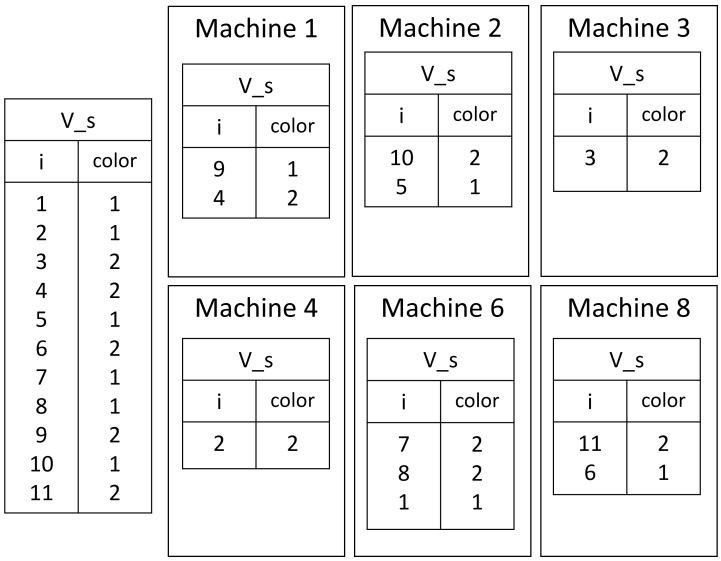


FIGURE 3.15 – Partitionnement des sommets dans V_s.

résultat de la jointure entre ces deux tables décrites dans les Figures 3.14 et 3.15. Notez que le partitionnement de la table *E_s_proxy* sur le modèle $k - machines$ est effectué en fonction des colonnes *i_color* et *j_color*, afin d'éviter la congestion vers une seule machine.
Enfin, Figure 3.17 présente le résultat de l'exécution de la requête SQL, permettant ainsi à chaque machine locale de collecter ses arêtes manquantes.
Après avoir appliqué le modèle RVP sur le graphe de l'exemple, il ne reste qu'à énumérer et compter localement les triangles sur chaque machine. Le résultat est donné dans Figure 3.18. Nous remarquons que chaque machine a renvoyé presque le même nombre de triangles.

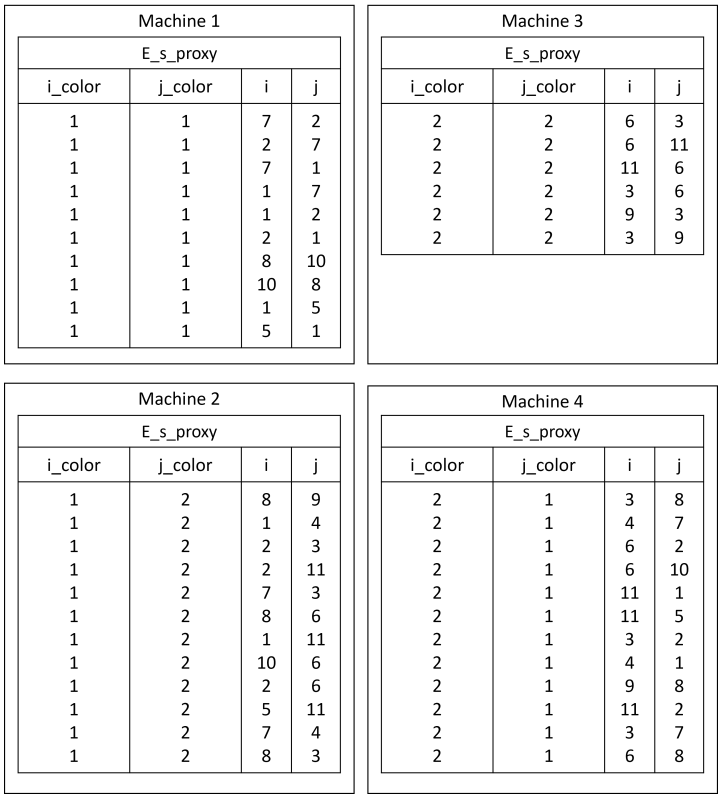
3.5.5 Équilibrage de charge de l'algorithme adapté

Dans le SGBD, le partitionnement s'effectue en LDD [5] à travers la définition de la clause de la segmentation [6], dans la requête de création de la table. En fait, le SGBD attribue à chaque machine un segment de valeurs de hachage. La valeur de hachage de chaque tuple est calculée à l'aide de la clause de segmentation. Puis, le tuple est envoyé à la machine possédant cette valeur de hachage dans son segment.
Nous avons défini une clause de segmentation dans la création de la table *E_s_local*, qui permet d'assurer une répartition équilibrée d'arêtes sur l'ensemble des machines physiques. Plus précisément, la segmentation des données s'effectue en utilisant la colonne *machine*. Par conséquent, chaque tuple de la table *E_s_local* est envoyé à la machine physique, possédant le même identifiant figurant dans la colonne *machine*. Ainsi, chaque machine aura que ses arêtes correspondantes et l'équilibrage de charge discuté dans la Section 3.4.4 est assuré.

3.5.6 Complexité de l'algorithme adapté

La complexité temporelle de l'algorithme adapté *peut être légèrement supérieure* à la complexité de l'algorithme randomisé dans (3.29), à cause du coût des jointures. Dans la suite, nous examinerons en détail la complexité de l'algorithme adapté tout en identifiant les requêtes les plus gourmandes en ressources.
Rappelons qu'il y a quatre étapes principales dans l'algorithme randomisé d'énumération et de comptage de triangles. Notre étude de complexité temporelle concerne principalement les

5. *Langage de Définition de Données*
6. La segmentation est une terminologie qui fait référence à la répartition des données sur les machines physiques.



Machine 1

E_s_proxy

i_color	j_color	i	j
1	1	7	2
1	1	2	7
1	1	7	1
1	1	1	7
1	1	1	2
1	1	2	1
1	1	8	10
1	1	10	8
1	1	1	5
1	1	5	1

Machine 3

E_s_proxy

i_color	j_color	i	j
2	2	6	3
2	2	6	11
2	2	11	6
2	2	3	6
2	2	9	3
2	2	3	9

Machine 2

E_s_proxy

i_color	j_color	i	j
1	2	8	9
1	2	1	4
1	2	2	3
1	2	2	11
1	2	7	3
1	2	8	6
1	2	1	11
1	2	10	6
1	2	2	6
1	2	5	11
1	2	7	4
1	2	8	3

Machine 4

E_s_proxy

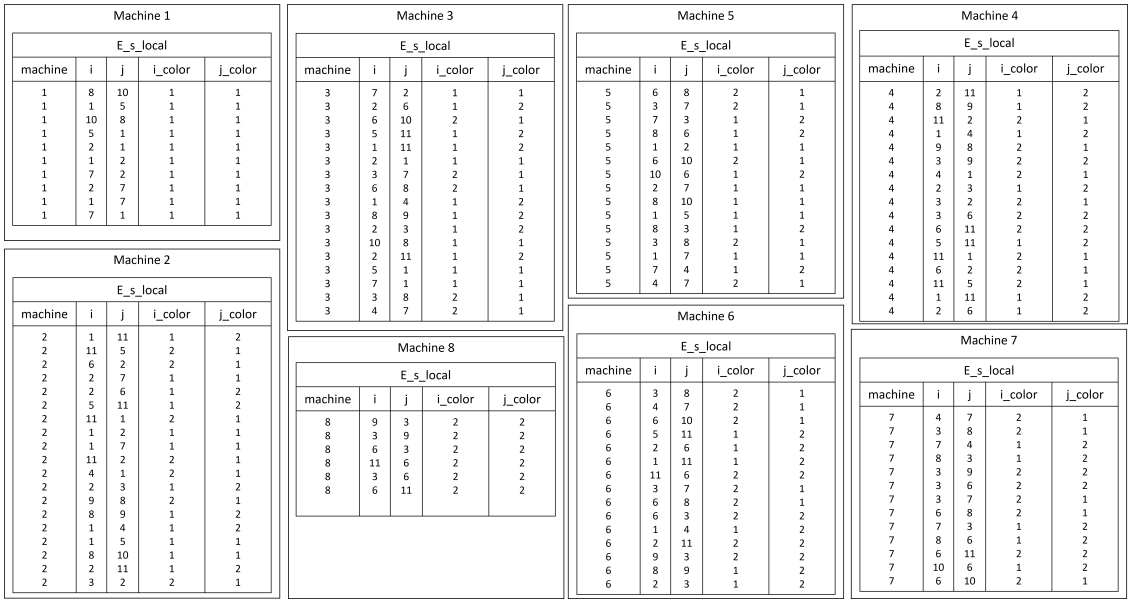
i_color	j_color	i	j
2	1	3	8
2	1	4	7
2	1	6	2
2	1	6	10
2	1	11	1
2	1	11	5
2	1	3	2
2	1	4	1
2	1	9	8
2	1	11	2
2	1	3	7
2	1	6	8

FIGURE 3.16 – Envoie des arêtes aux proxys sur la table *E_s_proxy*.

opérations de jointure dans l'algorithme adapté. En effet, nous avons constaté que les requêtes les plus chronophages sont : (1) la requête d'insertion dans la table *E_s_local*, et (2) la requête de comptage et d'énumération de triangles. Elles sont coûteuses car elles font appel à plusieurs opérations de jointure.
D'abord, la première étape de l'algorithme adapté s'agit de la requête de la création de la table V_s, dans laquelle nous attribuons une couleur aléatoire à chaque sommet présent dans la table E_s. Pour obtenir l'ensemble des sommets V, nous sélectionnons toutes les valeurs distinctes des deux colonnes i et j de la table E_s et les unissons ensemble. Ainsi, la table E_s sera lue deux fois dans cette requête. Notez qu'il existe un algorithme de tri derrière SELECT DISTINCT. Vue que la table est déjà triée par la colonne i, nous n'avons besoin de scanner la colonne i sur chaque machine que lors de l'exécution de SELECT DISTINCT i avec une complexité temporelle de $O(m)$. Cependant, la colonne j n'est pas triée donc un algorithme de tri est exécuté lors de l'exécution de SELECT DISTINCT j. La complexité temporelle du tri est de :

$$O(m \cdot log(m)) \tag{3.35}$$

Ensuite, les arêtes sont envoyées aux proxys via la création de la table *E_s_proxy*. La requête inclut deux jointures entre la table E_s et la table V_s. L'étape basique est d'établir une jointure entre E_s et V_s, puis la table résultante dérivée est jointe avec V_s une fois de plus. Ainsi, la table V_s est lue deux fois et la table E_s est lue une seule fois dans cette étape. Communément, les algorithmes de jointure peuvent être soit une jointure par hachage, soit une jointure par fusion dans le SGBD. La complexité temporelle de l'opérateur de jointure est de $O(m \cdot log(m))$ pour une jointure de tri-fusion si l'une des deux tables de jointure est triée, et elle devient $O(m)$ si les deux tables de jointure sont triés. La jointure par fusion est utilisée lorsque les tables jointes sont triées sur les colonnes de jointure. La jointure par hachage, quant à elle, est utilisée lorsque les tables jointes ne sont pas triées sur les colonnes de jointure, ce qui est le cas pour une auto-jointure. Par conséquent, l'optimiseur crée une table de hachage et la charge dans



Machine 1

E_s_local

machine	i	j	i_color	j_color
1	8	10	1	1
1	1	5	1	1
1	10	8	1	1
1	5	1	1	1
1	2	1	1	1
1	1	2	1	1
1	7	2	1	1
1	2	7	1	1
1	1	7	1	1
1	7	1	1	1

Machine 2

E_s_local

machine	i	j	i_color	j_color
2	1	11	1	2
2	11	5	2	1
2	6	2	2	1
2	2	7	1	1
2	2	6	1	2
2	5	11	1	2
2	11	1	2	1
2	1	2	1	1
2	1	7	1	1
2	11	2	2	1
2	4	1	2	1
2	2	3	1	2
2	9	8	2	1
2	8	9	1	2
2	1	4	1	2
2	1	5	1	1
2	8	10	1	1
2	2	11	1	2
2	3	2	2	1

Machine 3

E_s_local

machine	i	j	i_color	j_color
3	7	2	1	1
3	2	6	1	2
3	6	10	2	1
3	5	11	1	2
3	1	11	1	2
3	2	1	1	1
3	3	7	2	1
3	6	8	2	1
3	1	4	1	2
3	8	9	1	2
3	2	3	1	2
3	10	8	1	1
3	2	11	1	2
3	5	1	1	1
3	7	1	1	1
3	3	8	2	1
3	4	7	2	1

Machine 8

E_s_local

machine	i	j	i_color	j_color
8	9	3	2	2
8	3	9	2	2
8	6	3	2	2
8	11	6	2	2
8	3	6	2	2
8	6	11	2	2

Machine 5

E_s_local

machine	i	j	i_color	j_color
5	6	8	2	1
5	3	7	2	1
5	7	3	1	2
5	8	6	1	2
5	1	2	1	1
5	6	10	2	1
5	10	6	1	2
5	2	7	1	1
5	8	10	1	1
5	1	5	1	1
5	8	3	1	2
5	3	8	2	1
5	1	7	1	1
5	7	4	1	2
5	4	7	2	1

Machine 6

E_s_local

machine	i	j	i_color	j_color
6	3	8	2	1
6	4	7	2	1
6	6	10	2	1
6	5	11	1	2
6	2	6	1	2
6	1	11	1	2
6	11	6	2	2
6	3	7	2	1
6	6	8	2	1
6	6	3	2	2
6	1	4	1	2
6	2	11	2	2
6	9	3	2	2
6	8	9	1	2
6	2	3	1	2

Machine 4

E_s_local

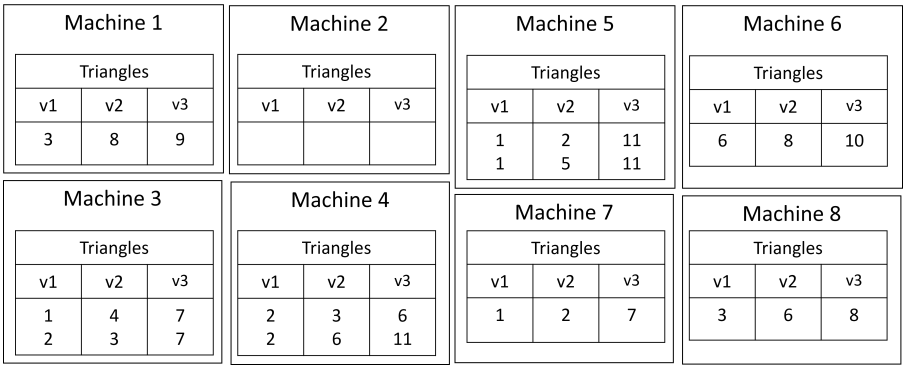
machine	i	j	i_color	j_color
4	2	11	1	2
4	8	9	1	2
4	11	2	2	1
4	1	4	1	2
4	9	8	2	1
4	3	9	2	2
4	4	1	2	1
4	2	3	1	2
4	3	2	2	1
4	3	6	2	2
4	6	11	2	2
4	5	11	1	2
4	11	1	2	1
4	6	2	2	1
4	11	5	2	1
4	1	11	1	2
4	2	6	1	2

Machine 7

E_s_local

machine	i	j	i_color	j_color
7	4	7	2	1
7	3	8	2	1
7	7	4	1	2
7	8	3	1	2
7	3	9	2	2
7	3	6	2	2
7	3	7	2	1
7	6	8	2	1
7	7	3	1	2
7	8	6	1	2
7	6	11	2	2
7	10	6	1	2
7	6	10	2	1

FIGURE 3.17 – Répartition finale du graphe dans la table *E_s_local*.



Machine 1

Triangles

v1	v2	v3
3	8	9

Machine 2

Triangles

v1	v2	v3

Machine 5

Triangles

v1	v2	v3
1	2	11
1	5	11

Machine 6

Triangles

v1	v2	v3
6	8	10

Machine 3

Triangles

v1	v2	v3
1	4	7
2	3	7

Machine 4

Triangles

v1	v2	v3
2	3	6
2	6	11

Machine 7

Triangles

v1	v2	v3
1	2	7

Machine 8

Triangles

v1	v2	v3
3	6	8

FIGURE 3.18 – Résultat d'énumération de triangles.

la mémoire principale afin de pouvoir rechercher les correspondances entre la table externe et la table de hachage. La complexité temporelle de la jointure par hachage est de $O(m^2)$ dans le pire des cas, en particulier avec des données fortement asymétriques (cliques). En revanche, la moyenne estimée de sa complexité temporelle est de $O(m)$, en supposant une distribution uniforme du graphe avec des sommets de faible degré. Dans cette requête, la table *E_s* et la table *V_s* sont répartitionnées par la colonne i et triées par la même colonne sur chaque machine. Ainsi, l'implémentation de la première jointure est la jointure par fusion avec une complexité temporelle de $O(m)$, car les tables de jointure sont triées par la colonne de jointure. Cependant, la deuxième condition de jointure est $E\_s.j = V\_s.i$, qui n'est pas la même que la colonne de tri. Une resegmentation s'avère nécessaire, en considérant une jointure par hachage. Comme expliqué précédemment, la complexité temporelle de la jointure par hachage est de $O(m)$ en moyenne, mais peut être de $O(m^2)$ dans le pire des cas.

Ensuite, les arêtes sont collectées à partir des proxys via la création de la table *E_s_local*. Lors de l'insertion dans cette table, trois opérations de jointure entre la table *Triplet* et la table *E_s_proxy* sont effectuées, et les résultats sont réunis ensemble avec deux opérations d'union. Par conséquent, la table *E_s_proxy* est lue trois fois. Ces trois opérations de jointure sont toutes des jointures par hachage, car les tables de jointure ne sont pas triées par les colonnes de jointure. Il est important de rappeler que la petite table *Triplet* est répliquée sur toutes les machines physiques du cluster. Cette table demeure en mémoire principale pendant tout le processus de

jointure. Chaque fois qu'un enregistrement de la table *E_s_proxy* est extrait, il est joint avec la table *Triplet* dans son intégralité. Ainsi, tous les enregistrements de la table *E_s_proxy* sont récupérés une fois dans chaque jointure, et la table *Triplet* reste dans la mémoire principale et est utilisé à plusieurs reprises. La complexité temporelle de la jointure par hachage pour cette requête est de $O(m)$, sachant que la taille de *E_s_proxy* est $O(m)$.
Enfin, la table *E_s_local* est jointe avec elle-même deux fois (auto-jointure) dans la requête d'énumération et de comptage des triangles. Donc, elle est lue deux fois. Dans la première jointure, $E1$ est joint avec $E2$ sur $E1.machine = E2.machine$ et $E1.j = E2.i$. Rappelons que la table *E_s_local* est créée de manière à ce que chaque machine locale ait les arêtes requises pour traiter localement le comptage et l'énumération de triangles. Ainsi, aucune segmentation n'est nécessaire pour la première opération de jointure. Dans l'étape suivante, la table dérivée est jointe avec $E3$ sur $E2.machine = E3.machine$, $E2.j = E3.i$ et $E1.i = E3.j$. De même, aucune segmentation n'est nécessaire pour la deuxième opération de jointure. Vu que Ces deux opérations de jointure sont des auto-jointures, l'algorithme utilisé est l'algorithme de jointure par hachage. Étant donné que la table est segmentée sur l'ensemble des machines du cluster, et qu'aucune arête n'est communiquée entre les machines, la jointure par hachage pourrait être parfaitement parallélisée. Ceci est grâce à notre stratégie de partitionnement qui a éliminé toutes les communications entre les machines durant ces opérations de jointure. Supposons que la taille de la table *E_s_local* est $O(m)$, la complexité temporelle de l'opération de jointure est $O(m/k)$ pour un arbre et $O(m^2/k)$ pour un graphe fortement asymétrique où k est le nombre de machines dans le cluster.
En résumé, la table *E_s* est lue quatre fois, la table *E_s_proxy* est lue trois fois et la table *E_s_local* est lue deux fois pendant tout le processus. La taille de toutes ces tables est de $O(m)$. Ainsi, le coût des E/S est de $O(9m)$, c'est-à-dire

$$O(m) \tag{3.36}$$

En se basant sur cette étude de complexité, nous avons constaté qu'élargir le modèle $k-machines$ ne garantit pas une accélération linéaire. Effectivement, afin d'exécuter localement la requête de comptage et d'énumération des triangles, un rééquilibrage de charge doit être effectué via le répartitionnement du graphe sur l'ensemble des machines. Cette communication entre les machines du modèle est coûteuse, lorsque le graphe est de petite à moyenne taille alors que le modèle $k - machines$ est très large.

3.6 Conclusion

À travers ce chapitre, nous avons présenté notre contribution qui consiste principalement en une étude d'adaptation et d'optimisation de l'algorithme randomisé d'énumération et de comptage de triangles avec les requêtes SQL et avec Python Pandas. Dans un premier lieu, nous avons présenté théoriquement l'algorithme randomisé tout en détaillons ses deux phases principales : (1) le partitionnement du graphe, et (2) l'énumération locale des triangles. Nous avons ensuite, discuté l'exactitude des résultats de l'algorithme, où nous avons prouvé que ce dernier est de type Las Vegas (c-à-d. toujours correcte). Nous avons également analysé l'équilibrage de charge parfait assuré par cet algorithme. En effet, nous avons affirmé que chaque machine renvoie $1/k$ du total des triangles dans le graphe, tout en préservant un traitement local lors de la phase d'énumération des triangles. Nous avons, enfin, discuté la complexité de l'algorithme randomisé qui est $\max\{\tilde{O}(m/k^{2/3}, n/k^{1/3})\}$ avec une probabilité élevée.
Dans un second lieu, nous nous sommes focalisés sur l'implémentation et l'optimisation de l'algorithme adapté pour l'énumération et le comptage des triangles. D'abord, nous avons élaboré un modèle conceptuel de données permettant l'exécution de l'algorithme randomisé sur une base de données relationnelle. Le modèle proposé consiste en cinq tables qui permettent la sauvegarde

des résultats de chaque étape de l'algorithme. Nous avons ensuite repris les étapes de l'algorithme randomisé et présenté les requêtes SQL et les expressions de Pandas les traduisant. À chaque étape, nous avons fourni une expression en algèbre relationnelle, une requête SQL et une expression en Python Pandas, tout en précisant quelle table du modèle conceptuel à utiliser. En outre, nous avons illustré notre solution avec un exemple d'un graphe de 11 sommets. Cet exemple nous a permis de donner un aperçu des résultats de chaque étape de l'algorithme adapté. Ensuite, nous avons examiné l'équilibrage de charge assuré par nos requêtes, suivi d'une étude de la complexité de ces requêtes. Nous avons conclu que la complexité moyenne est de l'ordre de $O(m)$.
Le chapitre suivant fera l'objet d'une démonstration d'une application Web que nous avons développée dans le cadre de cette thèse. Nous allons présenter l'architecture physique et logique de l'application, ainsi qu'une démonstration détaillée pour deux types d'utilisateurs : (1) débutants, et (2) experts.

Chapitre 4

PandaSQL : Énumération randomisée et parallèles des triangles à base des requêtes SQL

Hiding within those mounds of data is the knowledge that could change the life of a patient, or change the world.

— Atul Butte

Sommaire

4.1 Introduction

Dans l'introduction générale, nous avons exposé quelques applications réelles qui font appel à l'analyse du problème des triangles. Dans cette perspective, nous avons développé ***PandaSQL***[FBO+20b] ; une application Web qui permet d'analyser le problème d'énumération et de comptage de triangles avec l'algorithme adapté, ainsi de comparer ses résultats avec une approche classique d'énumération de triangles. Cette application implémente les deux algorithmes en utilisant les requêtes SQL sur un SGBDR.
Dans ce chapitre, nous présentons l'outil ***PandaSQL*** avec ses différentes fonctionnalités. Nous décrivant d'abord l'architecture physique et logique de notre application, suivi d'une présentation de l'algorithme de référence utilisé pour la comparaison avec les résultats produits par l'algorithme adapté dans notre outil. Ensuite, une démonstration détaillée de ce dernier est fournie pour en prenant en compte le niveau de l'utilisateur : (1) une démonstration basique pour les débutants dans les SGBD, et (2) une démonstration approfondie pour les experts. Ce chapitre se conclut par une discussion des avantages et des limitations de notre application.

4.2 Architecture physique

Rappelons que notre système se base sur le modèle $k - machines$, qui est utilisé pour le calcul distribué et parallèle. Pour cela, nous avons opté pour une architecture physique de type client/serveur pour le déploiement de notre outil. Précisément, le client est un client léger sous forme d'une application Web, et le serveur prend en charge la base de données et le serveur web. Le serveur est construit sur une architecture pair-à-pair composée de k machine. Dans notre système, nous avons opté pour un cluster de 8 machines, néanmoins cela peut facilement être adapté selon la taille du cluster disponible, si ce dernier respecte la règle du modèle $k-machines$ défini dans 3.2 ou 3.4.
Le choix de cette architecture est justifié par le fait que notre solution est fondée principalement sur l'exécution de certaines requêtes SQL. Le fait que le serveur Web est peu sollicité, nous avons décidé de mettre le serveur Web et le serveur de base de données sur le même serveur (un cluster de machine).
Figure 4.1 donne un aperçu général de l'architecture physique adoptée par notre système. Initialement, le client soumet la requête au SGBD distribué, via son API de connecteur de base de données correspondante. La machine recevant la requête devient un nœud initiateur, tandis que les autres machines demeurent des nœuds exécuteurs. Le nœud initiateur choisit un plan d'exécution, et le partage avec les exécuteurs. Puis, toutes les machines exécutent la requête (durant le rééquilibrage de charge, les données peuvent être échangées entre les machines par passage de message, en fonction de la requête et de la distribution des données). Enfin, le résultat de la requête est envoyé au client pour l'affichage.

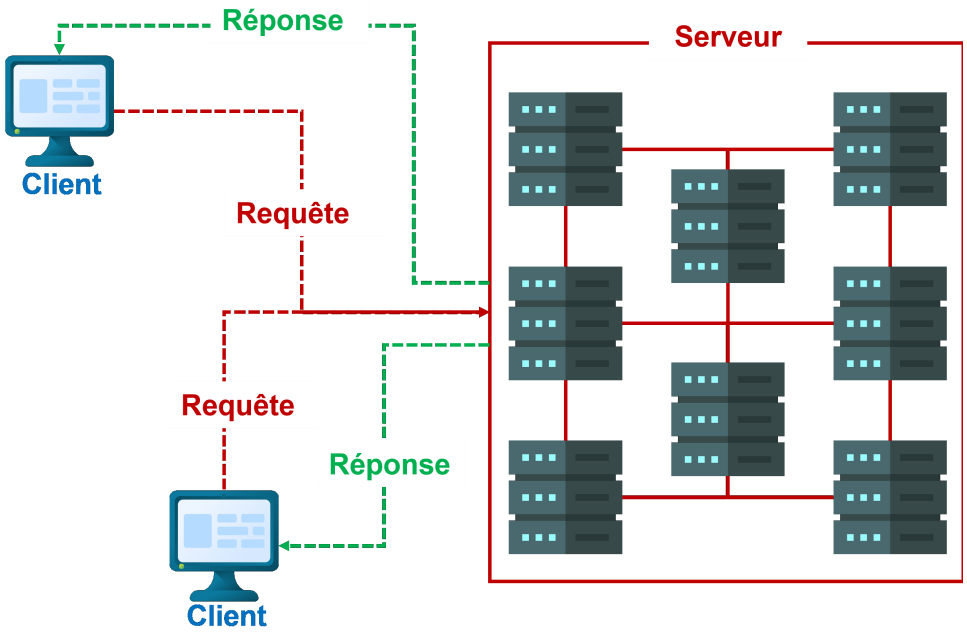


FIGURE 4.1 – Architecture physique de *PandaSQL*.

4.3 Architecture logique

Pour développer notre outil, nous avons choisi l'architecture logique MVC. Cette architecture permet la séparation entre la logique métier et l'affichage des données/résultats, ce qui garantit une meilleure maintenance de l'outil. Cette architecture est principalement composée de trois parties :

(1) Modèle : permet de gérer les données et la logique métier.

(2) Vue : sert pour la gestion de l'affichage.

(3) Contrôleur : intermédiaire entre le modèle et la vue, il achemine les commandes entre les deux parties.

Le diagramme de classes présenté dans Figure 4.2 présente la conception logique adopté pour l'outil PandaSQL. Il est composé de quatre classes : (1) DB_model qui permet de représenté le modèle dans l'architecture MVC, (2) Présentation qui représente la vue, (3) Controller qui permet la communication entre le modèle et la vue, et (4) Main qui est la classe principale permettant d'instancier les différentes classes.
Nous présentons la description des différentes opérations des classes dans Table 4.1.

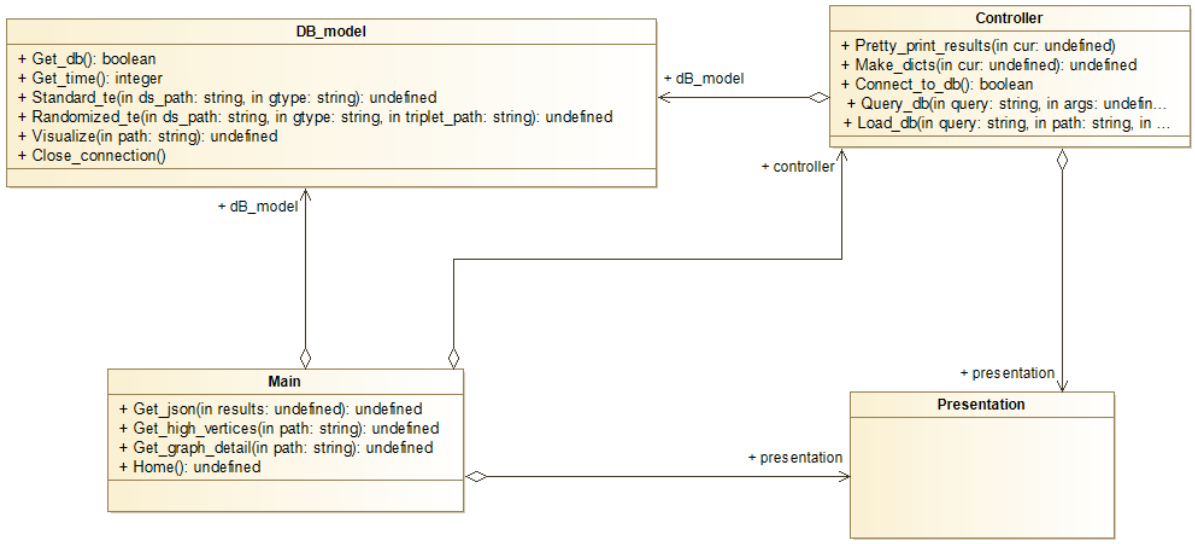


FIGURE 4.2 – Diagramme de classe de *PandaSQL*

TABLEAU 4.1 – Description des opétaions du diagramme de classes

Opération	Description
Classe : DB_model	
Get_db()	Permet de vérifier si la connexion à la base de données est établie.
Get_time()	Permet de renvoyer le temps d'exécution d'une requête sur la base de données.
Classic_te()	Lance les requêtes d'exécution de l'algorithme classique d'énumération de triangles.
Randomized_te()	Lance les requêtes d'exécution de l'algorithme randomisé d'énumération de triangles.
Close_connection()	Permet de fermer la connexion à la base de données après l'exécution des requêtes.
Classe : Contrôler	
Make_dict()	Transforme les résultats renvoyés par une requête sous forme d'un dictionnaire (clé,valeur).
Connect_to_db()	Initialise une connexion à la base de données.
Query_db()	Initialise un requête SQL, elle est utilisé par Classic_te() et Randomized_te().
Load_db()	Permet de charger un graphe dans la base de données.
Get_high_vertices()	Récupère les top 15 des sommets qui ont le degré le plus élevé pour l'affichage.
Classe : Main	
Get_json()	Transforme les résultats en une forme json pour l'affichage.
Get_graph_detail()	Récupère la taille et l'ordre du graphe introduit.
Home()	Permet de gérer l'affichage.

4.4 PandaSQL

Le système PandaSQL [FBO+20b] permet de compter les triangles intégrés dans les graphes à grande échelle en utilisant les requêtes SQL. Il utilise l'algorithme adapté présenté dans le chapitre précédent et compare ses résultats avec l'algorithme classique de comptage et d'énumération de triangle. Ce dernier est inspiré de l'algorithme de [Coh09] et reproduit avec des requêtes SQL.

4.4.1 Algorithme classique d'énumération et de comptage de triangles

Nous avons présenté plusieurs techniques pour l'énumération et le comptage de triangles dans le chapitre 2. Parmi ces approches, nous avons cité l'approche de [Coh09], qui permet d'énumérer les triangles en utilisant MapReduce dans l'environnement Hadoop. Rappelons les étapes de cette approche :

(1) Lister tous les triades ouverts, c-à-d deux arêtes qui ont un sommet en commun,

(2) Pour chaque triade, vérifier s'il existe une arête dont les extrémités sont les mêmes que les extrémités du triade.

Du point de vue algorithmique, cette approche peut être traduite par trois boucles imbriquées, où la première permet d'extraire tous les triades ouverts, en revanche la deuxième vérifie s'il existe une arête pour fermer le triade et former un triangle (voir Algorithme 4).

Algorithm 4 Algorithme classique d'énumération de triangles.

Input: $G = (V, E)$
Output: $\Delta(G)$
Graph reading
for $e_1 \leftarrow E$ **do**
  for $e_2 \leftarrow E$ **do**
    for $e_3 \leftarrow E$ **do**
      if $(e_1.j == e_2.i \wedge e_2.j == e_3.i \wedge e_3.j == e_1.i)$ **then**
        $\Delta(G) \leftarrow \Delta(G) \cup (e_1, e_2, e_3)$
      end
    end
  end
end

Dans notre système, ceci peut être traduit par une double auto-jointure sur la table des arêtes E_s. Autrement dit, $E1 \bowtie E2 \bowtie E3$ sur $E1.j = E2.i$, $E2.j = E3.i$ et $E3.j = E1.i$ respectivement avec $E1$, $E2$, $E3$ des tables alias de E_s. À noter que pour éliminer les résultats redondants, nous considérons que les triangles dont les sommets respectent un ordre lexicographique entre eux, c-à-d du plus petit ID au plus grand ID ($v_1 < v_2 < v_3$). Formellement, nous présentons ci-dessous une expression en algèbre relationnelle, et en requête SQL pour exprimer l'algorithme classique :

Algèbre relationnelle

$$\Delta(G) \leftarrow \sigma_{E1.i<E1.j,E2.i<E2.j,E1.i=E3.j}((E1 \bowtie_{E1.j=E2.i} E2) \bowtie_{E2.j=E3.i} E3) \tag{4.1}$$

SQL

```
/*La requete d'enumeration de triangles avec E_dup est une replique de E_s*/
SELECT E1.i As v1,E1.j AS v2,E2.j As v3
   FROM
      E_s E1 JOIN E_dup E2 ON E1.j=E2.i
```

```
      JOIN E E3 ON E2.j=E3.i
   WHERE E1.i<E1.j AND E2.i<E2.j AND E3.j=E1.i;

/*La requete de comptage de triangles avec E_dup est une replique de E_s*/
SELECT COUNT(*)
   FROM
      E_s E1 JOIN E_dup E2 ON E1.j=E2.i
      JOIN E E3 ON E2.j=E3.i
   WHERE E1.i<E1.j AND E2.i<E2.j AND E3.j=E1.i;
```

4.4.2 Algorithme randomisé d'énumération et de comptage de triangles

Nous rappelons les étapes de l'algorithme randomisé d'énumération et de comptage de triangles à travers Figure 4.3.

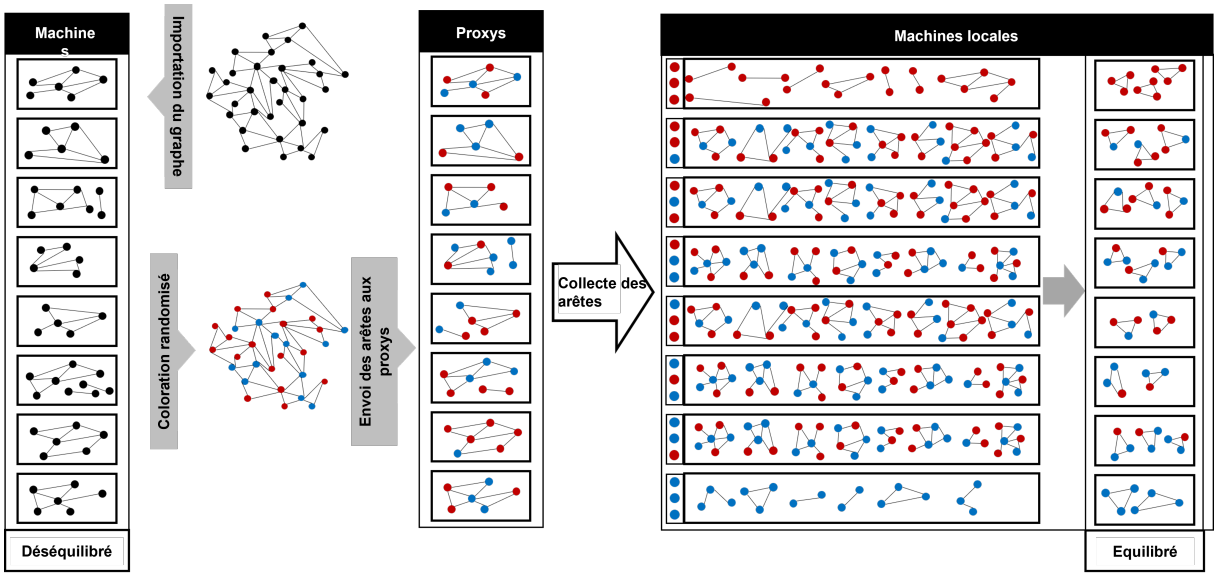


FIGURE 4.3 – Etapes de l'algoithme randomisé.

Figure 4.3 représente un exemple de l'algorithme randomisé sur un modèle de $k - machines$ où $k = 8$. L'étape initiale est le chargement du graphe sur le système, suivi par la coloration de ses sommets. Ensuite, les arêtes entre chaque couple de sommets sont envoyés aux machines proxys. En parallèle, chaque machine local reçois un triplet de couleurs qui lui permet de collecter ses arêtes (les extrémités de ses arêtes collectés doivent être dans deux sous-ensembles de couleurs parmi les trois sous-ensembles de couleurs affectés à elle). Enfin, les triangles sont énuméré localement vue que la redistribution des arêtes est équilibrée. Rappelons que l'algorithme adapté est fondé sur l'algorithme randomisé, il est utilisé dans notre outil PandaSQL pour énumérer et compter les triangles dans les graphes introduits, avec les requêtes SQL.

4.4.3 Comparaison entre l'algorithme adapté et l'algorithme classique

La différence majeure entre l'algorithme adapté et l'algorithme classique pour l'énumération et le comptage des triangles est la stratégie du partitionnement sur laquelle repose chaque algorithme. En effet, l'algorithme adapté se base sur le partitionnement aléatoire de l'ensemble des sommets V. Ceci est en utilisant un ensemble de couleurs généré à partir du nombre des machines disponibles pour le traitement. Ensuite, un remaniement des arêtes est effectué afin de construire des sous-graphes, où chacun porte une portion équivalente du total des triangles du graphe. Ceci permet d'éliminer tout échange d'arêtes entre les machines durant la tâche

d'énumération et de comptage de triangles. En revanche, l'algorithme classique ne définit aucune stratégie de partitionnement, donc il repose principalement sur la stratégie du partitionnement utilisé par le système sur lequel il est exécuté. Dans notre cas, il utilise la stratégie prédéfinie du partitionnement du SGBD utilisé. Cette stratégie ne garantit pas l'équilibrage de charge pour le traitement du problème des triangles.
À travers notre outil PandaSQL, nous voudrions étudier l'impact de la stratégie du partitionnement randomisé (le modèle RVP) sur l'équilibrage de charge, le temps d'exécution et l'exactitude des résultats renvoyés. Ainsi, l'algorithme classique sert pour une référence triviale pour faire cette étude et d'en tirer les conclusions.

4.5 Démonstration de l'outil

Comme expliqué dans l'introduction, l'énumération des triangles est fondamentale pour résoudre des problèmes de graphes plus complexes. Par conséquent, notre objectif majeur à travers cette démonstration est d'illustrer comment un algorithme distribué aléatoire intelligent peut fonctionner au-dessus d'un SGBD traditionnel, dans l'espérance que notre système pourra ouvrir la possibilité de programmer d'autres algorithmes aléatoires avec des requêtes.

4.5.1 Objectifs de la Démonstration

Nos principaux objectifs, allant des aspects logiques de la base de données aux aspects physiques et de traitement, sont les suivants :

(1) Donner un aperçu du modèle de calcul parallèle, en le contrastant par rapport au nombre de machines (nœuds de traitement),

(2) mettre en évidence les avantages de SQL par rapport à C++ ou Java (plus lent, mais plus évolutif et bénéficiant des fonctionnalités du SGBD),

(3) comprendre les définitions des tables dans le modèle relationnelle de données,

(4) expliquer comment les triplets de couleur sont attribués aux machines,

(5) comprendre le déséquilibre de la distribution des arêtes lorsqu'il existe des sommets fortement asymétriques,

(6) expliquer pourquoi une jointure distribuée ne s'adapte pas à un grand nombre de machines,

(7) expliquer pourquoi une requête SQL traditionnelle peut rencontrer un goulot d'étranglement avec des sommets asymétriques,

(8) expliquer pourquoi une affectation aléatoire des arêtes rééquilibre la charge de travail pour résoudre l'énumération de triangles.

4.5.2 Présentation de la démonstration

Notre scénario de démonstration s'adresse à la fois aux débutants et experts qui souhaitent apprendre comment les algorithmes randomisés peuvent fonctionner en SQL. Comme le montre l'interface graphique intuitive dans Figure 4.4, l'utilisateur choisira un graphe à analyser, où ce graphe est stocké sous forme de liste d'arêtes (i, j) dans la table E_s. Le système affichera m (le nombre d'arêtes) et n (le nombre de sommets). Ensuite, le système affichera les top sommets asymétriques par ordre décroissant de degré (top 15). Ensuite, pour résoudre l'énumération de triangles, l'utilisateur choisira entre les requêtes SQL traditionnelles (l'algorithme classique) ou les nouvelles requêtes randomisés (l'algorithme adapté). Après avoir évalué les requêtes, l'utilisateur visualisera des graphiques circulaires créés dynamiquement, montrant la répartition des triangles entre les machines (c'est-à-dire la charge de travail). Pour l'algorithme classique, l'utilisateur pourra voir des sommets asymétriques sur les machines qui ralentiront le calcul des jointures (résultats intermédiaires plus importants). En revanche, pour la solution randomisée,

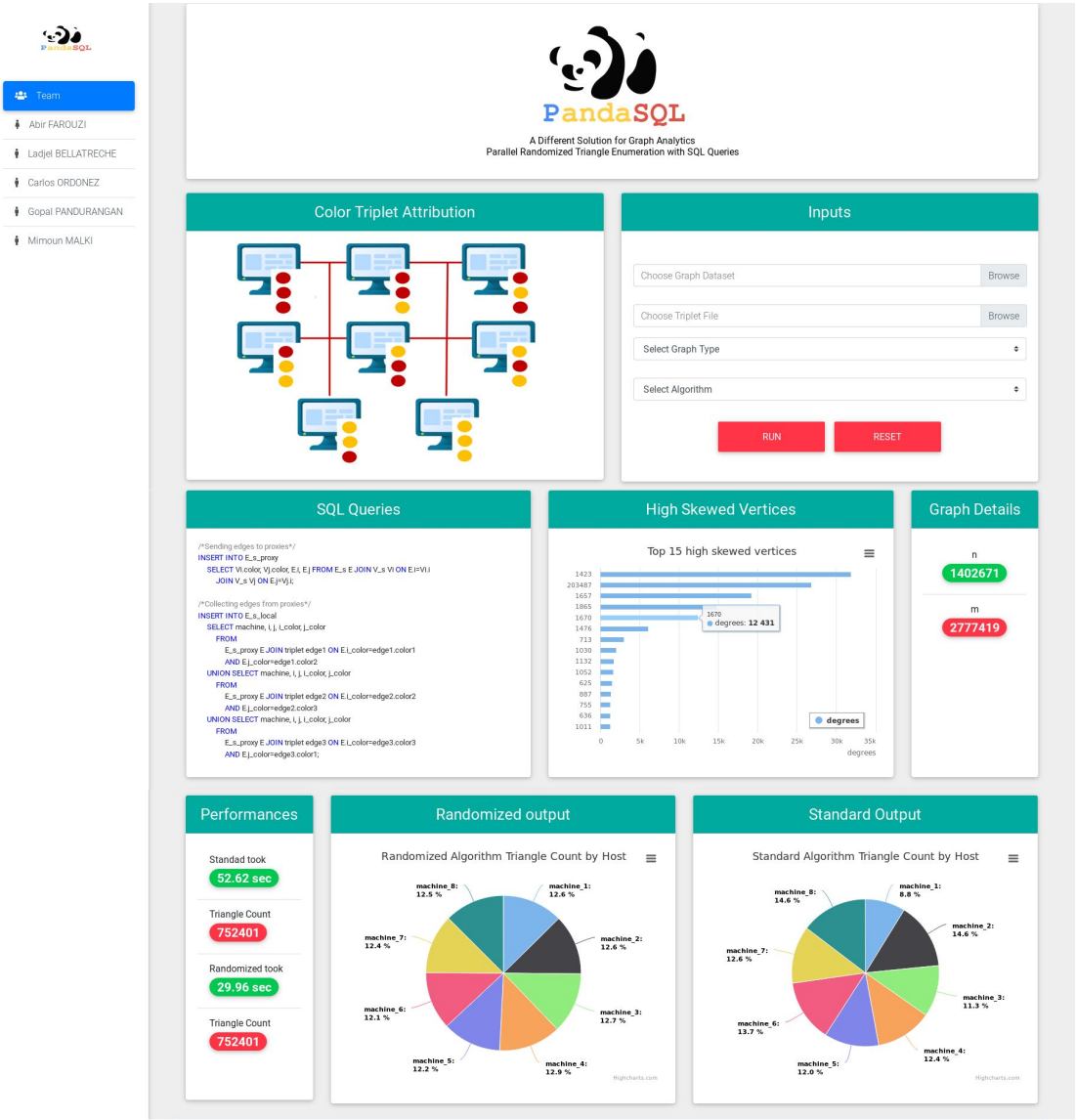


FIGURE 4.4 – Interface graphique de *PandaSQL*.

l'utilisateur remarquera que les arêtes impliquant des sommets asymétriques sont réaffectées, puis le nombre d'arêtes par sommet sur chaque machine est approximativement le même. Par conséquent, l'utilisateur comprendra que la réaffectation randomisée des arêtes accélère la jointure distribuée. Cette dernière devient locale et parfaitement en parallèle. Un tutoriel de l'interface graphique pour les débutants peut être retrouvé sur ce lien : `https://youtu.be/pwcYkOUV8_s`. Pour les utilisateurs expérimentés, nous allons discuter ci-dessous un aperçu plus technique de notre solution randomisé.

(1) **Le nombre de machine doit être un cube d'un entier supérieur exactement à 1 :** PandaSQL se base sur le modèle $k - machines$, donc le nombre de machine doit respecter la formule dans 3.2. La stratégie du partitionnement de l'algorithme randomisé consiste à partitionner l'ensemble V en $c = k^{1/3}$ sous-ensembles de couleurs. Vu que le triangle est formé de trois sommets, l'idée principale est d'attribuer une couleur c à chaque sommet. Par ailleurs, chaque machine est distinguée par un triplet de couleurs unique formé à partir des c couleurs puis affecté aléatoirement à elle. En se servant de son unique triplet de couleurs, chaque machine s'en charge de ramasser toutes arêtes dont les sommets portent les mêmes couleurs que son triplet de couleur. Si $k > c^3$, il y'aura des machines qui ne traitent aucun triangle, car aucun triplet de couleur ne sera affectée à elles. En revanche, si $k < c^3$

certaines machines effectueront plus de travail que les autres. Ces machines recevront plus d'un triplet de couleurs, donc elles traiteront plus de triangles par rapport aux autres. Ce qui dé-équilibre la charge entre les machines. Néanmoins, comme expliqué dans la chapitre précédent, le nombre de machine k peut être optimisé en utilisant la formule 3.4, en admettant une légère augmentation dans le temps d'exécutions des requêtes randomisées.

(2) **L'accélération potentielle :** Lorsque le nombre des machines k augmente, il y aura une accélération dans le traitement des triangles. Le traitement d'un graphe avec $m = 10M$ est plus rapide sur un cluster de 27 machines que celui de 8 machines. Le sous-graphe envoyé à chaque machine lorsque $k = 27$ est moins dense que celui envoyé aux machines lorsque $k = 8$. Ceci réduit le temps de réponse de la requête d'énumération des triangles (cette réquête s'exécute localement et en parallèle). Néanmoins, lorsque la taille du modèle $k - machines$ devient très large avec un graphe de petite à moyenne taille ($m < 1k$), l'accélération n'est pas garantit car le rééquilibrage (la redistribution des arêtes) devient coûteux.

(3) **Accès efficace aux arêtes dans les tables :** La table des arêtes $E\_s(i, j)$ est ordonnée par la colonne i. Lorsqu'une sélection de la colonne i est effectuée, un accès par une recherche dichotomique est choisi en utilisant un arbre B+ sur le SGBD. En contrepartie, la sélection de la colonne j, qui n'est pas triée, nécessite un accès par index (pour les SGBD en ligne) ou par projections (pour les SGBD en colonnes).

(4) **Le partitionnement des tables :** PandaSQL utilise cinq tables pour redistribuer les arêtes du graphe. Toutes ces tables sont partitionnées par les SGBD distribué. Ce partitionnement joue un rôle primordiale dans la performance des jointures. En effet, le partitionnement des tables par rapport à une colonne particulière permet de minimiser la charge de travail et paralléliser les opérations de sélection et de jointure.

(5) **La distribution équilibrée réelle des arêtes après leur envoi aux proxys :** Chaque machine locale reçoit un triplet de couleur unique lors de la redistribution des arêtes. Ce triplet de couleur est utilisé pour collecter, à partir des proxys, les arêtes dont les sommets ont les mêmes couleurs que deux couleurs parmi le triplet de couleurs de la machine locale. A la fin de cette phase, chaque machine locale aura un sous-ensemble d'arêtes qui peut former $1/k$ du total des triangles. Tous les triplets d'arêtes dont les sommets ont les mêmes couleurs que le triplet de couleurs de la machine locale, et qui peuvent former des triangles sont présents sur la même machine locale. De ce fait, la distribution des arêtes est équilibrée, vue qu'aucune communication ne sera nécessaire après l'étape de la collecte des arêtes par les machines locales.

(6) **Les jointures locales :** La table *E_s_local* est partitionnée par la colonne *machine* ce qui permet à chaque machine local de traiter que ses arêtes correspondantes. Les opérations de jointures sont donc exécutées d'une manière locale et en parallèle sur l'ensemble des machines du modèle.

(7) **L'énumération des triangles ne nécessite pas la redistribution des arêtes à la fin :** Comme expliquer dans les points précédents, la collecte des arêtes par les machines locales équilibre la distribution des arêtes. Pour cela, aucune arête n'est échangé lors de l'exécution des opérations de jointure dans la dernière requête d'énumération et de comptage de triangles. Plus précisément, chaque machine locale possède les arêtes nécessaires pour détecter tous les triangles, dont les extrémités ont le même ordre et couleurs que son triplet de couleurs.

Un tutoriel sur l'ensemble des requêtes utilisées par le système PandaSQL peut être consulté sur le lien suivant : `https://youtu.be/78Dd0rnMR4Q`, ainsi le code source de l'application est disponible sur : `https://github.com/lias-laboratory/pandasql`.

4.6 Conclusion

Nous avons présenté à travers ce chapitre une démonstration de notre système *PandaSQL* pour l'énumération de triangles fondée sur les requêtes randomisés. PandaSQL permet une comparaison visuelle (avec des graphiques) entre les résultats de l'algorithme adapté et ceux de l'algorithme classique, tout en mettant l'accent sur l'équilibrage de charge. Notre démonstration est composée de deux parties : (1) nous avons élaboré une démonstration détaillée de l'outil graphique pour les débutants, (2) ensuite, qu'une démonstration technique pour les experts.
Le chapitre suivant se concentra sur l'étude expérimentale du comportement de notre algorithme adapté, ainsi que son positionnement par rapport aux autres solutions, y compris la solution classique et les systèmes Big Data.

Chapitre 5

Évaluation de l'algorithme adapté pour l'énumértion et du comptage des Triangles

"In our world of Big Data, businesses are relying on data scientists to glean insight from their large, ever-expanding, diverse set of data ... while many people think of data science as a profession, it's better to think of data science as a way of thinking, a way to extract insights using the scientific method."

— Bob E. Hayes

Sommaire

5.1 Introduction

Dans les deux précédents chapitres, nous avons défini l'algorithme randomisé pour l'énumération et le comptage des triangles. Nous l'avons ainsi exploité pour développer l'algorithme adapté selon deux perspectifs :

(1) une implémentation Inside-DBMS avec le langage déclaratif SQL,

(2) une implémentation Outside-DBMS avec le langage impératif Python en employant sa librairie Pandas.

Ensuite, nous avons élaboré une analyse théorique de l'algorithme adapté, en étudiant sa complexité temporelle, son équilibrage de charge ainsi que l'exactitude de ses résultats. Nous avons également proposé une optimisation pour le modèle de calcul, qui permet de réduire sa taille sans impacter l'équilibrage de la charge de travail. Enfin, nous avons présenté PandaSQL qui est un outil avec une interface graphique qui permet d'énumérer les triangles sur les SGBDR.
Ce chapitre est consacré pour l'évaluation expérimentale de notre solution proposée. Nous commençons par la définition du modèle de calcul ainsi que la configuration logicielle et matérielle adoptée pour exécuter notre solution. Nous enchaînons avec la définition des techniques d'optimisation des requêtes SQL, où nous expliquons comment accélérer l'exécution des jointures et les requêtes complexes. Ainsi, nous effectuerons des expériences qui permettent de confirmer cette accélération. Ensuite, nous évaluons l'algorithme adapté sur deux aspects :

(1) une évaluation isolée qui permet d'évaluer les performances, l'équilibrage de charge et l'exactitude des résultats de l'algorithme adapté,

(2) une évaluation avec d'autres solutions qui permet de comparer les résultats de notre solution avec la solution classique sur les SGBDR et avec les systèmes Big Data. Ainsi, Nous pouvons positionner notre solution par rapport à d'autres solutions dans le même environnement, ainsi que par rapport à d'autres solutions dans des environnements différents.

Nous prévoyons également d'étudier l'impact d'élargir la taille du modèle de calcul sur le comportement et l'accélération de l'algorithme adapté et nous discutons dans quel cas l'augmentation du nombre des noeuds de calcul est plus intéressante. Enfin, nous résumons tous les résultats dans Table 5.12 et nous présentons une discussion approfondie sur les aspects positifs et négatifs de notre solution et ses cas d'utilisation les plus adéquats.

5.2 Modèle de calcul

Nous avons opté pour un modèle $k - machines$ constitué de 16 machines. Ce modèle est fondé sur une architecture shared-nothing, où aucun partage de mémoire ou de disque est autorisé entre les machines. C'est un modèle optimisé construit en suivant la formule 3.4. Ceci nous permet de valider les résultats avec un nombre de couleurs (nombre de sous-ensembles de V) plus large. Dans la suite de cette section, nous présentons et nous justifions nos choix matérielles et technologiques employés pour préparer le modèle de calcul.

5.2.1 Configuration matérielle

Les expérimentations ont été menées sur un cluster de 16 machines virtuelles. Chaque machine possède un processeur de 4 cœurs fonctionnant à $2,2\ Ghz$ en moyenne, 4 Go de mémoire principale, 500 To de stockage, 32 Ko de cache L1 et 1 Mo de cache L2. Le total de la RAM sur le cluster est de 64 Go, et le total du stockage sur disque est de 8 To, avec un total de 64 cœurs pour le traitement. Les machines sont connectées sur des cartes réseau de 1 Go avec une bande passante de 128 Mo/s.

5.2.2 Configuration logicielle

Nos expérimentations ont été exécuté sur le système d'exploitation Linux Ubuntu server 18.04, qui permet d'héberger tous les environnements d'évaluation, y compris le SGBD et celui du Big Data. Tout d'abord, nous avons opté pour le SGBD Vertica [LFV+12] pour examiner nos requêtes. Vertica [LFV+12] est un SGBD en colonnes utilisant SQL pour interroger la base de données. Elle est basée sur une architecture shared-nothing et sur le MPP. Ce choix est motivé par le fait que les SGBD en colonne sont $\times 10$ plus rapide que les SGBD en ligne pour les problèmes de graphe [JMCH15, AOB18]. En effet, les SGBD en colonne exploitent moins de mémoire pour interroger les données. Comme les SGBD en colonnes stockent les données par projections (une collection de colonnes) et non par rangées, elles peuvent stocker davantage de données dans un plus petit espace de mémoire. Cependant, tout autre SGBD parallèle fournissant un contrôle de partitionnement peut être utilisé. Nous avons également utilisé Python comme un langage de programmation pour générer nos requêtes et les soumettre à la base de données. Python est plus rapide et offre une meilleure gestion de la mémoire comparant à JDBC [1].
Nous présentons Figure 5.1 qui résume la configuration logicielle et matérielle utilisée pour exécuter les requêtes de l'algorithme adapté pour l'énumération et le comptage des triangles.

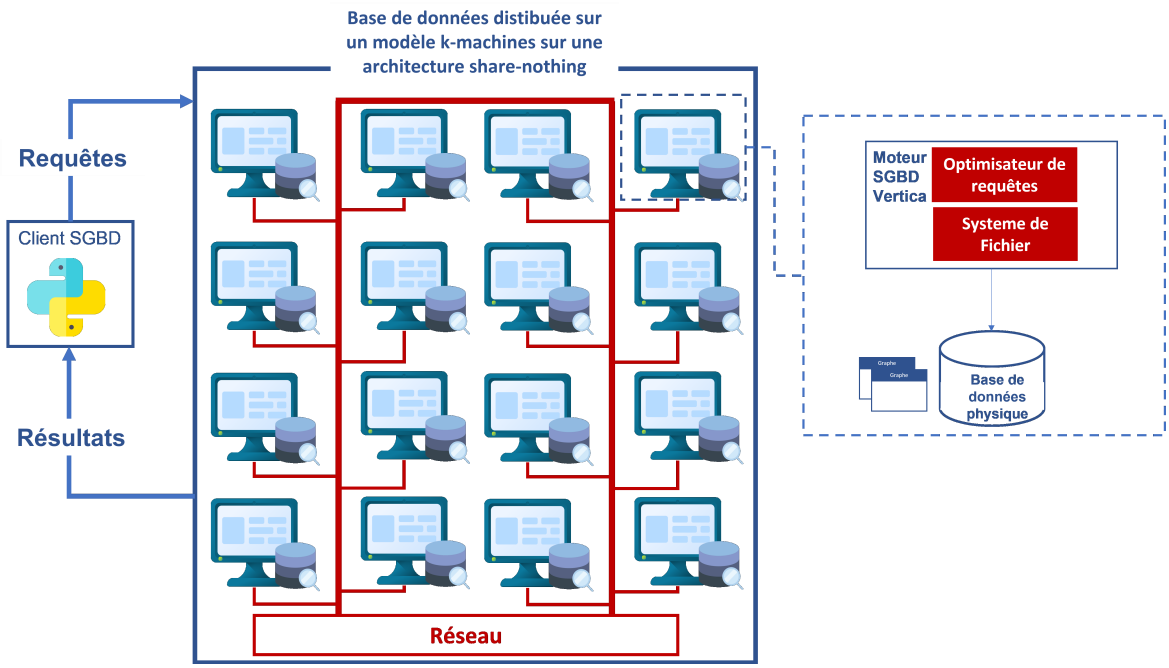


FIGURE 5.1 – Modèle de calcul.

Ainsi, comme l'algorithme adapté [FBO+20a, FZB+23a] propose une solution en utilisant la librairie Pandas de Python, nous avons choisi d'assurer la communication entre les machines du modèle en l'implémentant avec MPI. MPI [GGH+96] est une spécification pour une bibliothèque standard pour le modèle de transmission de messages utilisé dans les calculs parallèles. Il définit une communication de point à point et des opérations collectives.
De plus, pour vérifier l'efficacité de l'algorithme adapté, nous l'avons comparé avec la fonction de comptage des triangles pré-implémnetée dans *Spark GraphX*. Spark est un système de calcul distribué pour le traitement des données massives. Il définit un ensemble de modules, entre autres Spark GraphX qui s'agit d'un module pour le traitement des graphe. Il offre une suite de fonctions élémentaires pour l'exploration des graphes, dont la fonction du comptage des triangles. Enfin, nous avons également comparé notre solution avec G-thinker [YGRC+20]. G-thinker est un système centré sur les sous-graphes pour l'exploration des graphes et l'extraction des motifs

1. *Java Database Connectivity interface*

d'intérêt. Il s'agit d'un moteur d'exécution basé sur les tâches fonctionnant avec un petit budget de mémoire, tout en gardant les cœurs du processeur pleinement utilisés. G-thinker définit plusieurs fonctions pour résoudre différents problèmes de recherche de sous-graphes, y compris le comptage de triangles. Il est construit au-dessus d'un cluster Hadoop et utilise MPI et le threading. Selon [YGRC+20], G-thinker utilise tous les cœurs CPU du cluster, en exploitant un cache de sommets et une approche de planification légère de tâches. Cependant, ce système est fortement dépendant de la configuration du système, il nécessite de nombreuses compétences pour configurer les dépendances avant de l'utiliser. Nous jugeons qu'une comparaison avec ce système est intéressante car notre solution, particulièrement celle avec SQL, présente le même niveau de concurrence et de performance.

5.3 Les données utilisées dans nos expérimentations

Nous présentons dans Table 5.1 les ensembles de données utilisés pour évaluer notre solution d'énumération et de comptage de triangles. Nous avons mené nos expérimentations sur des graphes réels obtenus à partir des ensembles de données de référence (Benchmarks). Précisément, nous avons utilisé un jeu de données composé de neuf ensembles de données réels. Pour chaque graphe, nous présentons dans Table 5.1 sa taille (n), son ordre (m), le nombre des triangles attendus (TC), son degré maximal (d_{max}) et sa source.

Tableau 5.1 – Ensembles de données réels des graphes.

Ensemble de Données	n	m	TC	d_{max}	Source
web-google	875k	5,105k	13,391k	6,353	SNAP [LK14]
WikiTalk	2,394k	5,021k	9,203k	488,182	SNAP [LK14]
as-Skitter	1,696k	11,095k	28,769k	35,455	SNAP [LK14]
Hyves	1,402k	2,777k	752k	31,883	KONECT [Kun13]
Youtube	2,987k	1,134k	3,056k	28,754	KONECT [Kun13]
flick-links	1,715k	15,551k	548,174k	27,224	KONECT [Kun13]
live-journal	3,997k	69,362k	177,820k	14,815	KONECT [Kun13]
DBpedia	18,268k	172,183k	328,743k	632,558	KONECT [Kun13]
Dimacs10_*uk*	18,483k	261,787k	4,451,687k	194,955	KONECT [Kun13]

Dans ces ensembles de données, nous avons opté pour deux graphes orientés : web-google et WikiTalk. Le premier possède une distribution uniforme et des sommets à faible degré, tandis que le deuxième définit une distribution asymétrique des sommets. Nous avons également choisi un ensemble de données avec de multiples arêtes (DBpedia) pour montrer l'efficacité de notre solution même avec plusieurs arêtes entre les couples de sommets. Par ailleurs, tous les autres ensembles de données sont non-orientés avec des arêtes uniques entre les couples de sommets. Ils définissent une distribution asymétrique et possède plusieurs sommets à haut degré.
Par ailleurs, nous avons opté pour les ensembles de données synthétiques présentés dans Table 5.2. Ces ensembles sont utilisés pour pouvoir étudier l'impact de l'application des optimisations sur la taille du graphe et sur le temps d'exécution global. Pour cela, nous avons généré les graphes *cliqueLinear* en faisant varier linéairement les tailles des cliques (nous avons généré les cliques de tailles 2,4,6,..etc).

Tableau 5.2 – Ensembles de données synthétiques.

Ensemble de Données	n	m	TC
cliqueLinear1M	10k	1M	17,666k
cliqueLinear10M	48k	10M	387,278k
cliqueLinear100M	224k	100M	8,321M

5.4 Techniques d'optimisation des requêtes

Les requêtes SQL peuvent devenir lentes, lorsqu'elles sont composées de plusieurs jointures. Pour cela, il faut appliquer certaines techniques d'optimisation lors de la définition des tables, pour assurer l'accélération de l'interrogation de la base de données. D'autres techniques peuvent être directement introduites dans la requête SQL afin d'éviter la redondance et optimiser l'espace mémoire exploité. Dans ce qui suit, nous présentons les différentes techniques, tout en mentionnant l'étape de leur application.

5.4.1 Optimisation de l'espace mémoire dans les SGBD en colonnes

L'optimisation de l'espace mémoire lors de l'exécution des requêtes permet de réduire le temps d'exécution nécessaire pour extraire les résultats requis. En effet, les SGBD orientés colonnes sont optimisés pour la lecture. Ils permettent de charger en mémoire que les attributs nécessaires pour la requêtes et d'y analyser. Ceci permet d'éviter le chargement des attributs inutiles pour la requêtes et de saturer l'espace mémoire vainement. Les SGBD orientés colonnes stockent ses données sous forme de projections de colonnes, ce qui permet de récupérer directement les colonnes requises pour l'interrogation de la base de données. En outre, comme les attributs de la même colonne ont le même type, ils peuvent être encodés pour un stockage et une récupération plus efficaces.
Notons que pour les SGBD orientés lignes, le stockage des données se fait par tuple. Par conséquent, l'exécution des jointures nécessite le chargement entier des tables concernées. Ce qui augmente l'espace mémoire requis pour l'exécution de la requête. En outre, chaque ligne des SGBD orientés ligne possède une définition différentes entre ses attributs. Par conséquent, l'application des techniques de compression sur les SGBD en ligne s'avère délicat. Dans la suite, nous nous concentrons sur la présentation des projections et des techniques d'encodage, qui permettent d'accélérer l'exécution de l'algorithme adapté.

5.4.1.1 Projection des colonnes

Les projections sont des collections optimisées de colonnes de table qui fournissent un stockage physique pour les données. Elles peuvent contenir une partie ou toutes les colonnes d'une ou de plusieurs tables [LFV+12]. En effet, les projections offrent un moyen de stockage physique des colonnes triées par une clé, où une projection peut contenir une colonne ou plusieurs colonnes de la même table, ou plusieurs colonnes de différentes tables, à condition que la clé étrangère soit mentionnée. Par exemple, pour le modèle relationnelle de l'algorithme adapté :

```
E_s(i,j)
V_s(i,color)
E_s_proxy(i, j, i_color, j_color)
E_s_local(machine, i, j, i_color, j_color)
```

Les projections peuvent être :

```
Super_E_s(i, j | i)
Super_V_s(i, color| i)
```

```
Super_E_s_proxy(i, j, i_color, j_color | i)
Super_E_s_local(machine, i, j, i_color, j_color | machine)
```

avec i l'attribut sur lequel les projections E_s, V_s et E_s_proxy seront triées et $machine$ est l'attribut sur lequel la projection E_s_local sera triée (i et $machine$ sont appelés une clé de tri). Chaque projection produite est divisée en plusieurs segments, où chacun d'eux reçoit un SID. Cette segmentation est fondée sur les valeurs de la clé de tri, où chaque segment représente une portion de ces clés. Effectivement, les SGBD en colonnes permettent la gestion distribuée des données. Pour cela, ils permettent la segmentation des projections sur l'ensemble des machines de traitement, où chacune d'elle représente un SID et héberge tous les clés qui appartient à son segment.
La définition des projections doit permettre une reconstruction de la table initiale. Donc, les différentes projections doivent construire un *ensemble recouvrant* [SAB+18]. En fait, la reconstruction d'une table requiert l'utilisation des indices de jointure et les clés de stockage. Les clés de stockage sont utilisées pour représenter une ligne logique ou un tuple dans le segment donné. Ainsi, les index de jointure utilisent les clés de stockage pour créer les tables originales à travers plusieurs segments. Supposons que les deux projections T1 et T2 peuvent couvrir une table entière T. Alors l'entrée de l'index de jointure consiste en (SID, clé de stockage) de T2 pour les clés correspondantes dans T1. Par exemple, soit les deux projections :

```
T1(i, j | j)
T2(i, i_color | i_color)
```

La construction de la table $T(i, j, i\_color)$ est effectué sur l'attribut i comme montré sur Figure 5.2.

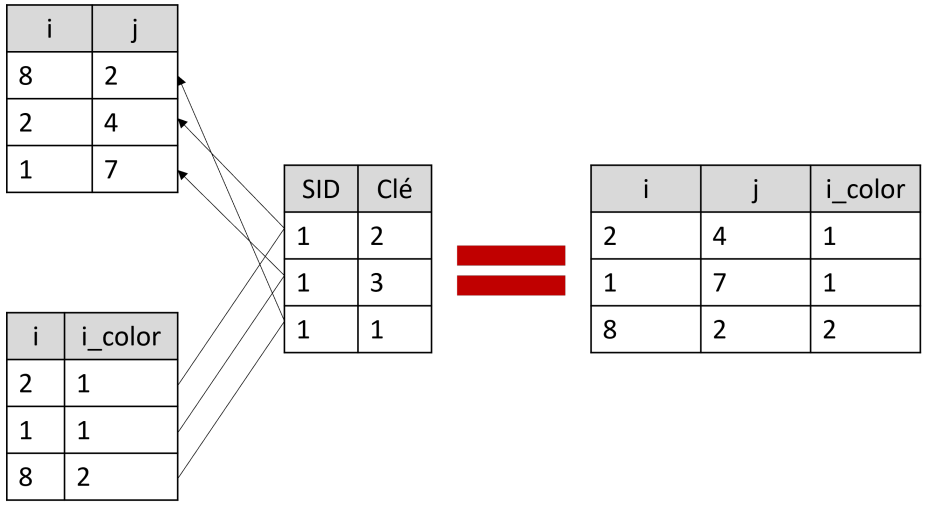


FIGURE 5.2 – Reconstruction de la table T avec l'index de jointure

Pour optimiser l'exécution des requêtes, les projections stockent les résultats réels des requêtes sur le disque, plutôt que de les recalculer à chaque interrogation. Le chargement des données actualise automatiquement ces projections ainsi que les index de jointures avec de nouvelles données ou des données mises à jour. Les SGBD en colonnes repose sur le principe de séparation entre les opérations de lecture et d'écriture, afin d'optimiser la lecture et d'éviter les problèmes de concurrence et de blocage avec l'écriture [SAB+18]. Pour cela, ils utilisent deux ensembles de logiques : Writable-Store(WS) and Readable-Store(RS). Toutes les écritures sont effectuées dans le WS et un composant de déplacement de tuple déplace périodiquement les enregistrements modifiés vers le RS à l'aide d'un processus de fusion. Notons que sur Vertica, le RS est composé de plusieurs conteneurs qui permettent le stockage physique des données sur un système de fichiers

standard. Pour permettre un accès sans verrou au RS, une capture du RS (très récente) est créée dans des petits lapses de temps et peut ensuite être utilisé pour des requêtes sans verrou. Ainsi, la soumission d'une nouvelle requête permet à l'optimiseur de requêtes d'assembler directement un plan de requête en tenant compte des propriétés de projection pour calculer le résultat demandé.

5.4.1.2 Compression des données

Le SGBD en colonnes utilise la compression pour réduire et économiser l'espace de stockage total requis. Il convertit les données dans un format standard, afin d'augmenter les performances et réduire les d'E/S durant l'exécution de la requête. Particulièrement, les projections peuvent améliorer la compression et réduire considérablement le temps de calcul des opérations de jointure. Comme mentionné dans la section précédente, les projections stockent les collections de colonnes, sachant que chaque colonne définit les attributs sous le même type. Par conséquent, l'encodage des données devient plus efficace. Notamment, Vertica [LFV$^+$12] trie les données avant leurs enregistrement, ce qui permet de faciliter et d'améliorer la compression des projections. Vertica [LFV$^+$12] définit plusieurs types de compression :

(1) **Auto** : Le système choisit automatiquement le type d'encodage le plus avantageux, en fonction des propriétés des données elles-mêmes. Ce type d'encodage est le type par défaut, et il est choisi lorsqu'il n'y a pas de type d'encodage prédéfini.

(2) **RLE** [2] : Cet encodage convertit une liste contiguë de valeurs identiques en un ensemble de paires de type $< occurrence, valeur >$. Ce codage est appliqué aux colonnes à faible cardinalité ayant des valeurs contiguës identiques.

(3) **Valeur Delta** : Les données sont enregistrées sous forme de différence par rapport à la plus petite valeur d'un bloc de données. Ce type est utilisé de préférence pour les colonnes à valeurs multiples ou à nombres entiers non triés.

(4) **Dictionnaire de bloc** : Dans un bloc de données, les valeurs distinctes des colonnes sont stockées dans un dictionnaire, et les valeurs réelles sont remplacées par des références au dictionnaire. Ce type est idéal pour les colonnes non triées à faible valeur.

(5) **Plage de différence compressée** : Stocke chaque valeur comme une différence par rapport à la valeur précédente. Ce type convient aux colonnes à valeurs multiples flottantes qui sont soit triées, soit limitées à une plage.

(6) **Différence commune compressée** : Le système construit un dictionnaire de tous les différence dans le bloc, et stocke ensuite les index dans le dictionnaire en utilisant le codage entropique [3]. Ce type est idéal pour les données triées avec des séquences prévisibles, et des ruptures de séquence occasionnelles. À titre d'exemple, il peut être appliquer sur les horodatages enregistrés à intervalles périodiques, ou les clés primaires.

Pour optimiser nos requêtes randomisées, nous avons opté pour la technique *RLE*. Ce codage est appliqué aux colonnes des différents projections, vu qu'elles sont toutes à faible cardinalité ayant des valeurs contiguës identiques (les différents sommets composant les arêtes).
En résumé, la compression transforme les données dans un format compact. Elle permet à un SGBD en colonnes de posséder moins de stockage et d'effectuer ainsi moins d'E/S.

5.4.1.3 Partitionnement et segmentation des données

Les SGBD en colonnes définissent deux logiques pour assurer le traitement parallèle des données distribuées :

2. *Run Length Encoding*

3. Le codage entropique (ou codage statistique à longueur variable) est une méthode de codage sans pertes basée sur les statistiques sur une source, dont le but est de transformer la représentation d'une source de données (qui peut être un texte, une image ou un signal numérique) par un code pour sa compression et/ou sa transmission via un canal de communication. Nous recensons deux principaux types de codage entropique qui sont le codage de Huffman et le codage arithmétique.

(1) Partitionnement des données : s'agit de la division d'une grande table/structure en petits morceaux en fonction de la veleur d'une ou de plusieurs colonnes. Cette division est définie de différentes façons sur les SGBD en colonnes. Par exemple, C-Store[SAB+18] utilise le *partitionnement horizontal* intra-nœud comme un moyen d'améliorer les performances en augmentant le parallélisme au sein d'un seul nœud. En revanche, Vertica [LFV+12] n'exige pas une séparation des structures physiques sur le disque, afin d'assurer un parallélisme intra-nœud. Il le fait en divisant chaque structure sur le disque en régions logiques au moment de l'exécution, puis il traite ces régions parallèlement. Néanmoins, Vertica fournit un moyen de garder les données séparées dans les structures physiques. Ceci est effectué par rapport à une colonne par une syntaxe simple, définie lors de la création de la table ou de la projection :

```
CREATE TABLE .. PARTITION BY
CREATE PROJECTION .. PARTITION BY
```

Les principaux avantages du partitionnement de données sont :

(a) *Suppression rapide de données en masse* : Il est courant de conserver les données séparées dans des fichiers basés sur une combinaison de mois et d'année, de sorte que la suppression d'un mois spécifique de données du système est aussi rapide et simple que la suppression de fichiers d'un système de fichiers. Ceci permet de récupérer immédiatement de l'espace de stockage.

(b) *Amélioration des performances* : Vertica stocke les valeurs minimales et maximales des données de la colonne dans chaque RS. Ceci permet de libérer rapidement les conteneurs (qui ne peuvent pas passer les prédicats de la requête) au moment de l'évaluation des plan d'exécution. Le partitionnement rend cette technique plus efficace, en empêchant les valeurs de colonnes mélangées dans le même conteneur RS.

Dans l'algorithme adapté, toutes les projections sur les tables E_s, V_s, E_s_proxy et E_s_local sont partitionnés par l'attribut i.

(2) Segmentation des projections : Les SGBD en colonnes permettent le traitement distribué des données, en créant des segments qui sont pris en charge par un nœud de traitement. Particulièrement, Vertica dispose d'un système de stockage distribué entièrement implémenté qui attribue des tuples à des nœuds de calcul spécifiques. Cette segmentation est fondée sur un partitionnement horizontal inter-nœuds (qui divise les tuples entre les machines). La segmentation est spécifiée pour chaque projection, où elle fournit une correspondance déterministe entre la valeur du tuple et la machine de traitement. Ceci permet de nombreuses optimisations importantes, y compris l'exécution locale des jointures, ainsi que le traitement accéléré des agrégations distribuées, ce qui est particulièrement efficace pour le calcul d'agrégats distincts à haute cardinalité. Notamment, les données de la projection peuvent être soit repliquées, soit segmentées. La projection répliquée permet de stocker une copie de chaque enregistrement sur chaque machine de traitement (voir Sect. 5.4.2 pour plus de détails). En contrepartie, les projections segmentées stockent chaque tuple sur exactement un nœud de projection spécifique. Le nœud sur lequel le tuple est stocké est déterminé par une clause de segmentation $< expr >$ dans la définition de la projection :

```
CREATE PROJECTION .. SEGMENTED BY <expr>
```

D'une manière générale, chaque machine héberge tous les tuples qui appartient a son segment. Les segments sur Vertica sont calculés avec une fonction de hachage déterministe $Hash : col \rightarrow seg$. Cette fonction cherche le segment de chaque tuple, où les plages de

valeurs pour chaque segment sont comme suivant :

$$Machine1 \in [0, \frac{C_{max}}{k}]$$
$$Machine2 \in [\frac{1 * C_{max}}{k}, \frac{2 * C_{max}}{k}]$$
$$Machine3 \in [\frac{2 * C_{max}}{k}, \frac{3 * C_{max}}{k}]$$
$$..$$
$$Machinek \in [\frac{(k-1) * C_{max}}{k}, C_{max}]$$

avec $C_{max} = 2^{64}$ dans Vertica[LFV+12]. La segmentation est utilisé principalement, dans notre solution, avec la table *E_s_local* afin d'envoyer toutes les tuples qui ont la même valeur de l'attribut machine au même nœud de traitement. Ce qui permet d'effectuer des jointures purement locales.

Bref, il y a une différence entre partitionner une table et segmenter une projection dans les SGBD en colonnes, le premier s'effectue localement sur chaque machine du cluster, alors que le second permet de diviser les données sur le cluster en affectant une partie de la projection à chaque machine.

5.4.2 Réplication des petites tables

La réplication des petites tables signifie la création d'une copie intégrale de la table sur chaque machine de traitement. C'est une technique plus avantageuse que la segmentation, pour le calcul des jointures et des agrégations. En effet, la réplication des petites tables sur chaque machine du cluster représente une technique d'optimisation très efficace. Elle garantit l'accès rapide et la disponibilité des données requises sur chaque machine, empêchant ainsi l'échange de données pendant les opérations de jointure et d'agrégation. Dans les ordinateurs modernes, une petite table statique peut être mise en cache dans la mémoire principale sur chaque nœud, accélérant ainsi le calcul local des jointures. La table *Triplet* est une petite table statique introduite par l'utilisateur, et qui n'évolue pas dans le temps. Elle est répliquée sur l'ensemble des machines du modèle de calcul. Sur Vertica, ceci est défini lors de la création de la projection en utilisant l'expression suivante :

```
CREATE PROJECTION .. UNSEGMENTED ALL NODES
```

5.4.3 Replication de la table de l'auto-jointure

Sur un SGBDD, il est recommandé d'éviter les auto-jointure car elle augmente le coût de communication entre les nœuds de traitement. Nous avons expliqué précédemment que le SGBD Vertica repose sur le partitionnement des tables et la segmentation de projections pour accélérer le calcul des jointures. Explicitement, lors de l'exécution d'une jointure $R_1 \bowtie_{c_1=c_2} R_2$, la relation R_1 est partitionnée par c_1, tandis que la relation R_2 est partitionnée par c_2. Ainsi, des projections sur les colonnes c_1 et c_2 sont créées. Ceci permet d'effectuer la jointure en exploitant les deux projections sur les colonnes concernées. Par conséquent, la jointure est exécutée plus efficacement et plus rapidement, en minimisant les échanges entre les machines de calcul pour communiquer les données. Il existe plusieurs solutions pour remplacer les auto-jointures, entre autres, la réplication de la table concernée. Dans les requêtes SQL, l'utilisation de la table répliquée permet d'accélérer le traitement des jointures locales. En partitionnant la table R par c_1 et la son réplicat par c_2, les deux tables correspondantes sont divisées en petits morceaux en fonction des colonnes susmentionnées. De plus, deux projections sur c_1 et c_2 sont créées. Ceci

accélère l'exécution de la jointure locale sur $c_1 = c_2$. À ce propos, nous présentons la requête suivante, qui permet d'optimiser la requête d'énumération des triangles présentée dans le chapitre 3. Notons que *E_s_local_dup* et *E_s_local_rep* sont deux tables répliquées de la table *E_s_local* :

```
SELECT E1.machine as machine,E1.i as v1,E1.j as v2,E2.j as v3
   FROM
      E_s_local E1 JOIN E_s_local_dup E2
      ON E1.machine=E2.machine AND E1.j=E2.i
      JOIN E_s_local_rep E3
      ON E2.machine=E3.machine AND E2.j=E3.i
   WHERE E1.i<E1.j AND E2.i<E2.j AND E1.i=E3.j;
```

Figure 5.3 présente un aperçu de l'effet de la réplication sur l'organisation des données et sur l'accélération la jointure $E\_s\_local \bowtie_{j=i} E\_s\_local$. Chaque fragment coloré sur la figure représente une partition. Notons qu'avec la réplication, la table *E_s_local_dup* est triée par la colonne i et la table *E_s_local* par la colonne j. Donc, il est plus simple de réaliser la jointure localement sur chaque partition par un algorithme de fusion au lieu d'un algorithme de tri-fusion appliqué sur l'auto-jointure (voir le cas à gauche). Par conséquent, la compléxité d'une telle jointure passe de $O(m \times log(m))$ à $O(m)$, où m est le nombre de tuples dans la table *E_s_local*.

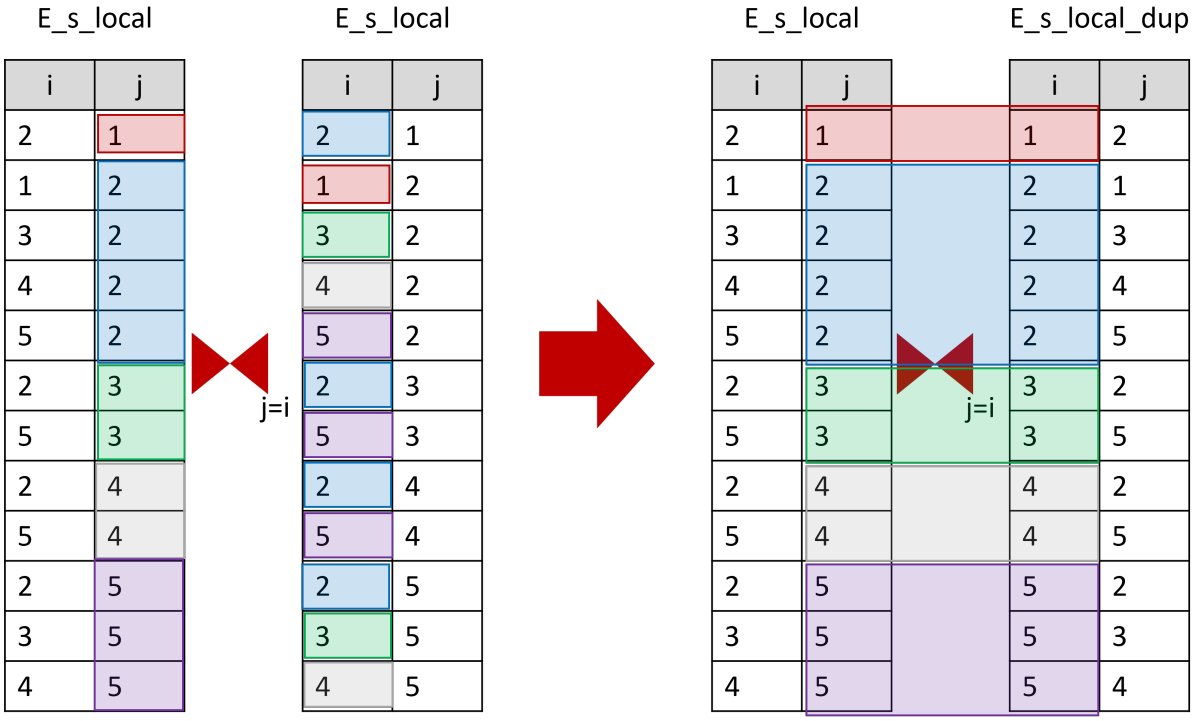


FIGURE 5.3 – Auto-jointure versus replication des tables.

5.4.4 Élimination des résultats redondants

Les résultats redondants affectent sérieusement les performances des requêtes randomisées. En effet, les résultats superflus permettent de saturer rapidement la mémoire, ce qui augmente le nombre d'E/S. Pour remédier à ce problème, il est préférable de les éliminer lors de l'exécution de la requête d'énumération et de comptage de triangle. Pour cela, seuls les triangles définissant l'ordre lexicographique (de l'ID de sommet le plus bas à l'ID de sommet le plus élevé) sont pris en compte. En d'autres termes, pour $\Delta_{(v_1,v_2,v_3)}$, seul le triangle (v_1, v_2, v_3) avec $v_1 < v_2 < v_3$ est énuméré et compté, en revanche les autres alternatives comme (v_2, v_3, v_1) ou (v_3, v_1, v_2) sont

ignorés. Concrètement, la condition $E1.i < E1.j \ AND \ E2.i < E2.j$ est rajoutée à la requête ci-dessous :

```
SELECT E1.machine as machine,E1.i as v1,E1.j as v2,E2.j as v3
   FROM
      E_s_local E1 JOIN E_s_local_dup E2
      ON E1.machine=E2.machine AND E1.j=E2.i
      JOIN E_s_local_rep E3
      ON E2.machine=E3.machine AND E2.j=E3.i
   WHERE E1.i<E1.j AND E2.i<E2.j AND E1.i=E3.j;
```

Lors de l'exécution de la requête, un filtre pour sélectionner que les arêtes dont la source est plus petite que la destination, est appliqué. Ceci élimine la sélection des arêtes qui n'interviennent pas dans la construction des triangles. Par conséquent, le nombre d'E/S est réduit ainsi que l'espace mémoire requis pour exécuter la jointure locale.

5.5 Validation des résultats de l'algorithme adapté

Nos expérimentations visent les objectifs suivants :

(1) évaluer l'impact des techniques d'optimisation sur l'accélération des requêtes,

(2) évaluer l'équilibrage de charge de l'algorithme adapté pour démontrer l'efficacité de la stratégie de partitionnement,

(2) comparer l'algorithme adapté avec les systèmes concurrents, afin de prouver que nos requêtes sont efficaces en plus d'être courtes, élégantes et abstraites.

Chaque expérience a été répété cinq fois et la moyenne des mesures du temps est rapportée. Avant chaque expérimentation sur le SGBD en colonnes, le cache a été nettoyé et tous ces données sont effacées. Ainsi, le temps d'exécution considéré est celui émis par le SGBD (pas celui du client).

Nos expériences sont présentées en trois ensembles, comme illustré par Figure 5.4, le premier ensemble vise à évaluer les optimisations de requêtes : (1) l'effet d'avoir une table répliquée lors d'une auto-jointure locale et (2) l'impact de pousser la clause "WHERE" dans la requête d'énumération et de comptage local de triangles, afin d'éliminer les résultats redondants. De plus, il évalue le comportement de l'algorithme adapté sur plusieurs configurations optimisées du modèle $k - machines$. Ce qui permet d'étudier l'influence de l'optimisation du modèle sur l'exécution de l'algorithme adapté. Cet ensemble d'expérimentation permet également d'étudier l'effet du partitionnement en dehors du SGBD et de vérifier si cette optimisation accélère le temps d'exécution des requêtes adaptées. Le deuxième ensemble concerne une évaluation de l'algorithme adapté en termes d'équilibrage de charge et d'accélération, en augmentant le nombre de couleur permettant le partitionnement de l'ensemble des sommets V. Le but de cette partie des expérimentations est de prouver la justesse et l'efficacité de l'algorithme adapté. La dernière série d'expérimentation évalue l'algorithme adapté avec plusieures solutions concurrentes dans différents environnements distribués : (1) avec la solution classique sur le SGBD en colonne, (2) avec la fonction intégrée par défaut dans la bibliothèque Spark GraphX pour le comptage de triangles, (3) avec G-thinker qui s'agit d'un système distribué et parallèle pour l'énumération des sous-graphes, et (4) avec NetworkX qui est une librairie de Python destinée pour l'exploration et l'analyse des graphes. Nous voulons montrer que notre solution est robuste sur le SGBD en colonne, grâce à sa stratégie de partitionnement qui assure une distribution équitable de la charge de travail entre les machines. Ceci permet de traiter des graphes très large avec une distribution fortement asymétrique des sommets. En outre, notre comparaison avec les différents systèmes concurrents vise à démontrer l'efficacité de notre approche face à un système de calcul de Big Data, tout en discutant les limites des différents systèmes. À la fin de cette section, nous présentons un résumé de tous les résultats sur les différents systèmes.

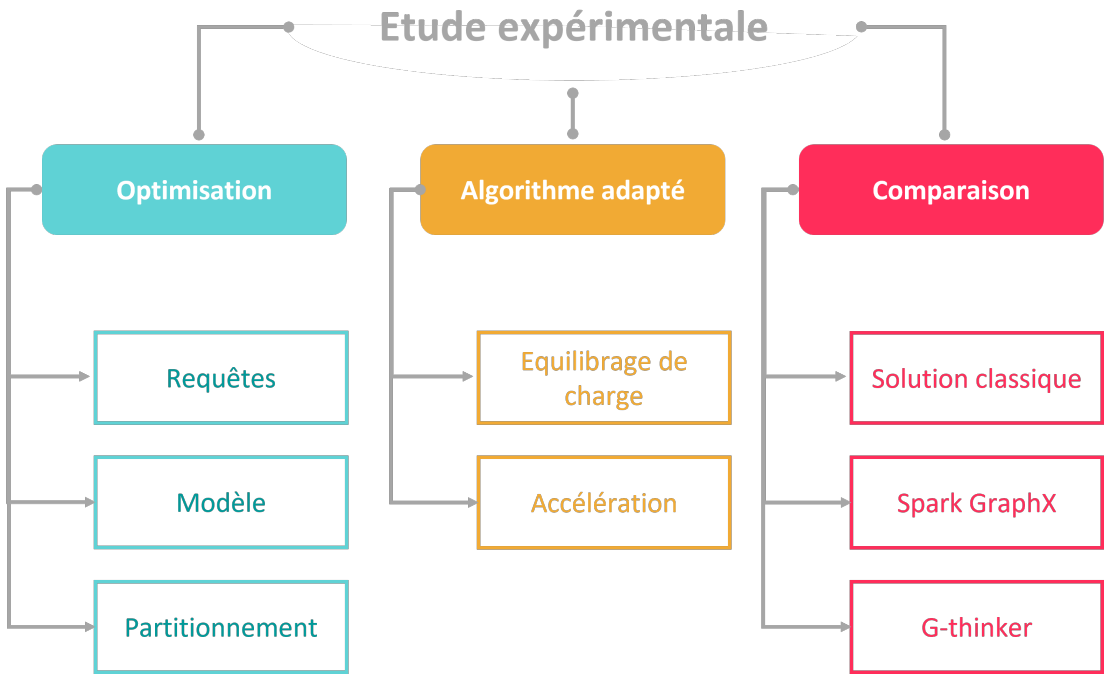


FIGURE 5.4 – Organisation de l'étude expérimentale.

5.5.1 Évaluation des optimisations

Nous étudions dans cette section l'influence des techniques d'optimisation présenté dans Sec. 5.4 sur l'exécution des requêtes. Nous nous intéressons particulièrement à la réplication des tables lors d'une auto-jointure, et l'élimination des résultats redondants. Nous étudions également l'effet de la réduction de la taille du modèle de calcul en appliquant la formule 3.4 sur le temps d'exécution. Enfin, nous évaluons l'impact du partitionnement de l'ensemble des sommets V dans et en dehors du SGBD.

5.5.1.1 Optimisation des requêtes

Dans cette expérience, nous comparons les avantages d'avoir une réplique de la table des arêtes lors de l'exécution de la requête d'énumération et de comptage de triangles. Cette requête est basée sur l'auto-jointure locale c-à-d elle est exécutée localement sur chaque machine et en parallèle (il n'y a pas de communication entre les machines lors de son exécution). Nous voulons montrer que cette optimisation réduit le temps d'exécution de cette requête. Pour cela, nous avons utilisé l'ensemble de données synthétique cliqueLinear en augmentant la taille (m) du graphe. Figure. 5.5 représente une comparaison entre le temps d'exécution de la requête avec et sans cette optimisation. Comme le montre la Figure, la réplication de la table diminue considérablement le temps d'exécution de l'auto-jointure locale, lorsque la taille du graphe devient de plus en plus importante. Ceci peut être expliqué par le fait que la table *E_s_local* est partitionnée par la colonne j et la table *E_s_local_dup* est partitionnée par i. Par conséquent, l'exécution de la jointure locale est plus rapide.
Figure. 5.6 présente une comparaison entre le nombre de triangles générés sur le graphe *cliqueLinear* par l'algorithme adapté, avec et sans prise en compte de l'ordre lexicographique des sommets des triangles. Rappelons que ce dernier est effectué en ajoutant la condition "WHERE $E1.i < E1.j$ AND $E2.i < E2.j$" dans la requête de comptage local des triangles. Il est évident que des triangles redondants seront générés sans pousser la condition WHERE. L'élimination des redondances permet de sélectionner au départ les arêtes qui interviennent dans la construction des triangles résultants. Par conséquent, moins d'espace mémoire est requis pour l'exécution de

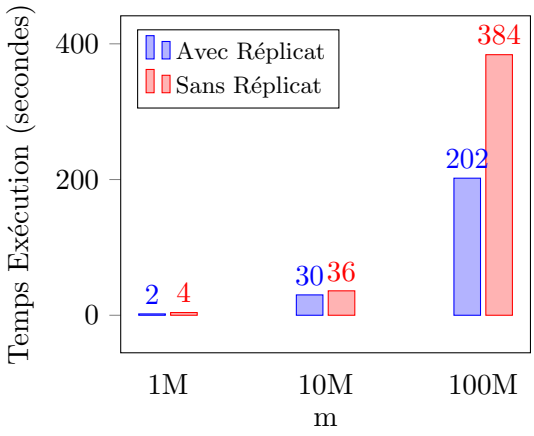


FIGURE 5.5 – Comparaison du temps d'exécution dans le SGBD en colonnes : (1) Avec un *réplicat* de la table des arêtes et (2) *Sans réplicat* de la table des arêtes.

la requête d'énumération locale des triangles.

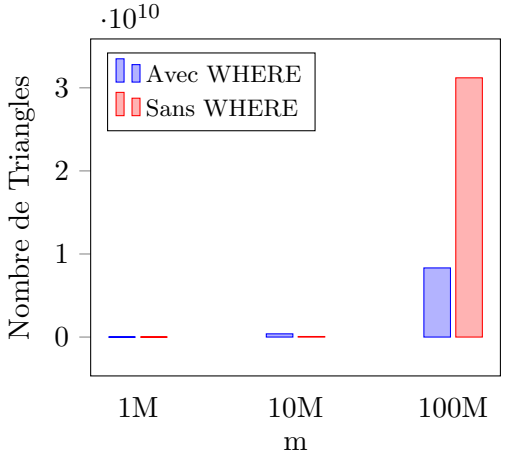


FIGURE 5.6 – Comparaison du comptage de triangles dans le SGBD en colonnes : (1) Avec la condition WHERE et (2) Sans la condition WHERE.

5.5.1.2 Optimisation du modèle

Nous avons expliqué précédemment que le modèle de calcul peut être optimisé en utilisant la formule (3.4). Ainsi, nous présentons dans Table 5.3 l'impact de cette optimisation. En effet, nous comparons dans la Table 5.3 le temps d'exécution de l'algorithme adapté sur un modèle à 8 machines et sur un modèle optimisé à 4 machines.
Nous pouvons remarquer que l'exécution de l'algorithme adapté, avec les requêtes SQL, sur le modèle optimisé est plus efficace avec les ensembles de données de taille petite à moyenne, comme hyves et web-google. Ceci peut être expliqué par le fait qu'il y a moins de communication de données entre les machines et moins d'E/S principalement durant l'étape du partitionnement. En revanche, le temps d'exécution avec les grands ensembles de données, comme live-journal, sur le modèle optimisé est plus coûteux que celui sur le modèle original. Cela s'explique par le fait que, dans ce dernier modèle, chaque machine traite moins de données que dans le premier, ce qui signifie moins d'E/S.
D'autre part, l'exécution de l'algorithme adapté utilisant Python prend beaucoup de temps sur le modèle à 4 machines, en raison du coût de la communication, c'est pourquoi il échoue sur la plus part des graphes, alors que le temps d'exécution sur le modèle à 8 machines est raisonnable, car

Tableau 5.3 – Impact de l'optimisation du modèle (en secondes).

Ensembles de données	4 Machines		8 Machines	
	Python	SGBD	Python	SGBD
Web-google	26	29	23	39
WikiTalk	échec	36	47	40
Hyves	39	22	38	42
Youtube	23	16	18	14
as-Skitter	échec	55	512	135
Flickr-links	échec	224	échec	320
Live-Journal	échec	588	échec	757
DBpedia	échec	7689	échec	4427
Dimacs10_uk	échec	5229	échec	4012

il y a moins de traitement pour chaque machine. En résumé, la variation du temps d'exécution est principalement due à la quantité de données communiquées entre les machines pendant le rééquilibrage des arêtes pour l'algorithme adapté avec Python ou au coût des E/S pour l'algorithme adapté avec les requêtes SQL. Dans la suite, nous utilisons principalement le modèle optimisé pour évaluer l'algorithme adapté et le comparer avec les systèmes concurrents.

5.5.1.3 Partitionnement en dehors de la base de données

Dans cette section, nous étudions l'impact de l'importation de jeux de données de graphes à sommets colorés dans le SGBD, et nous analysons quel modèle peut bénéficier le plus d'une telle optimisation ; le modèle original ou le modèle optimisé. Ainsi, nous avons choisi d'effectuer cette expérience sur le modèle à 4 machines et le modèle à 8 machines. Les résultats sont présentés dans Table 5.4.

Tableau 5.4 – Coloriage avant importation.

Ensembles de données	4 Machines		8 Machines	
	Coloriage Avant Importation	Coloriage Après Importation	Coloriage Avant Importation	Coloriage Après Importation
Web-google	51	29	35	39
WikiTalk	43	36	33	40
Hyves	28	22	23	42
Youtube	10	16	7	14
as-Skitter	105	55	72	135
Flickr-links	514	224	251	320
Live-Journal	795	588	424	757
DBpedia	6939	7689	4201	4427
Dimacs10_uk	6189	5229	4531	4012

Nous constatons que la coloration [4] en dehors du SGBD réduit le temps d'exécution total, surtout avec de grands ensembles de données sur un modèle à 8 machines. Ceci est dû au fait que la coloration à l'intérieur du SGBD est effectué sur un large cluster, ce qui augmente le coût de communication des données entre les machines. Sur le modèle optimisé (4 machines), la coloration en dehors du SGBD n'est pas intéressante car elle devient plus couteûse, vue que sur ce modèle le coût de la communication est réduit mais la charge de travail envoyée à chaque machine est doublée. Dans ce modèle, le SGBD importe 4 colonnes pour les petits graphes lors

4. Le partitionnement de l'ensemble V

de la coloration en dehors du SGBD (il charge la table *E_s_proxy*), alors qu'il lit deux colonnes lorsqu'il partitionne l'ensemble V à l'intérieur du SGBD (il charge la table *E_s*). Ainsi, il crée plus de projections avec le premier, ce qui consomme plus de temps (le coût de communication des données est faible). D'autre part, l'importation de 4 colonnes est moins coûteuse pour les grands graphes que la coloration à l'intérieur du SGBD, puisque il y a une grande quantité de données à traiter pour la coloration et beaucoup d'accès au disque. En résumé, la coloration avant importation est intéressante sur les modèles de $k - machines$ avec tous les types de graphes, et moins utile sur le modèle optimisé. Dans le reste des expérimentations, nous n'utilisons pas cette optimisations, vu que tous nos tests seront exécutés sur des modèles optimisés.

5.5.2 Évaluation des résultats de l'algorithme adapté

5.5.2.1 Sur le SGBD

La présente section étudie l'équilibrage de charge de l'algorithme adapté exécuté avec les requêtes SQL. Elle analyse, en outre, l'accélération de l'algorithme en l'exécutant sur plusieurs modèles de calcul optimisés pour étudier son comportement et valider notre analyse théorique présentée dans le chapitre 3.

Équilibrage de charge La contribution ultime de l'algorithme adapté est de préserver une charge de travail équilibrée, lors du calcul des triangles. Figure 5.7, Figure 5.8 et Figure 5.9 présentent des histogrammes montrant le nombre de triangles calculés par chaque machine sur un cluster de 4, 9 et 16 machines (k = 4, 9, 16 et c = 2, 3, 4).

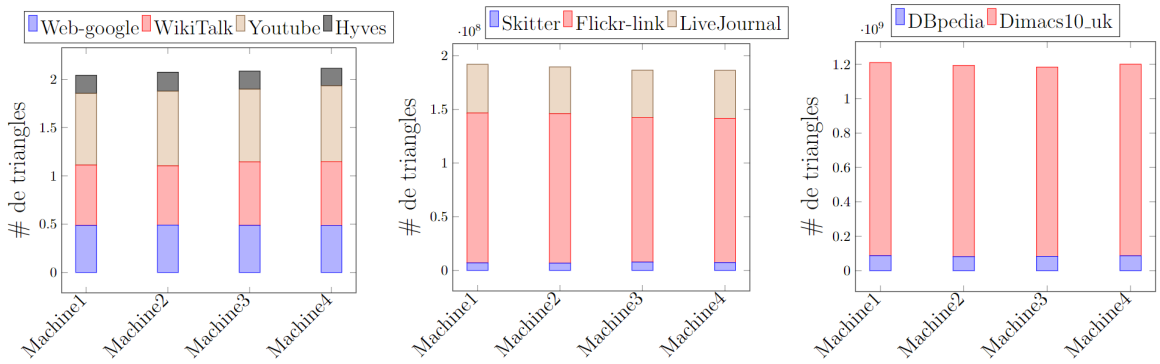


FIGURE 5.7 – Nombre de triangles par machine ($k = 4$ et $c = 2$) : charge équilibrée.

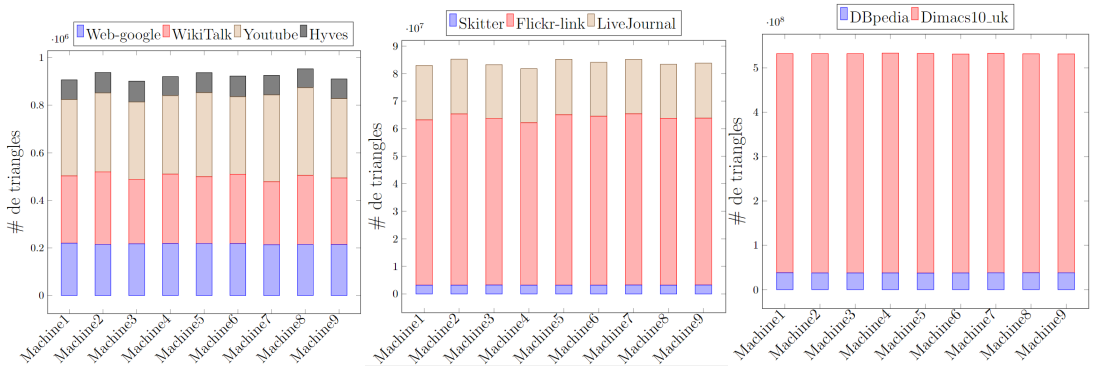


FIGURE 5.8 – Nombre de triangles par machine ($k = 9$ et $c = 3$) : charge équilibrée.

Dans chaque modèle, chaque machine produit la même quantité de triangles que les autres machines, pour chaque ensemble de données de graphe. Cela est dû au rééquilibrage des données effectué avant l'énumération et le comptage locaux des triangles. Comme mentionné dans le

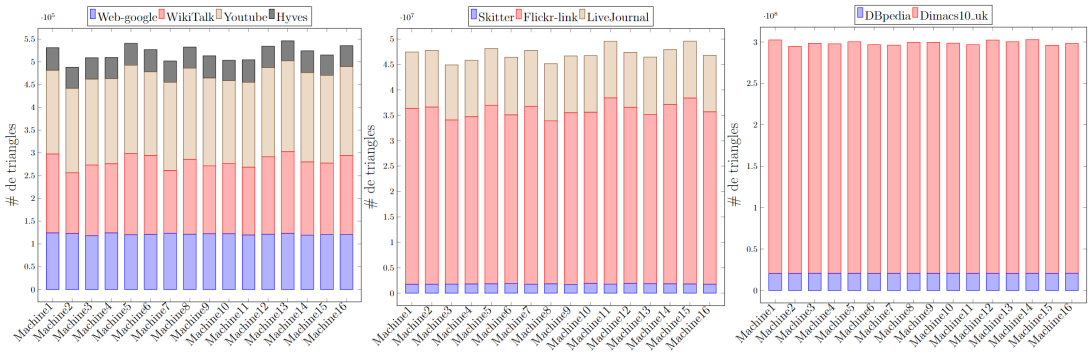


FIGURE 5.9 – Nombre de triangles par machine ($k = 16$ et $c = 4$) : charge équilibrée.

chapitre 3, le nombre d'arêtes de chaque sous-graphe sur chaque machine est relativement équilibré avec une forte probabilité. Ainsi, chaque machine traite essentiellement le même nombre de triangles, ce qui permet d'équilibrer la charge de travail.

Accélération Pour étudier l'accélération de notre algorithme implémenté avec SQL, nous fournissons Table 5.5, dans laquelle nous présentons le temps d'exécution de l'énumération/comptage des triangles, en faisant varier la taille du cluster de 1 à 16 ($k = \{1, 4, 9, 16\}$ et $c = \{1, 2, 3, 4\}$). Lorsque nous exécutons l'algorithme adapté sur une machine ($k = 1$), l'étape de partitionnement est ignorée et seule l'énumération/comptage des triangles est exécutée.

TABLEAU 5.5 – Temps d'exécution (en secondes) et l'accélération de l'algorithme adapté implémenté avec les requêtes SQL ($k = \{1, 4, 9, 16\}$ et $c = \{1, 2, 3, 4\}$ resp.).

Ensemble de données	1 Machine	4 Machines	9 Machines	16 Machines	4 → 9	4→16
Web-google	11	29	39	20	0.7	1.5
WikiTalk	27	36	40	22	0.9	1.6
Hyves	12	22	16	12	1.7	1.8
Youtube	9	16	13	9	1.2	1.7
as-Skitter	57	55	47	30	1.5	1.8
Flick-links	531	224	160	81	1.4	2.7
Live-journal	818	588	187	89	3.1	6.6
DBpedia	échec	7689	4026	3760	1.9	2
Dimasc10_uk	échec	5229	3962	3102	1.3	1.6

4→9 : l'accélération entre le modèle à 4 machines et le modèle à 9 machines ;
4→16 : l'accélération entre le modèle à 4 machines et le modèle à 16 machines ;

Lorsque le nombre de machines augmente dans le modèle de calcul, le temps d'exécution de notre algorithme diminue, car il y aura plus de machines pour traiter les triangles. Rappelons que notre algorithme préserve une charge de travail équilibrée avec $O(m/k)$, donc les machines dans les grands modèles traiteront moins de données localement et en parallèle. Nous pouvons constater que l'accélération de l'algorithme adapté à base de SQL est considérable, principalement pour les grands graphes avec des données asymétriques comme DBpedia et as-Skitter ou éparses tel que Live-journal. Sur ces graphes, le gain de vitesse double entre les modèles à 4 machines, à 9 machines et à 16 machines. Notez également qu'il n'y a pas de partitionnement sur une seule machine, c'est pourquoi le temps d'exécution sur de petits ensembles de données est meilleur. Cependant, lorsque le graphe devient plus grand, le temps d'exécution sur un cluster plus important reste plus efficace.
De outre, Table 5.6 présente une analyse de la consommation de mémoire de nos requêtes. Elle indique les pics de mémoire pour chaque exécution de requête sur différentes tailles de cluster ($k = 4, 9, 16$ et $c = 2, 3, 4$). Notre attention se portera sur les requêtes d'insertion *E_s_proxy* et

E_s_local, en plus de la requête d'énumération locale des triangles. Ces requêtes sont coûteuses car elles impliquent de nombreuses jointures. La requête d'importation du graphe et la requête d'insertion *V_s* sont presque identiques pour tous les ensembles de données, quelle que soit leur taille.

TABLEAU 5.6 – Pics de mémoire (en Go) de l'algorithme adapté implémenté avec les requêtes SQL sur la requête de *E_s_proxy*, la requête de *E_s_local* et la requête de l'énumération locale des triangles (TE) ($k = \{4, 9, 16\}$ et $c = \{2, 3, 4\}$ resp.).

Ensemble de données	4 machines			9 machines			16 machines		
	E_s_ proxy	E_s_ local	TE	E_s_ proxy	E_s_ local	TE	E_s_ proxy	E_s_ local	TE
Web-google	0.69	0.69	0.88	0.69	0.69	0.69	0.69	0.69	0.69
WikiTalk	0.69	0.69	0.88	0.69	0.69	0.69	0.69	0.69	0.69
Hyves	0.69	0.69	0.88	0.69	0.69	0.69	0.69	0.69	0.69
Youtube	0.69	0.69	0.88	0.69	0.69	0.69	0.69	0.69	0.69
as-Skitter	0.74	0.85	2.2	0.69	0.69	1.2	0.69	0.69	0.69
Flick-links	0.74	0.85	2.2	0.69	0.69	2.2	0.69	0.69	1.2
Live-journal	1.1	1.6	1.4	0.69	0.69	2.4	0.69	0.69	2.4
DBpedia	1	2.6	2.4	0.9	1	2.4	0.89	1	1.4
Dimasc10_uk	1	2.6	1.4	0.92	2.4	2.4	0.89	1.4	2.4

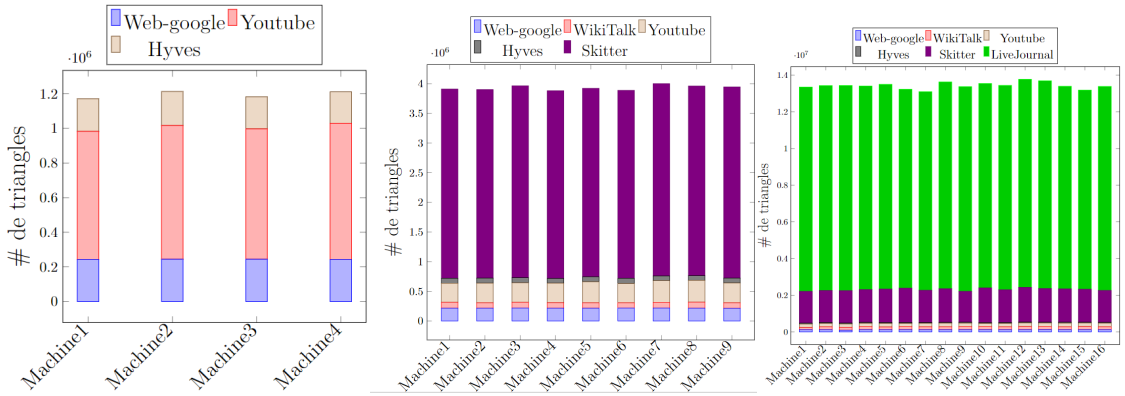
En analysant Table 5.6, nous pouvons constater que, pour de petits ensembles de données tels que YouTube et Hyves, les pics de mémoire sont les mêmes entre les trois modèles. Cela est dû au fait que le SGBD alloue la mémoire en fonction de la gestion des données requise sans dépasser un seuil d'allocation. Par exemple, Vertica alloue des parties de mémoire qui sont des puissances de 2 pour traiter les données. Ces parties deviennent plus importantes lorsque les données traitées deviennent de taille plus élevée. Cependant, le processus d'allocation garantit l'allocation de la mémoire disponible, et le processus de transaction gère l'exécution des transactions sans dépasser la mémoire allouée. C'est pourquoi nous pouvons constater que, pour des ensembles de données de taille moyenne à grande, les pics de mémoire diminuent du plus petit modèle au plus grand, car il y a plus de mémoire disponible sur ce dernier.

5.5.2.2 Programmé avec Python et MPI

Dans cette section, nous étudions l'équilibrage de charge et l'accélération de l'algorithme adapté programmé avec Python et MPI sur plusieurs modèles de calcul optimisés.

Équilibrage de charge Comme mentionné précédemment, la principale contribution de notre algorithme est de garantir un équilibrage de charge parfait. Ainsi, nous présentons Figure 5.10 qui représente l'équilibrage de charge assuré par notre algorithme programmé en Python et MPI sur différents clusters ($k = 4, 9, 16$ et $c = 2, 3, 4$). Nous présentons principalement les ensembles de données pour lesquels l'exécution a réussi, en excluant les autres.
Comme le montrent les histogrammes, chaque machine génère $\frac{1}{k}$ du total des triangles $\Delta(G)$. Cela est dû au fait que notre stratégie de partitionnement redistribue les arêtes de manière à ce que chaque machine acquiert ses arêtes respectives via le dataframe *E_s_local*, afin d'effectuer l'énumération des triangles localement en fonction de ses triplets de couleurs.

Accélération Nous présentons dans Table 5.7 une comparaison des temps d'exécution de l'énumération des triangles (comptage) en variant la taille du modèle de calcul optimisé de 4 à 16 ($k = 4, 9, 16$ et $c = 2, 3, 4$). Notez que l'exécution sur une seule machine est exclue sur laquelle tous les ensembles de données échouent.



(a) 4 Machines (b) 9 Machines (c) 16 Machines

FIGURE 5.10 – Nombre de triangles par machine ($k = \{4, 9, 16\}$ et $c = \{2, 3, 4\}$) : charge équilibrée.

TABLEAU 5.7 – Accélération et pics de mémoire de l'algorithme adapté programmé avec Python et MPI (en secondes and $k = \{4, 9, 16\}$ et $c = \{2, 3, 4\}$ resp.).

Ensemble de données	4 Machines		9 Machines		16 Machines		4→9	4→16
	Temps	Pics	Temps	Pics	Temps	Pics		
Web-google	25	32MB	70	21MB	36	18MB	1.6	2.6
WikiTalk	échec	na	46	25MB	66	21MB	na	0.69
Hyves	39	27MB	23	20MB	15	19MB	1.6	2.6
Youtube	23	43MB	16	18MB	11	20MB	1.4	2
as-Skitter	échec	na	88	24MB	54	20MB	na	1.6
Flick-links	échec	na	échec	na	échec	na	na	na
Live-journal	échec	na	échec	na	541	na	na	na
DBpedia	échec	na	échec	na	échec	na	na	na
Dimasc10_uk	échec	na	échec	na	échec	na	na	na

4→9 : l'accélération entre le modèle à 4 machines et le modèle à 9 machines ; 4→16 : l'accélération entre le modèle à 4 machines et le modèle à 16 machines ;

L'efficacité de MPI provient de son utilisation excessive d'une grande quantité de mémoire, c'est pourquoi l'algorithme adapté programmé avec Python et MPI échoue pour de grands ensembles de données tels que as-Skitter et Live-journal sur un modèle de calcul à 4 machines, et donne de meilleurs résultats sur un modèle à 16 machines. En conséquence, nous pouvons déduire qu'augmenter le nombre de couleurs pour partitionner l'ensemble des sommets réduit considérablement le temps d'exécution. De plus, l'accélération avec l'algorithme adapté est significative du plus petit au plus grand cluster. Nous pensons qu'avec une mémoire plus importante et une infrastructure plus sophistiquée, l'algorithme adapté programmé avec Python et MPI donnera de meilleurs résultats. En résumé, l'algorithme adapté programmé avec Python et MPI est plus adapté aux petits ensembles de données sur n'importe quel modèle et plus efficace pour les ensembles de données à grande échelle sur des modèles de calcul sophistiqués.

De plus, Table 5.7 présente les pics de mémoire de l'exécution de notre algorithme. La gestion de la communication de MPI (passage de message) et les opérations de fusion de Pandas de Python nécessitent un budget mémoire considérable. En conséquence, les pics de mémoire des petits ensembles de données sont bas, mais lorsque le graphe devient plus grand, il nécessite un espace mémoire plus important pour exécuter ces opérations. C'est pourquoi notre algorithme programmé avec Python et MPI échoue pour les grands ensembles de données.

5.5.3 Efficacité de l'algorithme adapté sur l'environnement parallèle

Notre objectif à travers cette section est d'étudier la robustesse et l'efficacité de l'algorithme adapté, en le comparons avec plusieurs solutions sur plusieurs environnements. En premier lieu, nous nous focalisons sur la comparaison de l'algorithme adapté avec la solution classique dans le SGBD. Le but de cette comparaison est de prouver l'efficacité de la stratégie de partitionnement sur l'accélération du temps d'exécution total de l'énumération et de comptage des triangles. Dans un deuxième lieu, nous comparons l'algorithme adapté avec deux solutions de Big Data, qui permettent l'exploration des graphes, à savoir Spark GraphX et G-thinker. Ces deux systèmes sont définis en détails dans la section de la comparaison avec notre solution.
Dans un troisième lieu, nous comparons l'algorithme adapté avec Networkx sur l'environnement Python. Cette comparaison a pour objectif d'évaluer notre solution avec un API natif d'exploration de graphes sur Python. Ces comparaisons permettent de positionner notre solution par rapport aux autres systèmes sur les différents environnements.

5.5.3.1 Algorithme adapté avec les requêtes SQL

Dans cette section, nous nous intéressons principalement à la comparaison de l'algorithme adapté implémenté avec les requêtes SQL avec (1) la solution classique à base de SQL pour l'énumération de triangles, avec (2) la fonction d'énumération de triangles de Spark GraphX, et avec (3) l'algorithme d'énumération de triangles de G-thinker.

Comparaison dans l'environnement de SGBD La comparaison entre l'algorithme adapté et l'algorithme classique sur l'environnement de base de données a pour objectif de confirmer l'efficacité des optimisations de notre approche de rééquilibrage d'arêtes sur le temps d'exécution de l'énumération des triangles (comptage). Nous présentons ainsi Table 5.8 qui affiche le temps d'exécution et le nombre des triangles obtenus par l'exécution de la solution classique et notre algorithme adapté.

Tableau 5.8 – Comparaison entre le temps d'exécution de l'algorithme adapté et l'algorithme classique (en secondes) et le nombre des triangles généré ($k = \{4, 9, 16\}$ et $c = \{2, 3, 4\}$ resp.).

Ensemble de données	Algorithme classique				Algorithm adapté			
	Temps d'exécution			Nombre de triangles	Temps d'exécution			Nombre de triangles
	4M	9M	16M		4M	9M	16M	
Web-google	9	6	5	13 391 903	33	25	20	13 391 903
WikiTalk	101	76	61	9 203 519	36	28	22	9 203 519
Hyves	124	103	52	752 401	22	16	12	752 401
Youtube	52	23	14	3 056 386	16	13	9	3 056 386
as-Skitter	170	114	77	28 769 868	55	47	30	28 769 868
Flickr-links	1649	592	336	548 174 465	224	160	81	548 174 465
LiveJournal	474	236	168	177 820 130	588	187	89	177 820 130
DBpedia	12646	5377	3930	5 774 698 667	7689	4026	3760	328 743 411
Dimacs10_uk	5323	4646	3318	4 451 687 605	5229	3962	3102	4 451 687 605

4M : 4 Machines ; 9M : 9 Machines ; 16M : 16 Machines.

D'après Table 5.8, nous pouvons conclure que notre algorithme adapté présente de meilleures performances. Ceci est particulièrement constaté pour les graphes à distribution asymétrique, comme WikiTalk, Youtube et Hyves. Pour ces graphes, l'algorithme classique a besoin d'échanger beaucoup d'arêtes entre les machines, afin d'énumérer tous les triangles. De plus, le rééquilibrage des arêtes des graphes comme as-skitter, flickr-links et DBpedia, dans l'algorithme adapté réduit considérablement le temps d'exécution du comptage local des triangles. Sachant que ces graphes sont de grands graphes avec une forte asymétrie et beaucoup de cliques. En revanche, l'algorithme

classique est extrêmement coûteux, du fait qu'il présente une communication excessive entre les machines (en raison du grand nombre de sommets fortement asymétriques). Enfin, Live-journal, qui est un grand ensemble de données, et Web-google, qui a une distribution uniforme des sommets, présentent un meilleur temps d'exécution pour l'algorithme classique, car il s'agit de graphes symétriques clairsemés. Brièvement, l'algorithme adapté convient aux graphes fortement asymétriques, quelle que soit leur taille.
Nous présentons dans Figure 5.11 et Figure 5.12 une comparaison entre l'algorithme adapté et l'algorithme classique, en termes du nombre de triangles renvoyés par chaque machine. Remarquons que toutes les lignes des graphiques de l'algorithme adapté sont presque alignées. Ceci indique que l'équilibrage de la charge est préservé, tout en calculant les triangles, en particulier avec des graphes très asymétriques comme WikiTalk, Hyves et les Flickr-links. En revanche, l'algorithme classique déséquilibre la charge de travail, ce qui explique son résultat déséquilibré et son temps d'exécution élevé.

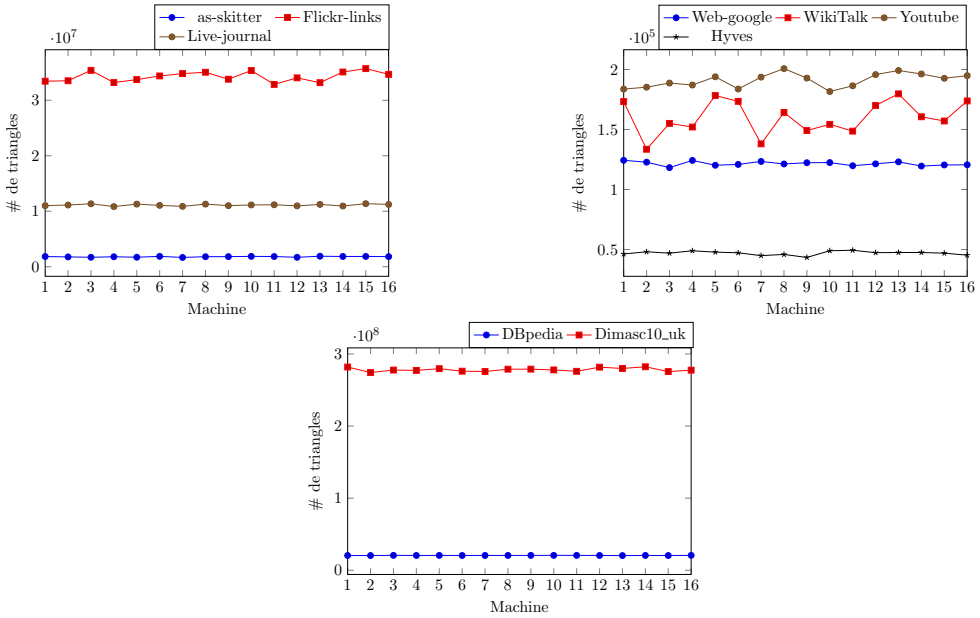


FIGURE 5.11 – Nombre de triangles par machine avec l'algorithme adapté.

Comparaison avec les systèmes Big Data Dans cette section, nous comparons notre algorithme adapté exécuté sur un SGBD avec G-thinker [YGRC+20] et Spark GraphX. G-thinker est un moteur d'exécution basé sur les tâches pour le traitement de graphes. Il comprend de nombreux algorithmes de graphes tels que la recherche de sous-graphes et le comptage des triangles. Il bénéficie de l'efficacité de Hadoop en tant que système de traitement de données volumineuses et de MPI en tant que bibliothèque d'exécution de tâches parallèles. Ce système constitue une bonne référence pour le comparer avec notre solution. Il s'agit d'un système de type Pregel qui inclut MPI et Hadoop.
D'un autre côté, Spark est un système de calcul distribué pour le traitement des données massives, il définit un ensemble de modules, entre autres Spark GraphX. GraphX est un module pour le traitement des graphe qui offre une suite de fonctions élémentaires, dont la fonction du comptage des triangles. Dans Spark GraphX, un ensemble de fonctions intégrées pour le partitionnement des arêtes est inclus. Ce partitionnement applique la technique du vertex-cut (coupure de sommet). Il s'agit d'une technique de partitionnement du graphe, basée sur la détermination du seul sommet dont la suppression déconnecte le graphe. Dans ce cas, les sommets peuvent avoir besoin d'être répliqués. Les arêtes sont partitionnées par l'une des stratégies suivantes :

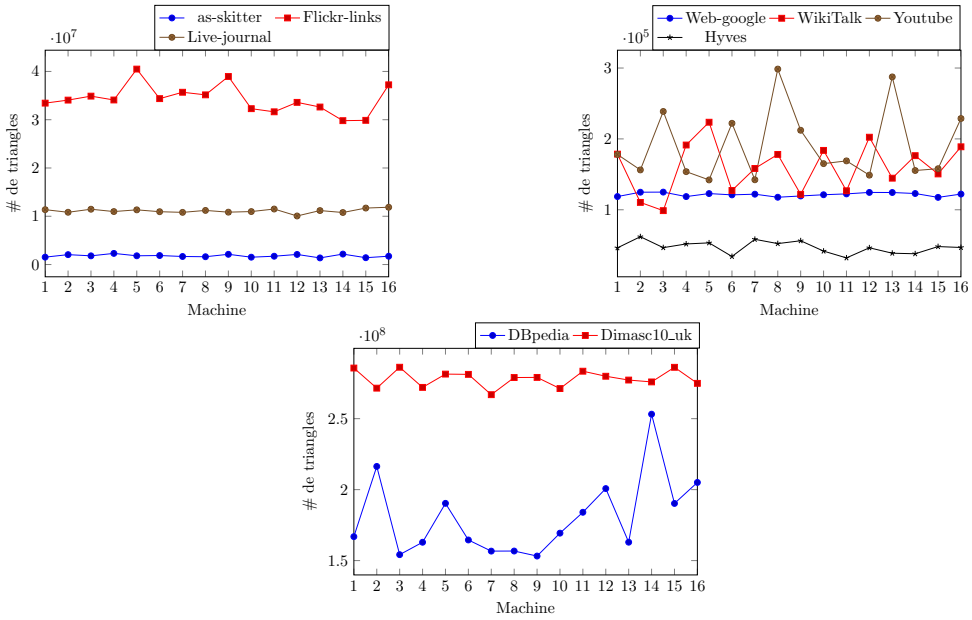


FIGURE 5.12 – Nombre de triangles par machine avec l'algorithme classique.

(1) **Random vertex-cut** : le graphe est partitionné en attribuant ses arêtes aux machines de calcul de manière aléatoire.

(2) **Edge Partition 1D** : la matrice d'adjacence du graphe d'entrée est partitionnée horizontalement.

(3) **Edge Partition 2D** : la matrice d'adjacence est partitionnée en grille, à la fois horizontalement et verticalement.

(4) **Canonical Random vertex cut** : les arêtes sont affectées aux partitions en hachant les identifiants de sommet source et de destination dans une direction canonique, ce qui entraîne une coupe de sommet aléatoire qui colocalise toutes les arêtes entre deux sommets, quelle que soit la direction.

Notre choix de G-thinker et Spark GraphX est justifié par le fait que ces deux systèmes sont robustes et similaires au SGBD en termes de capacités.
Dans Table 5.9, nous résumons la comparaison entre les trois systèmes sur un modèle de calcul à 4 machines. Nous avons choisi ce modèle car : (i) nous voulons évaluer l'efficacité de notre système par rapport aux autres systèmes sur des ressources mémoire et de traitement limitées, (ii) G-thinker est un moteur basé sur les tâches et peut fonctionner parfaitement avec un budget de mémoire réduit, et (iii) Spark GraphX est un puissant moteur de traitement parallèle de graphes. Nous sommes convaincus que les trois systèmes peuvent donner des résultats parfaits sur des clusters plus puissants et avec des configurations matérielles sophistiquées. Cependant, nous souhaitons positionner notre solution, parmi d'autres, sur un cluster avec des limitations de CPU et de mémoire.
Avant d'exécuter l'algorithme G-thinker pour le calcul des triangles, nous convertissons tous les ensembles de données au format de liste d'adjacence. Notez que le formatage des ensembles de données prend du temps, surtout pour les ensembles de données volumineux. Cependant, nous n'avons pas inclus ce temps dans le temps total d'exécution de G-thinker. De plus, pour garantir une comparaison équitable entre les trois systèmes, nous avons exécuté G-thinker avec 4 threads, car nous disposons de 4 cœurs de CPU sur chaque machine.
Pour les ensembles de données de petite à moyenne taille comme Hyves, Youtube et as-Skitter, l'algorithme adapté est plus efficace avec un meilleur temps d'exécution. Notez que pour l'ensemble de données Hyves, G-thinker renvoie moins de triangles car il s'agit d'un ensemble de

TABLEAU 5.9 – Comparaison de l'algorithme adapté avec SQL contre Spark GraphX et G-thinker en termes de temps d'exécution (en secondes), et du nombre de triangles (TC) sur 4 machines ($k = 4$ et $c = 2$).

Ensembles de données	GraphX		G-thinker		Adapté avec SQL	
	Temps	TC	Temps	TC	Temps	TC
Web-google	75	13,391,903	8	13,391,903	33	13,391,903
WikiTalk	176	9,203,519	35	9,203,519	36	9,203,519
Hyves	119	752,401	23	538,362	22	752,401
Youtube	112	3,056,386	30	3,056,386	16	3,056,386
as-Skitter	échec	na	101	28,769,868	55	28,769,868
Flick-links	échec	na	92	548,174,465	224	548,174,465
Live-journal	échec	na	91	177,820,130	588	177,820,130
DBpedia	échec	na	28,579	154,277,630	7,689	328,743,411
Dimasc10_uk	échec	na	échec	na	5,229	4,451,687,605

données très déséquilibré avec $d\_max = 31,883$, et Spark GraphX donne le pire temps d'exécution. En revanche, G-thinker donne le meilleur temps d'exécution sur de plus grands ensembles de données tels que Flickr-links et Live-journal. Pour ces ensembles de données, G-thinker bénéficie de sa stratégie de partitionnement sur HDFS pour compter efficacement les triangles. En contre partie, Spark GraphX échoue en raison d'un manque de mémoire. En outre, G-thinker prend beaucoup de temps pour l'ensemble de données DBpedia en renvoyant un nombre erroné de triangle et plante pour l'ensemble de données Dimacs10_uk en raison d'un problème de mémoire (erreur de segmentation). Enfin, l'algorithme adapté termine l'énumération des triangles sur ces deux ensembles de données en environ 2 heures sur un cluster modeste. Notamment, les trois systèmes comptent efficacement les triangles, mais seul l'algorithme adapté génère *la liste des triangles en plus du nombre total de triangles*. En fait, Spark GraphX, par exemple, se limite au comptage de triangles (pas d'énumération). Sa fonction de comptage de triangles incrémente le nombre de triangles chaque fois que trois sommets adjacents ont des arêtes entre eux, quel que soit leur sens. Pour compter les triangles, Spark crée des tâches sur le nœud maître et les envoie aux workers pour exécution. Chaque worker utilise des conteneurs pour lire les données, puis produit des résultats et les envoie au nœud maître, qui collecte tous les résultats des workers pour les afficher. Ainsi, le nœud maître attend que tous les workers aient terminé avant d'envoyer les résultats finaux. Toutes ces étapes intermédiaires consomment du temps et de l'espace. De plus, Spark dépend de la configuration de l'écosystème. Il est plus adapté aux calculs à haute performance, où chaque nœud définit suffisamment de mémoire et de capacités de traitement. C'est une raison de l'échec de Spark GraphX pour traiter de grands graphes tels que as-Skitter et Flickr-links, et de la longue durée d'exécution pour de petits graphes comme Youtube sur notre configuration.
En résumé, l'algorithme adapté s'exécute parfaitement sur un cluster modeste et fournit le nombre exact de triangles pour tous les ensembles de données, quelle que soit leur taille ou leur type (degré d'asymétrie).

5.5.3.2 Algorithme adapté programmé avec Python et MPI

Cette section sert pour évaluer notre solution programmée avec Python et MPI et la positionner par rapport aux systèmes de Big Data. Cette solution développée sous Python avec la library Pandas qui utilise la structure de données *DataFrame* pour manipuler les données du graphe et MPI pour établir la communication entre les machines du modèle de calcul. Rappelons que MPI [GGH+96] est une spécification pour une bibliothèque standard pour le modèle de transmission de messages utilisé dans les calculs parallèles. Il définit une communication de point à point et des opérations collectives.

Dans un premier lieu, nous comparons notre solution programmée avec Python et MPI contre notre solution avec SQL. Ensuite, nous la positionnons par rapport à NetworkX et Spark GraphX.

Comparaison contre l'algorithme adapté avec SQL Table 5.10 synthétise le temps d'exécution de l'algorithme adapté sur les deux plateformes : (1) Python et MPI, et (2) sur SGBD, sur différents clusters ($k = \{4, 9, 16\}$ et $c = \{2, 3, 4\}$).

TABLEAU 5.10 – Comparaison de l'algorithme adapté avec Python et MPI contre l'algorithme adapté avec SQL (en secondes avec $k = \{4, 9, 16\}$ et $c = \{2, 3, 4\}$ resp.).

Ensembles de données	Algorithme adapté avec Python et MPI			Algorithme adapté avec SQL		
	4M	9M	16M	4M	9M	16M
Web-google	26	71	36	33	25	20
WikiTalk	échec	44	66	36	28	22
Hyves	39	23	15	22	16	12
Youtube	23	16	11	16	13	9
as-Skitter	échec	88	54	55	47	30
Flick-links	échec	échec	échec	224	160	81
Live-journal	échec	échec	541	588	187	89
DBpedia	échec	échec	échec	7689	4026	3760
Dimasc10_uk	échec	échec	échec	5229	3962	3102

4M : 4 Machines ; 9M : 9 Machines ; 16M : 16 Machines.

L'algorithme adapté programmé avec Python et MPI renvoie des résultats équivalents à l'algorithme adapté avec des requêtes SQL. Cependant, MPI dépend principalement de la taille de la mémoire principale, c'est pourquoi il plante pour les grands ensembles de données, lorsqu'il ne peut pas allouer suffisamment de mémoire. L'algorithme adapté programmé avec Python et MPI devient plus efficace lorsque la taille du cluster augmente et que la mémoire principale devient suffisamment grande pour traiter l'ensemble des données du graphe. En revanche, l'algorithme adapté avec des requêtes SQL réussit son exécution sur tous les ensembles de données quelque soit leur taille. En effet, la structure du système de base de données relationnelle définit une exécution efficace en mémoire principale et des techniques d'E/S optimisées lorsque les données dépassent la taille de la mémoire principale.

Synthèse SQL contre MPI : En comparaison avec Python et MPI, le langage SQL est plus élégant, abstrait et plus facile à comprendre avec des connaissances de base en SQL. Il utilise principalement les opérations SPJA (Sélection, Projection, Jointure et Agrégation). De plus, tous les opérateurs relationnels en SQL fonctionnent naturellement en parallèle. Par conséquent, il n'est pas nécessaire d'envoyer/recevoir explicitement des messages avec des tampons et de dupliquer des fichiers dans les implémentations SQL. Ainsi, une implémentation SQL peut fonctionner à la fois sur un nœud unique et sur un cluster parallèle sans nécessiter de reprogrammation, tandis qu'adapter une solution sérielle à une solution parallèle avec MPI est compliqué et chronophage. Il est vrai que le langage SQL présente certaines limitations : (1) il nécessite le chargement de fichiers, (2) chaque entrée et résultat intermédiaire doivent être stockés dans des tables, et (3) il est difficile d'exprimer une logique compliquée en raison du peu de contrôle sur les structures de données en RAM. Cependant, dans le cas du problème du triangle, toutes ces limitations sont acceptables. Le graphe d'entrée et les triangles de sortie peuvent être stockés au format de table, et notre algorithme peut être exprimé en SQL avec l'aide de stratégies de partitionnement fournies par les systèmes de base de données. De plus, les solutions SQL n'ont pas de limitations de mémoire, en revanche l'implémentation avec Python et MPI échoue lorsque

les graphes sont trop grands pour tenir dans la mémoire principale.

Comparaison contre Networkx et Spark GraphX NetworkX [HSS08] est un package Python pour la création, la manipulation et l'étude de la structure, de la dynamique et des fonctions des graphes. Il définit un ensemble de fonctionnalités pour charger et stocker des graphes dans des formats de données standard et non standard, générer de nombreux types de graphes aléatoires et classiques, analyser la structure du graphe, dessiner des graphes ... etc.
Ainsi, nous comparons l'algorithme adapté implémenté avec Python et MPI, contre la fonction prédéfinie pour le calcul des triangles de Netwokx et celle de Spark GraphX. Notre choix de la librairie NetworkX est justifié par le fait que nous voulions comparer notre solution avec une solution native sur l'environnement Python. En revanche, notre comparaison avec Spark GraphX sert pour le positionnement de notre solution par rapport un moteur d'analyse de graphe. Pour élaborer une comparaison équitable entre notre solution et la fonction de calcul des triangles pré-définie dans Spark GraphX. Nous avons choisi le partitionnement *Canonical Random vertex cut* pour partitionner le graphe sur Spark.
Nous présentons Table 5.11, dans laquelle nous résumons les résultats de temps d'exécution des trois solutions sur un modèle de calcul à 16 machines. Notre choix de ce modèle est justifié par le fait que nous voudrions étudier l'impact de la communication et du modèle de partitionnement sur le temps d'exécution global de l'énumération et du comptage de triangles.

Tableau 5.11 – Comparaison de l'algorithm adapté avec Python et MPI contre Spark GraphX et Networkx en termes de temps d'exécution (en secondes), et le nombre de triangles (TC) sur 16 machines ($k = 16$ et $c = 4$).

Ensemble de données	Networkx		Spark GraphX		Algorithme adapté avec Python et MPI	
	Temps	TC	Temps	TC	Temps	TC
Web-google	354	13,391,903	64	13,391,903	36	13,391,903
WikiTalk	échec	na	138	9,203,519	66	9,203,519
Hyves	289	752,401	92	752,401	15	752,401
Youtube	échec	na	90	3,056,386	11	3,056,386
as-Skitter	échec	na	échec	na	54	28,769,868
Flick-links	échec	na	échec	na	échec	na
Live-journal	échec	na	échec	na	541	177,820,130
DBpedia	échec	na	échec	na	échec	na
Dimasc10_uk	échec	na	échec	na	échec	na

A partir de Table 5.11, nous pouvons déduire que notre algorithme codé en Python et MPI donne de meilleurs temps d'exécution sur les graphes de petite taille et réussit l'exécution pour l'ensemble de données Live-journal, tandis que Spark et Networkx échouent. Il faut noter que Networkx renvoie le nombre de triangles avec la répétition, par conséquent, il faut le diviser par trois, pour retrouver la valeur exacte des triangles. En outre, les trois systèmes échouent pour les ensembles de données grands et très grands en raison du manque de mémoire. Notre algorithme affiche un temps d'exécution satisfaisant en comparaison avec Spark GraphX et NetworkX, cependant, il reste tributaire de la taille de la mémoire principale et ne peut rivaliser en efficacité avec l'algorithme adapté mis en œuvre à l'aide de requêtes SQL.

5.6 Discussion

Dans cette section, nous résumons tous les résultats précédents pour pouvoir en tirer les conclusions, puis nous positionnons notre solution par rapport aux autres sur les différents plateformes.

Table 5.12 résume tous les temps d'exécution des différents solutions/plateformes présentés dans ce chapitre. Nous présentons pour tous les ensembles de données, le temps d'exécution : (1) sur le SGBD entre la solution classique et notre algorithme adapté avec les requêtes SQL, (2) sur Python entre NetworkX et notre algorithme adapté avec Pandas, (3) sur Spark GraphX, et (4) sur G-thinker. Nous avons utilisé des modèles de calcul optimisé de taille 4, 9 et 16 machines. Notez que les résultats sur G-thinker ne sont donnés que sur le modèle à 4 machines. Ce système dépend entièrement d'une configuration complexe combinée de Hadoop, MPI et Java et nous n'avons pas réussi à l'utiliser que sur le modèle susmentionné.

Tableau 5.12 – Résumé des résultats (en secondes).

Ensembles de données		SGBD		Python		GraphX	G-thinker
		Adapté	Classique	Adapté	Networkx		
web-google	4M	33	9	26	échec	75	8
	9M	25	6	71	échec	69	/
	16M	20	5	36	254	64	/
WikiTalk	4M	36	101	échec	échec	176	35
	9M	28	76	44	échec	155	/
	16M	22	61	66	échec	138	/
Hyves	4M	22	124	39	échec	75	23
	9M	16	103	23	échec	68	/
	16M	12	52	15	289	55	/
Youtube	4M	16	52	23	échec	échec	30
	9M	13	23	16	échec	105	/
	16M	9	14	11	échec	90	/
as-Skitter	4M	55	170	échec	échec	échec	101
	9M	47	114	88	échec	échec	/
	16M	30	77	54	échec	échec	/
Flick-links	4M	224	1649	échec	échec	échec	92
	9M	160	592	échec	échec	échec	/
	16M	81	336	échec	échec	échec	/
Live-journal	4M	588	474	échec	échec	échec	91
	9M	187	236	échec	échec	échec	/
	16M	89	168	541	échec	échec	/
DBpedia	4M	7689	12646	échec	échec	échec	28,579
	9M	4026	5377	échec	échec	échec	/
	16M	3760	3930	échec	échec	échec	/
Dimasc10_uk	4M	5229	5323	échec	échec	échec	échec
	9M	3962	4646	échec	échec	échec	/
	16M	3102	3318	échec	échec	échec	/

En analysant Table 5.12 et les résultats d'exécution présentés dans ce chapitre, nous avons appris les leçons suivantes :

(1) L'optimisation du modèle est plus adaptée aux graphes de petite et moyenne taille. En revanche, elle est légèrement plus coûteuse pour les graphes de grande taille que le modèle original. En effet, en comparant les résultats d'exécution des petits graphes comme *web-google* ou *Hyves*, nous nous rendons compte que le gain en temps d'exécution est négligeable, contrairement aux résultats d'exécution sur les grands graphes tels que *Flickr-links* et *Live-journals*, où la différence en temps d'exécution entre les différents modèles de calcul, est très importante.

(2) L'algorithme adapté préserve un équilibrage de charge parfait quelles que soient la taille et la nature (clairsemé ou asymétrique) du graphe. La réaffectation des arêtes lors de

l'étape du partitionnement a permis d'envoyer toutes les arêtes, qui constitue un triangle à une seule machine. Cet envoie a été effectué d'une façon équitable et aléatoire entre les machines du modèle de calcul. Par conséquent, chaque machine renvoie presque le même nombre de triangles (sans redondance). Ceci préserve l'équilibrage de la charge de travail entre les machines.

(3) L'algorithme adapté présente une excellente accélération. Lorsque la taille du modèle augmente, le temps d'exécution diminue. Il peut présenter un temps d'exécution élevé avec Python, en raison du coût de communication. Cependant, le temps d'énumération locale diminue considérablement. Effectivement, le modèle de calcul utilisé consomme une portion importante du temps d'exécution lors du calcul des triangles. En élargissant la taille du modèle de calcul, la charge de travail diminue sur chaque machine durant l'énumération et le comptage des triangles (chaque machine traite un nombre plus faible de triangles). En outre, le SGBD est doté de techniques optimisées qui permettent de réduire le temps de communication et d'E/S. Par conséquent, l'accélération est parfaite sur la solution à base de requêtes SQL. En contre partie, nous constatons une accélération acceptable sur la solution à base de Pandas. Ceci est expliqué par le fait que le modèle de communication MPI est fortement influencé par le type des canaux de communication et les protocoles de réseaux utilisés, y compris la bande passante définie sur le réseau. Lorsque le nombre de machine augmente dans le modèle de calcul, la communication deviennent plus importante, particulièrement pour les petits graphes. En revanche, le traitement local sur chaque machine devient plus optimisé. Ainsi, nous recensons un gain important en temps d'exécution au niveau du calcul local des triangles, et une petite perte en temps d'exécution au niveau du partitionnement.

(4) L'équilibrage de la charge et l'accélération de notre algorithme assurent son scalabilité. Lorsque la taille du modèle ou du graphe augmente, l'algorithme adapté donne des résultats corrects en un temps d'exécution optimisé. Nous pouvons constater ceci plus clairement avec la solution sur le SGBD. Nous avons pu exécuter l'algorithme adapté sur des graphes avec des structures complexes (cliques) sur des différents tailles du modèle de calcul. Ainsi, nous nous sommes convaincus que cet algorithme permet le passage effective à l'échelle.

(5) La coloration avant l'importation est plus intéressante sur le modèle $k-machines$ avec tous les graphes, et sur un modèle optimisé avec des graphes larges et asymétriques. En affect, les graphes denses asymétriques possèdent un nombre important d'arêtes, ce qui augmente le temps d'exécution de l'étape de coloration. Ainsi, lorsque cette dernière est effectuée en dehors du SGBD, le temps d'exécution total du calcul des triangles est considérablement diminué.

(6) L'algorithme adapté avec les requêtes SQL convient aux graphes asymétriques, quelle que soit leur taille. Il n'échoue jamais à cause de la limitation de la mémoire, puisque le SGBD définit des techniques optimisés d'E/S par rapport à Spark GraphX. Ce dernier repose sur la configuration du système hôte, donc il dépend de la taille de la mémoire principale. Ceci explique le fait que Spark GraphX échoue avec les graphes à grande échelle.

(7) L'algorithme adapté implémenté avec Pandas et MPI est efficace et donne un temps d'exécution plus faible par rapport à Networkx. Effectivement, Networkx calcule les triangles avec redondance. Ainsi, pour renvoyer le nombre exacte des triangles, il faut éliminer tous les triangles en double. En revanche, l'algorithme adapté avec Pandas élimine la redondance lors de l'étape du partitionnement grâce à la technique de coloriation, ainsi que l'ordre lexicographique défini lors du calcul local des triangles.

(8) L'algorithme adapté avec Python est plus adéquat avec le modèle $k-machines$ de taille moyenne, où le coût de communication est plus optimisé. Il peut également être appliqué à des graphes épars (qui contiennent un nombre faible d'arêtes), qui ne nécessitent pas une taille importante de mémoire principale. Ainsi, il peut être facilement utilisé dans des domaines comme la biologie, la CAO, etc., où l'énumération des triangles n'est qu'une étape

dans leurs algorithmes. D'autre part, l'algorithme adapté sur le SGBD fonctionne parfaitement avec les graphes denses asymétriques, et avec un modèle $k-machines$ modeste, il est donc pratique pour les organisations, qui possèdent une configuration matérielle modeste, et des graphes plus compliqués à explorer.

Pour résumer, l'algorithme adapté utilisant les requêtes SQL peut être appliqué sur plusieurs systèmes, qui offrent un support pour un langage équivalent à SQL et définissent une architecture permettant le partitionnement des données et le calcul parallèle. Il peut être donc exécuté sur des systèmes d'exploration de graphes comme *TigerGraph*, avec des systèmes de Big Data tel que *Spark SQL* ou avec des SGBDR orientés lignes, colonnes ou table (array). D'autre part, l'algorithme adapté sur Python peut être appliqué pour explorer des petites structures (graphes), afin de résoudre une étapes, parmi plusieurs dans un grand projet. En effet, le projet nécessitent l'intervention de plusieurs solutions/technologies pour les conduire. Ainsi, l'API Pandas est souvent utilisé sur plusieurs étapes. Ceci permet l'intégration facile et efficace notre algorithme adapté dans le pipeline de la conduite du projet.

5.7 Conclusion

Dans ce chapitre, nous avons présenté une évaluation expérimentale de notre algorithme adapté pour l'énumération et le comptage des triangles. Dans un premier lieu, nous avons défini le cadre matériel et logiciel exploité dans cette étude expérimentale. Notre configuration matérielle est adéquate avec la définition du modèle de calcul $k-machines$, où nous avons exploité un cluster de 16 machines, construit sur une architecture shared-nothing. Ainsi, notre configuration logicielle implique le SGBD orienté colonnes Vertica et l'API Pandas de Python pour l'implémentation de l'algorithme adapté. Ce dernier est comparé avec l'API Networkx, Spark GraphX et G-thinker, considérés comme des solutions référentielles. Ensuite, nous avons détaillé l'ensemble des techniques d'optimisation que nous pouvons appliquer sur notre solution, afin d'assurer une exécution efficace et robuste. Pour cela, nous avons présenté la structure interne du SGBD en colonne, ainsi que ses différentes solutions appliquées pour optimiser l'espace et le temps d'exécution. Puis, nous avons expliqué l'application de ces techniques sur l'algorithme adapté. En outre, nous avons défini les différentes règles pour l'élimination des résultats redondants le plutôt possible dans l'algorithme. Dans un deuxième lieu, nous avons élaborer une évaluation expérimentale de l'algorithme adapté selon trois axes :

(1) Une évaluation des techniques d'optimisation appliquées : nous nous sommes principalement concentrés sur l'influence de répliquer la table lors de l'exécution d'une auto-jointure, et l'élimination des triangles redondants, en considérant que les triangles lexicographiquement triés.

(2) Une évaluation de l'efficacité et de la robustesse de notre algorithme en termes d'équilibrage de charge, d'accélération et de scalabilité.

(3) Une évaluation de notre algorithme dans différents environnements : nous avons comparé l'algorithme adapté avec la solution classique dans le SGBD, avec Networkx dans Python, et avec Spark GraphX et G-thinker comme solution de Big Data.

Nous avons enfin résumé les résultats obtenus et les analyser. Nous avons tiré plusieurs conclusions, entre autres que l'algorithme adapté est efficace avec les graphes asymétriques et de grande taille, il préserve un équilibrage parfait de charge de travail et il permet le passage à l'échelle.

Conclusion et perspectives

Conclusion

Le partitionnement et le traitement parallèle demeurent un défi et un enjeu majeur dans le domaine d'analyse et d'exploration des graphes. Très particulièrement lorsqu'il s'agit des graphes construits à partir des données relationnelles. En effet, les SGBD relationnels ont pour rôle stocker et manipuler une quantité importante des données. Notons que ces dernières peuvent être facilement modélisées et explorées sous forme de graphes. Dans ce contexte, cette thèse vise à explorer les graphes avec des requêtes qui peuvent être exécutées sur des SGBD relationnels ou des librairies équivalentes comme Pandas de Python. Le principale objectif de cette thèse est d'énumérer les triangles des graphes à grand échelle sur une infrastructure distribuée. Cette énumération est effectuée à l'aide des requêtes d'une manière locale et parallèle, tout en préservant l'équilibrage de charge de travail entre les nœuds de traitement. Ainsi, notre contribution principale s'agit d'un algorithme adapté pour l'énumération équilibrée des triangles dans les graphes à grande échelle. Nous résumons dans la suite les différents aspects de notre solution.

Représentation des graphes

Le graphe en entrée est représenté sous forme d'une table d'arêtes E_s avec deux colonnes, où la première colonne représente le sommet source et la deuxième le sommet de destination. Le choix de cette représentation est justifié par le fait de sa simplicité, et son efficacité lors de sa reproduction sur la base de données ou avec Pandas. De plus, nous avons construit la table E_s de telle sorte que toutes les arêtes qui possèdent le même sommet source sont listé consécutivement. Autrement dit, la table E_s n'est rien qu'une liste d'adjacence listée sur deux colonnes.

Modèle de calcul et le système d'exécution

Notre solution s'exécute sur une infrastructure distribuée qui a motivé notre choix d'utiliser le modèle de calcul $k-machines$. Ce modèle se définit par un ensemble de machines indépendantes interconncectées entre elles via un réseau. Elles communiquent par passage de message et ne partage aucune ressource y compris la mémoire et le disque. Notre solution exige un cluster de taille $k > 2$ machines, respectant la formule ci-après :

$$k = c^3 \ / \ c \in \{2, 3, 4 \cdots\}$$

Pour l'exécution de notre solution, nous avons opté pour Vertica ; un SGBD orienté colonne, qui permet le contrôle du partitionnement de données. En effet, Vertica est un SGBD basé sur le MPP et compatible avec l'architecture shared-nothing. En outre, nous avons utilisé MPI avec Pandas pour évaluer notre solution programmée avec Python. Les deux systèmes sélectionnés respectent les exigences de notre solution et sont compatibles avec le modèle $k - machines$.

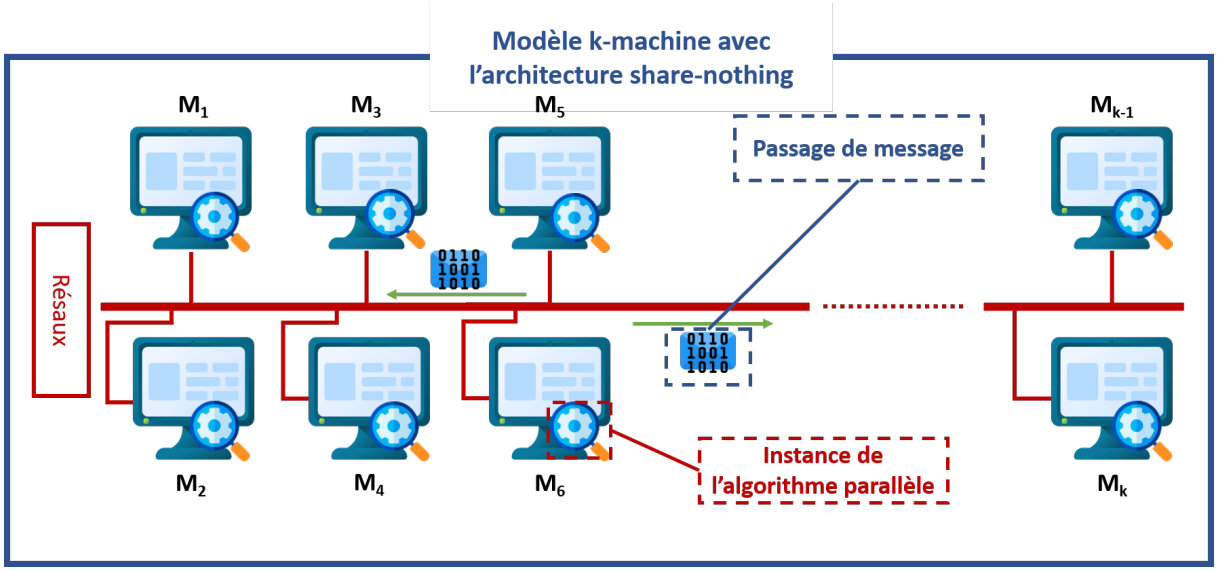


FIGURE 6.1 – Modèle $k-machines$.

Partitionnement du graphe

Le partitionnement représente l'étape la plus importante dans notre algorithme. Elle joue un rôle primordiale dans la division équitable de la charge de travail entre les nœuds de traitement, tout en garantissant un traitement local de triangles. Notre stratégie de partitionnement a pour objectif de diviser l'ensemble de sommet V en c ensembles de taille $O(V/c)$ avec $c = k^{1/3}$. Ensuite, chaque machine du modèle $k-machines$ construit un sous-graphe induit à partir des différents triplets de sommets, selon une technique de coloriage. Ces sous-graphes induits sont utilisés pour calculer localement et parallèlement tous les triangles. Précisément, chaque machine du modèle $k-machines$ reçois un triplet de couleurs construit à partir des différents permutations entre les c couleurs. Ces couleurs c sont utilisés pour partitionner l'ensemble V, où chaque sommet choisit aléatoirement et uniformément une couleur parmi celles disponibles. Ensuite, chaque arête entre les différents sommets colorés est envoyée aléatoirement à une machine proxy. Cette dernière sert pour un point de collecte pour construire les sous-graphes induits. En effet, chaque machine locale récupère ses arêtes manquantes à partir des machines proxys, en se servant de son triplet unique de couleurs.

Énumeration locale des triangles

Après avoir partitionné le graphe, chaque machine exécute la requête d'énumération et de comptage de triangles. Cette opération est totalement locale, car chaque machine possède tous les arêtes nécessaires pour former ses triangles. Ceci est garantit par la stratégie de partitionnement. Par conséquent, aucune communication entre les machines n'est requise, et aucune arête n'est échangée lors du calcul des triangles.

PandaSQL

Pour rendre accessible notre algorithme adapté aux utilisateurs quel que soit leur niveau de connaissance, nous avons avons développé une application Web, appelée *PandaSQL* accessible sur `https://github.com/lias-laboratory/pandasql`. Cette application permet d'évaluer visuellement l'équilibrage de charge de l'algorithme adapté et le comparer avec celui de l'algorithme de Cohen [Coh09] implémenté sur un SGBD relationnel. PandaSQL offre aux utilisateurs débutants la possibilité de charger un graphe sous forme de liste d'arêtes et d'utiliser un modèle $k-machines$, pour exécuter l'algorithme adapté. L'utilisateur aura en sortie trois informations :

(a) les topk des sommets asymétriques, (b) la taille et l'ordre du graphe d'entrée, et (c) deux graphiques circulaires qui représentent respectivement l'équilibrage de charge de l'algorithme adapté et celui de l'algorithme classique de Cohen. PandaSQL offre aux experts la possibilité de charger les requêtes SQL en lignes de commande et explorer la taille des partitions générées. PandaSQL a été présenté lors de ACM CIKM 2020 comme un article de démonstration.

Complexité de l'algorithme adapté

Cette complexité dépend fortement de l'étape du partitionnement, de la taille et de la densité du graphe en entrée. Nous avons effectué une analyse détaillée dans le chapitre 3. Dans cette étude, nous avons conclu que la complexité de l'algorithme est de $O(m)$ en considérant une distribution uniforme du graphe (graphe creux), où m représente la taille du graphe (le nombre d'arêtes). Elle peut atteindre $O(m^2)$ pour les graphes denses dont la structure est très complexe (ils contiennent plusieurs cliques d'ordre p).

Équilibrage de charge et scalabilité

La contribution principale de notre solution est de fournir une stratégie de partitionnement garantissant un équilibrage de charge parfait entre les machines de traitement. Cet équilibrage est la conséquence directe de la stratégie du partitionnement utilisée. Effectivement, cette dernière assure une distribution équitable de sommets avec une forte probabilité. Ceci permet de partitionner les arêtes sur l'ensemble des machines d'une manière équilibrée. Ainsi, chaque machine effectue la tâche d'énumération de triangles dans un temps relativement similaire aux autres. De plus, elle renvoie le même nombre de triangles que les autres machines ($O(\Delta/k)$). Par conséquent, notre stratégie préserve l'équilibrage de charge de travail entre les machines, tout en garantissant un traitement local de triangles.
Notons que l'équilibrage de charge est une condition ultime pour le passage à l'échelle. Cette condition est bien respectée par l'algorithme adapté. Ce dernier définit une scalabilité en termes de la taille des données traitées ainsi que la taille du modèle de calcul. Précisément, la stratégie du partitionnement de l'algorithme adapté permet de diviser le graphe en entrée en un ensemble de sous-graphes induits, en utilisant une stratégie par coloriage de sommets. Ceci permet de traiter tout type de graphes, y compris les graphes à grande taille et ceux ayant une distribution fortement asymétrique. En outre, la taille du modèle de traitement peut être facilement ajustée (élargie ou réduite) sans perte en performance. Ceci est grâce à l'équilibrage de charge parfait préservé par l'algorithme adapté entre les machines du modèle.

Validation de nos propositions

Nous avons conduit une étude expérimentale dans le chapitre 5. Elle nous a permet de positionner notre algorithme par rapport à d'autres solutions de traitement de graphes sur différents environnements. Nous avons conclu que l'algorithme adapté est plus performant en utilisant les requêtes SQL, car les SGBDR définissent des optimisations par défaut pour les E/S. Il est particulièrement plus rapide avec les graphes de distribution asymétrique, qu'une solution classique qui ne définit pas une stratégie de partitionnement. Ainsi, son implémentation avec Pandas est plus performante que la méthode native de comptage de triangle de Networkx. Ensuite, en comparant l'algorithme adapté avec la méthode de comptage de triangles pré-définie dans Spark GraphX, nous avons constaté que l'algorithme adapté renvoie des résultats plus pertinents particulièrement pour les graphes à grande échelle. Enfin, la comparaison conduite entre notre algorithme adapté et la solution G-thinker nous a permis de conclure que l'algorithme adapté peut être plus précis et plus performant sur les petits modèles de calcul définissant un budget limité en termes de mémoire.

Perspectives

L'étude d'énumération de triangles avec les requêtes conduite dans cette thèse nous a ouvert plusieurs portes pour explorer de multiples pistes de recherche. D'ailleurs, plusieurs perspectives peuvent être découlées de ce travail de recherche, que nous les présentons dans la suite.

Partitionnement du graphe

L'équilibrage de charge demeure un enjeu majeur pour de nombreux problèmes d'analyse de graphes. En effet, avec l'évolution de la technologie, les interconnexions entre les différents objets sur Internet deviennent gigantesques, tout comme les graphes qui les modélisent. Par conséquent, une stratégie de partitionnement efficace est nécessaire. Celle-ci doit limiter la communication entre les machines de traitement tout en fournissant des résultats corrects et performants.
Notre stratégie de partitionnement peut donc être adaptée pour résoudre les problèmes d'exploration et d'analyse de graphes sur ce type de structure de grande ampleur. Nous mentionnons à titre d'exemple, l'énumération et le comptage de structure de taille (n^α dans le graphe ; où α est l'ordre de la structure). Elle peut être également utilisé pour résoudre le problème de fermeture transitive et encore le problème de plus court chemin et le voyageur de commerce.

Application des triangles

Les triangles représentent une étape fondatrice et une mesure incontournable pour plusieurs problèmes d'analyse de graphes. Ils permettent de révéler des structures significatives et d'explorer des motifs complexes dans les graphes. Nous citons ci-dessous quelques applications et problèmes d'analyse de graphes qui peuvent tirer profits de notre algorithme adapté lors de leur résolution, et nous expliquons comment adapter notre solution pour qu'elle corresponde aux problèmes à résoudre.

(1) Détection de communautés : les triangles peuvent être utilisés pour déterminer l'âge de la communauté. En effet, la communauté qui possède une densité élevée de triangles représente une vieille communauté. Cette information peut être exploitée pour plusieurs application comme la recommandation. Nous pouvons ainsi utiliser notre solution d'algorithme adapté pour déterminer tous les triangles dans plusieurs communautés, puis exploiter le résultat dans une application de recommandation.

(2) Centralité : cette mesure se définit par les sommets dont le degré est le plus élevé. Concrètement et dans le contexte des réseaux sociaux, ceci pourra être traduit par une célébrité avec des millions de suivis. Ces sommets sont facilement identifiables lorsqu'ils sont impliqués dans plusieurs triangles. Notre algorithme est ainsi utilisé pour les détecter. Précisément, nous considérons l'énumération de triangles dans notre algorithme. Ainsi, nous pouvons les lister et détecter les sommets qui sont impliqués dans plusieurs triangles.

(3) Détermination de cliques maximales : le triangle représentent la plus petite clique dans le graphe. En suivant le même principe de son énumération, nous pouvons détecter itérativement des cliques de plus grand ordre jusqu'à la détermination de la clique maximale dans le graphe.

(4) Compression de graphes : le traitement des graphes à grande échelle devient de plus en plus problématique, car il est difficile de l'héberger entièrement en mémoire principale. Ceci nécessite de nombreuses E/S pour pouvoir le traiter. Une solution potentielle est de compresser le graphe en utilisant les triangles. En effet, chaque triangle représente un super-sommet et les arêtes inter-triangles deviennent des super-arêtes. Cette compression permet de réduire l'espace mémoire requis pour la manipulation du graphe.

(5) Détection des fraudes : l'identification de groupes de sommets connectés d'une manière inhabituelle permet de détecter des transactions frauduleuses sur un réseau. Ainsi, en cher-

chant les sommets qui forment des triangles de manière répétée, nous pouvons facilement les empêcher l'exécution de ces transactions.

Optimisation des résultats

Notre algorithme pourrait bénéficier d'une optimisation en utilisant des plateformes d'exécution multi-cœurs ou basées sur des GPU. En effet, nous avons l'intention de modifier nos requêtes pour pouvoir les exécuter sur une seule machine dotée de k-cœurs. Cette solution a pour but de privilégier le traitement local des triangles sur chaque cœur du CPU, d'offrir une solution qui optimise l'utilisation de la mémoire principale et qui minimise les coûts relatifs à l'acquisition du matériel pour former le modèle de $k - machines$.
De plus, nous souhaitons développer une solution qui puisse être exécutée sur un GPU. Cette plateforme offre une plus grande capacité de traitement grâce au grand nombre de cœurs de traitement qu'elle possède. Le GPU permet d'imiter un modèle de $k - machines$ avec des centaines de machines (chaque machine étant représentée par un cœur de traitement sur le GPU). Ainsi, nous pouvons répartitionner le graphe en entrée en utilisant plus de couleurs, ce qui permet d'améliorer les performances de notre algorithme.

Exécution sur le cloud

Le cloud computing est désormais un élément incontournable pour les entreprises en quête de flexibilité, d'agilité et de performance. En effet, les services cloud permettent de stocker des données, de déployer des applications, d'héberger des sites web et d'exécuter des logiciels de gestion de manière rapide, fiable et évolutive. Face à ce constat, notre perspective est d'adapter notre solution pour qu'elle puisse être exécutée sur le cloud, en optimisant les coûts de son exécution et de son déploiement. Nous souhaitons ainsi offrir une solution complète à base de requêtes SQL qui peuvent être exécutées sur des SGBDR hébergés sur le cloud, ainsi qu'une solution basée sur MPI et Python. Contrairement aux solutions exécutées sur les clusters locaux, les solutions basées sur le cloud tire profits de la disponibilité de ressources de traitement, qui sont quasiment illimitées. Cependant, il est important de surveiller les coûts relatifs à l'utilisation de ces ressources qui peuvent être significatifs s'ils ne sont pas gérés efficacement. Par conséquent, nous voudrions proposer une solution qui permette d'optimiser l'utilisation de ces ressources tout en minimisant les coûts associés.

Bibliographie

[AB09] Sanjeev Arora and Boaz Barak. *Computational complexity : a modern approach*. Cambridge University Press, 2009. (Cité en page 20)

[ABG15] Ariful Azad, Aydin Buluç, and John Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 804–811. IEEE, 2015. (Cité en page 56), (Cité en page 58)

[AET21] Aly Ahmed, Keanelek Enns, and Alex Thomo. Triangle enumeration for billion-scale graphs in RDBMS. In Leonard Barolli, Isaac Woungang, and Tomoya Enokido, editors, *Proceedings of the 35th of AINA*, volume 226, pages 160–173. Springer, 2021. (Cité en page 48)

[AKM13] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. Patric : A parallel algorithm for counting triangles in massive networks. In *CIKM*, pages 529–538, 2013. (Cité en page 56), (Cité en page 58)

[AKM15a] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. A fast parallel algorithm for counting triangles in graphs using dynamic load balancing. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 1839–1847. IEEE, 2015. (Cité en page 56), (Cité en page 58)

[AKM15b] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. A space-efficient parallel algorithm for counting exact triangles in massive networks. In *HPCC*, pages 527–534, 2015. (Cité en page 56), (Cité en page 58)

[ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1) :11–33, 2004. (Cité en page 60)

[ALT+17] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded : A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4) :1–44, 2017. (Cité en page 47), (Cité en page 48)

[AO09] Krzysztof R. Apt and Ernst-Rüdiger Olderog. Distributed programs. In *Verification of Sequential and Concurrent Programs*, pages 373–406. Springer London, 2009. (Cité en page 26)

[AOB18] Sikder Tahsin Al-Amin, Carlos Ordonez, and Ladjel Bellatreche. Big data analytics : Exploring graphs with optimized SQL queries. In *DEXA Workshops*, pages 88–100, 2018. (Cité en page 3), (Cité en page 37), (Cité en page 47), (Cité en page 48), (Cité en page 112)

[AR04] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, page 120–124, New York, NY, USA, 2004. Association for Computing Machinery. (Cité en page 37)

[ARK06] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10 pp.–, 2006. (Cité en page 37)

[ASSU13] Foto N Afrati, Anish Das Sarma, Semih Salihoglu, and Jeffrey D. Ullman. Upper and lower bounds on the cost of a MapReduce computation. *PVLDB*, 6(4) :277–288, 2013. (Cité en page 54), (Cité en page 55), (Cité en page 56)

[ATS04] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4) :335–371, dec 2004. (Cité en page 30)

[AvdBF+92] Peter M. G. Apers, Carel A. van den Berg, Jan Flokstra, Paul W. P. J. Grefen, Martin L. Kersten, and Annita N. Wilschut. PRISMA/DB : A parallel main memory relational DBMS. *IEEE Trans. Knowl. Data Eng.*, 4(6) :541–554, 1992. (Cité en page 36)

[AYR+19] Seher Acer, Abdurrahman Yaşar, Sivasankaran Rajamanickam, Michael Wolf, and Ümit V Catalyürek. Scalable triangle counting on distributed-memory systems. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–5. IEEE, 2019. (Cité en page 57), (Cité en page 58)

[AYZ97] Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3) :209–223, 1997. (Cité en page 50), (Cité en page 51), (Cité en page 57)

[BAC+90] Haran Boral, William Alexander, Larry Clay, George P. Copeland, Scott Danforth, Michael J. Franklin, Brian E. Hart, Marc G. Smith, and Patrick Valduriez. Prototyping bubba, A highly parallel database system. *IEEE Trans. Knowl. Data Eng.*, 2(1) :4–24, 1990. (Cité en page 36)

[BBJ+14] Yingyi Bu, Vinayak Borkar, Jianfeng Jia, Michael J. Carey, and Tyson Condie. Pregelix : Big(ger) graph analytics on a dataflow engine. *Proc. VLDB Endow.*, 8(2) :161–172, October 2014. (Cité en page 46)

[BCV93] B. Bergsten, M. Couprie, and P. Valduriez. Overview of Parallel Architectures for Databases. *The Computer Journal*, 36(8) :734–740, 01 1993. (Cité en page 31), (Cité en page 32), (Cité en page 33)

[BFN+15] Jonathan W. Berry, Luke A. Fostvedt, Daniel J. Nordman, Cynthia A. Phillips, C. Seshadhri, and Alyson G. Wilson. Why Do Simple Algorithms for Triangle Enumeration Work in the Real World ? *Internet Mathematics*, 11(6) :555–571, 2015. (Cité en page 65)

[BFV96] Luc Bouganim, Daniela Florescu, and Patrick Valduriez. Dynamic load balancing in hierarchical parallel database systems. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 436–447. Morgan Kaufmann, 1996. (Cité en page 36)

[BH10] Mohammad Beydoun and Ramzi A. Haraty. Directed graph representation and traversal in relational databases. In *Networked Digital Technologies - Second International Conference, NDT 2010, Prague, Czech Republic, July 7-9, 2010. Proceedings, Part II*, volume 88 of *Communications in Computer and Information Science*, pages 443–455. Springer, 2010. (Cité en page 40)

[BIPP18] Sayan Bandyapadhyay, Tanmay Inamdar, Shreyas Pai, and Sriram V. Pemmaraju. Near-optimal clustering in the k-machine model. In *Proceedings of the 19th International Conference on Distributed Computing and Networking*, ICDCN '18, New York, NY, USA, 2018. Association for Computing Machinery. (Cité en page 67)

[BK13] Subhrajyoti Bordoloi and Bichitra Kalita. Article : Designing graph database models from existing relational databases. *International Journal of Computer Applications*, 74(1) :25–31, July 2013. (Cité en page 84)

[BKL98] Ladjel Bellatreche, Kamalakar Karlapalem, and Qing Li. Derived horizontal class partitioning in oodbs : Design strategies, analytical model and evaluation. In *17th International Conference on Conceptual Modeling*, pages 465–479, 1998. (Cité en page 35)

[BKR02] Abraham Bookstein, Vladimir A. Kulyukin, and Timo Raita. Generalized hamming distance. *Information Retrieval*, 5(4) :353–375, Oct 2002. (Cité en page 55)

[BKS00] Ladjel Bellatreche, Kamalakar Karlapalem, and Ana Simonet. Algorithms and support for horizontal class partitioning in object-oriented databases. *Distributed Parallel Databases*, 8(2) :155–179, 2000. (Cité en page 34)

[BS13] Charles-Edmond Bichot and Patrick Siarry. *Graph partitioning.* John Wiley & Sons, 2013. (Cité en page 37)

[BW09] Ladjel Bellatreche and Komla Yamavo Woameno. Dimension table driven approach to referential partition relational data warehouses. In *ACM 12th International Workshop on Data Warehousing and OLAP (DOLAP)*, pages 9–16. ACM, 2009. (Cité en page 34)

[CC12] Shumo Chu and James Cheng. Triangle Listing in Massive Networks. *ACM Trans. Knowl. Discov. Data*, 6(4) :17, 2012. (Cité en page 65)

[CDD+09] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M Hellerstein, and Caleb Welton. Mad skills : new analysis practices for big data. *Proceedings of the VLDB Endowment*, 2(2) :1481–1492, 2009. (Cité en page 46)

[CEK+15] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges : Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12) :1804–1815, 2015. (Cité en page 45), (Cité en page 68)

[CGK+18] Valeriy M. Chernenkiy, Yuriy E. Gapanyuk, Yuriy Kaganov, Ivan Dunin, Maxim Lyaskovsky, and Vadim Larionov. Storing metagraph model in relational, document-oriented, and graph databases. In *Selected Papers of the XX International Conference on Data Analytics and Management in Data Intensive Domains (DAMDID/RCDL 2018), Moscow, Russia, October 9-12, 2018*, volume 2277 of *CEUR Workshop Proceedings*, pages 82–89. CEUR-WS.org, 2018. (Cité en page 40)

[CNW83] S. Ceri, S. Navathe, and G. Wiederhold. Distribution design of logical database schemas. *IEEE Transactions on Software Engineering*, SE-9(4) :487–504, 1983. (Cité en page 35)

[CO82] Stefano Ceri and Susan S. Owicki. On the use of optimistic methods for concurrency control in distributed databases. In *Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks, Asilomar, February 16-19, 1982*, pages 117–129. Technical Information Department, Lawrence Berkeley Laboratory, University of California, Berkeley CA, 1982. (Cité en page 35)

[CO17] Wellington Cabrera and Carlos Ordonez. Scalable parallel graph algorithms with matrix–vector multiplication evaluated with queries. *DAPD Journal*, 35(3-4) :335–362, 2017. (Cité en page 6), (Cité en page 47), (Cité en page 48)

[Coh09] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science Engineering*, 11(4) :29–41, 2009. (Cité en page 54), (Cité en page 55), (Cité en page 56), (Cité en page 66), (Cité en page 81), (Cité en page 82), (Cité en page 102), (Cité en page 138)

[CRA13] Amlan Chatterjee, Sridhar Radhakrishnan, and John K. Antonio. On analyzing large graphs using gpus. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 751–760, 2013. (Cité en page 53), (Cité en page 54)

[CS16] Fan Chung and Olivia Simpson. Distributed algorithms for finding local clusters using heat kernel pagerank, 2016. (Cité en page 67)

[CSC+19] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. Powerlyra : Differentiated graph computation and partitioning on skewed graphs. *ACM Trans. Parallel Comput.*, 5(3), jan 2019. (Cité en page 37)

[CXCL20] Yi Cui, Di Xiao, Daren BH Cline, and Dmitri Loguinov. Improving i/o complexity of triangle enumeration. *IEEE Transactions on Knowledge and Data Engineering*, 2020. (Cité en page 51), (Cité en page 58)

[CXL19] Yi Cui, Di Xiao, and Dmitri Loguinov. On efficient external-memory triangle listing. *IEEE Transactions on Knowledge and Data Engineering*, 31(8) :1555–1568, 2019. (Cité en page 50), (Cité en page 51), (Cité en page 58)

[Dav18] Timothy A. Davis. Graph algorithms via suitesparse : Graphblas : triangle counting and k-truss. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6, 2018. (Cité en page 52), (Cité en page 54), (Cité en page 58)

[Dav21] Timothy A. Davis. SuiteSparse : GraphBLAS, 2021. (Cité en page 52)

[DFN+17] Ketan Date, Keven Feng, Rakesh Nagi, Jinjun Xiong, Nam Sung Kim, and Wen-Mei Hwu. Collaborative (cpu + gpu) algorithms for triangle counting and truss decomposition on the minsky architecture : Static graph challenge : Subgraph isomorphism. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017. (Cité en page 54)

[DG92] David DeWitt and Jim Gray. Parallel database systems : The future of high performance database systems. *Commun. ACM*, 35(6) :85–98, 1992. (Cité en page 38)

[DGS+90] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. The gamma database machine project. *IEEE Trans. Knowl. Data Eng.*, 2(1) :44–62, 1990. (Cité en page 36)

[DLP12] Danny Dolev, Christoph Lenzen, and Shir Peled. "tri, tri again" : Finding triangles and small subgraphs in a distributed setting. In Marcos K. Aguilera, editor, *Distributed Computing*, pages 195–209, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. (Cité en page 69)

[DR93] A. Delis and N. Roussopoulos. Performance comparison of three modern dbms architectures. *IEEE Transactions on Software Engineering*, 19(2) :120–138, 1993. (Cité en page 29)

[DVMT13] Roberto De Virgilio, Antonio Maccioni, and Riccardo Torlone. Converting relational to graph databases. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, 2013. (Cité en page 84)

[DVS05] D. Del Vecchio and S.H. Son. Flexible update management in peer-to-peer database systems. In *9th International Database Engineering & Application Symposium (IDEAS'05)*, pages 435–444, 2005. (Cité en page xi), (Cité en page 30), (Cité en page 31)

[FBO+20a] A. Farouzi, L. Bellatreche, C. Ordonez, G. Pandurangan, and M. Malki. A scalable randomized algorithm for triangle enumeration on graph based on SQL queries. In *DaWaK Conference*, 2020. (Cité en page 81), (Cité en page 84), (Cité en page 112)

[FBO+20b] Abir Farouzi, Ladjel Bellatreche, Carlos Ordonez, Gopal Pandurangan, and Mimoun Malki. Pandasql : Parallel randomized triangle enumeration with sql queries. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, CIKM '20, page 3377–3380, New York, NY, USA, 2020. Association for Computing Machinery. (Cité en page 99), (Cité en page 102)

[FLC86] M. T. Fang, Richard C. T. Lee, and C. C. Chang. The idea of de-clustering and its applications. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, page 181–188, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc. (Cité en page 34)

[FQK+15] G. C. Fox, J. Qiu, S. Kamburugamuve, S. Jha, and A. Luckow. Hpc-abds high performance computing enhanced apache big data stack. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 1057–1066, Los Alamitos, CA, USA, may 2015. IEEE Computer Society. (Cité en page 67)

[FRP15] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M Patel. The Case Against Specialized Graph Analytics Engines. In *CIDR*, 2015. (Cité en page 6), (Cité en page 49)

[FZB+23a] Abir Farouzi, Xiantian Zhou, Ladjel Bellatreche, Mimoun Malki, and Carlos Ordonez. Balanced parallel triangle enumeration with an adaptive algorithm. *Distributed and Parallel Databases*, Jul 2023. (Cité en page 81), (Cité en page 84), (Cité en page 112)

[FZB+23b] Abir Farouzi, Xiantian Zhou, Ladjel Bellatreche, Mimoun Malki, and Carlos Ordonez. Parallel pattern enumeration in large graphs. In Christine Strauss, Toshiyuki Amagasa, Gabriele Kotsis, A. Min Tjoa, and Ismail Khalil, editors, *Database and Expert Systems Applications*, pages 408–423, Cham, 2023. Springer Nature Switzerland. (Cité en page 49)

[GGH+96] Al Geist, William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing L. Lusk, William Saphir, Anthony Skjellum, and Marc Snir. MPI-2 : extending the message-passing interface. In *Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon, France, August 26-29, 1996, Proceedings, Volume I*, volume 1123 of *Lecture Notes in Computer Science*, pages 128–135, 1996. (Cité en page 56), (Cité en page 112), (Cité en page 131)

[GI96] Minos N. Garofalakis and Yannis E. Ioannidis. Multi-dimensional resource scheduling for parallel queries. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 365–376. ACM Press, 1996. (Cité en page 36)

[GJZ+11] Jun Gao, Ruoming Jin, Jiashuai Zhou, Jeffrey Xu Yu, Xiao Jiang, and Tengjiao Wang. Relational approach for shortest path discovery over large graphs. *Proc. VLDB Endow.*, 5(4) :358–369, 2011. (Cité en page 47)

[GLG+12] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph : Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, October 2012. USENIX Association. (Cité en page 45)

[GMB12] Oded Green, Robert McColl, and David A. Bader. Gpu merge path : A gpu merging algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, page 331–340, New York, NY, USA, 2012. Association for Computing Machinery. (Cité en page 53)

[GXD+14] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx : Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 599–613, USA, 2014. USENIX Association. (Cité en page 45), (Cité en page 46)

[GYM14] Oded Green, Pavan Yalamanchili, and Lluís-Miquel Munguía. Fast triangle counting on the gpu. In *Proceedings of the 4th Workshop on Irregular Applications : Architectures and Algorithms*, pages 1–8, 2014. (Cité en page 53), (Cité en page 54)

[GZY+19] Chuangyi Gui, Long Zheng, Pengcheng Yao, Xiaofei Liao, and Hai Jin. Fast triangle counting on gpu. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2019. (Cité en page 53)

[HJC+19] Loc Hoang, Vishwesh Jatala, Xuhao Chen, Udit Agarwal, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Disttc : High performance distributed triangle counting. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2019. (Cité en page 53), (Cité en page 54)

[HKJ+18] Mohamed S. Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G. Aref, and Mohammad Sadoghi. Grfusion : Graphs as first-class citizens in main-memory relational database systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1789–1792, New York, NY, USA, 2018. Association for Computing Machinery. (Cité en page 49)

[HKSH17] Yang Hu, Pradeep Kumar, Guy Swope, and H. Howie Huang. Trix : Triangle counting at extreme scale. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017. (Cité en page 53), (Cité en page 54)

[HLH18] Yang Hu, Hang Liu, and H. Howie Huang. High-performance triangle counting on gpus. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–5, 2018. (Cité en page 53), (Cité en page 54)

[HRS+12] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. The madlib analytics library or MAD skills, the SQL. *Proc. VLDB Endow.*, 5(12) :1700–1711, 2012. (Cité en page 46)

[HS75] Jeffrey A. Hoffer and Dennis G. Severance. The use of cluster analysis in physical data base design. In *Proceedings of the 1st International Conference on Very Large Data Bases*, VLDB '75, page 69–86, New York, NY, USA, 1975. Association for Computing Machinery. (Cité en page 33)

[HSH07] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. Architecture of a database system. *Foundations and Trends® in Databases*, 1(2) :141–259, 2007. (Cité en page 33)

[HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008. (Cité en page 133)

[HTC13] Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. Massive graph triangulation. In *ACM SIGMOD*, pages 325–336, 2013. (Cité en page 50), (Cité en page 51), (Cité en page 58)

[IR78] Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4) :413–423, 1978. (Cité en page 45), (Cité en page 49), (Cité en page 51), (Cité en page 57), (Cité en page 66)

[JMCH15] Alekh Jindal, Samuel Madden, Malú Castellanos, and Meichun Hsu. Graph analytics using vertica relational database. In *IEEE International Conference on Big Data*, pages 1191–1200. IEEE Computer Society, 2015. (Cité en page 47), (Cité en page 112)

[JRW+14] Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande, and Mike Stonebraker. Vertexica : Your relational friend for graph analytics ! *Proc. VLDB Endow.*, 7(13) :1669–1672, August 2014. (Cité en page 46)

[KB96] Arthur M. Keller and Julie Basu. A predicate-based caching scheme for client-server database architectures. *VLDB J.*, 5(1) :35–47, 1996. (Cité en page 29)

[KBCP09] Milind Kulkarni, Martin Burtscher, Calin Cascaval, and Keshav Pingali. Lonestar : A suite of parallel irregular programs. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 65–76, 2009. (Cité en page 54)

[KHKM11] Jin Kim, Inwook Hwang, Yong-Hyuk Kim, and Byung-Ro Moon. Genetic approaches for graph partitioning : A survey. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, page 473–480, New York, NY, USA, 2011. Association for Computing Machinery. (Cité en page 37)

[KK95a] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *IEEE/ACM SC95 Conference*, pages 29–29, Los Alamitos, CA, USA, dec 1995. IEEE Computer Society. (Cité en page 37)

[KK95b] George Karypis and Vipin Kumar. Multilevel graph partitioning schemes. In *Proceedings of The International Conference on Parallel Processing*, 1995. (Cité en page 37)

[KNPR15] Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. Distributed computation of large-scale graph problems. *Proceedings of the 26th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 391–410, 2015. (Cité en page 67)

[KTF09] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus : A peta-scale graph mining system implementation and observations. In *2009 Ninth IEEE International Conference on Data Mining*, pages 229–238, 2009. (Cité en page 45)

[Kun13] Jérôme Kunegis. Konect : The koblenz network collection. Association for Computing Machinery, 2013. (Cité en page 113)

[KVH18] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. High-level programming abstractions for distributed graph processing. *IEEE Transactions on Knowledge and Data Engineering*, 30(2) :305–324, 2018. (Cité en page 46)

[Lam13] Christoph Lameter. An overview of non-uniform memory access. *Commun. ACM*, 56(9) :59–54, sep 2013. (Cité en page 31)

[Lat08] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical computer science*, 407(1-3) :458–473, 2008. (Cité en page 50), (Cité en page 51), (Cité en page 57)

[LBG+12] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab : A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8) :716–727, April 2012. (Cité en page 45)

[LFV+12] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The vertica analytic database : C-store 7 years later. *Proc. VLDB Endow.*, 5(12) :1790–1801, 2012. (Cité en page 112), (Cité en page 114), (Cité en page 116), (Cité en page 117), (Cité en page 118)

[LGS+79] V. Lum, S. Ghosh, M. Schkolnick, R. Taylor, D. Jefferson, S. Su, J. Fry, T. Teorey, B. Yao, D. Rund, B. Kahn, S. Navathe, D. Smith, L. Aguilar, W. Barr, and P. Jones. 1978 new orleans data base design workshop report. Los Alamitos, CA, USA, oct 1979. IEEE Computer Society. (Cité en page 34)

[LK14] Jure Leskovec and Andrej Krevl. SNAP Datasets : Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014. (Cité en page 113)

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel : A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 135–146, New York, NY, USA, 2010. Association for Computing Machinery. (Cité en page 45), (Cité en page 68)

[MD95] Manish Mehta and David J. DeWitt. Managing intra-operator parallelism in parallel database systems. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 382–394. Morgan Kaufmann, 1995. (Cité en page 36)

[MDQ+18] Vikram S. Mailthody, Ketan Date, Zaid Qureshi, Carl Pearson, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. Collaborative (cpu + gpu) algorithms for triangle counting and truss decomposition. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7, 2018. (Cité en page 54)

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge university press, 1995. (Cité en page 73)

[NCWD84] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical partitioning algorithms for database design. *ACM Trans. Database Syst.*, 9(4) :680–710, dec 1984. (Cité en page 35)

[NRR13] Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew strikes back : new developments in the theory of join algorithms. *SIGMOD Record*, 42(4) :5–16, 2013. (Cité en page 3), (Cité en page 65)

[NRR14] Hung Q Ngo, Christopher Ré, and Atri Rudra. Skew strikes back : New developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4) :5–16, February 2014. (Cité en page 4), (Cité en page 5)

[OCG17] Carlos Ordonez, Wellington Cabrera, and Achyuth Gurram. Comparing columnar, row and array DBMSs to process recursive queries on graphs. *Information Systems*, 63 :66–79, 2017. (Cité en page 6), (Cité en page 47)

[OGM+12] Saher Odeh, Oded Green, Zahi Mwassi, Oz Shmueli, and Yitzhak Birk. Merge path - parallel merging made simple. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 1611–1618, 2012. (Cité en page 53)

[OHE96] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Client/Server Survival Guide, 2nd Edition*. Wiley, 1996. (Cité en page 30)

[Ord10] C. Ordonez. Optimization of linear recursive queries in sql. *IEEE Transactions on Knowledge and Data Engineering*, 22(2) :264–277, 2010. (Cité en page 46)

[Ord13] Carlos Ordonez. Can we analyze big data inside a dbms ? In *DOLAP*, pages 85–92, 2013. (Cité en page 6), (Cité en page 37), (Cité en page 47), (Cité en page 65)

[OT18] Carlos Ordonez and Predrag T. Tosic. Time complexity and parallel speedup of relational queries to solve graph problems. In Sven Hartmann, Hui Ma, Abdelkader Hameurlain, Günther Pernul, and Roland R. Wagner, editors, *Database and Expert Systems Applications*, pages 339–349, Cham, 2018. Springer International Publishing. (Cité en page 48)

[ÖV91] M. Tamer Özsu and Patrick Valduriez. Distributed database systems : Where are we now ? *Computer*, 24(8) :68–78, 1991. (Cité en page 27), (Cité en page 28)

[ÖV20] M. Tamer Özsu and Patrick Valduriez. *Introduction*, pages 1–31. Springer International Publishing, Cham, 2020. (Cité en page xi), (Cité en page 27), (Cité en page 28), (Cité en page 29), (Cité en page 30), (Cité en page 31), (Cité en page 32), (Cité en page 33), (Cité en page 34), (Cité en page 36), (Cité en page 37)

[PC13] Ha-Myung Park and Chin-Wan Chung. An efficient MapReduce algorithm for counting triangles in a very large graph. In *ACM CIKM*, pages 539–548, 2013. (Cité en page 48), (Cité en page 54), (Cité en page 55), (Cité en page 56)

[PMC+90] Hamid Pirahesh, C. Mohan, Josephine Cheng, T. S. Liu, and Pat Selinger. Parallelism in relational data base systems : Architectural issues and design approaches. In *Proceedings of the Second International Symposium on Databases in Parallel and Distributed Systems*, DPDS '90, page 4–29, New York, NY, USA, 1990. Association for Computing Machinery. (Cité en page 31), (Cité en page 33)

[PMK16] Ha-Myung Park, Sung-Hyon Myaeng, and U. Kang. PTE : Enumerating trillion triangles on distributed systems. In *ACM SIGKDD*, KDD '16, page 1115–1124, 2016. (Cité en page 48)

[Pol16] Adam Polak. Counting triangles in large graphs on gpu. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 740–746, 2016. (Cité en page 53), (Cité en page 54)

[PP16] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on gpus. *SIGPLAN Not.*, 51(10) :1–19, October 2016. (Cité en page 54)

[PRS16] Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. Fast distributed algorithms for connectivity and mst in large graphs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, page 429–438, New York, NY, USA, 2016. Association for Computing Machinery. (Cité en page 67)

[PRS18a] Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. Fast distributed algorithms for connectivity and mst in large graphs. *ACM Trans. Parallel Comput.*, 5(1), 2018. (Cité en page 67)

[PRS18b] Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. On the Distributed Complexity of Large-Scale Graph Computations. In *SPAA*, pages 405–414, 2018. (Cité en page 11), (Cité en page 36), (Cité en page 37), (Cité en page 57), (Cité en page 58), (Cité en page 65), (Cité en page 67), (Cité en page 69), (Cité en page 74), (Cité en page 80), (Cité en page 81)

[PS08] Eric . Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF (W3C recommendation), January 2008. (Cité en page 46)

[PS14] Rasmus Pagh and Francesco Silvestri. The input/output complexity of triangle enumeration. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '14, page 224–233, New York, NY, USA, 2014. Association for Computing Machinery. (Cité en page 50), (Cité en page 51), (Cité en page 58)

[PT12] Rasmus Pagh and Charalampos E. Tsourakakis. Colorful triangle counting and a mapreduce implementation. *Information Processing Letters*, 112(7) :277–281, 2012. (Cité en page 52)

[QCQ+18] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment*, 11(12) :1876–1888, 2018. (Cité en page 3)

[Rah96] Erhard Rahm. Dynamic load balancing in parallel database systems. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon, France, August 26-29, 1996, Proceedings, Volume I*, volume 1123 of *Lecture Notes in Computer Science*, pages 37–52. Springer, 1996. (Cité en page 36)

[Ray09] Chhanda Ray. *Distributed Database Concepts*, pages 31–42. Pearson Education, Cham, 2009. (Cité en page 28)

[RG77] James B Rothnie and Nathan Goodman. An overview of the preliminary design of sdd-1 : A system for distributed databases. 1977. (Cité en page 34)

[RGG03] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. *Database management systems*, volume 3. McGraw-Hill New York, 2003. (Cité en page 5)

[RH10] Saeed K. Rahimi and Frank S. Haug. *Distributed Database Management Systems – A Practical Approach*. Wiley-IEEE Computer Society Pr. Wiley, 2010. (Cité en page 27)

[RM95] Erhard Rahm and Robert Marek. Dynamic multi-resource load balancing in parallel database systems. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 395–406. Morgan Kaufmann, 1995. (Cité en page 36)

[SAB+18] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. *C-Store : A Column-Oriented DBMS*, page 491–518. 2018. (Cité en page 115), (Cité en page 117)

[SAL+17] Benjamin A. Steer, Alhamza Alnaimi, Marco A. B. F. G. Lotz, Felix Cuadrado, Luis M. Vaquero, and Joan Varvenne. Cytosm : Declarative property graph queries without data migration. In *Proceedings of the Fifth International Workshop on Graph Data-Management Experiences & Systems*, GRADES'17, New York, NY, USA, 2017. Association for Computing Machinery. (Cité en page 49)

[Sch07] Thomas Schank. *Algorithmic Aspects of Triangle-Based Network Analysis*. PhD thesis, 2007. (Cité en page 50), (Cité en page 51), (Cité en page 57), (Cité en page 66)

[Sch19] Matthias Schmid. An approach to efficiently storing property graphs in relational databases. In *Grundlagen von Datenbanken*, pages 56–61, 2019. (Cité en page 40)

[SCP10] Sriganesh Srihari, Shruti Chandrashekar, and Srinivasan Parthasarathy. A framework for sql-based mining of large graphs on relational databases. In Mohammed J. Zaki, Jeffrey Xu Yu, B. Ravindran, and Vikram Pudi, editors, *Advances in Knowledge Discovery and Data Mining*, pages 160–167, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. (Cité en page 46)

[Sey98] AYSE YASEMIN Seydim. An overview of distributed database management. *vol*, 4 :207–214, 1998. (Cité en page 28)

[sf] Apache software foundation. Apache giraph. (Cité en page 68)

[SFS+15] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. SQLgraph : An efficient relational-based property graph store. In *ACM SIGMOD*, pages 1887–1901, 2015. (Cité en page 49)

[ST15] Julian Shun and Kanat Tangwongsan. Multicore triangle computations without tuning. In *ICDE*, pages 149–160, 2015. (Cité en page 52), (Cité en page 54), (Cité en page 58)

[SV11] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614, 2011. (Cité en page 54), (Cité en page 55), (Cité en page 56)

[SW85] Domenico Sacca and Gio Wiederhold. Database partitioning in a cluster of processors. *ACM Trans. Database Syst.*, 10(1) :29–56, mar 1985. (Cité en page 35)

[TD13] Serafettin Tasci and Murat Demirbas. Giraphx : Parallel yet serializable large-scale graph processing. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing*, pages 458–469, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. (Cité en page 45)

[Tea20] The Neo4j Team. The neo4j manual v4.2, 2020. (Cité en page 46)

[Tin21] Apache TinkerPop. Tinkerpop 3.4.10, 2021. (Cité en page 46)

[TKMF09] Charalampos E Tsourakakis, U Kang, Gary L Miller, and Christos Faloutsos. Doulion : counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 837–846, 2009. (Cité en page 56)

[TSA+17] Ancy Sarah Tom, Narayanan Sundaram, Nesreen K. Ahmed, Shaden Smith, Stijn Eyerman, Midhunchandra Kodiyath, Ibrahim Hur, Fabrizio Petrini, and George Karypis. Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017. (Cité en page 52), (Cité en page 54), (Cité en page 58)

[TST+19] Yuanyuan Tian, Wen Sun, Sui Jun Tong, En Liang Xu, Mir Hamid Pirahesh, and Wei Zhao. Synergistic graph and sql analytics inside ibm db2. *Proc. VLDB Endow.*, 12(12) :1782–1785, August 2019. (Cité en page 49)

[TTP+19] Yuanyuan Tian, Suijun Tong, Mir Hamid Pirahesh, Wen Sun, En Liang Xu, and Wei Zhao. Synergistic graph and SQL analytics inside IBM db2. *Proc. VLDB Endow.*, 12(12) :1782–1785, 2019. (Cité en page 6)

[TXZ+20] Yuanyuan Tian, En Liang Xu, Wei Zhao, Mir Hamid Pirahesh, Suijun Tong, Wen Sun, Thomas Kolanko, Md. Shahidul Haque Apu, and Huijuan Peng. IBM db2 graph : Supporting synergistic and retrofittable graph queries inside IBM db2. In *SIGMOD 2020*, pages 345–359. ACM, 2020. (Cité en page 49)

[UGF+20] Mikhail Urubkin, Vasiliy Galushka, Vladimir Fathi, Denis Fathi, and Alla Gerasimenko. Representation of graphs for storing in relational databases. In *E3S Web of Conferences*, volume 164, page 09014. EDP Sciences, 2020. (Cité en page 39)

[VLPP17] Chad Voegele, Yi-Shan Lu, Sreepathi Pai, and Keshav Pingali. Parallel triangle counting and k-truss identification using graph-centric methods. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017. (Cité en page 54)

[WDB+17] Michael M. Wolf, Mehmet Deveci, Jonathan W. Berry, Simon D. Hammond, and Sivasankaran Rajamanickam. Fast linear algebra-based triangle counting with kokkoskernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017. (Cité en page 52), (Cité en page 54), (Cité en page 58)

[WDJ91] Christopher B. Walton, Alfred G. Dale, and Roy M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 537–548. Morgan Kaufmann, 1991. (Cité en page 36)

[WFA95] Annita N. Wilschut, Jan Flokstra, and Peter M. G. Apers. Parallel evaluation of multi-join queries. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 115–126. ACM Press, 1995. (Cité en page 36)

[WS98a] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393 :440–442, 1998. (Cité en page 3), (Cité en page 65)

[WS98b] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world' networks. *nature*, 393(6684) :440–442, 1998. (Cité en page 2)

[WXSW14] Lu Wang, Yanghua Xiao, Bin Shao, and Haixun Wang. How to partition a billion-node graph. In *2014 IEEE 30th International Conference on Data Engineering*, pages 568–579, 2014. (Cité en page 37)

[WZTT10] Nan Wang, Jingbo Zhang, Kian-Lee Tan, and Anthony K. H. Tung. On Triangulation-based Dense Neighborhood Graph Discovery. *Proc. VLDB Endow.*, 4(2) :58–68, 2010. (Cité en page 3), (Cité en page 65)

[XKD15a] Konstantinos Xirogiannopoulos, Udayan Khurana, and Amol Deshpande. Graphgen : Exploring interesting graphs in relational data. *Proceedings of the VLDB Endow.*, 8(12) :2032–2035, 2015. (Cité en page 6), (Cité en page 84)

[XKD15b] Konstantinos Xirogiannopoulos, Udayan Khurana, and Amol Deshpande. Graphgen : Exploring interesting graphs in relational data. *Proc. VLDB Endow.*, 8(12) :2032–2035, 2015. (Cité en page 49)

[YGRC+20] Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, M. Tamer Özsu, Wei-Shinn Ku, and John C. S. Lui. G-thinker : A distributed framework for mining subgraphs in a big graph. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1369–1380, 2020. (Cité en page 57), (Cité en page 58), (Cité en page 112), (Cité en page 113), (Cité en page 129)

[YQZ+21] Michael Yu, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. Dptl+ : Efficient parallel triangle listing on batch-dynamic graphs. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1332–1343, 2021. (Cité en page 52)

[ZFBO23] Xiantian Zhou, Abir Farouzi, Ladjel Bellatreche, and Carlos Ordonez. Bitwise algorithms to compute the transitive closure of graphs in python. In Christine Strauss, Toshiyuki Amagasa, Gabriele Kotsis, A. Min Tjoa, and Ismail Khalil, editors, *Database and Expert Systems Applications*, pages 345–351, Cham, 2023. Springer Nature Switzerland. (Cité en page 2)

[ZJ14] Mohammed J. Zaki and Wagner Meira Jr. *Data Mining and Analysis : Fundamental Concepts and Algorithms*. Cambridge University Press, 2014. (Cité en page 20)

[ZJW+19] Yongxuan Zhang, Hong Jiang, Fang Wang, Yu Hua, Dan Feng, and Xianghao Xu. LiteTE : Lightweight, Communication-Efficient Distributed-Memory Triangle

Enumerating. *IEEE Access*, 7 :26294–26306, 2019. (Cité en page 57), (Cité en page 58)

[ZO19] X. Zhou and C. Ordonez. Computing complex graph properties with sql queries. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 4808–4816, 2019. (Cité en page 47), (Cité en page 48)

[ZSMF18] Jiyuan Zhang, Daniele G. Spampinato, Scott McMillan, and Franz Franchetti. Preliminary exploration of large-scale triangle counting on shared-memory multicore system. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6, 2018. (Cité en page 52), (Cité en page 54), (Cité en page 58)

[ZSYZ19] K. Zhao, J. Su, J. X. Yu, and H. Zhang. Sql-g : Efficient graph analytics by sql. *IEEE TKDE*, pages 1–1, 2019. (Cité en page 48)

[ZY17] Kangfei Zhao and Jeffrey Xu Yu. All-in-one : Graph processing in RDBMSs revisited. In *SIGMOD*, pages 1165–1180, 2017. (Cité en page 47), (Cité en page 65)

[ZZQC17] Yuanyuan Zhu, Hao Zhang, Lu Qin, and Hong Cheng. Efficient MapReduce algorithms for triangle listing in billion-scale graphs. *DAPD Journal*, 35(2) :149–176, 2017. (Cité en page 54), (Cité en page 55), (Cité en page 56)

Annexe A

Résumé des symboles utilisés

Cet annexe constitue un repère et un index pour les symboles et les caractères les plus importants utilisés dans cette thèse.

- **G=(V,E)** : graphe simple non-orienté non-pondéré.
- **V** : ensemble des sommets.
- **E** : ensemble des arêtes.
- **n** : le nombre de sommets.
- **m** : le nombre d'arêtes.
- **i** : un sommet.
- **e** ou **(i,j)** : une arête.
- **k** : le nombre de machines dans le modèle de calcul.
- **c** : le nombre de couleurs pour partitionner l'ensemble V dans l'algorithme adapté.
- $\Delta_{(v_1,v_2,v_3)}$: un triangle.
- $\Delta(G)$: l'ensemble des triangles du graphe G.

Annexe B

Plan d'exécution

Définition des requêtes

Dans cette sectionn, nous rappelons et nous identifions les différentes requêtes utilisées pour l'énumération des triangles sur les graphes à grande échelle :

SQL

```
Q1.1 : CREATE TABLE E_s(i int,j int);
Q1.2 : COPY E_s FROM "link/to/graph_data_set";
Q1.3 : INSERT INTO E_s SELECT j,i FROM E_s;
```

SQL

```
/*UNSEGMENTED ALL NODE permet d'avoir une copie de la table sur chaque machine*/
Q2.1 : CREATE TABLE Triplet(machine int,color1 int, color2 int, color3 int)
   UNSEGMENTED ALL NODES;
/*Un exemple d'affectation des triplets de couleurs sur un modele de 8 machines*/
Q2.2 : INSERT INTO Triplet VALUES ((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))
```

SQL

```
/*Chaque sommet choisi une couleur aleatoire parmi les c couleurs*/
Q3.1 : CREATE TABLE V_s(i int,color int);
Q3.2 : INSERT INTO V_s
   SELECT i,randomint(c)+1
   FROM
      (SELECT DISTINCT i FROM E_s
      UNION
      SELECT DISTINCT j FROM E_s)V;
```

SQL

```
/*Envoi des aretes aux proxys*/
Q4.1 : CREATE TABLE E_s_proxy(i_color int,j_color int, i int,j int);
Q4.2 : 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;
```

SQL

```
/*Collect des aretes a partir des proxys*/
Q5.1 : CREATE TABLE E_s_local(machine int,i int,j int, i_color int,j_color int);
```

```
Q5.2 : 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;
```

SQL

```
/*Enumeration des triangles*/
Q6.1 : SELECT E1.machine as 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
   WHERE E1.i<E1.j AND E2.i<E2.j AND E1.i=E3.j;
/*Comptage des triangles*/
Q6.2 : SELECT count(*)
   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
   WHERE E1.i<E1.j AND E2.i<E2.j AND E1.i=E3.j;
```

Dans la suite, nous nous intéressons particulièrement aux requêtes suivantes : Q3.2, Q4.2, Q5.2 et Q6. Ces requêtes impliquent une ou plusieurs jointures. Par conséquent, elles impactent le temps d'exécution et nécesitent ainsi notre attention.

Plan d'exécution

La requête Q3.2

L'inspection du plan d'exécution de cette requête, renvoie le résultat suivant :

```
1 QUERY PLAN DESCRIPTION:
2 ------------------------------
3
4 -> explain INSERT INTO V_s SELECT i,randomint(2)+1 FROM (SELECT DISTINCT i FROM
      E_s UNION SELECT DISTINCT j FROM E_s)V;
5
6  Access Path:
7  +-DML INSERT [Cost: 0, Rows: 0]
```

```
| Target Projection: public.V_s_super_b1 (RESEGMENT)
| Target Projection: public.V_s_super_b0
| Target Prep:
| Execute on: All Nodes
| Execute on: All Nodes
| Execute on: All Nodes
| Execute on: All Nodes
| +---> GROUPBY HASH (SORT OUTPUT) (LOCAL RESEGMENT GROUPS) [Cost: 9K, Rows: 20K
    (NO STATISTICS)] (PATH ID: 4)
| |    Group By: "*SELECT* 1".i
| |    Execute on: All Nodes
| | +---> UNION [Cost: 9K, Rows: 20K (NO STATISTICS)] (PATH ID: 5)
| | |   Execute on: All Nodes
| | |   Execute on: All Nodes
| | | +---> GROUPBY PIPELINED [Cost: 4K, Rows: 10K (NO STATISTICS)] (PATH ID: 7)
| | | |   Group By: E_s.i
| | | |   Execute on: All Nodes
| | | | +---> STORAGE ACCESS for E_s [Cost: 4K, Rows: 6M (1K RLE) (NO
    STATISTICS)] (PATH ID: 8)
| | | | |  Projection: public.E_s_super_b0
| | | | |  Materialize: E_s.i
| | | | |  Execute on: All Nodes
| | | +---> GROUPBY HASH (GLOBAL RESEGMENT GROUPS) (LOCAL RESEGMENT GROUPS)
    [Cost: 5K, Rows: 10K (NO STATISTICS)] (PATH ID: 10)
| | | |   Group By: E_s.j
| | | |   Execute on: All Nodes
| | | | +---> STORAGE ACCESS for E_s [Cost: 4K, Rows: 6M (1M RLE) (NO
    STATISTICS)] (PATH ID: 11)
| | | | |  Projection: public.E_s_super_b0
| | | | |  Materialize: E_s.j
| | | | |  Execute on: All Nodes
```

Analyse : Pour la sous-requête : $SELECTDISTINCTjFROME\_s$, l'opération $DISTINCT$ signifie qu'une opération de regroupement est nécessaire. Ainsi, l'opération $GroupByE\_s.j$ nécessite que la table E_s soit partitionné par $E\_s.j$ vue qu'elle ne l'est pas. Pour cela, il est nécessaire d'envoyer les arêtes vers d'autres machines (GLOBAL RESEGMENT GROUPS), puis de réaliser une répartition locale (LOCAL RESEGMENT GROUPS) en triant les tuples par $E\_s.j$, afin d'effectuer le regroupement distinct par j.
En revanche, l'opération $DISTINCT$ de la sous-requête : $SELECTDISTINCTiFROME\_s$ nécessite qu'une opération de regroupement. De plus, l'opération $GroupByE_s.i$ ne requiert gère un partitionnement supplémentaire, car la table E_s est déjà segmenté et partitionné par i. De ce fait, il n'est pas nécessaire d'envoyer les arêtes aux machines ni de les trier. Notez que pour cela, l'opération $GROUPBYPIPELINED$ est utilisé.

La requête Q4.2

Le plan d'exécution de cette requête est présenté ci-dessous :

```
QUERY PLAN DESCRIPTION:
------------------------------

```

```
-> explain 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;

Access Path:
+-DML INSERT [Cost: 0, Rows: 0]
| Target Projection: public.E_s_proxy_super_b1 (RESEGMENT)
| Target Projection: public.E_s_proxy_super_b0 (RESEGMENT)
| Target Prep:
| Execute on: All Nodes
| Execute on: All Nodes
| Execute on: All Nodes
| +---> JOIN HASH [Cost: 30K, Rows: 6M (1M RLE) (NO STATISTICS)] (PATH ID: 3)
    Outer (RESEGMENT)(LOCAL ROUND ROBIN)
| |    Join Cond: (E.j = Vj.i)
| |    Materialize at Input: E.i, E.j
| |    Execute on: All Nodes
| | +-- Outer -> JOIN MERGEJOIN(inputs presorted) [Cost: 7K, Rows: 6M (1M RLE)
    (NO STATISTICS)] (PATH ID: 4)
| | |    Join Cond: (E.i = Vi.i)
| | |    Execute on: All Nodes
| | | +-- Outer -> STORAGE ACCESS for E [Cost: 4K, Rows: 6M (1K RLE) (NO
    STATISTICS)] (PATH ID: 5)
| | | |   Projection: public.E_s_super_b0
| | | |   Materialize: E.i
| | | |   Execute on: All Nodes
| | | |   Runtime Filter: (SIP1(MergeJoin): E.i)
| | | +-- Inner -> STORAGE ACCESS for Vi [Cost: 2K, Rows: 1M (NO STATISTICS)]
    (PATH ID: 6)
| | | |   Projection: public.V_s_super_b0
| | | |   Materialize: Vi.i, Vi.color
| | | |   Execute on: All Nodes
| | +-- Inner -> STORAGE ACCESS for Vj [Cost: 2K, Rows: 1M (NO STATISTICS)] (PATH
    ID: 7)
| | |    Projection: public.V_s_super_b0
| | |    Materialize: Vj.i, Vj.color
| | |    Execute on: All Nodes
```

Analyse : Pour cette requête, une table temporaire Vi est créée suite à l'exécution de $E\_sEJOINV\_sViONE.i = Vi.i$. Étant donné que les deux tables V_s et E_s sont toutes les deux segmentées et partitionnées par i, ces deux tables sont déjà triés par i. Par conséquent, il n'est pas nécessaire d'échanger les arêtes entre les machines, et une opération de jointures par fusion (merge-join) est exécutée (les arêtes sont triées).
De même, la table temporaire Vj est créée après l'exécution de $JOINV\_sVjONE.j = Vj.i$. Comme les deux tables V_s et E_s ne sont pas segmentées et partitionnées par j, les arêtes de la table temporaire doivent être répartitionnées et triées par j. Par conséquent, l'opération de jointure par hachage est exécutée (Hash-join).

La requête Q5.2

Le plan d'exécution de cette requête est :

```
QUERY PLAN DESCRIPTION:
------------------------------

-> explain 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;

 Access Path:
 +-DML INSERT [Cost: 0, Rows: 0]
 | Target Projection: public.E_s_local_super_b1 (SORT BY PROJECTION SORT ORDER)
     (RESEGMENT)
 | Target Projection: public.E_s_local_super_b0 (SORT BY PROJECTION SORT ORDER)
     (RESEGMENT)
 | Target Prep:
 | Execute on: All Nodes
 | Execute on: All Nodes
 | Execute on: All Nodes
 | +---> GROUPBY HASH (LOCAL RESEGMENT GROUPS) [Cost: 92K, Rows: 135 (NO
     STATISTICS)] (PATH ID: 3)
 | |    Group By: "*SELECT* 1".machine, "*SELECT* 1".i, "*SELECT* 1".j, "*SELECT*
     1".i_color, "*SELECT* 1".j_color
 | |    Execute on: All Nodes
 | | +---> UNION [Cost: 92K, Rows: 135 (NO STATISTICS)] (PATH ID: 4)
 | | |    Execute on: All Nodes
 | | |    Execute on: All Nodes
 | | | +---> GROUPBY HASH (LOCAL RESEGMENT GROUPS) [Cost: 31K, Rows: 45 (NO
     STATISTICS)] (PATH ID: 6)
 | | | |   Group By: edge1.machine, E.i, E.j, E.i_color, E.j_color
 | | | |   Execute on: All Nodes
 | | | | +---> JOIN HASH [Cost: 31K, Rows: 45 (NO STATISTICS)] (PATH ID: 7) Inner
     (FILTER)
 | | | | |  Join Cond: (E.i_color = edge1.color1) AND (E.j_color = edge1.color2)
 | | | | |  Execute on: All Nodes
 | | | | | +-- Outer -> STORAGE ACCESS for E [Cost: 30K, Rows: 6M (NO STATISTICS)]
     (PATH ID: 8)
 | | | | | |  Projection: public.E_s_proxy_super_b0
 | | | | | |  Materialize: E.i, E.j, E.i_color, E.j_color
 | | | | | |  Filter: (E.i < E.j)
 | | | | | |  Execute on: All Nodes
 | | | | | |  Runtime Filters: (SIP1(HashJoin): E.i_color), (SIP2(HashJoin):
     E.j_color), (SIP3(HashJoin): E.i_color, E.j_color)
 | | | | | +-- Inner -> STORAGE ACCESS for edge1 [Cost: 15, Rows: 8 (NO
     STATISTICS)] (PUSHED GROUPING) (PATH ID: 9)
 | | | | | |  Projection: public.triplet_super
 | | | | | |  Materialize: edge1.machine, edge1.color1, edge1.color2
 | | | | | |  Execute on: All Nodes
 | | | +---> GROUPBY HASH (LOCAL RESEGMENT GROUPS) [Cost: 31K, Rows: 45 (NO
```

```
    STATISTICS)] (PATH ID: 11)
37  | | | |   Group By: edge2.machine, E.i, E.j, E.i_color, E.j_color
38  | | | |   Execute on: All Nodes
39  | | | | +---> JOIN HASH [Cost: 31K, Rows: 45 (NO STATISTICS)] (PATH ID: 12) Inner
        (FILTER)
40  | | | | |  Join Cond: (E.i_color = edge2.color2) AND (E.j_color = edge2.color3)
41  | | | | |  Execute on: All Nodes
42  | | | | | +-- Outer -> STORAGE ACCESS for E [Cost: 30K, Rows: 6M (NO STATISTICS)]
        (PATH ID: 13)
43  | | | | | |  Projection: public.E_s_proxy_super_b0
44  | | | | | |  Materialize: E.i, E.j, E.i_color, E.j_color
45  | | | | | |  Filter: (E.i < E.j)
46  | | | | | |  Execute on: All Nodes
47  | | | | | |  Runtime Filters: (SIP4(HashJoin): E.i_color), (SIP5(HashJoin):
        E.j_color), (SIP6(HashJoin): E.i_color, E.j_color)
48  | | | | | +-- Inner -> STORAGE ACCESS for edge2 [Cost: 15, Rows: 8 (NO
        STATISTICS)] (PUSHED GROUPING) (PATH ID: 14)
49  | | | | | |  Projection: public.triplet_super
50  | | | | | |  Materialize: edge2.machine, edge2.color2, edge2.color3
51  | | | | | |  Execute on: All Nodes
52  | | | +---> GROUPBY HASH (LOCAL RESEGMENT GROUPS) [Cost: 31K, Rows: 45 (NO
        STATISTICS)] (PATH ID: 16)
53  | | | |   Group By: edge3.machine, E.i, E.j, E.i_color, E.j_color
54  | | | |   Execute on: All Nodes
55  | | | | +---> JOIN HASH [Cost: 31K, Rows: 45 (NO STATISTICS)] (PATH ID: 17) Inner
        (FILTER)
56  | | | | |  Join Cond: (E.i_color = edge3.color3) AND (E.j_color = edge3.color1)
57  | | | | |  Execute on: All Nodes
58  | | | | | +-- Outer -> STORAGE ACCESS for E [Cost: 30K, Rows: 6M (NO STATISTICS)]
        (PATH ID: 18)
59  | | | | | |  Projection: public.E_s_proxy_super_b0
60  | | | | | |  Materialize: E.i, E.j, E.i_color, E.j_color
61  | | | | | |  Filter: (E.i > E.j)
62  | | | | | |  Execute on: All Nodes
63  | | | | | |  Runtime Filters: (SIP7(HashJoin): E.i_color), (SIP8(HashJoin):
        E.j_color), (SIP9(HashJoin): E.i_color, E.j_color)
64  | | | | | +-- Inner -> STORAGE ACCESS for edge3 [Cost: 15, Rows: 8 (NO
        STATISTICS)] (PUSHED GROUPING) (PATH ID: 19)
65  | | | | | |  Projection: public.triplet_super
66  | | | | | |  Materialize: edge3.machine, edge3.color1, edge3.color3
67  | | | | | |  Execute on: All Nodes
```

Analyse : Dans cette requête, trois opérations de jointures de hachage sont utilisées, suivi d'un regroupement de hachage pour réunir tous les résultats. La table *E_s_proxy* est segmentée par *i_color* et *j_color*, qui correspondent aux colonnes de la condition de la jointure. Par conséquent, aucune arête est échangée entre les machines. Enfin, une union parallèle est effectuée.

La requête Q6

Nous présentons ci-après le plan d'exécution de cette requête :

```
QUERY PLAN DESCRIPTION:
------------------------------

-> explain SELECT local_node_name(), E1.machine, count(*) 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 WHERE E1.i<E1.j AND E2.i<E2.j AND
    E1.i=E3.j group by 1,2;

Access Path:
+-GROUPBY HASH (GLOBAL RESEGMENT GROUPS) (LOCAL RESEGMENT GROUPS) [Cost: 6, Rows:
    1 (NO STATISTICS)] (PATH ID: 1)
| Aggregates: count(*)
| Group By: E1.machine
| Execute on: All Nodes
| +---> JOIN HASH [Cost: 5, Rows: 1 (NO STATISTICS)] (PATH ID: 2) Outer
    (RESEGMENT)(LOCAL ROUND ROBIN) Inner (RESEGMENT)
| |    Join Cond: (E1.i = E3.j) AND (E2.machine = E3.machine) AND (E2.j = E3.i)
| |    Materialize at Input: E1.i, E1.j, E1.machine
| |    Execute on: All Nodes
| | +-- Outer -> JOIN HASH [Cost: 3, Rows: 1 (NO STATISTICS)] (PATH ID: 3)
| | |   Join Cond: (E1.machine = E2.machine) AND (E1.j = E2.i)
| | |   Execute on: All Nodes
| | | +-- Outer -> STORAGE ACCESS for E1 [Cost: 1, Rows: 1 (NO STATISTICS)] (PATH
    ID: 4)
| | | |   Projection: public.E_s_local_super_b0
| | | |   Materialize: E1.i, E1.j, E1.machine
| | | |   Filter: (E1.i < E1.j)
| | | |   Execute on: All Nodes
| | | |   Runtime Filters: (SIP1(HashJoin): E1.machine), (SIP2(HashJoin): E1.j),
    (SIP3(HashJoin): E1.machine, E1.j)
| | | +-- Inner -> STORAGE ACCESS for E2 [Cost: 1, Rows: 1 (NO STATISTICS)] (PATH
    ID: 5)
| | | |   Projection: public.E_s_local_super_b0
| | | |   Materialize: E2.i, E2.j, E2.machine
| | | |   Filter: (E2.i < E2.j)
| | | |   Execute on: All Nodes
| | +-- Inner -> STORAGE ACCESS for E3 [Cost: 1, Rows: 1 (NO STATISTICS)] (PATH
    ID: 6)
| | |   Projection: public.E_s_local_super_b0
| | |   Materialize: E3.i, E3.j, E3.machine
| | |   Execute on: All Nodes
```

Analyse : Cette requête inclue deux opérations d'auto-jointure. Par conséquent, l'opération de jointure par hachage (Hash-Join) est choisie et exécutée. Comme la table *E_s_local* est partitionné par la colonne *machine*, les jointures sont toutes locales. Il n'est donc pas nécessaire de repartitionner la table *E_s_local* pour les deux auto-jointures.

Résumé

Cette thèse s'inscrit dans le domaine de l'analyse des graphes à grande échelle, avec une attention particulière portée sur l'énumération des triangles. La recherche de triangles est un problème fondamental qui permet de détecter toutes les structures composées de trois sommets adjacents, tous interconnectés par des arêtes. Ces triangles jouent un rôle crucial dans de nombreuses applications liées aux graphes, telles que : la détermination du coefficient de clustering, l'évaluation de l'âge d'une communauté dans un réseau social, et la détection de cliques.
Pour aborder cette étude, nous avons effectué une revue approfondie de la littérature sur la détection de triangles dans les graphes. Nous avons identifié deux catégories principales d'approches : (a) celles qui supposent que le graphe peut être entièrement stocké en mémoire centrale, et (b) celles qui considèrent que le graphe ne peut pas être stocké entièrement en mémoire centrale. En analysant ces travaux, nous avons constaté que les premières approches (qui reposent sur une gestion en mémoire centrale) sont efficaces lorsque le graphe peut être complètement chargé en mémoire, mais cela ne s'applique pas toujours aux graphes à grande échelle qui ne tiennent pas entièrement en mémoire centrale. Par conséquent, il est essentiel de développer des algorithmes efficaces pour ces types de graphes, ce qui constitue un défi majeur pour la communauté scientifique.
Dans le cadre de cette thèse, nous avons proposé un algorithme randomisé (également connu sous le nom de l'algorithme adapté) pour l'énumération de triangles dans des graphes à grande échelle en utilisant la technologie des Systèmes de Gestion de Bases de Données (SGBD) relationnel distribué. Nous avons choisi Vertica [1], un SGBD qui repose sur un modèle de stockage orienté colonne, contrairement aux SGBD traditionnels qui utilisent le modèle orienté ligne. Notre solution a également exploité le modèle de calcul distribué $k - machines$. Dans cette approche, le graphe est représenté sous forme de tables et exploité à travers des requêtes SQL exécutées sur l'ensemble de machines, tout en veillant à équilibrer la charge de travail entre ces nœuds. Pour évaluer cet équilibrage de charge, nous avons développé un outil appelé **PandaSQL**, qui permet d'analyser la structure du graphe et de comparer notre solution aux algorithmes existants dédiés à l'énumération de triangles. En outre, nous avons apporté une autre contribution en étudiant le comportement de notre algorithme randomisé dans un environnement distribué en utilisant Python, tout en gérant la communication des données entre les machines via l'interface de passage de messages (MPI). Une analyse théorique de notre algorithme, mise en œuvre dans les deux environnements susmentionnés, a également été réalisée.
Enfin, nous avons mené une série d'expériences en utilisant des graphes à grande échelle présentant différentes structures, tout en variant le nombre de machines du modèle $k - machines$ (4, 9 et 16 machines). Ces expériences concernent les performances de l'algorithme randomisé et sa comparaison avec des systèmes concurrents sur différents environnements. Particulièrement, la solution classique sur le SGBD, Networkx sur Python, Spark Graphx et G-thinker sur un environnement Big Data. Les résultats obtenus au cours de cette thèse mettent en évidence l'intérêt de combiner la technologie des SGBD et des algorithmes parallèles pour l'énumération efficace de triangles dans des graphes à grande échelle.

Mots-clés : Théorie de graphe, énumération de triangles, comptage de triangles, traitement parallèle, solutions dirigées par de la technologie de bases de données, équilibrage de charge.

1. https ://www.vertica.com/fr/

Abstract

This thesis falls within the field of large-scale graph analysis, with a particular focus on triangle enumeration. The search for triangles is a fundamental problem that allows us to detect all structures composed of three adjacent vertices, all interconnected by edges. These triangles play a crucial role in many graph-related applications, such as determining the clustering coefficient, assessing the age of a community in a social network, and detecting cliques.
To address this study, we conducted an in-depth literature review on triangle detection in graphs. We identified two main categories of approaches : (a) those that assume the graph can fit in main memory, and (b) those that consider that the graph cannot be entirely stored in main memory. Analyzing these works, we found that the first approaches (which rely on main memory management) are effective when the graph can be fully loaded into memory, but this does not always apply to large-scale graphs that do not entirely fit into main memory. Therefore, it is essential to develop efficient algorithms for these types of graphs, which poses a major challenge for the scientific community.
In the context of this thesis, we proposed a randomized algorithm (also known as adaptive algorithm) for triangle enumeration in large-scale graphs using distributed Relational Database Management System (RDBMS) technology. We chose Vertica, an RDBMS that is based on a column-oriented storage model, unlike traditional RDBMSs that use a row-oriented model. Our solution also leveraged the distributed $k - machines$ computing model. In this approach, the graph is represented as tables and processed through SQL queries executed across a set of machines while ensuring workload balancing among these nodes. To evaluate this workload balancing, we developed a tool called "PandaSQL," which allows us to analyze the graph's structure and compare our solution to existing algorithms dedicated to triangle enumeration. Furthermore, we made another contribution by studying the behavior of our randomized algorithm in a distributed environment using Python, while managing data communication between machines via the Message Passing Interface (MPI). A theoretical analysis of our algorithm, implemented in the two aforementioned environments, was also conducted.
Finally, we conducted a series of experiments using large-scale graphs with different structures while varying the number of machines in the $k-machines$ model (4, 9, and 16 machines). These experiments focused on the performance of the randomized algorithm and its comparison against competing systems in different environments. Specifically, the classic solution on RDBMS, Networkx in Python, Spark Graphx, and G-thinker in a Big Data environment. The results obtained during this thesis highlight the benefits of combining RDBMS technology and parallel algorithms for efficient triangle enumeration in large-scale graphs.

Keywords : Graph theory, triangle enumeration, triangle counting, parallel processing, technology-driven solutions for database, load balancing.

Secteur de recherche : Informatique et applications

Laboratoire d'Informatique et d'Automatique pour les Systèmes
École Nationale Supérieure de Mécanique et d'Aérotechnique - Chassneuil du Poitou, France
Laboratoire de Recherche en Informatique
Ecole National Supérieure en Informatique- 08 Mai 1945- Sidi Bel Abbès, Algérie