Université Sultan Moulay Slimane
Ecole Nationale des Sciences Appliquées Khouribga
Département de Mathématiques et Informatique
Laboratoire d'Ingénierie des Procédés et Optimisation des Systèmes Industriels

THÈSE DE DOCTORAT EN INFORMATIQUE

Présentée par

YASSIR ROCHD

Approches Efficaces de Fouille de Motifs Fréquents sur Big Data

---

Soutenue le 2019 devant le jury

Pr. M.BAHAJ	Faculté des Sciences et Techniques Settat	Président Rapporteur
Pr. M.NACHAOUII	Faculté des Sciences et Techniques de Béni Mellal	Rapporteur
Pr. L.MOUSSAID	Ecole Nationale Supérieure d'Eléctricité et Mécanique	Rapporteur
Pr N.ABOUTABIT	Ecole Nationale des Sciences Appliquées Khouribga	Examinateur
Pr. I.CHERTI	Faculté des Sciences et Techniques Settat	Examinateur
Pr. I.HAFIDI	Ecole Nationale des Sciences Appliquées Khouribga	Directeur de thèse

Remerciements

J'adresse tout d'abord, ma profonde reconnaissance et mes vifs remerciements à M.Imad HAFIDI pour m'avoir fait l'honneur d'être mon directeur de recherche. Je le remercie pour son soutien, sa disponibilité, sa patience et ses précieux conseils durant toute la période de mon travail.

Je suis très honoré a remercier de la présence mon jury de thèse et je tiens à remercier :

• M. Mohamed BAHAJ : Professeur à la Faculté des Sciences et Techniques de Settat.

• Mme. Laila MOUSSAID : Professeur à l'Ecole Nationale des Sciences Electrique et Mécanique de Casablanca.

• M.Mourad NACHAOUII : Professeur à la Faculté des Sciences et Techniques de Béni Mellal .

• M. Nourdine ABOUTABIT : Professeur à l'Ecole Nationale des Sciences Appliquées de Khouribga.

• M. Ilyass CHERTI : Professeur à la Faculté des Sciences et Techniques de Settat .

Je dédie ce travail à toute ma famille, mes chers parents, pour leurs amours et leurs prières, et je leurs suis très reconnaissant pour les sacrifices qu' ils ont du faire pendant mes langues années d' études, et particulièrement au défunt Haj Mohamed YASSIR.

Je remercie ma femme, qui m'a aidé à surmonter les difficultés et d'avoir accepté tant de sacrifices durant ces dernières années, ainsi mes petits enfants MOHAMED WALID et ALAE qui suscitent en moi l'envie de me battre.

Un remerciement à toutes les personnes qui m'ont aidé de près ou de loin pour achever mon travail.

Résumé

La fouille de motifs fréquents est un domaine de recherche important dans la fouille de données ou Data Mining. Depuis son introduction, elle a attiré l'attention de nombreux chercheurs, et de nombreux algorithmes ont été proposés. Bien que ces algorithmes soient populaires et performants en raison de leurs intéressantes propriétés, ils présentent également des inconvénients , tels que les analyses multiples de bases de données et les constructions récursives d'arbres.

Dans l'ère actuelle du Big Data, des volumes importants d'une grande variété de données de grande valeur sur les différentes véracités peuvent être facilement recueillis ou générés à grande vitesse dans diverses applications de la vie réelle. Parmi ces 5V de grandes données, on se concentre dans cette thèse sur le traitement de gros volumes de données.

Récemment, de nombreux algorithmes parallèles basés sur des frameworks modernes (comme Apache Hadoop, Apache Spark) ont été développés, et qui sont capables d'exploiter le calcul distribué dans des clusters de machines. Cependant, la parallélisation des algorithmes d'extraction de motifs fréquents est loin d'être triviale, et même l'exploration de l'espace de recherche, sur laquelle toutes les techniques sont basées, n'est pas facilement partageable. Par conséquent, la fouille de motifs fréquents distribués est un problème difficile et devient alors un sujet de recherche intéressant.

Dans ce contexte, nos principales contributions consistent :

Premièrement, en une analyse théorique et expérimentale exhaustive des approches les plus performantes, dont nous avons mené plusieurs expériences afin d'évaluer et de discuter les performances des algorithmes par rapport aux différents cas d'utilisation réelle et à différentes distributions de données. Les résultats de cette analyse ont montré qu'aucun algorithme n'est universellement supérieur et que les performances sont fortement affectées par la distribution des données. De plus, elle nous permet d'identifier un manque de fiabilité en ce qui concerne l'extraction fréquente de motifs dans des cas d'utilisation des données massives ou Big Data.

Deuxièmement, les résultats et les questions en suspens ont motivé le développement d'une première approche distribuée appelée HFIMH, basée sur le fameux algorithme Apriori, qui utilise les deux dispositions horizontales et verticales pour représenter efficacement la base des transactions, et une méthode d'intersection des ensembles pour accélérer le calcul des supports.

Troisièmement, en une implémentation d'un nouvel algorithme distribué appelé

HPrePostPlus basé sur la structure des listes, et adoptant la structure HashMap pour accélérer la construction d'une arborescence de recherche, l'algorithme HPrePostPlus utilise aussi une méthode d'intersection des listes afin d'accélérer le calcul des supports.

Enfin, l'évaluation expérimentale nous permet d'obtenir de meilleures performances de nos approches par rapport aux algorithmes existants de l'état de l'art, et montrent leurs performances, leurs efficacités et leurs extensibilités.

Table des matières

Table des figures

Liste des tableaux

Introduction

0.1 Contexte et Motivations

Actuellement, les données ne cessent d'affluer quotidiennement. Les réseaux sociaux et les dispositifs portables ne sont que quelques exemples des sources de données actuelles. En effet, cette révolution dite du Big Data ne concerne pas seulement la quantité croissante de données. La véritable innovation est liée aux connaissances exploitables qui peuvent en être extraites [124].
Les données extraites et collectées doivent être analysées afin que les entreprises soient en mesure de développer des modèles prédictifs pour cibler chaque client avec un accompagnement approprié. Ces systèmes de recommandation sont utilisés pour proposer des produits aux clients dépendant des choix similaires d'autres clients.
A l'ère du Big Data où l'espace de stockage des données n'est plus un problème, toutes les entreprises veulent désormais tirer profit de ce grand volume de données qui peuvent les aider dans leur environnement interne (RH, organisation, process...), leur environnement externe (type de clients, parcours clients, image de l'entreprise...) et à anticiper les phénomènes qui s'y rattachent. Ces données deviennent alors une grande richesse à condition d'être bien exploitées.
A la frontière, des statistiques de l'intelligence artificielle et de l'informatique, le Data Mining – ou fouille de données – est une discipline qui vise à extraire les informations pertinentes d'un grand ensemble de données. La fouille de données [100], ou l'extraction de connaissance à partir des données, est un domaine pluridisciplinaire très en vogue depuis plus de deux décennies, où concourent des spécialistes de l'algorithmique,de l'apprentissage automatique, des mathématiques, des bases de données, etc. Motivé par l'accumulation faramineuse des volumes de données et émergea au l'impérieuse nécessité de les valoriser, le Data mining émargea le début des années quatre-vingt dix comme étant un processus non trivial d'extraction à

partir de gros volumes de données de l'information valide, compréhensible, préalablement inconnue et potentiellement utile pour l'utilisateur [43]. C'est un domaine de recherche très fertile qui a suscité l'intérêt des scientifiques, industriels, commerciaux, et de l'ensembles des acteurs de différents champs et organisations au vu de ses impacts scientifiques et socio-économiques importants.
Tout l'enjeu est de réussir à préparer, manipuler et analyser les données dans l'optique de les transformer en connaissance actionnable et en outil d'aide à la décision pour les entreprises. Dans [71], il est expliqué comment des données volumineuses peuvent en effet être des atouts précieux pour l'analyse prédictive. Les entreprises qui disposent d'un grand volume de données et des compétences nécessaires pour en tirer profit, peuvent obtenir un grand avantage concurrentiel.
D'autre part, dans le domaine académique, la conception d'algorithmes de Data mining -fouille de données- sur le Big Data représente un défi et une opportunité inspirante de recherche. En fait, l'application des techniques traditionnelles du Data mining à une vaste collecte de données est très difficile. De plus, à mesure que la quantité de données augmente, la proportion de personnes pouvant les interpréter diminue[90]. Pour cette raison, il y a un besoin concret et urgent d'une nouvelle génération d'outils évolutifs. Dans cette thèse, nous nous concentrons sur l'une des techniques populaires du Data mining : l'extraction des motifs fréquents (FIM).

0.2 Algorithmique d'extraction de motifs fréquents

L'extraction des règles d'association premièrement introduite dans [5] est un problème populaire et commun en fouille de données [43], avec une large étendue d'applications telles que l'analyse du panier de la ménagère, la classification, la segmentation, le Web mining et une variété d'autres tâches dont l'objectif est la découverte de corrélations et d'associations.
Traditionnellement, ce problème est décomposé en deux principaux sous-problèmes : Premièrement, l'énumération de l'ensemble de motifs fréquents, i.e., les motifs dont la cooccurrence dans l'ensemble de données dépasse un seuil minimal de support fourni comme paramètre, puis, la génération des règles d'association intéressantes à partir de ces motifs.
En dépit de sa simplicité, la première phase, c'est-à-dire le calcul de la collection

de motifs fréquents, constitue un problème complexe qui a été assez bien traité durant les deux dernières décennies.

Depuis l'introduction de l'algorithme Apriori [8], une panoplie d'autres algorithmes ont été proposés pour résoudre le problème de l'extraction de motifs fréquents. Sans prétendre à l'exhaustivité, nous pouvons catégoriser ces algorihmes en trois grandes classes : (i) les algorithmes avec énumération totale de l'ensemble de motifs fréquents (ii) les algorithmes avec énumération abrégée (iii) et les algorithmes avec énumération incrémentale [3, 20, 46].

La première classe de ces algorithmes vise l'extraction de la totalité des motifs fréquents. Les stratégies d'exploration de l'espace du problème peuvent être communément distinguées par la méthode de parcours et celle de calcul des supports des motifs [66].

Les techniques par niveaux adoptent un parcours en largeur, où un k-motif est dérivé en étendant un autre de longueur k-1 [8]. Le calcul du support d'un motif est réalisé par le parcours de la base de données. Dans ces méthodes, l'apport le plus significatif est l'heuristique-Apriori, qui stipule que tout sous-motif d'un motif fréquent est aussi fréquent, et tout sur-motif d'un motif infréquent doit être infréquent. Cette observation a permis un élagage considérable de l'espace du problème, et a été par conséquent largement utilisée dans tous les algorithmes ultérieurs. Toutefois, les techniques par niveaux souffrent de deux inconvénients majeurs : la génération d'un nombre excessif de candidats (motifs potentiellement fréquents), et un coût important en terme d'opérations d'entrée/sortie nécessaires pour les calculs des supports.

Dans [131], l'auteur considère la base de données d'un point de vue vertical. Il associe à chaque motif X la liste de transactions qui le couvrent (transactions contenant le motif) dite sa tid-list, et utilise l'intersection ensembliste entre ces listes comme mécanisme de calcul de support ; une approche qui a montré son efficacité. Néanmoins, et malgré la relation d'équivalence fondée sur des préfixes communs exploités pour décomposer l'espace du problème, cette approche exige, en particulier dans le cas de bases denses, des temps et espaces intermédiaires larges pour réaliser les intersections. Une version améliorée qui depasse cette dernière limitation a été introduite dans [129].

Afin de réduire la taille de la base de données et passer outre ses multiple scans, Han et al. ont introduit l'algorithme FPGrowth [61] qui exploite une structure de données compacte appelée FPTree (Frequent-Pattern Tree) ; cette dernière

structure est un arbre de préfixes (ou trie) augmenté par les supports des motifs. FPGrowth génère récursivement des projections conditionnelles de la base, pour lesquelles des FPTrees correspondants sont aussi construits. En adoptant la stratégie diviser-pour-résoudre, cet algorithme a présenté une idée excellente dans ce contexte offrant ainsi des gains remarquables. Cependant, la version originale de cet algorithme induit, des surcoûts abondants en temps et espace, dûs essentiellement aux opérations répétées de tri et de reconstructions des FP-Trees intermédiaires.

Les algorithmes de la deuxième classe se focalisent sur l'extraction de représentations condensées, dites bases, pour les motifs fréquents qui permettent de les dériver tous. C'est ainsi que les concepts de motifs fréquents maximaux [17] et fréquents fermés [98] ont été introduits. La préoccupation des algorithmes de la troisième classe est l'incrémentalité. C'est-à-dire, comment générer la collection des motifs fréquents, et surtout la maintenir dans le cas des ensembles de données dynamiques [95, 121] .

Plusieurs algorithmes efficaces d'extraction des motifs fréquents ont été proposés dans les dernières décennies. Ils sont très efficaces lorsque les données sont stockées dans la mémoire principale, cependant ,ce n'est plus le cas avec des données plus grandes et plus complexes. Dans ce contexte, l'extraction de motifs fréquents devient un problème difficile et intéressant. C'est la raison pour laquelle certaines techniques évolutives ont été introduites au cours des dernières années en s'appuyant sur des stratégies de distribution différentes. Ceci conduit à diverses performances liées aux différents cas d'utilisation et aux distributions de données.

0.3 Big Data et fouille de données

En raison de la popularité et de l'avancement des technologies de l'information et du Web, des quantités massives de données sont produites dans notre vie quotidienne. De grandes quantités d'informations, et des pétaoctets de données, sont enregistrées chaque jour. De toute évidence, l'ère des données massives ou Big Data est arrivée [87]. En plus de la taille des données (c.-à-d. le volume), les données massives ont d'autres caractéristiques, comme la variété et la vitesse. La première signifie que les données massives peuvent être composées d'une grande variété de données structurées, semi-structurées et non structurées, tandis que la seconde

fait référence à l'exigence de traitement et d'analyse en temps réel [45]. Par conséquent, l'analyse du Big Data par l'apprentissage automatique (machine learning) et les techniques de fouille de données (data mining) est devenue un important problème de recherche [103, 133].

L'extraction de connaissances à partir des données massives ou la fouille du Big Data (Big Data Mining) est une tâche très couteûse et difficile à gérer avec les méthodologies actuelles, en raison de leur grande taille, et complexité [42]. En d'autres termes, l'utilisation d'un seul ordinateur personnel (PC) pour exécuter la tâche d'extraction de connaissances sur des ensembles de données à grande échelle exige des coûts informatiques très élevés. Il est donc nécessaire d'utiliser des environnements informatiques plus puissants pour traiter et analyser efficacement les données volumineuses.

Selon [124], les solutions du problème de l'extraction de données à grande échelle peuvent être basées sur les plates-formes parallèles. En principe, le calcul parallèle consiste à diviser le (gros) problème choisi en problèmes plus petits et indépendants, dont chacun (c'est-à-dire le calcul) est effectué par un seul processeur individuellement, de sorte qu'un calcul composé d'un certain nombre de calculs est effectué simultanément d'une manière distribuée et parallèle [10]. Plus précisément, du point de vue des données, le paradigme du parallélisme des données peut être envisagé pour le traitement des ensembles de données à grande échelle. Dans le parallélisme des données, l'ensemble de données à grande échelle est partitionné entre un certain nombre de processeurs, dont chacun exécute le même algorithme sur une partition désignée [131].

Pour traiter les problèmes d'ensembles de données à grande échelle, le calcul de MapReduce est généralement implémenté comme un puissant cadre de programmation parallèle [32]. MapReduce est composée de deux fonctions essentielles : d'une part il répartit les tâches sur plusieurs nœuds au sein du cluster (fonction Map) et, d'autre part, il organise et agrège les résultats de chacun des nœuds pour apporter une réponse à une requête.

Plusieurs algorithmes d'extraction de motifs fréquents traditionnels ont été proposés. Ils sont très efficaces lorsque les ensembles de données peuvent être complètement chargés dans la mémoire principale. Cependant, ils ne peuvent pas gérer des données plus importantes et plus complexes. Pour cette raison, au cours des dernières années, différentes approches distribuées ont été introduites, capables d'effectuer l'extraction des ensembles d'éléments même dans les cas liés aux données massives ou Big Data. En fait, Hadoop [21]et Spark[128] ont été largement

adoptés dans le milieu de la recherche . Les raisons sont en partie liées à la gestion plus facile des données et à une meilleure tolérance aux pannes[104].

0.4 Plan de thèse et contribution

La thèse est structurée en quatre parties principales, qui suivent l'ordre de conception et d'application des algorithmes. En même temps, cette division regroupe mes principales contributions :

1. Une analyse approfondie des outils d'extraction de motifs fréquent les plus fiables et les plus utilisés pour traiter les données massives «Big Data » [108]. Dans tout projet de recherche, ce pré-requis est fondamental pour mieux comprendre le domaine et, surtout, découvrir les lacunes ou les enjeux possibles. Cette analyse est divisée en deux composantes principales :

   i Une analyse théorique introduisant l'extraction des motifs fréquents et décrivant les motivations et les défis inhérents à la parallélisation. Enfin, les meilleures approches distribuées sont décrites.

   ii Une comparaison expérimentale visant à comparer les performances des approches de pointe à travers différents cas d'utilisation caractérisés par une distribution de données différentes (à la fois synthétique et jeux de données réels), afin d'identifier les lacunes ou de possibles problèmes.
2. Proposition d'une version améliorée de l'algorithme Apriori en mode distribué appliqué aux Big Data, pour surmonter les nombreux inconvénients des algorithmes existants [105].
3. La conception et la mise en place parallèle d'une version parallele de l'algorithme d'extraction de motifs frequent PrePost, pour le traitement des données massives Big Data [106, 107].

0.4.1 Extraction des motifs fréquents : Motivations, Défis et Etat de l'art

Comme nous l'avons déjà mentionné, avec la quantité croissante de données générées, des algorithmes différents d'ensembles d'éléments fréquents, distribués et

évolutifs, ont été développés.
Dans cette section, nous allons d'abord présenter les motivations du besoin des algorithmes évolués d'extraction des motifs fréquents, et de la migration actuelle vers les frameworks de calcul distribué [21, 128]. Étant donné que toutes les approches distribuées et leurs choix de conception algorithmiques inhérents reposent forcement sur un ensemble d'algorithmes centralisés sous-jacents, les approches de base d'extractions fréquemment utilisées seront introduites. Ces connaissances permettent de mieux comprendre les défis associés à la parallélisation, qui seront décrits en détail ultérieurement. Une nouvelle taxonomie basée sur la stratégie de distribution sera également introduite. Enfin, cette section du travail décrit comment les meilleures approches d'extraction d'itemset fréquent distribués ont abordé les techniques de parallélisation pour extraire des items fréquents d'une grande quantité de données.

0.4.2 Evaluation expérimentale des approches de pointe : Etat de l'art

La description détaillée des choix algorithmiques des méthodes distribuées, pour l'exploration fréquente d'itemset, est suivie d'une analyse expérimentale comparant la performance des implémentations distribuées les plus récentes et les plus fiables sur des ensembles de données synthétiques et réelles[108]. Les avantages et les inconvénients des algorithmes sont examinés en détail en respectant les caractéristiques de l'ensemble de données telles que leurs distributions, longueurs moyennes de transaction et nombres d'enregistrement. Enfin, à partir des analyses théoriques et expérimentales, des axes de recherche pour la parallélisation de la problématique d'extraction d'itemset fréquent sont présentés. Les résultats de l'examen expérimental ont révélé qu'aucun algorithme n'est universellement supérieur et que les performances sont fortement influencées par les cas d'utilisation et les données d'entrée relatives. En outre, les expériences ont mis en évidence l'importance fondamentale de l'équilibrage de charge et des coûts de communication.

0.4.3 Algorithme HFIMH basé sur Hadoop pour le traitement de données massives

Comme nous l'avons mentionné dans la sous-section précédente (et comme nous l'avons clairement montré au chapitre 4), la plupart des algorithmes actuels d'extraction d'itemset fréquents sont conçus pour traiter des ensembles de données de faible dimension, ce qui donne de mauvaises performances dans les cas d'utilisation caractérisés par des données à haute dimension. Ensuite, au chapitre 4, mon algorithme proposé, HFIMH, sera présenté en détail [105]. En guise d'aperçu, HFIMH est basé sur l'algorithme Apriori, en appliquant une méthode de calcul parallèle et de deux dispositions différentes de la base des transactions. Les résultats montrent que HFIMH est très performant en termes de temps d'exécution et d'utilisation de la mémoire.

0.4.4 Extraction des motifs fréquents sur des données massives avec HPrePostPlus sur Hadoop

Au chapitre 4, un nouvel algorithme nommé HPrePostPlus sera introduit [107]. C'est une amélioration au niveau des structures de données de l'algorithme PrePost qui, dans la plupart des cas, surpasse les algorithmes de l'état de l'art. HPrePostPlus est ainsi un algorithme distribué basé sur Hadoop, afin de traiter de volumineuses données. Les résultats des expériences montrent l'efficacité de l'algorithme proposé en termes de temps d'exécution, de robustesse aux problèmes de mémoire et de scalabilité.

0.5 Organisation

Cette thèse est organisée comme suit .Le chapitre 1 présente le contexte liées à l'extraction des motifs fréquents et aux plates-formes distribuées impliquées, ainsi que les principaux défis et les stratégies de parallélisation seront décrits. Par la suite, au chapitre 2, une analyse approfondie des solutions les plus efficaces sera effectuée. En étudiant les performances des meilleures approches existantes dans l'état de l'art, grâce à l'utilisation de jeux de données synthétiques et réels, en mettant en évidence les limites actuelles et les perspectives de l'état de l'art académiques.

Ensuite, Les chapitres 3 et 4 présentent HFIMH et HPrePostPlus, deux nouveaux algorithmes d'extraction d'itemset fréquents basé sur Hadoop MapReduce pour le traitement des données massives « Big Data ».

Chapitre 1

Fouille de Motifs fréquents et Big Data : Motivations, Défis et Etat de l'art

la fouille de motifs est le sujet numéro un dans la sphère de recherche sur la fouille de données. Ce constat est fondé aussi bien sur des aspects historiques que bibliométriques. D'une part, l'initiateur papier d'Agrawal, Imielinski, et Swami présenté en 1993 et intitulé « Mining Association Rules between Sets of Items in Large Database » [5] est en fait, considéré comme un des oeuvres fondateurs du domaine. Dans ce papier, les auteurs introduisirent pour la première fois la notion des règles d'association qui empocha vite un intérêt grandissant. D'autre part, l'examen du nombre de papiers consacrés au sujet [51] conduit aisément à reconnaître l'ampleur des efforts et contributions, non seulement fondamentaux mais aussi applicatifs, fournis par la communauté fouille de données à ce problème basique.

1.1 Présentation du problème

Nous présentons ici les notions de base relatives au problème de la fouille de motifs comme initialement introduit dans [5]. Nous discutons aussi sa complexité.

1.1.1 Définitions

Historiquement, le problème de la fouille de motifs fréquents a été proposé dans le cadre de l'analyse du panier de la ménagère (Market-Basket Analysis), dont le but fut de rechercher la collection de produits corrélés souvent achetés ensemble, en examinant les tickets de caisse enregistrés par les supermarchés. Cette connaissance des co-occurrences existantes entre produits est ensuite exploitée pour générer ce qui est appelé les règles d'association qui sont des formes d'implications entre ensembles de produits. La mise en évidence de ces produits reliés sert dans divers usages comme les campagnes de promotion, l'organisation des rayonnages, etc.
Bien qu'élémentaire, ce problème a connu un succès fulgurant et a conséquemment envahi plusieurs autre tâches d'extraction de connaissances telles que la classification, la segmentation et le Web mining pour ne citer que les plus remarquables.Ci-après la formulation de ce problème.

Support et motif fréquent

Étant donné un ensemble fini non vide $A = \{a_1, a_2, \ldots, a_m\}$ de m éléments communément appelés items. Ces items peuvent désigner, comme déjà mentionné, des articles commandés d'un supermarché, des pages Web visités en surfant, ou, en général, une collection de variables, d'attributs, ou d'évènements.
Un motif ou encore un itemset I est un sous-ensemble de A. Il est qualifié de k-motif lorsque son cardinal est égal à k. Dans le contexte du modèle du panier de la ménagère, un k-motif symbolise donc l'ensemble des produits commandés par un client lors de ses courses. De même, une transaction $t_(i)$ est un sous-ensemble non vide de A. Elle est identifiée par son identificateur unique i. Un ensemble (ou une base) de données D est un ensemble de n transactions, qu'on note comme un multiset $D = \{t_1, t_2, \ldots, t_n\}$.
Notons que dans ce problème, le nombre n de transactions est généralement supposé très large et que la taille (longueur) d'une transaction est nettement inférieur au nombre total d'items. Dans une base de données D, l'ensemble de transactions qui contiennent un motif x est appelé sa couverture. Le support d'un motif x, dénoté $sprt(x, D)$ est le nombre de transactions qui contiennent x, i.e., le cardinal de sa couverture.

$$sprt(x, D)_{abs} = |t_k \in D / x \subseteq t_k| \tag{1.1}$$

Précisons que l'équation donne la valeur absolue du support (un entier positif inférieur ou égale à la taille de la base de données), qui peut aussi être exprimée en relatif, c-à-d, par un réel compris entre zéro et un, représentant un pourcentage

en le divisant par la taille de l'ensemble de données comme cela est montré dans la formule ci-dessous .

$$sprt(x, D)_{rel} = \frac{|t_k \in D/x \subseteq t_k|}{|D|} \tag{1.2}$$

Afin d'alléger l'écriture, nous le notons simplement $sprt(x)$ sans sous-mention abs ni référence à une base de transactions qui seront compris du contexte. Ceci est actuellement une tendance largement répondue dans la communauté fouille de données [20, 44, 46, 78].

Définition (Motif fréquent) Un motif x est fréquent si son support dépasse un seuil minimal s spécifié d'avance. Formellement, x est fréquent dans D relativement au seuil s ssi : $sprt(x) \geqslant s$

Le problème de la fouille de motifs consiste à énumérer l'ensemble $F_D(s)$ de tous les motifs fréquents dérivés de A présents dans la base de transactions D au regard du seuil minimal du support s. Pour alléger la notation, ce dernier ensemble sera, désormais, noté F tout court.

$$F_D(S) = x \in /sprt(x) \geqslant s \tag{1.3}$$

Règle d'association et confiance

Définition (Règle d'association) Une règle d'association est une implication de la forme $X \Rightarrow Y$ où X et Y sont des motifs non vides et disjoints

Une règle d'association exprime la liaison entre l'ensembles d'items de l'antécédent X et ceux du conséquent Y. De manière similaire et vu le nombre prohibitif de règles pouvant être générées, il est généralement admis d'opérer ici aussi un filtrage de façon à ne retenir que les règles intéressantes. Par conséquent, outre l'application du seuil de support minimal à toute règle d'association produite , une deuxième condition de confiance est aussi exigée. La confiance d'une règle exprime la proportion des transactions de la base contenant Y parmi celles qui contiennent X. Elle traduit donc la probabilité conditionnelle des transactions contenant Y sachant quelles contiennent X. Cette dernière contrainte permet de retirer les règles qui ne satisfont pas le seuil minimal de confiance c également fixé a priori par l'utilisateur ou l'expert du domaine. Les règles qui vérifient à la fois les deux seuils support et confiance sont parfois qualifiées de fortes. Les règles d'association sont écrites alors en spécifiant leurs importances :

$$X \Rightarrow Y[sprt = vs, conf = vc] \tag{1.4}$$

Où vs et vc sont les valeurs calculées pour le support et la confiance de la règle respectivement. Ci-dessous les deux formules permettant de mesurer l'intérêt d'une règle, comme définies dans l'énoncé original du problème [8, 132] :

$$sprt(X \Rightarrow Y) = sprt(X \cup Y) \tag{1.5}$$

$$conf(X \Rightarrow Y) = \frac{sprt(X \cup Y)}{sprt(X)} \tag{1.6}$$

La tâche d'extraction des règles d'association consiste, en considérant les deux paramètres s et c, à générer l'ensemble des règles fortes de la base. Afin d'éviter des temps de calcul importants, et comme nous pouvons le constater de la formule précédente, une règle ne peut être intéressante si elle ne satisfait pas la condition du support ; cela nous permet de filtrer l'ensemble de ces règles avant même d'entamer leurs examens. C'est ainsi que ce problème est souvent effectué, pour fins d'optimisation, en deux phases [5] :

• L'énumération d'abord des motifs fréquents satisfont le seuil de support s,

• La formation par la suite, parmi ces derniers, des règles intéressantes justifiant le seuil de confiance c.

Il est facile de remarquer que la phase difficile et gourmande en temps et espace est la première, car elle exige l'exploration de l'espace du problème afin de découvrir les motifs fréquents. En effet, une fois cette dernière achevée, la deuxième est relativement aisée vu que sa mission se résume à dériver de la précédente l'ensemble des règles possibles en se limitant à celles fortes.

Il est à souligner que l'approche d'extraction des règles d'association basée sur le découpage en deux sous-problèmes mentionnés en haut a été adoptée par la quasi totalité des travaux qui se sont intéressés au sujet .

Afin d'illustrer les concepts introduits, nous donnons, dans le tableau 1.1, un exemple d'un ensemble de données de dix transactions. Cet ensemble de données servira comme exemple de référence dans la suite du chapitre.

En retenant 3 et 1 comme seuils minimaux de support et de confiance respectivement, nous pouvons remarquer sans difficulté dans l'exemple que :

Tous les items formant les cinq singletons sont fréquents avec les supports

table1.1 exemple d'une base de 10 transactions sur l'ensemble d'items $A = \{a, b, c, d, e\}$

Suivants : {(a,10),(b,4),(c,8),(d,7),(e,6) } .

— Le motif {a,c,e} est fréquent (support = 4). Par contre ,{b,d} ne l'est pas (support = 2)

TABLEAU 1.1: Exemple d'une base de 10 transactions sur l'ensemble d'items $A = \{a, b, c, d, e\}$

i	t_i	i	t_i
1	{a,c,d,e }	6	{a,c,d,e }
2	{a,c,d }	7	{a,b,c,d }
3	{a,b,c,d }	8	{a,b,e }
4	{a,b,e }	9	{a,c,d }
5	{a,c,e }	10	{a,c,d,e }

— En se référant aux formules 2.5 et 2.6, la seule règle forte dérivable du motif fréquent {a,c,e} est :$\{c, e\} \Rightarrow \{a\}[sprt = 4, conf = 1]$
Nous donnons ci-dessous l'ensemble F tout entier (tous les motifs fréquents) de la base de notre exemple relativement à un seuil minimal de support égal à 3
$F = \{((\{a\}, 10), (\{b\}, 4), (\{c\}, 8), (\{d\}, 7), (\{e\}, 6), (\{a, b\}, 4), (\{a, c\}, 8), (\{a, d\}, 7), (\{a, e\}6),$
$(\{c, d\}, 7), (\{c; e\}, 4), (\{d, e\}, 3), (\{a, c, d\}, 6), (\{a, c, e\}, 4), (\{a, d, e\}, 3), (\{c, d, e\}, 3), (\{a, c, d, e\}, 3$

1.1.2 Espace d'états et complexité

La fouille de motifs est un problème complexe qui exige un calcul énorme. En effet, pour un ensemble d'items A, nous devons explorer tous les motifs possibles. L'ensemble puissance de A constitue donc l'espace de recherche de ce problème, qui compte donc $2^{|A|} - 1$ sous motifs candidats (pouvant être fréquents) en excluant le motif vide qui ne véhicule aucune information utile. L'espace du problème est clairement exponentiel avec le nombre d'items, et par conséquent «intractable» [96] même avec une collection faible d'items.
Quant à la complexité temporelle asymptotique, elle dépend en général du nombre de motifs candidats possibles multiplié par le temps nécessaire au calcul du support pour chaque instante. Ainsi, il est évident qu'une solution naïve à ce problème, montrée dans l'Algorithme 2 coûte : $O(|D||A|2^{|A|})$car le calcul du support d'un motif exige de le comparer avec chacune des transactions de la base, i.e., un temps en $O(|D||A|)$ au pire des cas. Ceci est clairement inefficace voire irréalisable, compte tenu du nombre d'items considérés dans les applications réelles souvent très abondant. Bien entendu, la complexité en pratique est tributaire à plusieurs autres aspects. Outre les caractéristiques des compilateurs et des machines, elle est aussi fonction, en autres, de la nature de la base de transactions (sparse ou dense), de la valeur du seuil du support dont les valeurs trop faibles induisent alors des temps

de calcul très prohibitifs, et enfin des structures de données utilisées offrant des gains sur plus d'un titre tels que dans la génération des candidats et le nombre d'entrées/sorties.

Dans le chapitre suivant, nous étudions quelques optimisations communes introduites afin d'améliorer le processus de fouille de motifs fréquents, et donnons des bornes de complexité plus fines pour ce problème. Néanmoins, signalons d'emblée que l'espace de recherche peut être représenté par un treillis complet$(2^A \subseteq)$ de l'ensemble2^A muni de l'ordre partiel induit par la relation d'inclusion ensembliste. Le diagramme de Hasse de cet espace dans le cas d'un ensemble de cinq items $A = \{a, b, c, d, e\}$ est illustré dans la figure 1.1, où les noeuds sont les motifs et les arrêtes correspondent aux relations d'inclusion.

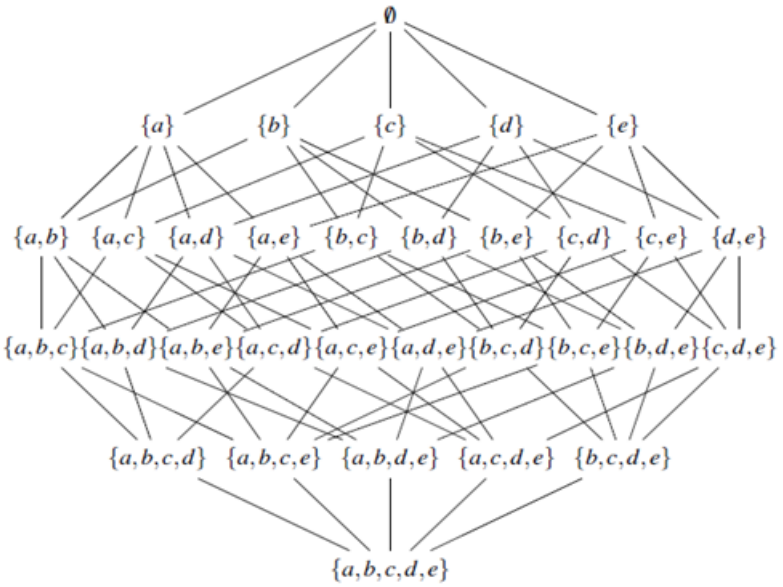


FIGURE 1.1: Espace de recherche dans la fouille de motifs comme un treillis

1.2 Approches de fouille de motifs fréquents

Un état de l'art sur les méthodes de fouille de motifs fréquents fera l'objet du présent chapitre. Nous commençons par évoquer les propriétés permettant une taxonomie des travaux de la littérature, pour ensuite s'attaquer à les distinguer. Pour chaque approche, nous donnons les traits primaires et présentons l'algorithme type, souvent l'initiateur, comme représentant des travaux qui y sont affectés. La présentation évoque : l'idée derrière le papier originel, le pseudo-code, un exemple

d'exécution, pour terminer avec une analyse et quelques fameuses extensions. La discussion ne prétend pas à l'exhaustivité vu le nombre important de travaux, mais se contentera plutôt des principales approches introduites, étant donné que la quasi-totalité des algorithmes sont dérivés de celles-ci moyennant des adaptations.
Comme cette thèse a essentiellement une vocation formelle, il est à mentionner que seules les approches de fouille exactes et complètes sont explorées. Ceci dit, les travaux qui visent une fouille approchée ou partielle ne sont pas considérés ici, même si un bref passage leur sera réservé en fin du chapitre. Notons également que le même exemple de la base de données introduit dans le chapitre précédent est retenu ici à des fins d'illustration

1.2.1 Représentation de données

Dans la fouille de motifs, un ensemble de données D est modélisé par une relation binaire entre un ensemble de transactions T et un ensemble d'items A. Le modèle binaire pur ($leproduitT \times A$)peut être représenté par une matrice binaire à deux dimensions, où une case (i, j) prend la valeur 1 ou 0 selon que la transaction i contient ou non l'item j. Pour des raisons de gain d'espace, une vue horizontale (resp. verticale) est souvent employée, permettant de ne considérer (à l'aide de listes ou tableaux) que les éléments (items/transactions) présents effectivement dans la base. La représentation horizontale donne les items présents pour chaque transaction. Elle est alors une vue centrée transactions ou lignes, tandis que celle verticale énumère la liste de transactions où apparaît un item donné, dite tidlist (pour transaction identifier list) de l'item considéré. La représentation verticale est, quant à elle, orientée items ou colonnes et peut être vue comme la transposée de la matrice binaire initiale. D'autres algorithmes ont adopté une combinaison des deux représentations afin de tirer profit des avantages de chacune. Cette dernière représentation estqualifiée d'hybride, dont l'implémentation célèbre est celle qui utilise les arbres de préfixes ou les tries [88], après l'adoption d'un ordre fixe sur les items. Le choix de l'une des représentations citée ci-dessus est en lien étroit avec (ou du moins supporte) les autres aspects tels que la stratégie d'exploration de l'espace de recherche ou le calcul des supports des motifs. Une illustration des principales représentations de données pour la fouille de motifs, à l'égard de notre exemple de référence présenté dans la table 1.1, est montrée dans la figure 1.2.

Trans.	a	b	c	d	e
t_1	1	0	1	1	1
t_2	1	0	1	1	0
t_3	1	1	1	1	0
t_4	1	1	0	0	1
t_5	1	0	1	0	1
t_6	1	0	1	1	1
t_7	1	1	1	1	0
t_8	1	1	0	0	1
t_9	1	0	1	1	0
t_{10}	1	0	1	1	1

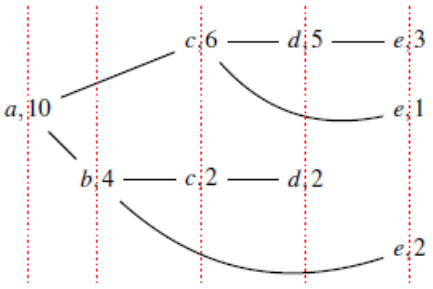
(a) Binaire

Trans.	Motifs
1	$a\ c\ d\ e$
2	$a\ c\ d$
3	$a\ b\ c\ d$
4	$a\ b\ e$
5	$a\ c\ e$
6	$a\ c\ d\ e$
7	$a\ b\ c\ d$
8	$a\ b\ e$
9	$a\ c\ d$
10	$a\ c\ d\ e$

(b) Horizontale

Items	Tidlists
a	1,2,3,4,5,6,7,8,9,10
b	3,4,7,8
c	1,2,3,5,6,7,9,10
d	1,2,3,6,7,9,10
e	1,4,5,6,8,10

(c) Verticale



(d) Hybride

FIGURE 1.2: Différentes représentations de la base de l'exemple de référence

1.2.2 Méthode d'exploration de l'espace de recherche

L'espace de recherche de notre problème est un treillis semblable à celui de la figure 2.1. Les algorithmes de fouille de motifs adoptent différents schémas de parcours de cette structure. Il est évident que la méthode naïve est inefficace car elle permet la génération redondante de motifs. Pour parer à ce problème, les approches adoptent un ordre quelconque mais fixe sur les items afin d'éviter de générer les même motifs plusieurs fois. Ceci est simplifié par la réduction du treillis en un arbre de préfixes en considérant l'ordre choisi sur les items. Dans cette structure, dite aussi arbre d'énumération, les nœuds des k-motifs partagent le même père (représentant un $(k-1)$motif) s'ils le possèdent comme un préfixe commun ; les nœuds associés aux motifs singletons constituent les fils de la racine de l'arbre qui dénote le motif nul. La figure 1.3 illustre cette structure, où les items sont traités comme des symboles de l'ensemble A.

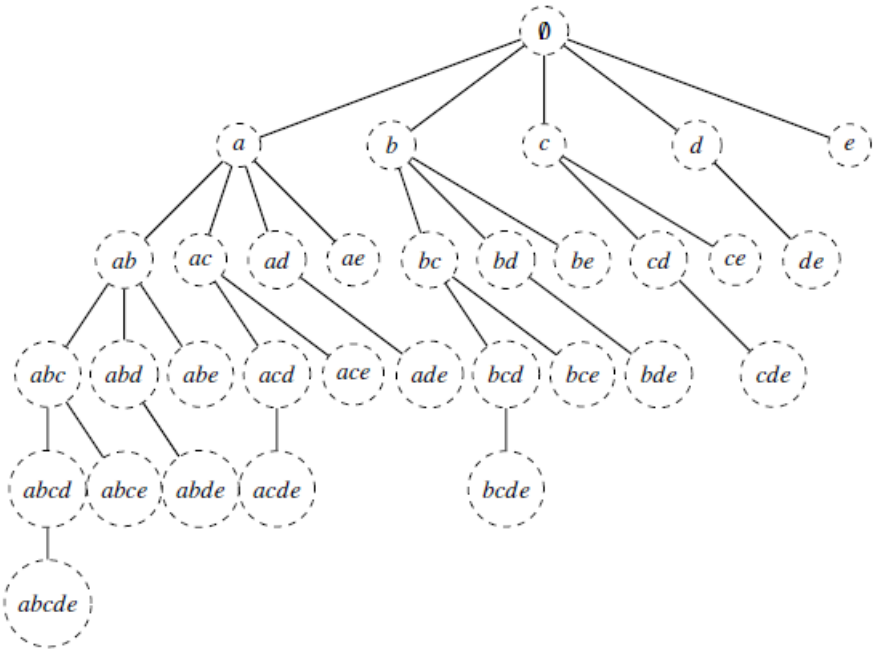


FIGURE 1.3: Arbre d'énumération (de préfixes) dérivé de l'espace de recherche dans la fouille de motifs le treillis $(2^A, \subseteq)$ des sous ensembles de $A = \{a, b, c, d, e\}$ en adoptant l'ordre lexicographique des items

L'exploration de l'arbre peut être effectuée selon les schémas bien connus en théorie des graphes. La première est l'exploration en largeur d'abord (Breadth-First Search (BFS)) qui examine, en partant de la racine, les nœuds de l'arbre par niveaux. La mission à l'étape k est de reporter les k-motifs fréquents. C'est une méthode qui scanne l'arbre des motifs des plus généraux à ceux plus spécifiques (les 1-motifs, puis le 2-motifs, etc.), qualifiée également d'horizontale ou descendante. Le même principe peut être envisagé mais dans le sens inverse ascendant, où les motifs plus spécifiques sont visités en premier pour remonter progressivement aux plus généraux. La deuxième manière de scruter l'espace du problème s'engage dans un parcours en profondeur d'abord (Depth-First Search (DFS)). L'espace est décomposé en sous-problèmes plus petits relativement à une relation de préfixes communs, en progressant à chaque fois par un nœud qui est traité souvent de façon récursive. Il est à noter que d'autre approches combinent les deux méthodes dans une stratégie hybride basée sur des heuristiques afin d'optimiser le processus global d'exploration.

1.2.3 Génération de candidats

À la recherche de motifs fréquents, la majorité des algorithmes suivent une stratégie connue sous le slogan générer et tester consistant à proposer dans une première phase des motifs candidats, i.e., potentiellement ou vraisemblablement fréquents, puis à valider ce caractère dans une deuxième phase en se référant à la base de transactions. Le nombre et la manière avec laquelle sont suggérés les candidats pour le test de fréquence influent considérablement sur l'efficacité de l'algorithme de fouille de motifs. Outre le procédé naïf, il existe principalement deux façons pour proposer des motifs candidats selon qu'elle utilise la jointure ou l'extension. Dans la première, deux k-motifs fréquents partageant un $(k-1)$ préfixe commun sont joints pour générer un $(k+1)$ motif. Par exemple, la jointure des deux 3-motifs $\{a,b,c\}$et$\{a,b,d\}$ qui partagent le 2-préfixe$\{a,b\}$ engendre le 3-motif$\{a,b,c,d\}$. Dans la deuxième, un k-motif est étendu par des items successeurs, selon un ordre établit à l'avance sur les items, pour donner naissance à un nouveau $(k+1)$-candidat. Pour le dernier exemple, le 3-motif $\{a,b,c\}$ peut être étendu en considérant l'ordre lexicographique des items par les item$\{d,e\}$, ceci fournira les deux 4-motifs $\{a,b,c,d\}$et $\{a,b,c,e\}$. Les deux méthodes sont équivalentes et constituent différents moyens de construction de l'arbre d'énumération que tout algorithme de fouille est censé générer.

1.2.4 Calcul des supports

Plusieurs techniques ont été proposées pour mesurer le support d'un motif, étant donné qu'il représente le facteur le plus pesant dans l'efficacité des algorithmes de fouille de motifs. Nous nous limitons ici à seulement deux versions. La première et la plus triviale, employée surtout dans les représentations horizontales, utilise le comp- tage par scans de la base. Un candidat (souvent un ensemble d'entre eux en pratique) est affronté aux transactions (toutes ou une partie d'entre elles selon l'approche d'exploration), où un compteur est incrémenté à chaque fois qu'une occurrence de celuici est détectée dans une transaction. L'autre technique emploie l'intersection des listes de transactions (tidlists). Dans le cadre d'une représentation verticale, pour connaître le support d'un motif $X \cup Y$il suffit, en partant des tidlists associées aux motifs X et Y, de calculer le cardinal de

l'intersection$|tidlist(X) \cap tidlist(Y)|$. Plusieurs autres optimisations ont été proposées afin d'améliorer l'efficacité de cette phase capitale. Certaines d'entre elles seront évoquées ultérieurement.

1.3 Énumération totale

L'énumération totale vise la découverte complète de l'ensemble F tout entier, sans omission ni erreur comme défini dans la formule 2.3. La sortie de l'algorithme doit inclure tout motif fréquent et en même temps omettre tout motif infréquent. La question est de savoir si une exploration exhaustive de l'espace de recherche est toujours exigée ? En effet, la réponse est négative comme a été noté par [8, 75, 85]. Une propriété clé permettant d'optimiser considérablement le processus de recherche de motifs fréquents et ainsi passer outre l'exploration de la totalité de l'espace du problème a été introduite. Cette propriété, pourtant évidente, est indéniablement perçue comme une avancée considérable dans le sujet et a été par conséquent adoptée plus tard par la totalité des algorithmes. Cette propriété connue sous l'appellation d'anti-monotonie du support ou de fermeture descendante de l'ensemble de motifs fréquents, stipule que si un motif est fréquent alors tous ses sous-motifs le sont aussi ; et dualement, si un motif est infréquent alors tous ses sur-motifs ne peuvent être fréquents. L'impact de cette astuce dans les implémentations se traduit à localiser, dans l'espace du problème, ce qui communément appelé la bordure de fréquence puis de restreindre ainsi la génération seulement aux motifs situés en dessus de cette frontière. Les autres motifs qui se situent en bas de cette limite sont donc infréquents et doivent être ignorés. Ces notions sont introduites ci-dessous et illustrées dans la figure 1.4.

Lemme : (Apriori ou anti-monotonie du support (Agrawal and Srikant, 1994)). Soit D une base de transactions, x et y deux motifs sur l'ensemble d'items A, nous avons :

$$x \subseteq y \Rightarrow prt(x) \geqslant sprt(y) \tag{1.7}$$

Démonstration. Triviale. Elle découle de la définition 2.1 relative à la couverture d'un motif et de la transitivité de l'inclusion. En effet, si une transaction t_i contient un motif y, elle inclut forcément tout sous-motif x de y

De ce lemme, il découle le corollaire utile suivant.

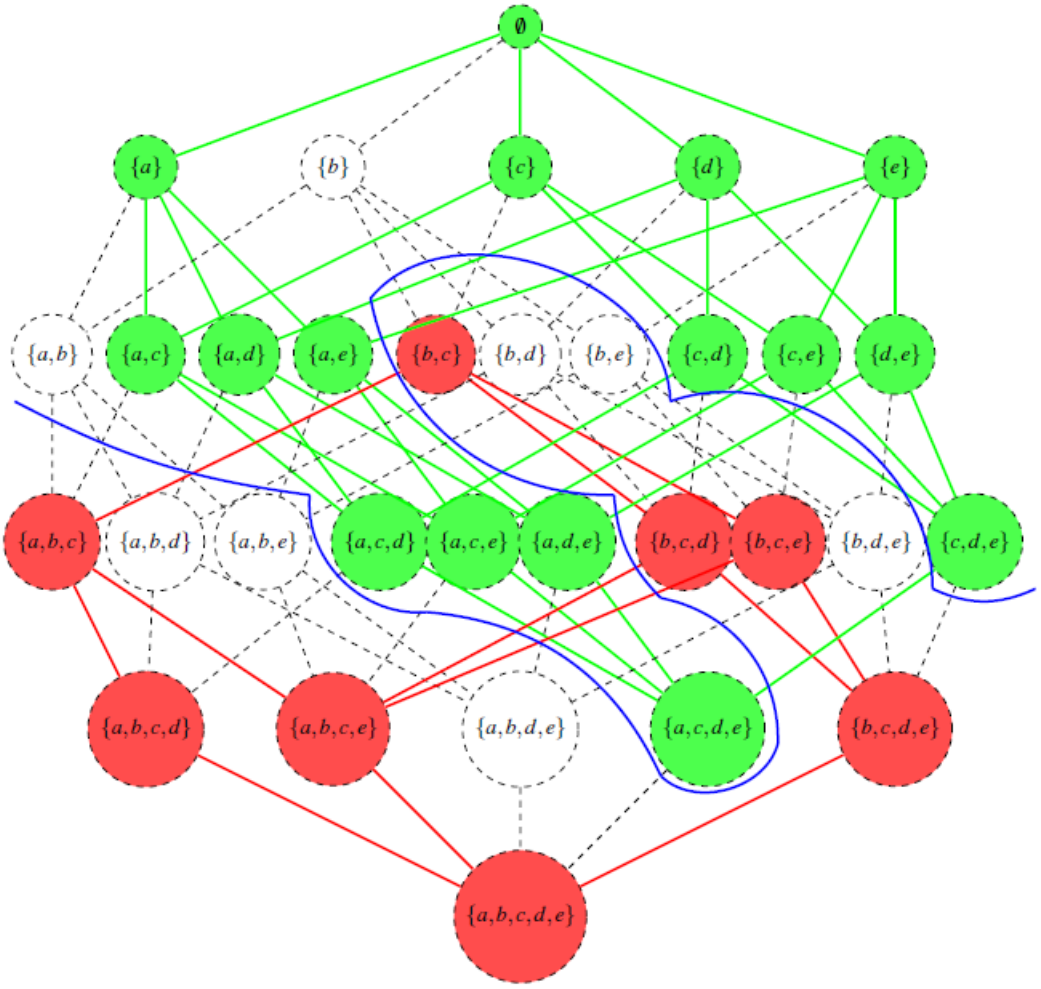


FIGURE 1.4: Illustration du principe d'anti-monotonie du support : $\{a, c, d, e\}$ est fréquent, tous ses sous motifs le sont aussi (couleur verte). De même, $\{b, c\}$ est infréquent ; tous ses sur-motifs sont également infréquents (couleur rouge)

Corollaire (Fermeture descendante des motifs fréquents). Soit D une base de transactions sur un ensemble d'item A, et s un seuil de support, nous avons :

$$xfréquent \Rightarrow \forall y \subseteq x, yestaussifréquent \tag{1.8}$$

Corollaire Soit D une base de transactions sur un ensemble d'item A, et s un seuil de support, nous avons :

$$xinfréquent \Rightarrow \forall y \supseteq x, yestaussiinfréquent \tag{1.9}$$

1.3.1 Approches par niveaux

Comme leur nom indique, ces méthodes extraient les motifs fréquents niveau par niveau. En premier lieu, les singletons (1-motifs) fréquents sont d'abord déterminés et ceux infréquents écartés. Ensuite, la mission de l'algorithme consiste à générer les paires fréquentes en combinant les singletons fréquents, puis les triplets fréquents à partir des doublets fréquents, et ainsi de suite. Souvent itératives, cette classe de méthodes utilisent une représentation horizontale de la base de transactions et le principe générer-tester par une stratégie d'exploration de type BFS. Ainsi, les candidats potentiels à l'étape k, notés par l'ensemble C_k, sont construits par jointure de l'ensemble des $(k-1)$ motifs fréquents, dénoté F_{k-1}, trouvés dans l'étape $(k-1)$. Le processus continue en alternance entre génération de candidats et calcul de support, via des balayages de la base de données, et s'arrête lorsque il n'est plus possible de découvrir de nouveaux motifs fréquents de longueurs supérieures. Le résultat de l'algorithme est alors l'union des différents k-motifs fréquents trouvés.

1.3.1.1 Algorithme Apriori

Apriori [8] est incontestablement l'algorithme pionnier de cette classe de méthodes et d'ailleurs pour les autres aussi. Il est introduit comme une amélioration de son prédécesseur AIS [5], baptisé selon les noms de ses inventeurs, qui n'emploie aucune optimisation. La force d'Apriori, dont le principal pseudo-code 1 est donné en algorithme 3, est dû à l'heuristique d'antimonotonie du support dite aussi la propriété Apriori tout court, qui a offert une réduction considérable du nombre de candidats à examiner. Cette dernière est exploitée selon deux optimisations : (i) à l'étape k, un k-candidat x est éliminé s'il possède un sous-motif infréquent de taille $(k-1)$ (ii) la totalité du sous arbre enraciné au motif x est ignoré si ce dernier s'avère infréquent. Notons que le principe Apriori fût indépendamment découvert également dans [75]. Les deux groupes de recherche consolident plus tard leur travaux dans un seul article [6].
De plus, pour accélérer le calcul des supports réalisé par balayages de la base, les motifs candidats sont consignés dans un arbre de hachage (hash tree). Dans cette structure, les motifs sont stockés dans les feuilles, et les noeuds internes au niveau d contiennent des tables de hachage qui pointent sur des noeuds du niveau

suivant $d+1$. Ainsi, lors du calcul des supports et pour comparer les candidats à une transaction, les items de cette dernière sont utilisés pour descendre (par application de la fonction de hachage aux différents items) dans l'arbre et atteindre les candidats possibles, pour lesquels les supports associés sont incrémentés.
Le déroulement de l'algorithme Apriori est illustré ci-dessous pour la base de notre exemple de référence. Les motifs barrés avec un point entre parenthèse sont les candidats éliminés en raison qu'ils ne respectent pas la clause (i) de la propriété Apriori (contiennent un sous-motif infréquent) ; ceux barrés avec le support entre parenthèses sont des candidats retenus mais identifiés comme infréquents à la suite du calcul de leurs supports, c'est-à-dire ne vérifiant pas la clause (ii).

Algorithm 1 APRIORI (D ; s)

Entrée : Une base de transactions D, et un seuil de support s
Sortie : L'ensemble F des motifs fréquents
$k \leftarrow 1$
$F_1 \leftarrow$Les singletons fréquents
tant que$F_k \neq \emptyset$ faire
Générer C_{k+1} par jointure des motifs de F_k
Éliminer de C_{k+1} les éléments qui violent la propriété Apriori
DéterminerF_{k+1} par calcul des supports des candidats retenus
$k \leftarrow k+1$
fin tant que
retourner$U_i F_i$

1.3.1.2 Analyse d'Apriori

La preuve de correction des algorithmes par niveaux dont Apriori est un représentant est donnée dans[85], en considérant ce problème comme celui de la recherche des instances d'un langage satisfaisant le prédicat de fréquence. Comme est indiqué dans le pseudo-code d'Apriori, le temps est dominé par les étapes de génération de candidats et de calcul des supports. Cet algorithme suit une génération en largeur, ceci borne son calcul par le plus long motif fréquent dans la collection, qui représente également le nombre de scans nécessaires à l'algorithme. Aussi, sa complexité asymptotique se réduit à $O(l|D|2l)$, où l est la taille du plus long motif fréquent. Donc, malgré l'amélioration apparente par rapport à l'approche naïve, la complexité de ces algorithmes demeure toujours exponentielle de l'ordre de $2|A|$. En pratique, outre les détails d'implémentations, plusieurs autres facteurs influent

sur le temps de réponse de ces algorithmes [117].

• Bien que n'apparaît pas directement sur l'estimation de la complexité, une valeur trop faible du seuil minimal du support s induit des calculs énormes, vu qu'elle élargit la bordure de fréquence et pousse à la hausse le nombre possible de motifs fréquents,

• Les algorithmes sont directement sensibles au nombre d'items $|A|$, qui représente la dimensionnalité des données et détermine la taille de l'espace de recherche,

• La taille de la base de données $|D|$ exprimant le nombre de transactions est aussi capitale dans cette complexité,

• La longueur moyenne d'une transaction, car des transactions plus larges qui caractérisent les bases de données denses tendent à compliquer les calculs en raison de la présence de nombreux candidats à examiner.

En tout état de cause, [11, 56] précisent le caractère difficile du problème, étant donné qu'il est énumératif. En fait, il a été prouvé que le comptage du nombre de motifs fréquents est ≠P-difficile [96]en fournissant une réduction polynômiale au problème de comptage d'assignations pour la satisfaction d'instances de problèmes $2SAT2$, et que la question de décider de l'existence, pour un k donné, d'un k-motif fréquent dans une base de transactions est NP-complet. C'est pour ces raisons que souvent l'examen des algorithmes de fouille de motifs préconise des complexités sensibles à la sortie. Il a été démontré par ailleurs dans [85, 119] que tout algorithme de fouille de motifs doit au moins évaluer la bordure de fréquence. Mais, à quel ordre cette frontière peut être large ? En effet, cette frontière est bornée comme stipule le lemme suivant.

Apriori est un algorithme intéressant qui a bénéficié de nombreuses implémentations publiées 3 [19, 20, 76], dont certaines sont réalisées dans divers outils de fouille de données comme nous avons vu à la section 2.3. Il est particulièrement efficace dans les bases de données sparses avec des motifs peu longs [99, 129]. Toutefois, deux inconvénients sont souvent infligés à Apriori. Le premier est le fait de générer un nombre considérable de candidats, ce qui dégrade significativement, comme déjà mentionné, ses performances. Le second à trait au surcoût causé par les multiples et couteuses opérations d'entrée/sortie nécessaires pour le calcul des supports. Aussi, plusieurs améliorations ont été introduites afin de l'améliorer. Ci-après, nous présentons brièvement quelques perfectionnements notables.

1.3.1.3 Quelques extensions à Apriori

AprioriTid et AprioriHybrid sont deux extensions mineures proposées dans le même papier d'Apriori [8]. Le premier permet de réduire le nombre de balayages de la base à seulement un seul. En effet, AprioriTid construit, après la première passe, des tables de transactions réduites aux motifs candidats, générées de la même manière qu'Apriori, que contiennent celles-ci. Ces structures remplacent la base qui n'est donc plus utilisée dans les calculs. Il a été montré que AprioriTid donne des résultats inférieurs comparés
à Apriori dans les toutes premières itérations, vu que les nouvelles structures induisent des surcoûts en mémoire (et en temps car les données sont à remplacer sur disque). Toutefois, au fur à mesure que l'algorithme progresse les résultats dépassent ceux d'Apriori, pour les raisons inverses. Ce constat a incité les auteurs à proposer AprioriHybrid qui combine les deux, dans lequel Apriori est invoqué en premier, puis dès que les structures d'AprioriTid
peuvent se loger en mémoire AprioriHybrid commute à AprioriTid de manière à tirer profit des avantages des deux algorithmes.
DHP (Dynamic Hashing and Pruning) [97]est introduit pour réduire le nombre de candidats. Dans DHP, les auteurs ont remarqué que durant la première passe d'Apriori une bonne partie de la mémoire reste libre et inexploitée. Pour ce faire, cet algorithme prépare, durant le calcul des supports des k-motifs, des indicateurs renseignant sur les fréquences des $(k+1)$-motifs. Une table de hachage est utilisée pour stocker des compteurs des candidats, regroupés en blocs, représentant les sous-motifs de la transaction en cours. Ainsi, dans l'étape de calcul des supports des $(k+1)$-motifs, seuls les candidats des blocs dont les compteurs dépassent le seuil minimal du support sont pris en considération, le reste sera ignoré. Cette astuce s'est avérée efficace précisément dans l'étape de calcul des pairs (2-motifs) fréquents qui dominent le calcul dans ce problème. De plus, DHP réduit progressivement la taille de la base et tronque les transactions en s'appuyant sur la condition nécessaire que tout item i peut être supprimé s'il n'apparait pas au moins k fois dans les k-motifs candidats d'une transaction.
Partition [111] est un algorithme qui réduit le nombre de scans de la base à seulement deux. Dans la première, il fractionne la base en plusieurs parties disjointes choisies telle que leurs tailles tiennent en mémoire, qui les traite séparément en extrayant leurs motifs fréquents locaux. Dans la seconde passe, les résultats locaux sont consolidés et vérifiés en confirmant les motifs fréquents globaux et écartant le

reste (les résultats de la première passe peuvent inclure des faux positifs). L'idée sur lequel repose Partition est qu'un motif ne peut être fréquent dans la base s'il n'est pas fréquent dans au moins une des partitions.
Sampling [119] travaille sur un échantillon aléatoire plus petit de la base et nécessite deux balayages au pire sur cette dernière. La technique a montrée sa précision et son efficacité notamment pour les bases trop larges. Dans le premier scan, l'algorithme extrait les motifs fréquents dans l'échantillon avec leur bordure négative. Les résultats (motifs et bordure) sont ensuite validés par un second balayage. Pour éviter de sauter des motifs fréquents dans la base, le seuil de support est diminué, lors de la première passe, à une valeur qui garantit avec une forte probabilité d'éviter ces oublis.
DIC (Dynamic Itemset Counting) [23] est motivé par la réduction du nombre de passes sur la base de données. La philosophie de cet algorithme entremêle génération de candidats et calcul des supports en anticipant la formation de motifs candidats plus long des motifs trouvés fréquents sans même attendre de voir toutes les transactions. Par analogie au train, la base est divisée en wagons de M transactions chacun, à chaque wagon de nouveau motif sont ajoutés/supprimés dynamiquement. La dynamicité est suivie en catégorisant les motifs en quatre classes : confirmés fréquents/infréquents et présumés fréquents/infréquents.

1.3.2 Approches verticales

Une représentation verticale de la base de transactions associée à un mécanisme de calcul de support via des intersections de tidlists et un parcours de l'espace du problème en profondeurs (DFS) sont les traits caractéristiques des approches verticales de fouille de motifs. Bien qu'Eclat [131], proposé par Zaki et al. Et décrit dans la suite, est considéré comme l'algorithme type de cette famille de méthodes, les fondements et les origines de cette approche remontent, à vrai dire, à bien avant. En effet, la représentation verticale comme structures inversées a été étudiée dans la communauté bases de données depuis les années 80 (Copeland and Khoshafian, 1985). En outre, l'astuce de calcul des supports par intersection des tidlists a été utilisée très tôt par Savasere et al. pour ce problème [111] dans le cadre de l'algorithme horizontale Partition. Les origines de l'astuce intersection peuvent être aussi trouvées dans les théories des treillis [16]et l'analyse formelle de concepts [123, 132].

1.3.2.1 Algorithme Eclat

Les atouts de l'algorithme ECLAT (Equivalence CLAss Transformation) sont attribués, outre l'heuristique Apriori, essentiellement à trois principes clés. D'une part, l'algorithme indexe la base de données par les items en fournissant leurs tidlist afin d'améliorer son efficacité. Ce principe en plus qu'il diminue les nombreux scans de la base, il offre aussi un calcul rapide des supports. En effet, Eclat exploite ces dernières listes pour la détermination des supports, qui sont alors les cardinaux des intersections résultantes. Par exemple, le support de $\{a, b\}$ est 4 représentant le cardinal de l'intersection des deux tidlists des items a et b qui vaut $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 1\} \cap \{3, 4, 7, 8\} = \{3, 4, 7, 8\}$. Le dernier principe sur lequel repose Eclat, est une stratégie de décomposition de l'espace de recherche en parties disjointes via la définition de classes d'équivalence sur cet espace. Cette décomposition permet d'alléger les traitements en réduisant ainsi la mémoire requise. La relation d'équivalence introduite est basée sur des préfixes communs (deux motifs x et y son équivalents, s'ils partagent un k-préfixe). L'ensemble des principes précédents sont combinés dans un algorithme récursif qui explore l'espace du problème en profondeur, comme il est montré par le pseudo-code de l'algorithme 4. Pour simplifier la présentation, une partie seulement du déroulement de l'algorithme Eclat relative aux motifs de la classe de l'item a (exceptés les singletons) est illustrée pour l'exemple de référence dans la figure 1.5

Algorithm 2 ECLAT(L ; s ;F)

Entrée : Une base de données transactionnelle D, et un seuil de support s
Sortie : L'ensemble F des motifs fréquents
$F \leftarrow \emptyset$
$L \leftarrow$les singletons fréquents
pour tout $P_i \in L$ faire
$F \leftarrow F \bigcup P_i$
$FP_i \leftarrow \emptyset$
pour tout $P_j \in L | j > i$ faire
si $|tidlist(P_i) \cap tidlist(P_j)| \geqslant s$ alors
$FP_i \leftarrow FP_i \bigcup \{P_i \bigcup P_j\}$
fin si
fin pour
si $FP_i \neq \emptyset$ alors
ECLAT$(FP_i; s; F)$
fin si
fin pour

1.3.2.2 Analyse d'Eclat

L'avantage d'Eclat par rapport aux méthodes horizontales est la rapidité dans le calcul des supports via des intersections intelligentes d'ensembles. En effet, pour accélérer le calcul des intersections, les tidlists dans Eclat sont, d'un coté, ordonnées de façon croissante ce qui produit un temps linéaire borné par la taille des deux listes. D'un autre coté, le calcul de cette intersection peut être rapidement abandonné prématurément par une technique dite intersection court-circuit, dès qu'on est certain que l'intersection en cours d'examen ne peut atteindre le seuil minimal du support.

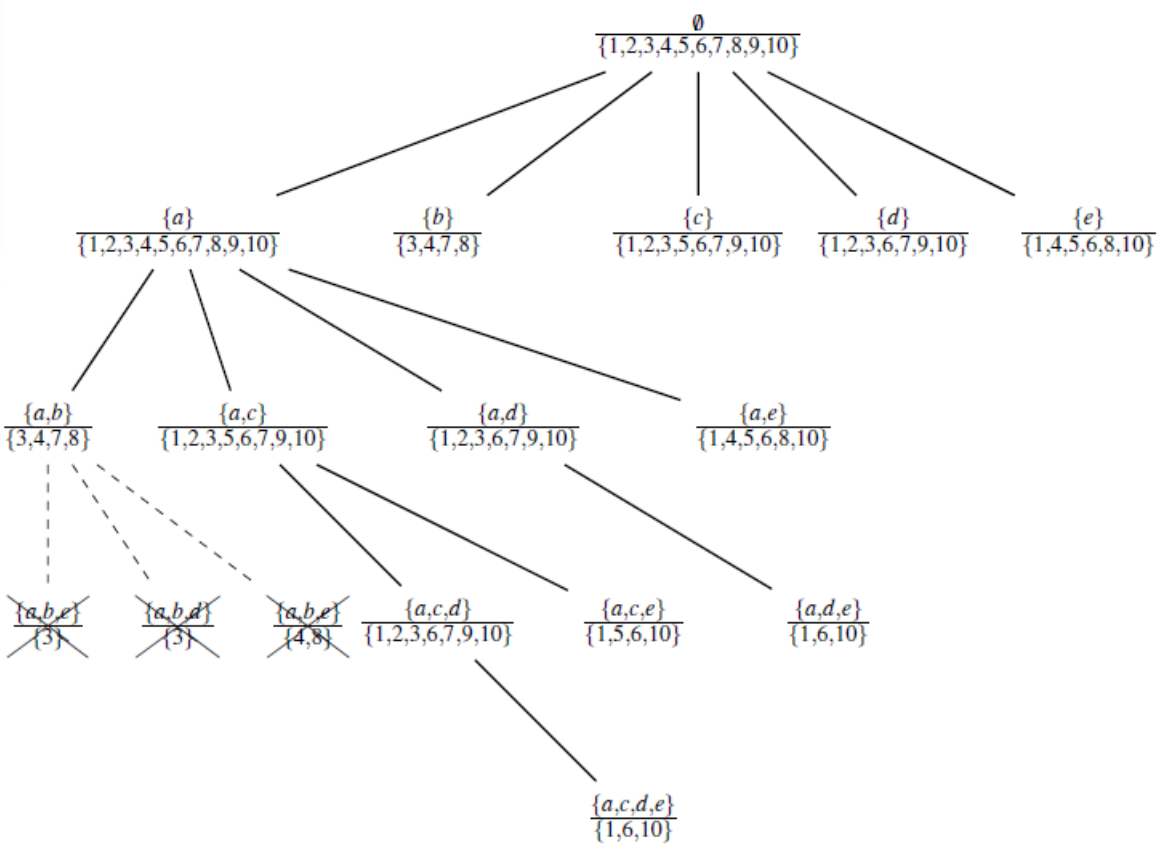


FIGURE 1.5: Eclat partiellement déroulé pour l'exemple de référence sur la classe d'équivalence de a. Les tidlists résultants des différentes intersections sont en parties basses des noeuds.

Pour toutes ces raisons, il a été trouvé qu'Eclat se comporte très bien devant des contextes de fouille pour des bases de données sparses qui impliquent souvent des intersections légères. Cependant, dans le cas des bases denses les résultats intermédiaires générés par Eclat deviennent énormes où il soufre alors de manque de

mémoire. Sa complexité est de l'ordre de $O(2|A||D|)$ au pire des cas, car l'intersection de deux tidlists est bornée par la taille de la base $|D|$.

1.3.2.3 Algorithme dEclat

Motivé par l'espace mémoire énorme qu'exige Eclat, notamment dans les cas de fouille complexes (bases de transactions denses/seuil de support trop faible), une version améliorée a été proposée appelée dEclat pour (diffset Eclat) [129]. Ce dernier algorithme part de l'idée qu'au lieu de considérer les listes complètes de transactions, il suffit de suivre leurs changements (différences) par rapport au parent en cours de traitement, ce qui permet de réaliser des gains importants en mémoire. Formellement, le diffset d'un motif $X = x_1....x_{k-1}x_k$, noté $d(X)$, est la différence entre les transactions supportant son préfixe et celles supportant son dernier item, i.e, $d(X) = tidlist(x_1...x_{k-1})\ tidlist(x_k)$. Cette définition conduit au fait que le diffset de deux motifs $X \cup Y$ partageant le même préfixe est la différence entre leur diffsets : $d(X \cup Y) = d(Y) - d(X)$, et que le support d'un motif X est la différence entre le support de son préfixe et le cardinal de son diffset. Ainsi dans notre exemple, le support du motif $\{a, b\}$ est :

$$sprt(\{a, b\}) = sprt(a) - |d(\{a, b\})| = 10 - |\{1, 2, 5, 6, 9, 10\}| = 10 - 6 = 4 \quad (1.10)$$

puisque $d(\{a, b\}) = d(\{b\}) - d(\{a\}) = \{1, 2, 5, 6, 9, 10\}/\emptyset = \{1, 2, 5, 6, 9, 10\}$.

1.3.3 Approches projectives

Afin de réduire la taille de la base de données transactionnelles, parer à ses multiplesbalayages, et atténuer l'effet de génération de candidats, Han et al. ont proposé l'algorithme FPGrowth [61]. Ce dernier exploite une structure de données compacte appelée FPTree (F requent Pattern tree) en guise de représentation de la base de données, qui est une sorte d'arbre de préfixes ou trie [88]augmenté par les informations de support. Ainsi, chaque nœud de l'arbre contient un item et un entier qui représente le support du motif formé des items trouvés sur le chemin depuis la racine à ce nœud. Reposant sur le principe diviser pour régner, FPGrowth adopte une exploration récursive en profondeur de l'espace de recherche via des extensions de suffixes (dans l'ordre inverse) et nécessite seulement deux passes sur l'ensemble de données. Durant la première passe, l'ensemble des items fréquents

sont identifiés (il constitue le nouveau ensemble d'items de la base) puis triés par ordre décroissant de leurs supports, et ceux infréquents sont tout simplement supprimés. La mission dans la deuxième passe est de construire l'arbre FPTree (qui contient initialement la racine modélisant le motif nul) de manière analogue à l'insertion de chaînes dans un trie, mais en respectant l'ordre décroissant des supports. En lisant la base ligne par ligne, chaque transaction est d'abord réécrite selon le nouveau ensemble d'items retenus et l'ordre retenu sur celui-ci puis insérée dans l'arbre. Les transactions partageant le même préfixes sont alors juxtaposés (les supports des noeuds concernés sont incrémenté à chaque nouvelle occurrence) ;de nouveaux noeuds sont créés pour les items restants (première occurrence) de la transaction où les supports associés sont initialisés à 1.

L'idée élégante de la structure compacte FPTree représente un pas remarquable pour cette classe de méthodes. En effet, l'heuristique de réécriture des transactions de la base selon les items fréquents par ordre décroissant du support a permis de réduire significativement la taille de la base à fouiller. D'une part, cette structure inclut uniquement des items fréquents (propriété Apriori), où les transactions sont alors tronquées des items infréquents. D'une autre part, les items les plus fréquents, c'est-à-dire, communs à plusieurs transactions, sont regroupés dans les niveaux supérieurs de l'arbre, ce qui produit un gain en espace en réalisant un taux important de compressions de données. Par ailleurs, en vue d'accélérer le processus de fouille, une table entête avec des pointeurs intra-noeuds sont ajoutés à l'arbre FPTree. Cette table maintient une liste pointée des items (fréquents) de la base. À chaque case correspond un pointeur, qui la lie au noeud le plus à gauche du même item dans l'arbre. Tous les noeuds de l'arbre portants le même item sont aussi connectés par des pointeurs pour faciliter leur accessibilité. En se référant à la table entête et commençant des items moins fréquents, FPGrowth réalise une succession de projections de la base de données sur le suffixe en cours d'extension de façon à produire une base conditionnelle (base conditionnée par l'item en question). La base résultante de la projection avec le suffixe considéré forment un nouveau contexte sur le quel FPGrowth, dont le pseudo code est montré dans l'algorithme 5, travaille de la même façon (reconstruction d'un nouveau FPTree, avec tri (sur le même critère) des items qui les composent et élimination des infréquents. Ce schéma de traitement est effectué dans une récursion. Le cas de base de cette récursion correspond à une base conditionnelle simple où l'arbre obtenu est vide ou forme un chemin simple. Dans ce cas, tous ses sous motifs possibles sont trouvés et les extensions associés au suffixe en cours sont reportées.

Un exemple de construction de l'FPTree de la base de l'exemple de référence est illustré ci-dessous dans la figure 2.6. De même, il facile de voir que la base conditionnelle de b est acd(2),ae(2) ; son FPTree conditionnel est montré dans la figure 1.7. L'application de l'algorithme sur cette dernière délivre seulement l'item $\{a\}$ avec un support égal à 4. Ce qui en résulte que le seul motif fréquent du suffixe $\{b\}$ est $\{a, b\}$.

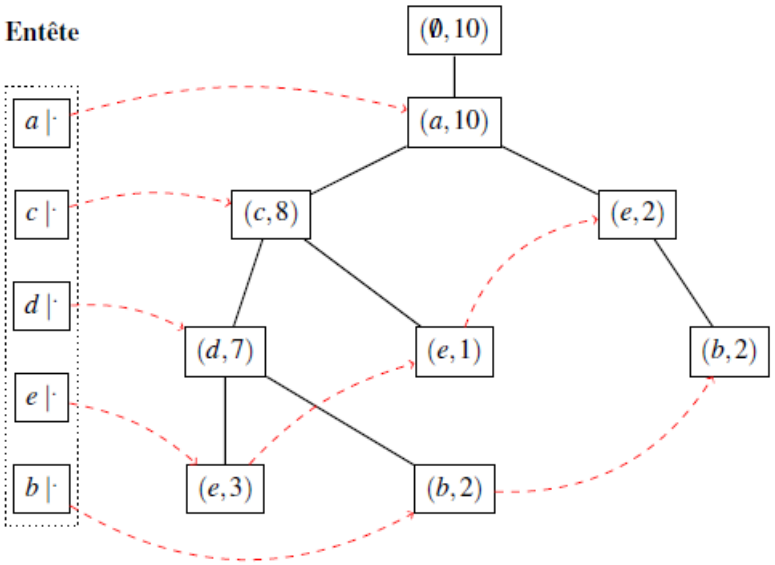


FIGURE 1.6: FPTree représentant la base de l'exemple de référence.

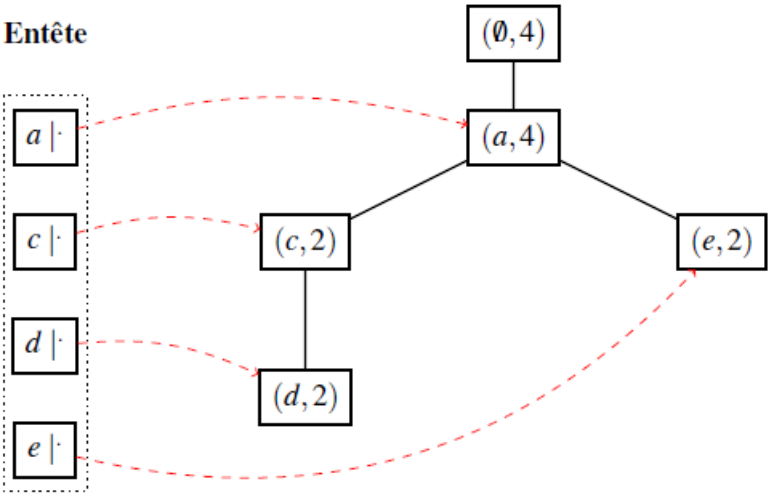


FIGURE 1.7: FPTree conditionnel de l'item b.

1.3.3.1 Analyse de FPGrowth

La représentation de données par FPTree adoptée dans FPGrowth bien qu'elle a permis de réaliser des économies notables sur la taille des données, elle reste juste une heuristique assez bonne mais qui ne garantit pas toutefois la minimalité (il a été trouvé dans plusieurs exemples de bases de transaction que leur représentations horizontales sont inférieurs à celles en utilisant FPTree). Certains chercheurs ont même considéré la structure FPTree comme une représentation horizontale compacte et d'autre comme une variante de la représentation verticale [20]. Ceci a été illustré, tout en début du chapitre, dans la figure 1.2
En dépit des gains réalisés en adoptant la structure compressée FPTree, l'ensemble du traitement dans cette approche reste équivalent à celui des algorithmes verticaux et par niveaux. De plus, les auteurs de FPGrowth avancent que leur algorithme n'effectue aucune génération de candidats, ce qui parait peu vrai [37, 52], puisque le processus récursif de projection est équivalent à une sélection de candidats qui est validé ensuite par l'algorithme. Plusieurs travaux conséquents ont constatés que la première version de l'algorithme FPGrowth souffre, hélas, de surcoûts considérables devant, en particulier, des bases de données sparses et très volumineuses [58, 99, 114]. En fait, dans ces cas l'arbre FPTree devient énorme, et l'espace nécessaire pour garantir les niveaux profonds de la récursion représente un grand défi. De plus, les multiples reconstructions des différents FPTree et les opérations onéreuses de tri induisent des charges additionnelles sur l'algorithme. La complexité asymptotique de l'algorithme demeure, quant à elle, toujours de l'ordre de $O(2|A||D|)$ au pire, bien qu'en pratique cette borne est loin d'être atteinte, car les arbres construits sont en général, comme expliqué en haut, nettement inférieurs à la base.

1.3.3.2 Améliorations à FPGrowth

Plusieurs travaux ont suivi la philosophie de FPGrowth en y incorporant des extensions remarquables [58, 102]. Les brèches par lesquelles a été attaqué FPGrowth sont, en premier, la structure FPTree qui peut être optimisée d'avantage, et deuxièmement son inconvénient dû aux nombreuses opérations couteuses de reconstruction et re-tri qu'il exige. Ainsi, dans (Sucahyo and Gopalan, 2004) les auteurs présentent un algorithme similaire à FPGrowth mais cette fois-ci avec un arbre dit CPFTree pour Compressed FPTree dont la taille peut être réduit à la moitie

Algorithm 3 FPGROWTH(FPT ; s ;P)

Entrée : FPT l'arbre compact d'une base de données transactionnelle D, un seuil de support s et le suffixe courant P
Sortie : L'ensemble F des motifs fréquents
si FPT est un chemin simple alors
pour tout combinaison c de noeuds de FPT faire
Écrire $(c \bigcup P)$
fin pour
sinon
pour tout item a dans FPT selon l'ordre croissant du support faire
$Q = \{a\} \bigcup P$
$F \leftarrow F \bigcup Q$
Temp $\leftarrow \emptyset$
pour tout chemin d depuis la racine à a faire
$d_1 \leftarrow d$ tronqué de a
insérer d_1 dans l'arbre Temp
fin pour
si Temp$\neq \emptyset$alors
FPGROWTH(Temp ; s ;Q)
fin si
fin pour
fin s

de celle d'un FPTree, sur lequel travaille un algorithme semblable à FPGrowth mais itératif cette fois-ci, dont la confrontation avec Apriori et FPGrowth a montré une nette supériorité. Une implémentation efficace de l'algorithme FPGrowth qui élimine les lourdes opérations de reconstruction a été présentée dans (Rácz, 2004).

1.3.4 Approches hybrides

À chaque algorithme ses avantages et ses inconvénients et un one-fits-all n'existe pas toujours. Par conséquent, des solutions hybrides ont été proposées dans le but d'éliminer les défauts des approches introduites et tirer profit, au même temps, de leurs avantages. Comme nous l'avons vu, AprioriHybrid [8] est la première hybridation proposée pour la fouille de motifs fréquents. Elle combine Apriori et AprioriTid. Dans [67] un algorithme hybride est conçu et analysé ; il démarre avec Apriori pour ensuite commuter à Eclat. Récemment, une solution hybride associant l'approche projective représentée par FPGrowth avec l'approche verticale a été suggérée dans [34]. Les auteurs introduisent l'algorithme Prepost+ qui étend la

structure FPTree en ajoutant des étiquettes aux noeuds de l'arbre. Cette extension offre des intersections efficaces pour le calcul des supports. L'évaluation de cet algorithme a montré des performances extraordinaires en ce qui concerne le temps de réponse.

1.4 Énumération abrégée

Le nombre de motifs fréquents dans une base de transaction est, comme nous l'avons mentionné, exponentiel avec le nombre d'items. Ainsi, les reporter tous via une énumération exhaustive est une tâche inextricable notamment pour les contextes de fouille complexes (bases denses et très volumineuses et/ou seuil de support trop bas). Ceci a vivement incité les chercheurs à prospecter des solutions permettant de réduire la complexité de cette tâche. Comme la compression de données est un sujet mature en technologie de l'information, l'appliquer à ce problème constitue une alternative prometteuse. Aussi, plusieurs représentations compactes pour les motifs fréquents qui permettent de les restaurer tous ont été proposées dans la littérature. Ces représentations condensées outre qu'elles permettent de rationner les ressources de calcul et de stockage nécessaires en éliminant les redondances présentes, elles offrent une aisance dans l'analyse des connaissances (règles) découvertes à partir de ces motifs. Dans cette section et afin d'adoucir le manuscrit, nous nous limiterons seulement aux deux représentations compactes les plus courantes à savoir les motifs fréquents maximaux et les motifs fréquents fermés ; pour une étude plus approfondie des représentations compactes des motifs fréquents veuillez vous référez aux références-survey cités en début du chapitre ou les papiers dédiés suivants : [25, 89]. Afin d'apprécier le gain obtenu en considérant ce type de représentation, la figure. 1.8 anticipe et montre, pour une base de donnée transactionnelle commune (voir le chapitre sur l'expérimentation), les rapports entre le nombre de motifs fréquents, fréquents fermés et fréquents maximaux.

1.4.1 Motifs fréquents maximaux (MFM)

Un motif x est dit fréquent maximal dans une base D par rapport à un seuil de support s si et seulement si x est fréquent et il n'existe pas un sur-motif de x qui est aussi fréquent autrement dit, tous ses sur-motifs sont infréquents. Une définition formelle est donnée ci-dessous.

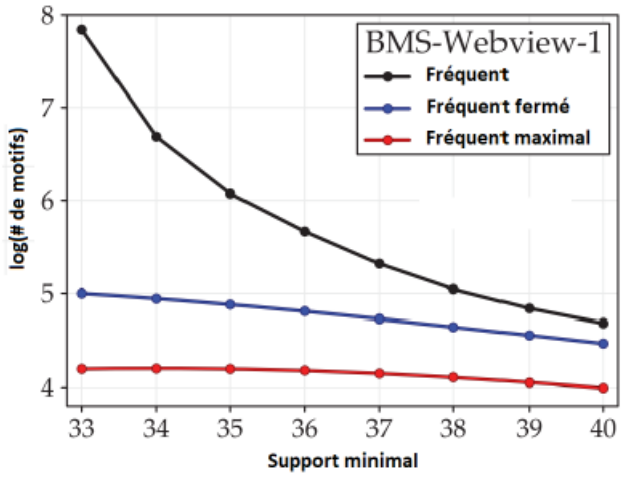


FIGURE 1.8: Nombre (en échelle logarithmique) de motifs fréquents, fréquents fermés et fréquents maximaux pour la base de données BMS-Webview-1 (Borgelt, 2012)

Définition(Motif fréquent maximal). Dans une base D et un seuil de support s, un motif x est fréquent maximal ssi x est fréquent et $\nexists y / y \supset x \wedge sprt(y) \geqslant s$
Partant de la propriété Apriori stipulant que tout sous-motif d'un motif fréquent est lui même fréquent, il suffit donc de connaitre les motifs fréquents maximaux (MFM) d'un ensemble de données. Ce dernier permet de dériver le reste en considérant leurs sous-motifs sans référence à la base. Notons que l'ensemble de MFM, notéM, correspond exactement à la bordure positive définie dans 3.2.2, qui une fois identifiée tout ce qui est situé en dessous d'elle est forcement fréquent. L'ensemble de MFM bien qu'il offre une réduction importante de l'espace du problème, il ne préserve pas cependant l'information relative au supports des motifs fréquents trouvés, ce qui requière un balayage additionnel de la base pour renseigner les supports associés aux sous-motifs fréquents qu'on pourra en dérivés. Les MFM, illustrés dans la figure 2.9, pour notre exemple de référence sont : $M = \{\{a, b\}, \{a, c, d, e\}\}$.
En principe, tous les algorithmes d'extraction totale de motifs fréquents sont adaptables à l'extraction de ceux fréquents maximaux. L'idée naïve est d'ajouter un test de maximalité après chaque motif fréquent trouvé de façon à ne retenir que les maximaux entre eux. Ainsi, si un motif fréquent extrait est un sous-motif d'un autre déjà marqué maximal il sera ignoré ; de même, s'il représente un sur-motif d'un autre classé maximal, ce dernier sera supprimé de la sortie. Nombreux sont les algorithmes introduits pour l'extraction de MFM. Ils sont fondés sur l'approche de base précédente à laquelle ils ajoutent d'autre optimisations afin d'accélérer le processus de fouille. Ci-après une sélection non intégrale d'algorithmes de cette

catégorie : Max- Miner [17], Pincer-Search [82], Depth Project [2], MAFIA [24], FPMax [54], LCM-Max[120], GenMax [129], Charm-MFI [115], etc. Loin d'une présentation exhaustive, nous décrivons brièvement, dans ce qui suit, trois algorithmes représentatifs (un pour chaque approche) de découverte de MFM.

MaxMiner

Cet algorithme [17] suit la même philosophie d'Apriori en adoptant un parcours en largeur de l'espace de recherche représenté par l'arbre d'énumération d'ensemble défini par Rymon [109], qui l'implémente aussi par un arbre de hachage. Les points forts de MaxMiner sont les optimisations qu'il introduit (fréquence de sur-motifs/infréquence de sous-motifs) et un mécanisme d'anticipation (lookahead) permettant de découvrir rapidement des motifs fréquents plus longs. Pour ce faire, l'auteur associe à chaque noeud correspondant à un groupe candidat deux motifs : le premier est la tête du groupe noté h(g) qui symbolise le motif que code le noeud et t(g) sa queue qui est un ensemble ordonné d'items pouvant constituer une extension possible de la tête du groupe. L'examen d'un noeud correspond au traitement du groupe candidat codé par celui ci, qui implique le calcul des supports pour les motifs suivants : $h(g), h(g) \bigcup t(g), eth(g) \bigcup \{i\}$ pour tout $i \in t(g)$.
Dans MaxMiner, l'optimisation fondée sur la fréquence de sur-motifs est implémenté en ignorant l'extension de tout groupe pour lequel $h(g) \bigcup t(g)$, qui représente le plus long sur-motif de celui du noeud, est fréquent puisque dans ce cas tous les descendants de ce groupe sont évidemment fréquents mais non maximaux. De plus, si le résultats de l'extension fréquente $h(g) \bigcup t(g)$ représente un sous-motif d'un autre connu déjà comme tel, ce candidat sera éliminé. D'un autre coté, l'optimisation fondée sur l'infréquence de sous-motifs est implémentée pour tous les items i où le motif $h(g) \bigcup \{i\}$ s'avère infréquent. Ces branches sont tout simplement abandonnées, car leurs expansions futures seront certainement infréquentes. Enfin, MaxMiner propose deux autres techniques permettant une optimisation plus fine. D'une part, les descendants d'un noeud sont triés par ordre croissant de leurs supports afin de permettre une efficacité de la première optimisation basée sur la fréquence des sur-motifs, car les items les plus fréquents se trouvent dans la majorité des noeuds ce qui accroit la chance de bénéficier de cette optimisation. Une borne inférieure sur le support d'un motif en exploitant les informations déjà disponibles sur les supports de ses sousmotifs permet, d'autre part, d'estimer quand est-ce qu' il serait fréquent avant même d'accéder à la base, offrant ainsi la possibilité de limiter le nombre de candidats à examiner.

GenMax

GenMax [129] est un algorithme récursif de découverte de MFM qui utilise le format de données vertical et un parcours en DFS de l'espace du problème. Cet algorithme génère exactement la collection de MFM et ne nécessite aucun post-traitement ; il profite de certaines optimisations connues déjà à l'instar de celles utilisées dans MaxMiner et MAFIA. Il définit, similairement à MaxMiner, pour chaque motif x son combine-set composé des items pouvant participer à une extension possible à x, qu'il ordonne selon l'ordre croissant des supports. Cette astuce, permet un élagage très-tôt de l'espace de recherche car les motifs moins fréquents ont peu de chance de former des combine-set plus larges dans le niveau suivant. En commençant par un ensembleM de MFM vide, à chaque étape l'union du motif en cours avec son combine-set est examiné. S'ils sont contenus dans un motif maximal, la branche toute entière sera alors élaguée ; sinon, GenMax passe à la formation de nouveaux motifs candidats et calcule leurs supports. Si l'ensemble des extensions possibles est non vide, un appel récursif est lancé pour trouver d'autres extensions plus longues. Le cas contraire correspondant à un motif fréquent maximal probable ; ce dernier n'est inséré dansM que s'il ne constitue pas un sous-motif d'un autre déjà dans M. De plus, GenMax utilise d'autres heuristiques pour réduire davantage son temps de réponse tels que : la propagation des diffsets introduit dans dEclat afin d'accélérer l'estimation des supports, et la «progressive focusing» consistant à diminuer le nombre des tests d'appartenance inutiles entre motifs en se limitant uniquement à un sous ensemble réduit deM, dit MFM locaux, et en fin, le maintient d'un indicateur de suivi de changements dans l'ensembleM pour signaler si ce dernier a été ou non modifié de manière à annuler toutes les vérifications d'inclusion dans le cas où M n'a pas changé.

FPMax

FPMax [54] est une extension, pour l'extraction de motifs fréquents maximaux, de l'algorithme FPGrowth duquel il emprunte plusieurs aspects. D'un coté, il est récursif et opère un parcours en DFS de l'espace du problème. Il utilise, d'un autre coté, en addition de l'arbre FPTree introduit par FPGrowth, une structure semblable dite MFI-Tree (Maximal Frequent Itemset Tree) dans les tests de maximalité et le suivi de la collection des MFM trouvés. L'idée sur laquelle repose FPMax est l'observation que si un motif fréquent n'est pas un sous-motif d'aucun des MFM existants, il est alors un nouveau MFM. Ainsi, cet algorithme crée l'arbre MFITree de façon similaire à l'arbre FPTree, qui inclut donc une table entête (ordonnée de la même manière) et des pointeurs de liens internes, mais sans les informations relatives aux supports. Une fois la structure initialisée, le

processus d'extraction est entamé par FPMax durant lequel les cas de bases des récursions de FPGrowth sont exploités pour insérer le motif fréquent trouvé (qui représente un chemin simple de l'FPTree) auquel est ajouté l'item en cours si bien évidement le motif fréquent en question est absent dans la structure MFI-Tree. A la fin du traitement, l'MFI-Tree contient la collection des MFM de la base de données

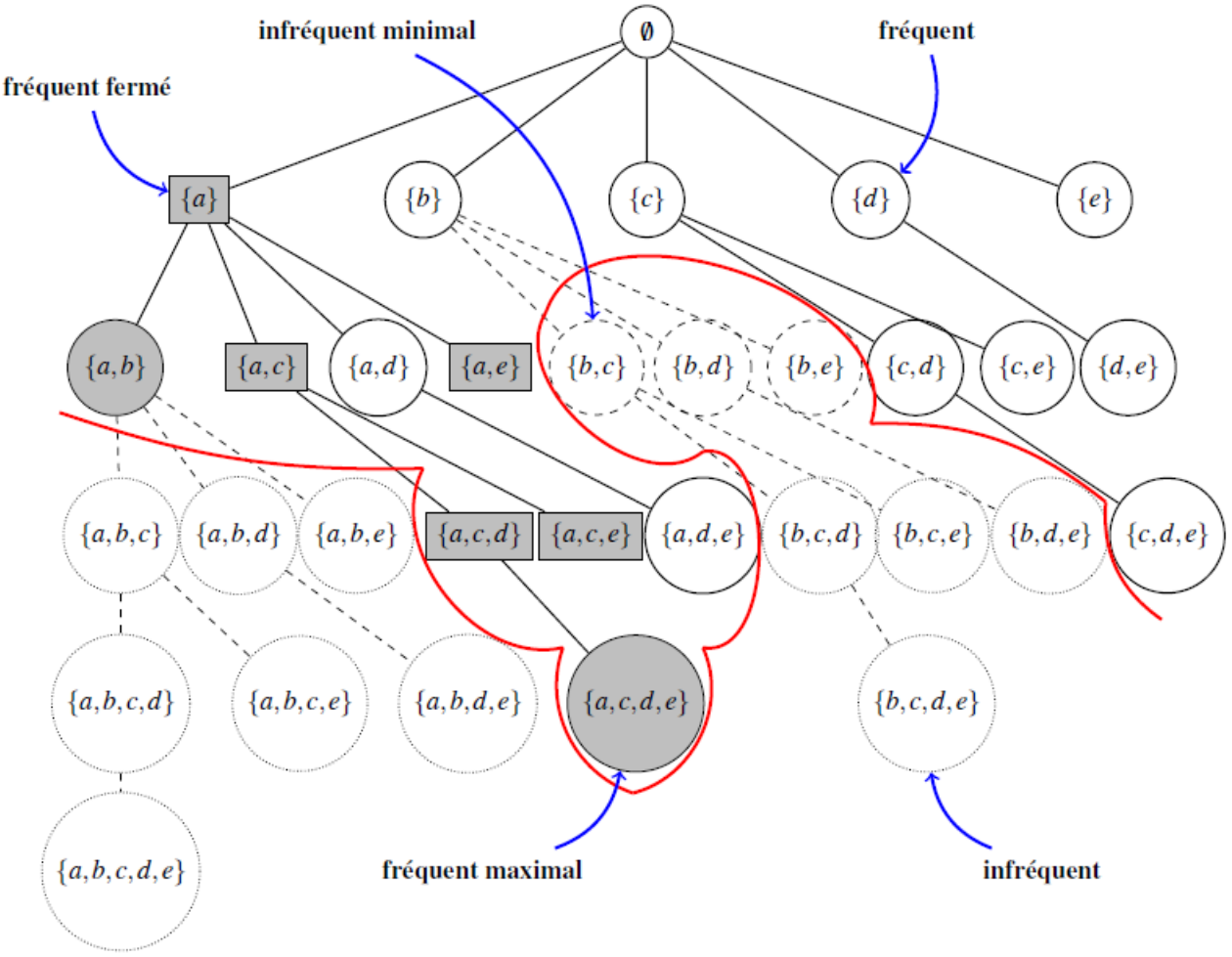


FIGURE 1.9: Représentations condensées de la base de l'exemple de référence : motifs fréquents maximaux (2 noeuds gris en cercles) et motifs fréquents fermés (7 noeuds gris)

1.4.2 Motifs fréquents fermés (MFF)

Un motif x est dit fréquent fermé dans une base D par rapport à un seuil de support s ssi x est fréquent et il n'existe pas un sur-motif de x qui a le même

support. Intuitivement, un motif fermé est un ensemble maximal d'items commun à un ensemble de transactions

Définition Motif fréquent fermé). Dans une base D et un seuil de support s, un motif x est fréquent fermé ssi x est fréquent et $\nexists y / y \supset x \wedge sprt(y) sprt(x)$

Formellement, un motif x est fermé par rapport à un opérateur γ si $x = \gamma(x)$.Cette notion de fermeture a été empruntée à la fouille de motifs premièrement mais séparément par (Pasquier et al., 1998) en France et (Zaki and Ogihara, 1998) aux USA. Les auteurs ont soulignés le lien entre la fouille de motifs fréquents et l'analyse formelle de concepts (AFC) (Barbut and Monjardet, 1970 ; Wille, 1982 ; Davey and Priestley, 1990), dans lequel les transactions représentent les objets, les items sont les attributs et la base de données joue le rôle du contexte d'analyse (la relation binaire entre les deux) noté par le triplet (T,A,D). Dans les papiers de Pasquier et al. et M.Zaki et al., il a été montré que la fouille de motifs fréquents peut être réduite à la construction seulement du treillis des motifs fréquents fermés dit treillis des concepts ou de Galois dans l'AFC. Ce dernier treillis est un sous ordre du treillis complet des sous ensembles de A, souvent de taille moins réduite. L'opérateur γ est ici la composition des deux applications $f : 2^T \rightarrow 2^A$ et $g : 2^A \rightarrow 2^T$ suivantes qui lient les transactions aux items et inversement, c'est-à-dire les fonctions définies comme suit où $X \subseteq A$et $Y \subseteq T$:

$$f(Y) = \{a \in A / \forall y \in Y : y \; contient \; a\} \tag{1.11}$$

$$f(T) = \{t \in T / \forall A \in X : t \; contient \; a\} \tag{1.12}$$

Le couple de fonctions (f, g) est une connexion de Galois où $\gamma = f \circ g (resp. \gamma = g \circ f)$ représente l'opérateur de fermeture (vérifie les conditions de fermeture vues en section 1.1) pour 2 A (resp. 2T) ; il associe des motifs à eux même (resp. des transactions à elles-mêmes). La fermeture d'un motif x correspond donc àl'unique sur-motif maximal ayant le même support que x. La collection de MFF constitue donc une autre représentation compressée pour les motifs fréquents d'un ensemble de données. Cette représentation est complète et sans perte d'information (Pasquier et al., 1998), puisque sa connaissance permet de dériver le reste des motifs fréquents avec les valeurs exactes de leurs supports à l'opposé de l'ensemble de MFM qui, comme déjà mentionné, ne préserve pas les supports. Il est aisé de constater que tout motif fréquent maximal est aussi fréquent fermé (l'inverse n'est pas vrai), ce qui signifie que l'ensemble des MFM M est inclut dans l'ensemble des MFF C , qui est à son tour inclut dans l'ensemble de tous les

motifs fréquents F ($M \subseteq C \subseteq F$).Le support d'un motif x est celui de sa fermeture : $sprt(x) = maxsprt(y)|x \subseteq y \in C$.Notons enfin que sur le plan théorique, il a été prouvé l'équivalence des problèmes suivants : la recherche de rectangles maximaux d'une relation binaire, l'énumération de toutes les cliques bipartites maximales d'un graphe et l'identification de motifs fermés d'une base transactionnelle [82, 132]. Pour la base de notre exemple de référence, il existe sept MFF $C = \{\{a\}, \{a, b\}, \{a, c\}, \{a, e\}, \{a, c, d\}, \{a, c, e\}, \{a, c, d, e\}\}$ qui sont montrés dans la figure 1.9 (notons la présence des deux MFM). Dans la suite, nous décrivons brièvement trois algorithmes représentatifs d'extraction de MFF

A-Close

A-Close [98] est un algorithme horizontal itératif qui étend Apriori pour découvrir les MFF. En exploitant les propriétés de fermeture, A-close identifie, niveau par niveau, d'abord l'ensemble de motifs générateurs qui permettent de dériver tous les MFF de la base par application de l'opérateur de fermeture. Un motif g est un générateur du motif fermé c s'il est le motif minimal (selon l'inclusion) dont la fermeture est c, autrement dit, le motif ayant le même support que c. À l'étape k , les k-générateurs candidats sont construits à partir des $(k-1)$-générateurs fréquents en examinant les supports calculés par balayage de la base. Tout générateur infréquent ou non minimal est alors supprimé. Par la suite, un ultime scan est effectué afin de calculer les fermetures des générateurs trouvés qui constituent la collections de MFF contenus dans la base.

Charm

Charm (Closed Association Rule Mining, avec un H sans correspondance) [130] est un algorithme de la famille des méthodes verticales initiées avec Eclat. Il utilise donc, d'une part, des paires motifs-listes de transactions associées et décompose l'espace de recherche, représenté par un arbre, en plusieurs classes d'équivalence qui sont traitées séparément. D'autre part, il exploite l'astuce des diffsets proposée dans dEclat pour les calculs des supports. Étant donné que la découverte de MFF requière des tests de fermeture, cet algorithme introduit plusieurs mécanismes d'élagage permettant de sauter des niveaux complets de l'espace du problème et ainsi de générer moins de candidats à vérifier. Pour ce faire, Charm examine pour chaque cas les tidlists impliquées dans l'extension d'un candidat en cours et distingue trois cas de figures impliquant les deux paires $(X_i, tidlist(X_i))$ et $(X_i, tidlist(X_i))$ selon que leurs listes de transactions sont égales, l'une contenue dans l'autres ou différentes : (i) le cas où $tidlist(X_i)) = tidlist(X_j)$, qui conduit aussi à l'égalité des fermetures de X_i, X_j et celle de leur union $X_i \cup X_j$Pour ce cas

toute occurrence de Xi est remplacée par $X_i \cup X_j$et la branche entière de X_j est élaguée, (ii) le cas où $tidlist(X_i) \subset tidlist(X_j)$ (resp. $tidlist(X_j) \subset tidlist(X_i)$) qui mène à l'inégalité des fermetures des deux motifs, mais à l'égalité de celle de X_i (resp.X_j) avec la fermeture de l'union $Xi \cup Xj$ qui implique que toute occurrence de X_i (resp. X_j) est remplacée par $X_i \cup X_j$tout en gardant cette fois la branche de X_i (resp. X_j), (iii) ce dernier cas est rencontré lorsque les deux listes de transactions sont différentes qui n'offre aucune optimisation. En opérant ainsi, il est prouvé que Charm produit correctement la liste des motifs fréquents fermés. En effet, il aisé de constater que cet algorithme explore son espace et que les branches éliminées sont celles correspondantes à des motifs non fréquents, ou fréquents mais non fermés.

LCM

LCM (Linear time Closed itemset Miner) [120]est un algorithme efficace de fouille de motifs fréquents. Cet algorithme n'appartient pas, comme on peut le croire de par sa position, à l'approche projective ; mais il fait partie plutôt des algorithmes hybrides (LCM est considéré à la fois comme un algorithme vertical et projectif). Nous l'avons présenté ici en raison de sa popularité, étant donné qu'il a remporté la deuxième compétition du workshop organisé en 2004 [17]. LCM est récusrsif et fondé sur une relation entre les motifs fréquents fermés tirée du domaine des cliques bipartites. Son efficacité est due à de simples structures de données exploitée de façon intelligente qu'il associe à plusieurs techniques pour améliorer le temps de calcul. L'espace du problème étant un arbre de préfixe parcouru ici en DFS, et ne nécessite aucun stockage des résultats précédents trouvés. Par ailleurs, LCM utilise des tableaux comme structures de données pour représenter les transactions et les items qu'il ordonne afin d'accélérer les recherche. Il bénéficie également de deux autres idées : d'un coté, l'occurrence deliver consistant à anticiper le calcul de tous les successeurs du motif en cours durant le même scan, et la technique des diffsets proposée dans dEclat. Vu que ces deux dernières heuristiques sont contradictoires, LCM les combine dans un schéma hybride de façon à trouver un compromis selon que la base à fouiller est dense ou sparse. Notons en fin, que cet algorithme propose aussi deux adaptations LCMfreq et LCMmax permettant, respectivement, l'extraction de tous les motifs fréquents et ceux fréquents maximaux.

1.5 Énumération incrémentale

Cette classe de méthodes concerne les projets de fouille pour les base de données dynamiques. Il est évident que les connaissances extraites deviennent caducs si l'ensemble de données est évolutif objet de mises à jour (insertion, suppression ou modification), ou si les paramètres du problèmes sont ajustés. Dans ces cas de figure, l'approche naïve consistant à relancer le processus de fouille à partir de zéro est manifestement impraticable. Il serait, par contre, judicieux de maintenir les résultats obtenus par l'exploitation des connaissances extraites actuelles en les confrontant seulement avec les fragments altérés de la base. Dans cette catégorie, plusieurs algorithmes ont été proposés. Pour citer des exemples, nous mentionnons les tous premiers travaux de Chen et al.[27, 28], suivis par l'approche qui utilise les treillis de Galois de [95], puis l'algorithme efficace ZIGZAG [121] qui s'attaque à l'extraction et la maintenance des motifs fréquents maximaux.

1.6 5 Aspects avancés

La fouille de motifs fréquents est un sujet très vaste qui a suscité d'innombrables contributions et il est hors de question de pouvoir le cerner par un rapport si limité. Dans cette dernière section, nous soulignons sommairement quelques aspects que nous pouvons qualifiés d'avancés dans la mesure où ils traitent des extensions au problème standard de la fouille de motifs.

1.6.1 Retour sur l'intérêt de motifs/règles

La totalité des solutions discutées dans cette thèse concernent le framework de fouille de motifs classique basé sur les mesures support et confiance. Il a été noté, dans plusieurs études, que se limiter aux règles fortes vérifiant les deux seuils support et confiance, s'il permet de générer certes des connaissances dignes d'intérêt, peut souvent conduire à l'omission d'autres connaissances utiles ; liées à un support plus bas par exemple. De plus, le réglage insouciant de ses paramètres est critique et peu produire des résultats étranges. En conclusion, le choix d'une configuration de fouille est fortement tributaire au type de données et au domaine d'application. Par conséquent, d'autre mesures pour quantifier l'importance d'un motif ou d'une

règle ont été proposés. Certains ont pour but de renforcer le framework de base par des mesures de corrélation [59, 60], d'autres suggèrent de nouveaux frameworks plus objectifs[113].

1.6.2 Motifs et fouilles complexes

Un motif dans le framework classique est simplement un ensemble d'items. Or dans plusieurs applications réelles, nous avons en face des motifs non triviaux comme les catégories de produits dans les grands magasins, des structures (données multi-dimensionnelles, sections de documents, pages web, réseaux sociaux, données biologiques, etc.) qui constituent des terrains fertiles à explorer. Il est donc naturel d'envisager l'extraction de ce genre de motifs plus élaborés. Ainsi, le même framework a été vite étendu à, par exemple, la fouille de séquences [9], d'arbres [130], de graphes [70], et enfin la fouille de motifs quantifiés, c'est-à-dire des bases ayant des quantités associées aux items dans les transaction [122].Par ailleurs, certains contextes imposent, pour le framework classique ou ses extensions, des conditions de travail trop contraignantes. Il s'agit, à titre d'exemple, de fouille en situation de mémoire limitée, ou la présence de données en ligne, temps réel, ou en flots. Ces situations nécessitent une prise en charge poussée, étant donné que ces contextes ne tolèrent ni repassage sur les données, ni leurs sauvegarde ou temporisation [61].

1.6.3 Parallélisation et distribution

Il a été observé très tôt [97] que la complexité du problème de la fouille de motifs fréquents dépasse les schéma de traitements séquentiels et les architectures conventionnelles. Dès lors, le recours aux paradigmes de traitements parallèles et/ou distribués constitue légitimement une alternative pour faire face à l'espace énorme impliquant des données massives et les traitements intensifs qu'engendre le problème étudié. Ce courant de recherche a récemment été stimulé par l'apparition d'architectures et modèles de calcul plus adéquats.Outre les premiers travaux des équipes de Park et Agrawal [7, 97], plusieurs autres travaux se sont intéressés à l'extraction parallèle/distribuée de motifs fréquents. Par exemple, FPF [79] est un des algorithmes célèbres de fouille parallèle de motifs fréquents ; il étend l'algorithme FPGrowth en se basant sur le modèle MapReduce [32]. Sur les nouveaux processeurs disposant du multicore, PLCMQS [92] est une extension parallèle de

l'algorithme LCM. Un algorithme distribué et parallèle pour la fouille incrémentale de motifs fréquents destiné aux bases de données dynamiques est présenté dans [94]. Ce dernier parallélise l'algorithme ZIGZAG [121] de fouille incrémentale de motifs fréquents.

1.6.4 Méta-heuristiques pour l'extraction de règles d'association

Pour clore ce chapitre, nous avons jugé opportun de mentionner un autre courant de recherche sur le sujet, qui vise plus particulièrement la phase de découverte de règles d'association en exploitant les méta-heuristiques [93]. Ces dernières, sont des classes de méthodes générales et approximatives conçues pour résoudre des problèmes complexes d'optimisation, où les techniques d'optimisation classiques ont échoué. Les méta-heuristiques tirent leur origines de différents phénomènes observés dans plusieurs domaines tels que : l'évolution en biologie, la dynamique de certain systèmes physiques, la propagation de la chaleur en métallurgie, la mécanique statistique, etc. Ces approches ont connu un succès grandissant ces dernières années et un large étendu d'applications ; elles incluent, entre autres, : le recuit simulé, la recherche tabou, les réseaux de neurones, les algorithmes évolutionnaires, la programmation génétique, les colonies de fourmis, les optimisations par essaims de particules, et les approches hybrides [116], etc. Le principe étant d'explorer efficacement (en temps raisonnable) l'espace de recherche, sans pour autant songer à l'énumération totale de l'ensemble de solutions, afin de trouver des solutions quasiment optimales. En assimilant la question à un problème d'optimisation combinatoire, la stratégie est de démarrer avec une(des) solution(s) réalisable(s), selon que l'approche est à base de solution unique ou de population d'entre elles, puis de progresser selon une(des) fonctions objectif(s) à optimiser (maximiser/minimiser). Le recours à ces méthodes pour l'extraction des règles d'association est justifié par la nature combinatoire du problème et sa complexité exponentielle [38]. Suite aux extensions introduites au problème standard, où les règles d'association quantitatives et floues ont été d'abord explorées [47, 77], plusieurs métaheuristiques ont été appliquées à l'extraction des règles d'association. Celles-ci introduisent des mesures autre que le support et la confiance (compréhension, intérêt, etc.) et les combinent dans des optimisations mono ou multi-objectifs. C'est ainsi que les

algorithmes génétiques ont été utilisés dans [86] et des métaheuristiques multi-objectifs exploitées par [50] et [73]. Des stratégies évolutionnaires multi-objectifs basées sur des populations BSO (Bee Swarm optimization) sont aussi proposées par[41] ; récemment, un algorithme de type Bat est introduit dans [64]. Pour finir, le papier[74] propose une d'hybridation entre ces méthodes optimisations évolutionnaires et celles exactes pour la découverte de règles d'association intéressantes.

1.7 Approches parallèles et distribuées de fouille de motifs fréquents

1.7.1 Introduction

Comme nous l'avons déjà mentionné, les algorithmes d'extraction de motifs fréquents existants se sont révélés très efficaces sur des ensembles de données typiques, mais très exigeants en ressources lorsque la taille de l'ensemble des données d'entrée augmente[79]. En général, l'application des techniques d'extraction de motifs fréquents sur des données massives a souvent nécessité de faire face à des coûts de calcul qui représentent un goulot d'étranglement critique. Pour cette raison, au cours des dernières années, de nombreux algorithmes d'extraction de données distribuées ont été développés et largement exploités dans différents domaines d'application (par exemple, soins de santé[12], données biologiques[30], données énergétiques[26], images[13]). Dans cette section, nous présenterons les raisons du besoin d'algorithmes d'extraction d'itemset fréquents évolutifs, et la migration actuelle vers les frameworks de calcul distribué (par exemple, Apache Hadoop[21], Apache Spark[128]). Enfin, cette section décrira en détail les principaux problèmes liés à la distribution des algorithmes d'extraction des motifs fréquents et la façon dont les meilleures approches distribués les ont traités.

1.7.2 Motivation

Plusieurs algorithmes d'extraction de données traditionnels ont été proposés (détaillés à la section 1.3), et ont montré leurs efficacités lorsque les ensembles de données peuvent être complètement chargé dans la mémoire principale. Cependant, ils ne peuvent pas faire face à des besoins plus grands et plus importants. Pour cette

raison, au cours des dernières années, différentes approches distribuées ont été introduites, capables d'effectuer l'extraction des ensembles d'éléments même dans les cas liés à un très grand nombre de données ou Big Data.. En fait, Hadoop et Spark ont été largement adoptés dans le milieu de la recherche[63, 72, 124], grâce à leurs gestion souple des données et à leurs meilleure tolérance aux pannes[104]. Sans oublier que le besoin d'approche distribuée de fouille de données est motivé par des facteurs différents :

Taille des données d'entrée : Le premier facteur , évidemment, est la quantité de données à traiter. Ce problème est fortement lié aux structures de données (par exemple, FP tree [99], Enumération tree[30], Prefix tree [132], ...) utilisées par les algorithmes afin d'explorer l'espace de recherche. En général, des ensembles de données plus importants entraînent des structures de données plus complexes, qui nécessitent une plus grande quantité de ressources informatiques et de mémoire pour être explorées et maintenues. Par conséquent, les approches traditionnelles trouvent de grande difficultés dans le traitement de très grands ensembles de données, Big Data.

Seuil minimum du support : La deuxième question est liée au seuil minimal du support, qui reflète directement la profondeur de l'analyse. Même pour des données simples, l'extraction des motifs fréquents avec un support très faible pourrait nécessiter une quantité énorme de ressources. Plus il est faible, plus l'extraction sera difficile en termes de ressources. les motivations ainsi sont liées à la structure interne utilisée par les algorithmes, afin d'explorer l'espace de recherche[52]. Un seuil minimum de support faible, conduit ainsi à une exploration plus approfondie de l'espace de recherche. Le cas extrême est celui des algorithmes avec générations des candidats, qui pourraient avoir un nombre des données de sortie plus grand que la taille des données d'entrée. De plus, veuillez noter que la complexité des structures algorithmiques n'évoluent pas de façon linéaire par rapport au seuil minimal du support[20]. Pour ces raisons, ce paramètre est très important afin d'évaluer la performance d'un algorithme d'extraction d'itemset fréquent. L'effet inverse de l'extraction de règles d'association et d'itemset à faible support est la très grande quantité de règles d'itemset générées. Dans ces cas, les chercheurs ont déployé de nombreux efforts pour réduire automatiquement le nombre de règles générés, avec l'introduction de différents indices d'intérêt et de mesures statistiques[49]. Cependant, la plupart de ces mesures pourraient être appliquées après l'extraction des motifs fréquents . En outre, seuls quelques-uns garantissent la propriété de fermeture vers le bas/vers le haut, ce qui leur permet de remplacer l'élagage basé sur

le support pour réduire l'espace de recherche[22]. En conclusion, il existe encore un besoin évident de techniques efficaces et pertinentes capables d'extraire des ensembles d'éléments a faible support , avec une grande quantité de données Big Data.

Distribution des données : Comme nous le verrons aux chapitres suivants, la distribution des données d'entrée a également un impact sur l'espace de recherche. Une longueur élevée d'une transaction, influe fortement sur la complexité de l'espace de recherche et, par conséquent, sur la structure des données à explorer et à maintenir[52]. Un ensemble de données dense tend à produire un ensemble d'éléments plus fréquent parce qu'un plus grand nombre d'éléments par transaction entraîne par nature un plus grand nombre de cooccurrences. Il est clair qu'une grande quantité de données d'entrée à traiter, n'est qu'un des facteurs affectant l'extraction des motifs fréquents , qui est fortement influencée également par la profondeur de l'analyse (c'est-à-dire le seuil minimum du support) et la distribution des données.

1.7.3 Stratégies de parallélisation de la fouille des motifs fréquents

La parallélisation des structures de données représente la contribution principale derrière le développement d'algorithmes d'extractions d'éléments fréquents distribués et parallèles. Dans les domaines distribués et parallèles, une approche idéale suppose de diviser le problème en sous-problèmes indépendants qui ne se chevauchent pas, et qui peuvent être affectés à des nœuds de clusters [65]. De cette façon, (i) les ressources sont complètement exploitées et (ii) les coûts de communication, qui présentent un goulet d'étranglement concret dans un environnement distribué, sont réduits autant que possible. Dans l'environnement de la fouille des motifs fréquents, la principale tâche à paralléliser est l'exploration de l'espace de recherche, qui s'effectue au moyen de structures de données. Les algorithmes distribués divisent et distribuent intelligemment le traitement de ces structures de données, la plupart d'entre elles adoptent le mode "diviser pour régner". Cette technique permet de surmonter les principaux problèmes de mémoire. Les meilleurs algorithmes basés sur MapReduce ont relevé les défis liés à l'extraction parallèle des motifs fréquents au moyen de deux approches algorithmiques principales : Ils sont significativement différents parce que (i) ils utilisent des solutions différentes

pour diviser le problème original en sous-problèmes et (ii) ils se basent sur différentes hypothèses concernant les données qui peuvent être enregistrées dans la mémoire principale de chaque tâche indépendante.

Approche de partitionnement des données. Il divise le problème en sous-problèmes similaires, exécutant la même fonction sur différents blocs de données. Plus précisément, chaque sous-problème calcule les supports locaux de tous les éléments candidats sur un morceau du jeu de données d'entrée (c'est-à-dire que chaque sous-problème fonctionne sur l'espace de recherche complet mais sur un sous-ensemble des données d'entrée). Enfin, les résultats au niveau local (c.-à-d. les supports locaux des ensembles d'éléments candidats) émis par chaque sous-problème/tâche sont fusionnés pour calculer le résultat final global (support global pour chaque ensemble d'éléments). Les principales hypothèses de cette approche sont que (i) le problème peut être divisé en sous-problèmes similaires travaillant sur différents morceaux des données d'entrée et (ii) l'ensemble des éléments candidats est suffisamment petit pour pouvoir être stocké dans la mémoire principale de chaque tâche.

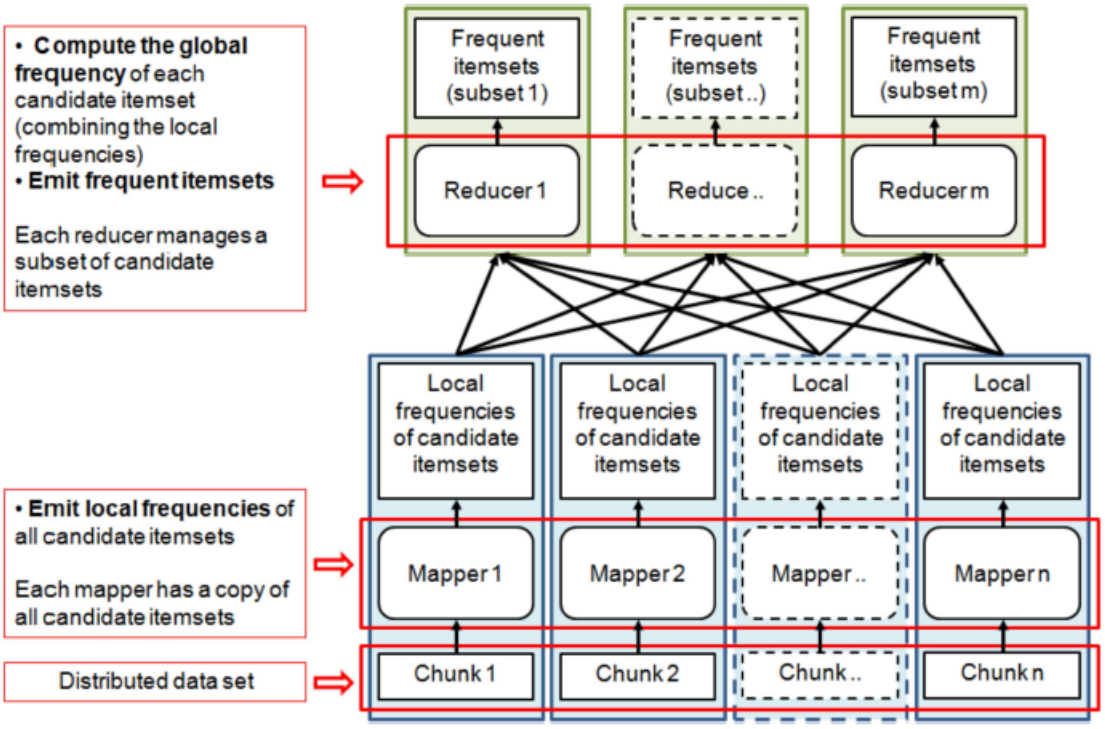


FIGURE 1.10: Approche de partitionnement des données

Approche par division de l'espace de recherche Il divise le problème en assignant à chaque sous-problème la consultation d'un sous-ensemble de l'espace

de recherche (c'est-à-dire que chaque sous-problème se rapporte à une partie du réseau). Plus précisément, cette approche génère, à partir de l'ensemble de données distribuées d'entrée, un ensemble de données projetées, suffisamment petit pour être stocké dans la mémoire principale d'une tâche unique. Chaque ensemble de données projeté contient toute l'information nécessaire pour extraire un sous-ensemble d'éléments (c.-à-d. que chaque ensemble de données contient toute l'information nécessaire pour explorer une partie du réseau) sans avoir besoin de la contribution des résultats des autres tâches. Le résultat final est l'union des sous-ensembles d'éléments extraits de chaque ensemble de données projeté.

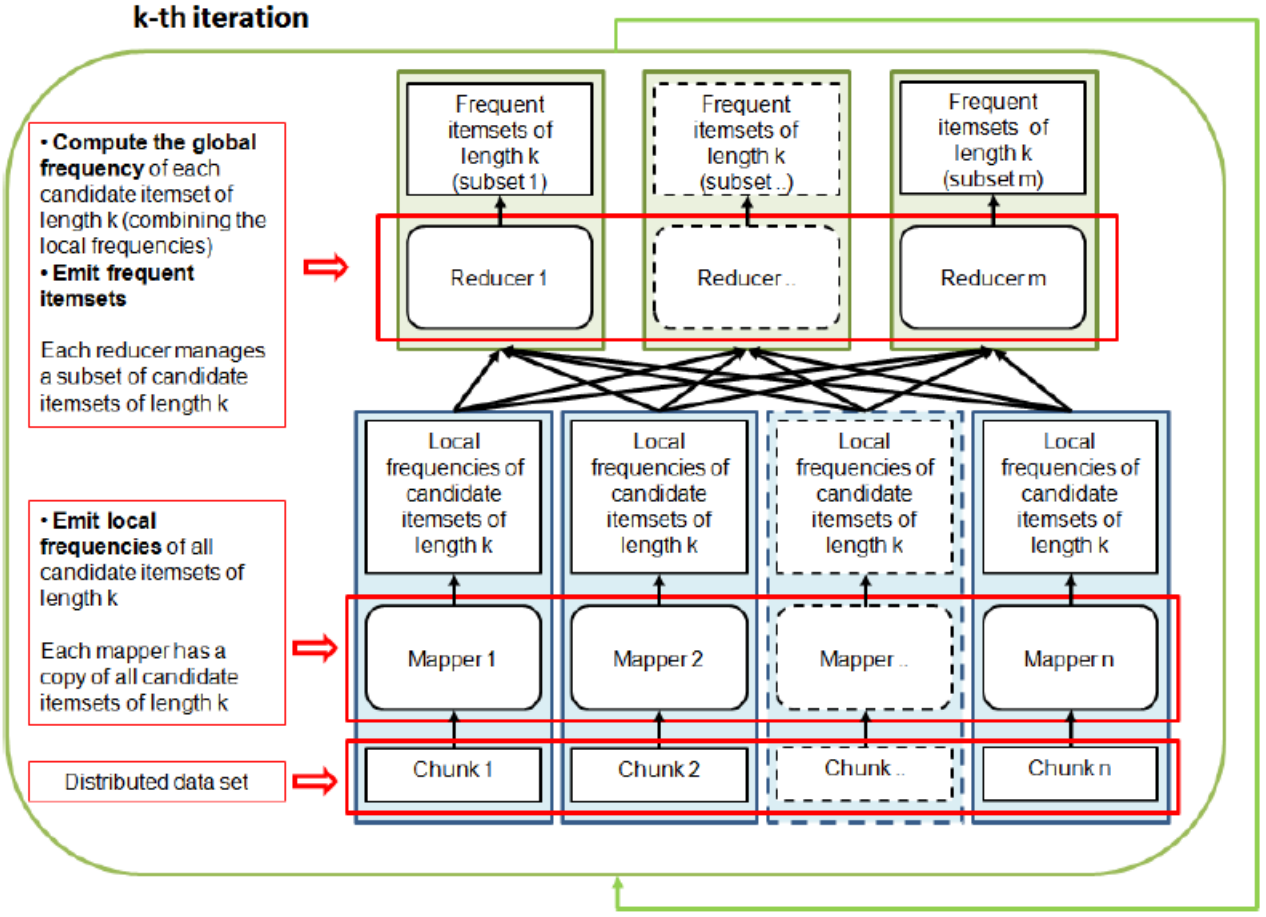


FIGURE 1.11: Itérations de partitionnement des données

Les figures 1.10 et 1.12 illustrent respectivement les première et deuxième stratégies de parallélisation. Dans l'approche du partitionnement des données (figure 1.1O), la phase Map calcule les supports locaux des ensembles d'éléments candidats dans son bloc de données (c'est-à-dire que chaque mapper exécute une extraction locale de jeux de données sur son bloc de données). Ensuite, la phase

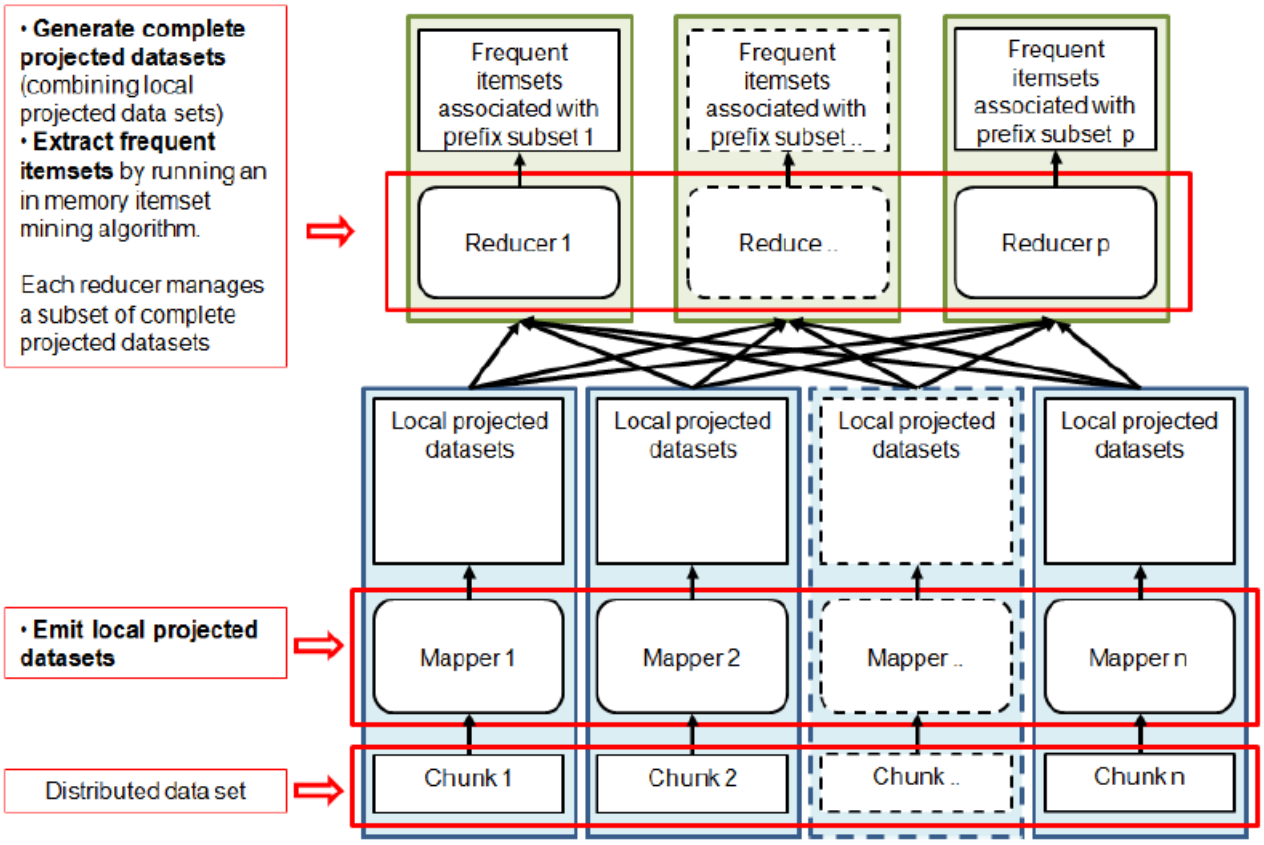


FIGURE 1.12: Approche par division de l'espace de recherche

de réduction fusionne les supports locaux de chaque itemset candidat pour calculer son support global. Cette solution exige que chaque mappeur garde une copie de l'ensemble complet des éléments candidats (c.-à-d. une copie du treillis). Cet ensemble doit être stocké dans la mémoire principale de chaque mappeur. Étant donné que l'ensemble complet des éléments candidats est généralement trop important pour être stocké dans la mémoire principale d'un seul mappeur, une solution itérative, inspirée des algorithmes centralisés par niveau, est utilisée. La figure 3.2 présente la solution itérative. A chaque itération k, seul le sous-ensemble de candidats de longueur k est pris en compte et donc stocké dans la mémoire principale de chaque mappeur. Cette approche, grâce aussi à l'exploitation du principe de l'algorithme Aapriori pour réduire la taille des ensembles candidats, permet d'obtenir des sous-ensembles d'éléments candidats qui peuvent être chargés dans la mémoire principale de chaque mappeur. Dans l'approche du partitionnement de l'espace de recherche (figure 3.3), l'idée principale est de créer plusieurs ensembles de données projetés indépendants qui peuvent être exploités en parallèle dans différentes tâches. La phase Map est donc utilisée pour diviser ou mapper l'ensemble

de données initial en plusieurs sous-échantillons qui, s'ils sont intelligemment agrégés, pourraient être explorés et exploités indépendamment. Précisément, à partir des données locales de l'ensemble de données initial, les mappeurs génèrent un ensemble de données locales projetées. Chaque ensemble de données locales projetées est la projection du bloc d'entrée par rapport à un préfixe p. Ensuite, la phase de réduction fusionne les ensembles de données locales projetées pour générer les ensembles de données projetées complets (les transactions relatives au même préfixe p, peuvent être contenues dans différents blocs de données). Les ensembles de données projetées agrégées sont fournis comme des données d'entrée par rapport aux réducteurs. Chaque réducteur exécute ensuite un algorithme standard centralisé d'extraction sur les ensembles de données projetés, en extrayant l'ensemble relatif d'éléments fréquents. Par conséquent, l'hypothèse principale, dans cette approche, est que chaque ensemble complet de données projetées doit occuper la mémoire principale d'un seul réducteur. Le tableau 3.1 résume les principales caractéristiques des deux approches de parallélisation par rapport aux critères suivants : type de répartition du problème, utilisation de la mémoire principale, coûts de communication, équilibrage de charge et parallélisation maximale (c.-à-d. nombre maximal de mapper et de réducteurs).

Type de fractionnement de l'espace de recherche La principale différence entre les deux approches de parallélisation est la stratégie adoptée pour diviser le problème en sous-problèmes, ce choix ayant un impact significatif sur les autres critères[14]. L'utilisation différente de la mémoire principale des tâches a un impact sur la fiabilité des deux approches. l'approche de fractionnement des données suppose que les éléments candidats de longueur k peuvent être stockés dans la mémoire principale de chaque mappeur. Par conséquent, il n'est pas en mesure de mettre à l'échelle des ensembles de données denses caractérisés par de grands ensembles de candidats. En d'autres termes, l'approche du fractionnement de l'espace de recherche suppose que chaque jeu de données projeté complet peut être stocké dans la mémoire principale d'une seule tâche. Par conséquent, cette approche ne dispose plus de mémoire lorsque de grands ensembles de données projetés complets sont générés.

Coûts de communication. Dans un algorithme parallèle MapReduce, les coûts de communication sont importants, car le réseau peut facilement devenir le goulot d'étranglement si de grandes quantités de données y sont envoyées. Les coûts de communication sont principalement liés aux sorties des mappeurs qui sont envoyées aux réducteurs sur le réseau. Pour l'approche du fractionnement des données, les

Critère	Approche itérative de fractionnement des données	Approche par répartition de l'espace de recherche
Type de fractionnement de l'espace de recherche	Chaque sous-problème analyse un sous-ensemble différent des données d'entrée et calcule les supports locaux de tous les items candidats de longueur k sur ses blocs de données. Le résultat final est donné par la fusion des résultats locaux.	Chaque sous-problème analyse un sous-ensemble différent d'éléments/une partie différente de l'espace de recherche. Le résultat final est le l'union des résultats locaux.
Utilisation de la mémoire principale	L'ensemble candidat de longueur k est stocké dans la mémoire principale d'une tâche unique.	L'ensemble des données projetées est stocké dans la mémoire principale d'une seule tâche.
Cout de communication	Nombre d'éléments candidats x nombre de mappeurs x nombre d'itérations.	Somme des tailles des ensembles de données locales projetées.
Equilibrage de charge	L'équilibrage de charge est réalisé en associant le même nombre d'éléments à chaque réducteur.	Les tâches pourraient être très déséquilibrées selon les caractéristiques des ensembles de données projetés assignés à chaque nœud.
Nombre maximum des mappeurs	Nombre de partitions	Nombre de partitions
Nombre maximum des réducteurs	Nombre des éléments candidats	Nombre des éléments

données envoyées sur le réseau sont linéaires en ce qui concerne le nombre d'éléments candidats, le nombre de mappeurs et le nombre d'itérations. Autrement dit, pour l'approche de l'espace de recherche, l'approche la quantité de données émises par les mappeurs est égale à la taille de la carte projetée ensembles de données.

Équilibrage de charge. La répartition différente du problème en sous-problèmes a un impact significatif sur l'équilibrage de charge. Pour l'approche du fractionnement des données, le temps d'exécution de chaque mappeur est linéaire par rapport au nombre de transactions d'entrée et le temps d'exécution de chaque réducteur est linéaire par rapport au nombre d'éléments attribués. Par conséquent, l'approche du fractionnement des données permet d'obtenir facilement un bon équilibrage de la charge en attribuant le même nombre de blocs de données à chaque Mapper, et le même nombre d'items candidats à chaque Réducteur. Différemment, l'espace de recherche est potentiellement déséquilibrée. En fait, chaque sous-problème est associé à un sous-ensemble différent du treillis, lié à un ensemble de données et à un

préfixe projeté spécifique, et, selon la distribution des données, la complexité des sous-problèmes peut varier. Une affectation intelligente d'un ensemble de sous-problèmes à chaque nœud permettrait d'atténuer le déséquilibre. Cependant, la complexité des sous-problèmes est difficilement décelable pendant la phase initiale de l'affectation. Les deux approches de parallélisation sont utilisées pour concevoir des systèmes efficaces d'implémentations parallèles d'algorithmes d'extraction centralisée bien connus. Plus précisément, l'approche du partitionnement des données est utilisée pour mettre en œuvre les versions parallèles des algorithmes de niveau (comme Apriori[9]), tandis que l'approche du partitionnement de l'espace de recherche est utilisée pour mettre en œuvre des versions parallèles des approches récursives en profondeur (comme FP-growth[52] et Eclat[53]).

1.7.4 Algorithmes distribués de fouille de motifs fréquents

Cette section décrit les algorithmes et les implémentations disponibles, représentant les solutions de pointe de l'état de l'art dans le contexte de l'extraction parallèle t distribues des motifs fréquents. Nous avons considéré les algorithmes suivants : YAFIM[101], PFP[79], BigFIM[91] et DistEclat[91]. YAFIM fait partie des algorithmes basés sur l'approche du fractionnement des données, tandis que PFP et DistEclat sont basés sur l'approche du fractionnement de l'espace de recherche. Enfin, BigFIM mélange les deux stratégies, visant à y exploiter les avantages de chacune . Pour le PFP, nous avons sélectionné deux implémentations populaires : Le PFP de Mahout et le PFP de MLlib, qui sont fondés sur Hadoop et Spark, respectivement. La description des quatres algorithmes sélectionnés et leurs mises en œuvre sont présentées dans les sous-sections suivantes.

YAFIM

YAFIM[69] est une implémentation distribuée d'Apriori développée en Spark. La nature itérative des algorithmes basés sur Mapreduce, a toujours représenté un défi pour leurs application dans le cadre du Big Data, a cause des frais généraux générés par le lancement de nouvelles tâches MapReduce et la nécessité de lire l'ensemble de données d'entrée à partir du disque à chaque itération. YAFIM exploite les RDD Spark pour faire face à ces problèmes. Précisément, il suppose que tous les jeux de données peuvent être chargés dans un RDD pour accélérer les opérations de comptage. Ainsi, après la première phase au cours de laquelle toutes les transactions sont chargées dans un RDD, YAFIM lance l'algorithme itératif Apriori qui organise les candidats dans un arbre de hachage pour accélérer la recherche.

Étant fortement basée sur Apriori, YAFIM a hérité une stratégie d'exploration et de partitionnement de l'espace de recherche et d'une préférence pour des distributions de données éparses. YAFIM exploite la fonctionnalité d'abstraction et de diffusion des variables sur Spark, qui permet aux programmeurs d'envoyer une seule fois des sous-ensembles de données partagées à chaque esclave , au lieu de les envoyer pour chaque tâche qui les utilise . Cette mise en œuvre réduit les coûts de communication (ce qui atténue le cout de communication entre les travaux), tandis que l'équilibrage de la charge n'est pas abordé.

Parallèle FP-growth

Parallèle FP-growth[79], appelé PFP, est une implémentation distribuée de FP-growth qui exploite le paradigme MapReduce pour extraire les k itemsets fermés les plus fréquents. Il est inclus dans la bibliothèque d'apprentissage machine, Mahout (version 0.9), et il est développé sur Apache Hadoop. Le PFP est fondé sur la stratégie de parallélisation de l'espace de recherche décrite à la section 1.2.2. Plus précisément, PFP est basé sur la construction d'arbres indépendants FP Tree (c.-à-d. des ensembles de données projetées) qui peuvent être traités séparément sur différents nœuds. L'algorithme se compose de 3 jobs MapReduce :

•Le Premier job, construit le tableau de la liste F, qui est utilisé pour sélectionner les éléments fréquents.

•Dans le second job, les mappers projettent par rapport à un groupe d'éléments (préfixes), toutes les transactions de l'ensemble de données d'entrée pour générer les contributions locales. Ensuite, les réducteurs regroupent les projections associées aux items d'un même groupe et construisent des arbres FP complets et indépendants à partir de ceux-ci. Chaque arbre FP est géré par un réducteur, qui exécute l'algorithme FP-growth dans la mémoire principale locale et extrait les éléments fréquents qui y sont associés.

•le dernier job MapReduce sélectionne les k items fermés les plus fréquents. Les arbres FP complets indépendants peuvent avoir des caractéristiques différentes et ce facteur a un impact sur le temps d'exécution des tâches d'extraction . Comme nous l'avons mentionné à la section 3.3, ce facteur a une incidence importante sur l'équilibrage de la charge. Plus précisément, lorsque les arbres FP ont des tailles et des caractéristiques différentes, les tâches ne sont pas équilibrées parce qu'elles abordent des sous-problèmes d'une complexité différente. Ce problème pourrait être résolu en divisant les arbres complexes en sous-arbres, chacun associé à un sous-problème indépendant du premier. Cependant, il n'est pas facile de définir

une métrique pour diviser un arbre de manière à obtenir des problèmes équivalents en termes de temps d'exécution. En fait, le temps d'exécution du processus d'exploration des itemset sur un arbre FP n'est pas seulement lié à sa taille (nombre de nœuds) mais aussi à d'autres caractéristiques (par exemple, nombre de branches et fréquence de chaque nœud). Selon les caractéristiques de l'ensemble de données, les coûts de communication peuvent être très élevés, en particulier lorsque les ensembles de données projetés se chevauchent de manière significative car, dans ce cas, la partie des données qui se chevauchent est envoyée plusieurs fois sur le réseau. Spark PFP[112] représente une transposition pure et simple de PFP sur Spark. Il est inclus dans MLlib, la bibliothèque d'apprentissage de la machine Spark. L'implémentation de l'algorithme PFP dans Spark est très proche de celle du frère Hadoop. La principale différence, en termes de problème traité, est que MLlib PFP exploite tous les ensembles d'articles fréquents, alors que Mahout PFP n'exploite que les k principaux ensembles d'articles fermés. Les deux implémentations, fortement inspirées de l'algorithme FP-growth , ils gardent de cet algorithme centralisé sous-jacent les caractéristiques liées à l'exploration de l'espace de recherche (profondeur d'abord) et la capacité d'extraire efficacement des ensembles de données denses.

DistEclat

DistEclat[91] est un algorithme d'extraction d'itemset fréquent basé sur Hadoop inspiré de l'algorithme Eclat, alors que BigFIM[91] est un algorithme mixte à deux phases qui combine une approche basée sur Apriori avec une approche basée sur Eclat. DistEclat est un algorithme d'itemset fréquent développé sur Apache Hadoop. Il exploite une version parallèle de l'algorithme Eclat pour extraire un super ensemble d'items fermés. L'algorithme se compose principalement de deux étapes. La première extrait des préfixes de taille k (c.-à-d. des ensembles fréquents d'éléments de longueur k) à l'égard desquels, dans la deuxième étape, l'algorithme construit des sous-arbres projetés indépendants, chacun étant associé à un sous problème indépendant. Même dans ce cas, l'idée principale est d'exploiter ces arbres indépendants dans différents nœuds, en appliquant l'approche de parallélisation par fractionnement de l'espace de recherche dont il est question à la section 3.3. L'algorithme est organisé en 3 tâches MapReduce :

•Dans le travail initial, un job MapReduce transpose l'ensemble de données en une représentation verticale.

•Deuxième job : Dans ce job MapReduce, chaque mappeur extrait un sous-ensemble des préfixes de taille k (items de taille k) en exécutant Eclat sur les items fréquents,

et les listes de tâches associées, qui lui sont affectées. Les préfixes de taille k et les listes de tâches associées sont ensuite divisés en groupes et assignés aux mappeurs de la dernière tâche.

•Troisième job : Chaque mappeur de la dernière tâche MapReduce exécute l'algorithme Eclat en mémoire principale sur son jeu de préfixes indépendants. L'ensemble finale des éléments fréquents est obtenu en fusionnant les sorties du dernier job. En remarque ainsi que l'extraction des ensembles d'éléments fréquents en deux étapes différentes , (c.-à-d. l'extraction des ensembles d'éléments de longueur k dans le deuxième job et l'extraction des autres ensembles d'éléments fréquents dans le dernier job) vise à améliorer la répartition de la charge de l'algorithme. En particulier, la division en deux étapes permet d'obtenir des sous-problèmes plus simples, qui sont potentiellement caractérisés par des temps d'exécution similaires, et des extractions globalement bien équilibrée. DistEclat est conçu pour être très rapide, mais il suppose que toutes les listes des éléments fréquents doivent être stockées dans la mémoire principale. Dans le pire des cas, chaque Mapper doit disposer d'un jeu de données complet, en format vertical, pour construire tous les 2-préfixes[68]. Cela a un impact négatif sur l'évolutivité de DistEclat par rapport à la taille du jeu de données. L'algorithme hérite de la version centralisée la stratégie de profondeur d'abord pour explorer l'espace de recherche et la préférence pour les ensembles de données denses.

BigFim

BigFIM [91]est une solution basée sur Hadoop très similaire à DistEclat. Tout comme DistEclat, BigFIM est organisé en deux étapes : (i) extraction des ensembles fréquents d'éléments de longueur inférieure ou égale au paramètre d'entrée k et (ii) exécution d'Eclat sur les sous-problèmes obtenus en divisant l'espace de recherche par rapport aux ensembles de k éléments. La différence réside dans la première étape, où BigFIM exploite un algorithme basé sur Apriori pour extraire les k-itemsets fréquents, c'est-à-dire qu'il adopte l'approche de la répartition des données (Section 3.3). Même si BigFIM est plus lent que DistEclat, BigFIM est conçu pour fonctionner sur de plus grands ensembles de données. La raison est liée à la première étape dans laquelle, en exploitant une approche basée sur Apriori, les préfixes k sont extraites en largeur tout d'abord. Par conséquent, les nœuds n'ont pas besoin de conserver de grandes listes d'éléments dans la mémoire principale, mais seulement l'ensemble des éléments candidats à compter. Toutefois, il s'agit également de la question la plus critique dans l'application de l'approche de parallélisation et de la répartition des données, car, selon la densité de l'ensemble de

données, l'ensemble d'éléments candidats peut ne pas être stocké en mémoire principale. En raison des deux techniques différentes utilisées par la BigFIM (division des données puis division de l'espace de recherche), l'algorithme BigFIM a obtenu les meilleures performances avec des ensembles de données eparses , tandis que dans un deuxième temps, il s'adapte mieux aux distributions de données denses. DistEclat et BigFIM sont les seuls algorithmes spécifiquement conçus pour pour traiter l'équilibrage de charge et le coût de communication au moyen du paramètre de longueur de préfixe k. En particulier, le choix de la longueur des préfixes générés pendant la première étape affecte à la fois l'équilibrage de charge et le coût de communication.

1.8 Conclusion

Dans ce chapitre nous avons dressé un panorama des travaux algorithmiques en fouilles de motifs fréquents. Nous avons vu que l'ensemble des travaux de la littérature sont classés dans trois principales approches : horizontale par niveau, verticales ou projectives.

Chapitre 2

Etude Comparative et Analyse Approfondie des Approches de l'état de l'art

Après l'analyse théorique, dans ce chapitre, nous présentons et commentons des comparaisons expérimentales et analyses approfondies entre toutes les approches introduites dans le Chapitre précédent. Cette étude est très utile pour évaluer la fiabilité des attentes liées à l'analyse théorique, ce qui nous a permis d'identifier les forces et les faiblesses des algorithmes concernant : les caractéristiques de l'ensemble de données d'entrée (distribution des données, longueur moyenne des transactions, nombre d'enregistrements. . .) et les réglages de quelques paramètres spécifiques (seuil du support, cout de communication. . .). Nous avons mené plusieurs expériences sur différents ensembles de données synthétiques et réelles, pour évaluer le temps d'exécution, l'équilibrage de charge et les coûts de communication de quatre approches de l'état de l'art dans le cadre des algorithmes parallèles et distribués d'extraction de motifs fréquents. Enfin, nous commentons et discutons les résultats, en introduisant des axes de recherche ouverts pour la parallélisation de la problématique de l'extraction de motifs fréquents à partir des données massives Big Data.

2.1 Expérience

L'évaluation expérimentale comprend les quatre algorithmes suivants décrits à la section 1.7.4 : l'algorithme Parallèle FP-Growth fournie dans Mahout 0.9 [57] (nommé Mahout PFP dans ce qui suit), l'algorithme Parallèle FP-Growth fourni dans MLlib pour Spark 1.3.0[112] (appelée MLlib PFP dans ce qui suit), l'algorithme BigFIM [91] et l'algorithme DistEclat [91].
Nous rappelons que Mahout PFP extrait les k items fermés les plus fréquents, BigFIM et DistEclat extraient un super ensemble des items fermés les plus fréquents, tandis que MLlib PFP extrait tous les items fréquents. Pour effectuer une comparaison équitable, Mahout PFP est obligé d'éditer tous les éléments fermés. Étant donné que l'extraction de l'ensemble complet des ensembles d'éléments fréquents est généralement plus gourmande en ressources que le traitement des éléments fermés fréquents, les temps d'exécution de Mahout PFP, BigFIM et DistEclat peuvent augmenter par rapport au MLlib PFP.
Nous avons défini un ensemble commun de valeurs de paramètres par défaut pour toutes les expériences. Des expériences spécifiques avec différents réglages sont explicitement indiquées. Le réglage par défaut de chaque algorithme a été choisi en tenant compte des caractéristiques physiques du cluster Hadoop, pour permettre à chaque approche d'exploiter au mieux la configuration hardware et software.
Pour Mahout PFP, la valeur par défaut de k est fixée à la valeur la plus basse. En particulier, avec des valeurs plus faibles du nombre de partitions, MLlib PFP ne peut s'adapter à des transactions très longues ou à des très faibles valeurs du minsup. En revanche, des valeurs plus élevées de partition n'entraînent pas une meilleure évolutivité, et affectent ainsi les performances. C'est pour ces raisons qu'on a pris des nombres de partition afin d'arriver au meilleur compromis entre la performance des algorithmes et leurs capacités, pour accomplir les tâches d'extraction sans problèmes flagrants de mémoire.
La valeur par défaut du paramètre de longueur du préfixe de BigFIM et de distEclat est définie sur 2, ce qui permet de trouver un bon compromis entre l'efficacité et l'évolutivité des deux approches. Nous n'avons pas défini de valeur par défaut de minsup, qui est un paramètre commun à tous les algorithmes, car il est fortement lié à la distribution des données et au cas d'utilisation. Ainsi cette valeur de paramètre est spécifiquement discutée dans chaque ensemble d'expériences.
Nous avons pris en compte à la fois des ensembles de données synthétiques et

réelles. Les données synthétiques ont été générées à l'aide du générateur de données IBM [4], fréquemment utilisé pour l'analyse comparative des performances d'algorithmes d'extraction des motifs fréquents. Nous avons ajusté les paramètres du générateur de données IBM (T = taille moyenne des transactions, P = longueur moyenne des motifs maximaux, I = nombre d'items différents, C = degré de corrélation entre les motifs, et D = nombre de transactions), pour analyser l'impact des différentes distributions de données sur la performance des différents algorithmes. La liste complète des ensembles de données synthétiques est présentée au tableau 2.1. Enfin, deux ensembles de données réelles ont été utilisés pour simuler des cas d'utilisation réels. Ils sont décrits à la section 4.1.8. Toutes les expériences, ont été

Tableau 2.1: Les propiétés des ensembles des données utilisées dans l'expérience

Type de Dataset	Nom	Nombre des différents Item	Nombre moyen des Items par transaction	Taille(GB)
Syntetique	Dataset 1	18000	10	0,5
Syntetique	Dataset 2	18000	20	1,2
Syntetique	Dataset 3	18011	30	1,8
Syntetique	Dataset 4	18017	10	6
Syntetique	Dataset 5	18017	10	30
Syntetique	Dataset 6	18017	10	60
Real life	Netlogs	160,900,000	15	0,6
Real life	Delicious	57,300,000	4	44

effectuées sur un cluster de 5 nœuds exécutant la distribution Cloudera d'Apache Hadoop (CDH5.3.1)[1]. Chaque nœud de cluster est un ordinateur Intel Core I5-3230M CPU 2,67 GHz, doté de 32 Go de mémoire principale et de disques durs SATA à 7 200 tr / min. La dimension de Yarn est réglée sur 6 Go. Cette valeur conduit à une exploitation complète des ressources de notre matériel, ce qui représente un bon compromis entre la quantité de mémoire attribuée à chaque tâche et le niveau de parallélisme. Des valeurs plus basses auraient augmenté le niveau de parallélisme au détriment de l'achèvement de la tâche, alors que des valeurs plus élevées auraient affecté le parallélisme, avec très peu de tâches réparties. D'un point de vue pratique, toutes les implémentations se sont révélées assez faciles à déployer et à utiliser. En réalité, la seule condition requise pour toutes les implémentations à exécuter était l'installation de Hadoop / Spark (d'un ordinateur à un grand cluster).

2.1.1 Impact du seuil de support « minsup »

Le seuil minimum de support (minsup) a un impact élevé sur la complexité de la tâche d'extraction des itemset. Afin d'éviter le biais dû à une distribution de données spécifiques, deux ensembles de données différents ont été considérés : Dataset 1 et Dataset 3 (tableau 4.1). Ils partagent la même longueur moyenne maximale des motifs (5), le nombre d'éléments différents (100 000), la valeur de corrélation entre les motifs (0,25) et le nombre de transactions (10 millions). La différence se situe au niveau de la longueur moyenne des transactions : 10 items pour Dataset 1 et 30 items pour Dataset 3. Étant donné que les autres caractéristiques sont constantes, les transactions plus longues entraînent une plus grande densité des ensembles de données, ce qui se traduit par un plus grand nombre d'ensembles d'éléments fréquents.

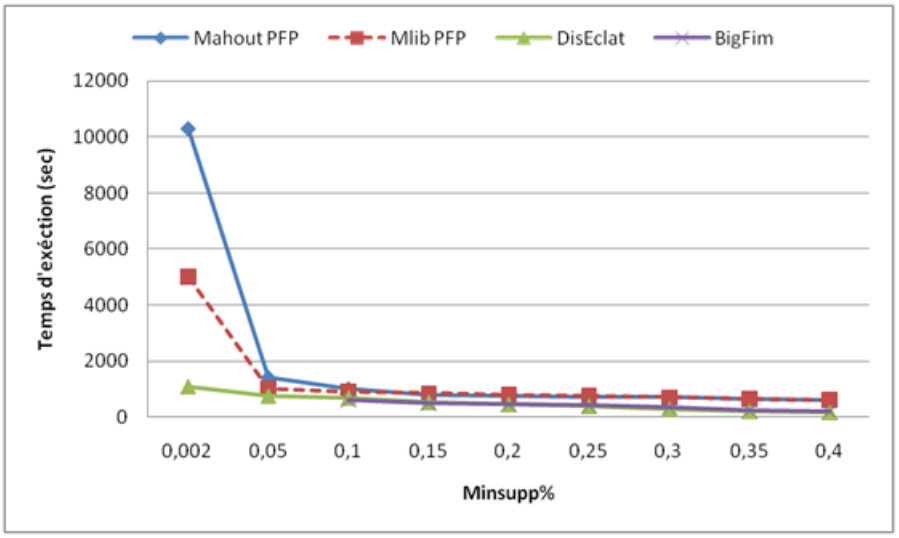


FIGURE 2.1: Temps d'exécution pour différentes valeurs du support (Dataset 1), durée moyenne des transactions 10.

La figure 2.1 indique le temps d'exécution des algorithmes lorsqu'on fait varier le seuil de Minsup de 0,002% à 0,4 sur l'ensemble de données Dataset 1. DistEclat est l'algorithme le plus rapide pour toutes les valeurs minsup considérées. Cependant, l'amélioration par rapport aux autres algorithmes dépend de la valeur de minsup. Lorsque minsup est supérieur ou égal à 0,2%, toutes les implémentations montrent des performances similaires. L'écart de performance augmente largement avec des valeurs minsup inférieures à 0,05. BigFIM est aussi rapide que DistEclat lorsque minsup est supérieur à 0,1%, mais en dessous de ce seuil BigFIM manque de mémoire pendant la phase extraction de 2-itemsets.

Dans la deuxième partie des expériences, nous avons analysé le temps d'exécution des algorithmes pour différentes valeurs minimales de support sur l'ensemble de données Dataset 3, qui est caractérisé par une longueur moyenne de transaction plus élevée (3 fois plus longue que l'ensemble de données Dataset 1), et une taille de données plus grande sur disque (4 fois supérieure), avec le même nombre de transactions (10 millions). Étant donné que la tâche d'extraction est plus intensive en calcul, les valeurs de minsup inférieures à 0,01% n'ont pas été prises en compte dans cet ensemble d'expériences, car cela s'est avéré être une limite pour la plupart des algorithmes en raison d'un épuisement de mémoire ou d'une durée expérimentale trop longue. Les résultats sont présentés à la figure 2.2.
L'algorithme MLlib PFP est beaucoup plus lent que Mahout PFP pour la plupart des valeurs minsup, tandis que BigFIM, comme dans l'expérience précédente, atteint des performances de haut niveau, mais ne peut évoluer vers des valeurs minsup faibles (le plus bas est 0,3%), en raison de contraintes de la mémoire pendant la phase de génération de k éléments. Enfin, DistEclat n'a pas pu fonctionner parce que la taille des listes initiales était déjà trop importante.

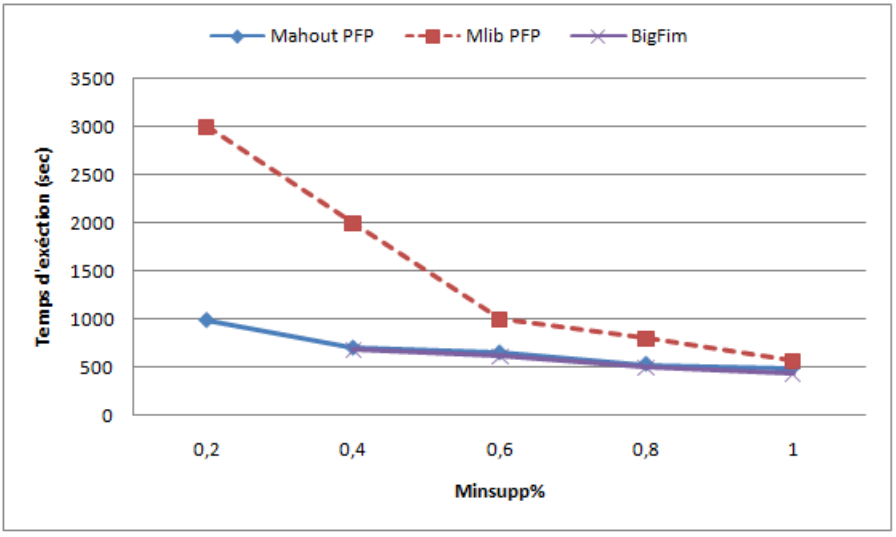


FIGURE 2.2: Temps d'exécution pour différentes valeurs du support (Dataset 3), durée moyenne des transactions 30.

Dans l'ensemble, comme prévu, DistEclat est l'approche la plus rapide lorsqu'il ne manque pas de mémoire. Mahout PFP est la mise en œuvre la plus fiable dans presque toutes les valeurs minsup, même si elle n'est pas toujours la plus rapide, et parfois avec de grands écarts derrière les autres algorithmes. MLlib PFP est un choix professionnel raisonnable, car il est constamment capable d'accomplir toutes les tâches dans un délai acceptable. Enfin, l'algorithme BigFim ne présente

pas d'avantages par rapport aux autres approches, car il n'est pas en mesure d'atteindre de faibles valeurs du support et de fournir des exécutions rapides.

2.1.2 Impact de la longueur moyenne des transactions

Nous avons analysé l'effet des différentes longueurs moyennes de transactions, de 10 à 100 éléments par transaction. Nous avons fixé le nombre de transactions à 10 millions. Pour ce faire, les ensembles de données 1 à 3 ont été utilisés (voir le tableau 2.1). Des transactions plus longues conduisent souvent à des ensembles de données plus denses, et à un plus grand nombre d'ensembles d'éléments longs et fréquents. Cela correspond généralement à des tâches de calcul plus intensives. Les temps d'exécution obtenus sont rapportés dans les figures 2.3 et 2.4, avec une valeur de minsup respective de 1% et 0,1%.
Dans l'expérience de la figure 2.3, les temps d'exécution BigFIM et DistEclat, pour des longueurs de transaction de 10 et 20, ne sont pas rapportés car, pour ces configurations, aucun ensemble de 3 éléments n'est extrait et donc les deux algorithmes se sont plantés. Pour des transactions de longueurs supérieures, DistEclat n'est pas inclus car il manque de mémoire pour les valeurs supérieures à 20 items par transaction. Les autres algorithmes ont des temps d'exécution similaires pour les transactions courtes, jusqu'à 30 articles.

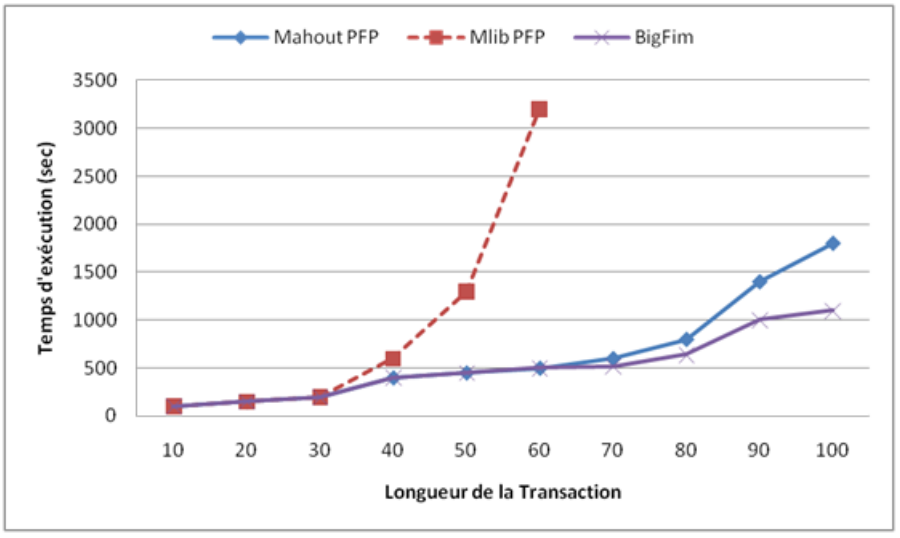


FIGURE 2.3: Temps d'exécution avec différentes longueurs moyennes de transactions (minsup 1%).

Pour les transactions de longueurs supérieures, une tendance claire s'affiche : (i) l'algorithme MLlib PFP est beaucoup plus lent que les autres, car il n'est pas

en mesure de traiter des transactions plus longues, et ses temps d'exécution augmentent brusquement jusqu'à épuisement de sa mémoire ; (ii) les algorithmes Mahout PFP et BigFIM ont une tendance similaire jusqu'à 70 éléments par transaction, jusqu'à ce que Mahout PFP devienne plus lent que BigFIM. Les expériences de la figure 4.4 montrent une tendance très similaire, à l'exception de l'algorithme BigFIM qui ne peut plus fonctionner.
Dans l'ensemble, bien que sa phase initiale soit basée sur Apriori, le BigFIM s'est avéré être la meilleure approche pour les transactions très longues et un minsup relativement élevé. Par contre, lorsque le Minsup est diminué, seul Mahout PFP est capable de faire face à la complexité de la tâche.

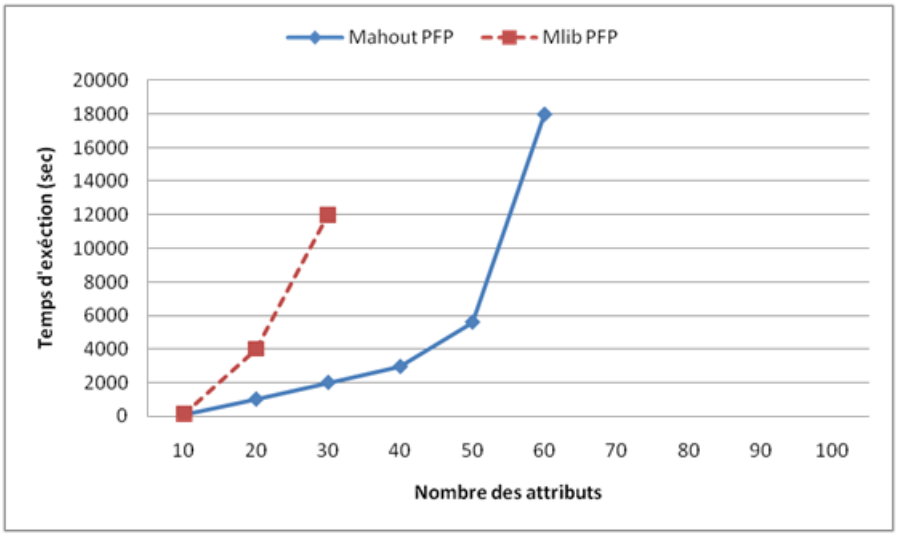


FIGURE 2.4: Temps d'exécution avec différentes longueurs moyennes de transactions (minsup 0.1%).

2.1.3 Impact du nombre de transactions

Nous avons évalué l'effet du nombre de transactions, qui représente la taille de l'ensemble de données, sans modifier les caractéristiques intrinsèques des données. Les expériences ont été effectuées sur les ensembles de données Dataset 1, Dataset 4 et Dataset 6 (voir le tableau 2.1), dont le nombre de transactions varie de 10 millions à 1 milliard. Le minsup est fixé à 0,4%, ce qui correspond à la valeur la plus élevée utilisée dans les expériences de la section 4.1.2.(impact minsup).
Puisque dans l'expérience le seuil de Minsup relatif est fixe, l'exploration de l'espace de recherche est similaire et n'est pas particulièrement exigeante, comme on peut le voir à la section 4.1.2.

Comme le montre la figure 2.5, tous les algorithmes considérés évoluent presque linéairement par rapport à la cardinalité de l'ensemble des transactions, BigFIM étant le plus lent, suivi de près par Mahout PFP. MLlib PFP est le plus rapide. Les implémentations PFP sont plus rapides que BigFIM parce qu'elles ne lisent que deux fois le jeu de données d'entrée sur le disque. BigFIM paie les opérations de lecture itérative du disque pendant sa phase initiale Apriori, lorsque le nombre de transactions de l'ensemble de données d'entrée augmente.
Enfin, DistEclat échoue car il stocke l'ensemble des données dans chaque nœud, alors il n'est plus en mesure d'effectuer l'extraction au-delà de 10 millions de transactions.

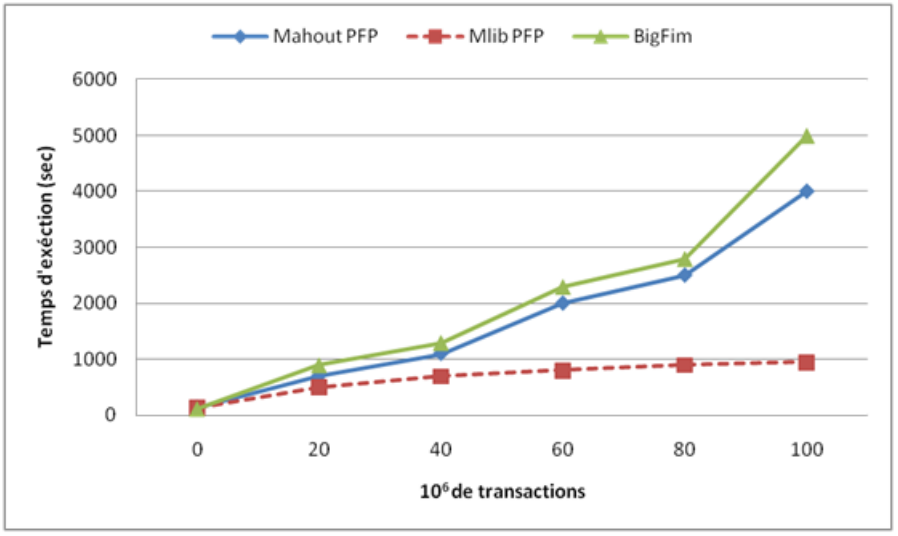


FIGURE 2.5: Temps d'exécution avec différents nombres de transactions

2.1.4 Répartition du temps d'exécution en phases

Afin d'étudier les goulots d'étranglement possibles au sein des algorithmes multiphases, nous avons comparé les temps d'exécution relatifs à chaque phase. Plus précisément, pour chaque algorithme, nous avons calculé le pourcentage du temps associé à l'exécution de chaque phase par rapport au temps total d'exécution de l'algorithme.
Nous avons choisi l'ensemble de données Dataset 1, et nous avons réglé la valeur du minsup à 0,15 %, ce qui nous a permis de compléter l'ensemble des expériences avec tous les algorithmes.
Comme le montre la figure 2.6, pour l'algorithme BigFIM, la longueur des préfixes extraits au cours de la première phase influe fortement sur le poids de cette phase

dans le processus global.
Pour DistEclat (Figure 2.7), au contraire, la différence n'est pas si importante.

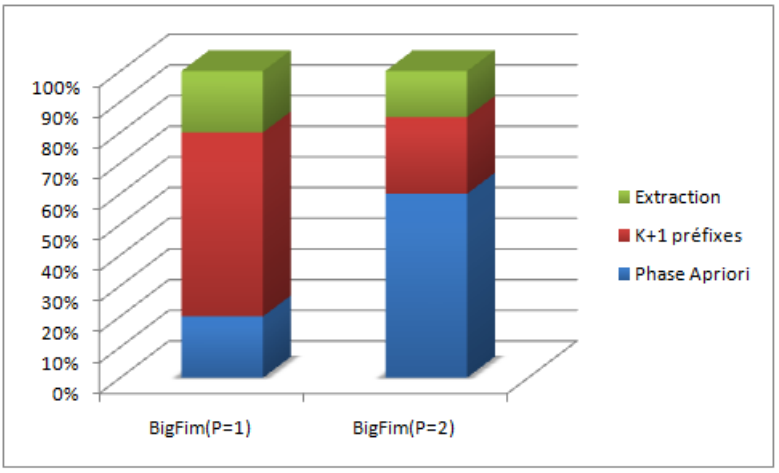


FIGURE 2.6: Temps d'exécution des phases de l'algorithme BigFIM

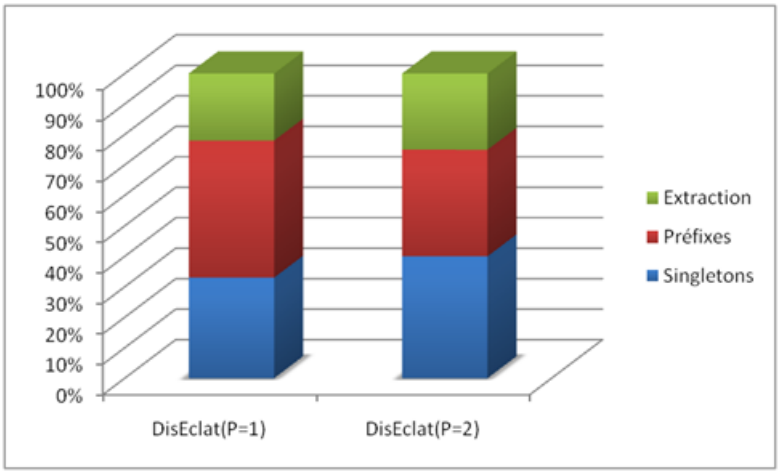


FIGURE 2.7: Temps d'exécution des phases de l'algorithme DisEclat

La dernière phase des deux algorithmes, qui est associée à l'extraction des ensembles d'éléments d'une longueur supérieure au seuil de longueur du préfixe, a un impact de plus faible ampleur sur le délai d'exécution des algorithmes, notamment lorsque le seuil est élevé.
Ces données, et les échecs rapportés dans les expériences des sous-sections précédentes, indiquent que les deux premières phases sont les principaux goulots d'étranglement pour les deux algorithmes.

Pour le BigFIM, chaque phase est fortement exposée aux problèmes de mémoire, comme le résume le tableau 2.2. Les expériences montrent que la phase Apriori est particulièrement difficile. Pour DistEclat, au contraire, la toute première étape

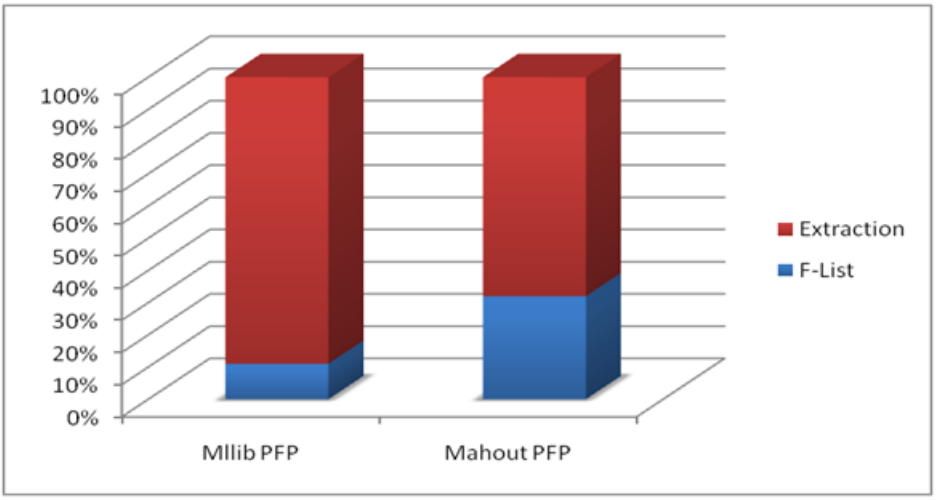


FIGURE 2.8: Temps d'exécution des phases des algorithmes Mahout PFP et MLlib PFP

est consacrée à l'extraction des 1-itemsets, et elle est principalement affectée par les coûts élevés de lecture de la base de données et de communication. Cependant, nous avons rencontré quelques problèmes de mémoire, qui sont probablement liés à la gestion des tid-listes. Les autres étapes sont plus susceptibles d'être affectées par des contraintes de mémoire.

La figure 2.8 présente les résultats de la mise en œuvre du PFP. Mahout PFP passe 1/3 du temps dans la première phase, au cours de laquelle la liste F est générée, tandis que MLlib PFP reste dans la deuxième phase pendant près de 90% du temps. La différence entre les deux approches est motivée par le traitement moins souple des différentes tâches par Hadoop par rapport à Spark. Même si, en particulier pour le Mahout PFP, la génération de la liste F pourrait prendre beaucoup de temps, ce n'est pas un goulot d'étranglement possible pour l'ensemble des extractions.

D'après la figure 2.8, le goulot d'étranglement pour les algorithmes basés sur FP-growth est la phase d'extraction des itemsets (c.-à-d. la deuxième phase de la PFP MLlib et de la PFP Mahout), fortement limitée par la mémoire.

La plupart des phases des différents algorithmes, sont fortement handicapées par des problèmes de mémoire. La disponibilité de la mémoire est le principal facteur affectant la capacité de chaque algorithme à effectuer l'extraction des itemset.

Tableau 2.2: Problèmes de phases

Algorithme	Phases	Obstacle
Algorithme base sur FP-Growth	F-List	Lecture, coût communication
	Extractions	Mémoire
BigFim	Phase Apriori	Mémoire
	K+ 1 Prefixes	Mémoire
	Extraction	Mémoire
DisEclat	Singletons	Lecture , coût.com et mémoire
	Prefixes	Mémoire
	Extraction	Mémoire

2.1.5 Équilibrage de charge

Nous avons analysé l'équilibrage de charge sur une durée d'une heure, sur l'ensemble de données Netlogs (tableau 2.1) avec une valeur de minsup fixée à 1 %. Nous considérons les tâches les plus déséquilibrées de chaque algorithme, et nous comparons les temps d'exécution des tâches les plus rapides et les plus lentes. Pour ce faire, nous ne nous intéressons pas au temps d'exécution absolu, mais surtout aux temps d'exécution normalisés, où la tâche la plus lente se voit attribuer une valeur de 100, et la tâche la plus rapide est comparée à cette valeur, comme le montre la figure 2.9.

L'algorithme MLlib PFP réalise le meilleur équilibrage de charge avec des temps d'exécution similaires pour toutes les tâches sur tous les nœuds, dont la différence est de l'ordre de 10%. Le Mahout PFP, au contraire, présente les pires résultats en ce qui concerne l'équilibrage de charges, les écarts pouvant atteindre 90 %. La différence entre MLlib PFP et Mahout PFP peut être corrélée à la granularité des sous-problèmes. Plus les sous-problèmes sont petits, plus l'équilibrage de charge est bon, car leurs temps d'exécution sont plus rapprochés.

MLlib PFP permet de spécifier le nombre de partitions, c'est-à-dire de sous-problèmes, ce qui a évidemment un impact sur la granularité de chaque sous-problème. Ainsi, en réglant convenablement ce paramètre, on obtient un bon résultat d'équilibrage de charge. Par ailleurs, Mahout PFP définit automatiquement le nombre de sous-problèmes, méthode qui fonctionne mal sur les ensembles de données considérés (des sous-problèmes non équilibrés sont générés).

Nous avons inclus BigFIM et DistEclat avec 2 tailles de préfixes différents dans la première phase. Pour ces algorithmes, l'expérience confirme qu'une configuration avec des préfixes plus longs conduit à des tâches d'extraction plus équilibrées

qu'une configuration avec des préfixes courts (comme mentionné dans la sous-section 3.4.3).

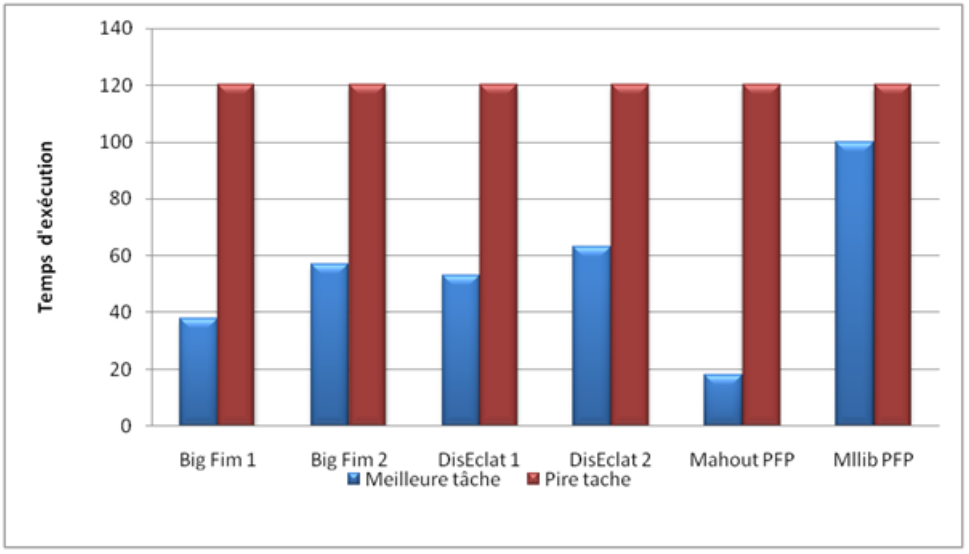


FIGURE 2.9: Temps d'exécution normalisé des tâches les plus déséquilibrées.

2.1.6 Coût de communication

Pour évaluer le coût de communication, nous mesurons la quantité de données transmises et reçues par les interfaces réseau des nœuds. Ces informations ont été récupérées à l'aide des utilitaires fournis par l'outil Cloudera Manager. Les expériences ont été effectuées sur l'ensemble de données Dataset 1, avec une valeur minsup fixe de 0,1 %, soit la valeur la plus faible pour laquelle tous les algorithmes ont complété l'extraction. La figure 2.10 présente, pour chaque algorithme, la valeur moyenne des trajets transmis et reçus par rapport au temps d'exécution total. Premièrement, les deux mesures ne semblent pas corrélées : des coûts de communication plus élevés sont associés à de faibles temps d'exécution pour BigFIM et DistEclat, alors que MLlib PFP reporte les deux mesures avec des valeurs élevées. Mahout PFP a un coût de communication 4 à 5 fois inférieur à tous les autres, qui échangent en moyenne 2 Gigaoctets de données. Le coût moyen des communications de l'algorithme Mahout PFP est d'environ 0,5 gigaoctets, ce qui correspond approximativement à la taille du jeu de données. La différence entre DistEclat et BigFIM n'est pas flagrante car avec seulement 2 préfixes de longueur, seule une itération supplémentaire est faite par BigFIM. Même si Mahout PFP est l'implémentation la plus optimisée en termes de coût de communication, sa très faible

quantité de données envoyées sur le réseau est liée à l'adoption de techniques de compression, ce qui conduit à des temps d'exécution plus longs.

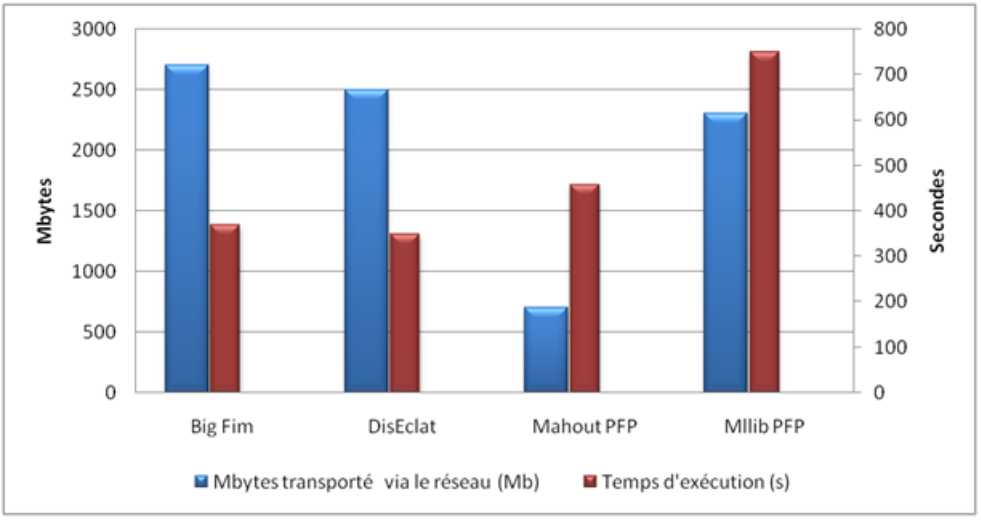


FIGURE 2.10: Coûts et performances de communication pour chaque algorithme (Datasets 1, minsup 0.1%) . Le graphique indique une moyenne entre les données transmises et les données reçues.

2.2 Conclusion

L'étude comparative présentée dans ce chapitre mentionne des orientations de recherche intéressantes pour améliorer les algorithmes distribués d'extraction de motifs fréquents sur les données massives . Comme nous l'avons expliqué dans cette étude, divers algorithmes ont été suggérés dans la littérature pour découvrir des ensembles d'éléments fréquents. Pourtant, l'extraction efficace de chaque algorithme repose sur des aptitudes et des capacités particulières, et L'analyste est tenu de choisir le meilleur. les algorithmes de l'état de l'art peuvent devenir inefficaces en raison des choix inadéquats des implémentations particulières et des configuration des paramètres, qui doivent être choisis avec soin. L'amélioration de la capacité d'utilisation des algorithme doit être abordée en concevant des cadres d'extraction de motifs fréquents auto-adaptatifs innovants, et capables de sélectionner judicieusement l'algorithme le plus adapté et de le configurer automatiquement.

Chapitre 3

Algorithme Apriori amélioré en utilisant une disposition hybride de données basée sur Hadoop pour le traitement de données volumineuses

Les algorithmes d'extraction de motifs fréquents existants se sont révélés très efficaces sur le traitement des ensembles de données simples mais très gourmands en ressources lorsqu'il s'agit de grandes bases de données. C'est pour cette raison que nous assistons à l'explosion des approches parallèles et distribuées, généralement basées sur des cadres distribués, comme Apache Hadoop[21] et Spark[128], afin de traiter efficacement les données massives.

L'algorithme Apriori [5]est une méthode itérative utilisée pour découvrir des motifs fréquents à partir d'un ensemble de données transactionnelles. Il scanne l'ensemble des données dans chaque itération pour obtenir le maximum d'ensembles d'éléments fréquents de différents cardinalités. L'algorithme Apriori est efficace pour les petites et moyennes bases de données, mais il n'est pas utile pour les grandes bases de données. Afin de résoudre le problème de traitement itératif des ensembles de données identifié par Apriori, nous présentons une nouvelle approche appelé Hybrid Frequent Itemset Mining on Hadoop (HFIMH) qui utilise une représentation verticale des ensembles de données pour résoudre le problème itératif du

traitement des données. Nous comparons notre approche avec une autre implémentation parallèle d'Apriori basée sur Hadoop sur différents ensembles de données. Les résultats expérimentaux démontrent que notre approche est meilleure.

3.1 Introduction

Récemment, les énormes progrès de la science et de la technologie ont affecté la taille des données. En plus de leur étendue, ces énormes données sont soit non structurées ou semi-non structurées, ce qui implique des difficultés de capture, de stockage, de partage, de recherche et d'analyse. d'où le nom de "Big Data" [63, 133].

L'extraction des motifs fréquents constitue une étape primordiale dans divers problèmes de la fouille de données, tels que l'extraction des règles d'association [8, 61, 131], la recherche de corrélations, et l'extraction des schémas séquentiels [60]. La performance de tout algorithme basé sur une recherche de tels motifs dépend en majeure partie des performances de cette étape. A titre d'exemple, la recherche des règles d'association est accomplie en deux étapes distinctes : la première, et la plus complexe, consiste à extraire tous les itemsets fréquents de la base de transactions, la seconde étape exploite ces itemsets fréquents pour générer, de façon quasi-automatique, les règles qui leur sont associées.

Comme nous l'avons déjà vu, beaucoup d'algorithmes dans la littérature adoptent une stratégie de recherche des motifs fréquents semblable à celle d'Apriori. Elle consiste à accomplir itérativement les deux opérations suivantes : génération d'un ensemble de motifs candidats de même taille k, puis calcul de leurs supports en parcourant la base de transactions.

A chaque itération on garde les candidats fréquents i.e. ayant des supports suffisants par rapport au support minimum. Ces mêmes motifs fréquents serviront à former les candidats de taille $k+1$ de l'itération suivante. Les itérations s'arrêtent lorsqu'on ne peut plus générer de candidats. La propriété adoptée pour élaguer l'espace des motifs possibles est dite d'anti-monotonie : si un itemset n'est pas fréquent, aucun de ses sur-ensembles ne sera fréquent.

Même si cette propriété réduit de façon considérable le nombre de candidats à considérer, la tache reste cependant complexe et coûteuse en temps et espace mémoire. De plus, il n'est pas toujours possible de charger la base de transactions en mémoire principale, ce qui occasionne des opérations d'entrées-sorties couteuses

sur des données résidant sur les disques.
L'algorithme Apriori est l'une des méthodes les plus utilisées pour découvrir l'ensemble des éléments de la base de données transactionnelle. L'utilisation de la version originale de l'algorithme Apriori est devenue inefficace, lorsque la taille des données a fortement augmenté. Les grandes données ont besoin d'un ensemble élevé de ressources pour le stockage et le traitement.
Ainsi, le calcul traditionnel sur une seule machine n'est plus suffisant pour traiter des données volumineuses. L'utilisation de plusieurs machines est devenu une obligation pour stocker et traiter d'énormes données en mode distribué.
Il existe dans l'état de l'art quelques implémentations distribuées de l'algorithme Apriori souffrant de quelques limitations telles que :
1.L'algorithme Apriori analyse l'intégralité du jeu de données à chaque itération pour calculer la prise en charge de chaque jeu d'éléments, ce qui consomme beaucoup de ressources.
2. Pour les grandes données transactionnelles, un nombre considérable d'ensembles de candidats sont générés à chaque itération.
3. Tous les ensembles d'éléments candidats ont besoin d'être stockés dans la mémoire principale pour une manipulation ultérieure, afin de produire des ensembles d'éléments fréquents.
Pour résoudre les limitations mentionnées, nous proposons un algorithme d'extraction de motifs fréquents efficace appelé (HFIMH), qui adopte une représentation verticale de l'ensemble de données, et la théorie d'intersection pour calculer les supports. Nous comparons notre algorithme avec une autre implémentation Apriori basée sur Hadoop.

3.2 Algorithme Apriori

L'algorithme Apriori , créé par Agrawal et Srikant en 1994 [8], procède en deux phases. Il est basé sur le principe lié à l'approche de support et de confiance. L'algorithme parcourt le treillis des itemsets pour rechercher les itemsets fréquents, et en déduire après les règles d'assocation dont la confiance dépasse le seuil de confiance minconf [69]. Le treillis des itemsets permet d'utiliser plus efficacement cet algorithme d'extraction en admettant les propriétés suivantes :

•Tout sous-ensemble d'un Itemset fréquent est fréquent.

•Tout sur-ensemble d'un itemset non fréquent est non fréquent.

3.2.1 Exemple de treillis des itemsets

Le nombre d'itemsets fréquents qui peuvent être générés de n items est de 2n. La génération des Itemsets fréquents est de complexité exponentielle. Il est alors essentiel de trouver la méthode de recherche la plus optimale. Ces Itemsets représentent un treillis d'Itemsets représenté sous la forme d'un diagramme de Hasse présenté à la figure (3,1)

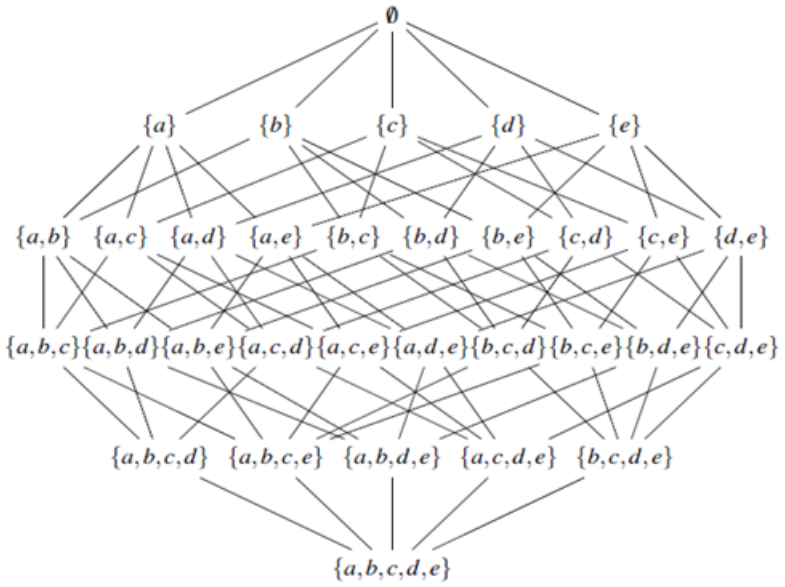


FIGURE 3.1: Espace de recherche dans la fouille de motifs comme un treillis des sous ensembles

3.2.2 Fonctionnement de l'algorithme Apriori :

D'une manière plus concise, l'exécution de l'algorithme Apriori se fait comme suit [3, 5] :

1. Générer les motifs candidats.

2. Calculer le support pour chaque motif candidat.

3. Apparier les motifs dont on a calculé le support avec le support choisi.

4.Rejeter les candidats dont le support est inférieur au minsup.

On termine comme résultats, avec tous les motifs fréquents dont le support est supérieur au support minimal.
En résumé le déroulement se fait comme suit :
L'algorithme Apriori fonctionne en deux phases. La première consiste en la recherche des ensembles d'items fréquents et la seconde utilise ces ensembles pour trouver les règles d'association dont la confiance est supérieure à un seuil prédéfini. Le processus de découverte des ensembles d'items fréquents est itératif, qui commence par la construction des ensembles fréquents avec un seul item. Ensuite on réitère pour construire des ensembles fréquent avec deux item, jusqu'à k-item , sachant que les ensembles d'items candidats avec n item sont construits à partir des ensembles d'items fréquents de taille $n-1$.

3.2.3 Avantages et inconvénients de l'Algorithme Apriori

Avantages

Il existe une multitude d'avantages dans l'utilisation de l'algorithme Apriori. On en énumère quelques-uns :
•La découverte rapide de règles d'association pertinentes entre objets.
•La facilité d'interprétation des résultats lors de l'extraction des règles d'association, malgré le nombre important de ces dernières.
Inconvénients :
Les inconvénients rencontrés lors d'une utilisation de l'algorithme Apriori sont les suivants :
•La génération d'un grand nombre de règles d'association.
•Un nombre important de configurations d'items ne peuvent pas engendrer de règles d'association.
•La recherche de règles d'association impose un temps considérable qui peut s'avérer désavantageux si l'on fait face à une grande base de données.

3.2.4 Hadoop Mapreduce

Hadoop est un framework libre et open source écrit en Java destiné à faciliter la création d'applications distribuées (au niveau du stockage des données et de leur traitement) et échelonnables (scalables) permettant aux applications de travailler

avec des milliers de nœuds et des pétaoctets de données. Ainsi chaque nœud est constitué de machines standard regroupées en grappe. Tous les modules de Hadoop sont conçus dans l'idée fondamentale que les pannes matérielles sont fréquentes et qu'en conséquence elles doivent être gérées automatiquement par le framework. Hadoop a été inspiré par la publication de MapReduce, GoogleFS et BigTable de Google. Hadoop a été créé par Doug Cutting et fait partie des projets de la fondation logicielle Apache depuis 2009.

Le noyau d'Hadoop est constitué d'une partie de stockage : HDFS (Hadoop Distributed File System), et d'une partie de traitement appelée MapReduce. Hadoop fractionne les fichiers en gros blocs et les distribue à travers les nœuds du cluster. Pour traiter les données, Hadoop transfère le code à chaque nœud et chaque nœud traite les données dont il dispose. Cela permet de traiter l'ensemble des données plus rapidement et plus efficacement que dans une architecture supercalculateur plus classique[réf. nécessaire] qui repose sur un système de fichiers parallèle où les calculs et les données sont distribués via les réseaux à grande vitesse.

Le framework Hadoop de base se compose des modules suivants :

• Hadoop Common

• Hadoop Distributed File System (HDFS) : le système de fichiers

• Hadoop YARN

• Hadoop MapReduce

Le terme Hadoop se réfère non seulement aux modules de base ci-dessus, mais aussi à son écosystème et à l'ensemble des logiciels qui viennent s'y connecter comme Apache Pig, Apache Hive, Apache HBase, Apache Phoenix, Apache Spark, Apache ZooKeeper, Apache Impala, Apache Flume, Apache Sqoop, Apache Oozie, Apache Storm.

MapReduce est un patron d'architecture de développement informatique, inventé par Google1, dans lequel sont effectués des calculs parallèles, et souvent distribués, de données potentiellement très volumineuses, typiquement supérieures en taille à 1 téraoctet.

Les termes « map » et « reduce », et les concepts sous-jacents, sont empruntés aux langages de programmation fonctionnelle utilisés pour leur construction (map et réduction de la programmation fonctionnelle et des langages de programmation tableau).

MapReduce permet de manipuler de grandes quantités de données en les distribuant dans un cluster de machines pour être traitées. Ce modèle connaît un vif

succès auprès de sociétés possédant d'importants centres de traitement de données tels Amazon.com ou Facebook. Il commence aussi à être utilisé au sein du Cloud computing. De nombreux frameworks ont vu le jour afin d'implémenter le MapReduce. Le plus connu est Hadoop qui a été développé par Apache Software Foundation. Mais ce framework possède des inconvénients qui réduisent considérablement ses performances notamment en milieu hétérogène. Des frameworks permettant d'améliorer les performances de Hadoop ou les performances globales du MapReduce, tant en termes de vitesse de traitement qu'en consommation électrique, commencent à voir le jour.

3.3 Travail connexe et énoncé de la problématique

De nombreux travaux de recherche ont été menés dans le domaine de l'extraction de motifs fréquents et de l'extraction de règles d'association. Un large éventail de techniques d'extraction de motifs fréquents sur les grandes bases de données a été discuté. En particuliers, de nombreuses versions améliorées de l'algorithme Apriori ont été proposées au cours des dernières décennies.
La plupart des variantes de l'algorithme Apriori ont été développées pour fonctionner avec des données de petite taille dans un système mono-machine. Mais avec l'introduction du Big Data, le système à une seule machine semble être incapable de gérer les grandes bases de données. De nombreuses recherches ont été faites dans un environnement informatique distribué avec un groupe de machine. Apache hadoop est l'un des principaux environnements informatiques distribués, qui a été adopté par de nombreux chercheurs pour l'extraction de motifs fréquents à partir de grandes bases de données. Une version parallèle de l'algorithme Apriori est fournie par Li et al.[80], qui produit de manière itérative les ensembles d'éléments fréquents en appliquant les fonctions de base de MapReduce. Yu et al[127] ont proposé un autre algorithme parallèle basé sur Apriori mais très coûteux en mémoire, qui génère les ensembles des candidats à partir de chaque transaction, puis extrait tous les ensembles d'éléments fréquents dans une itération individuelle. Récemment, deux nouveaux algorithmes nommées Dist-Eclat et BigFim ont été proposés[91]. L'algorithme Dist-Eclat est une version distribuée du fameux algorithme Eclat, qui consiste à diviser l'espace de recherche sur les nœuds de calcul. L'algorithme BigFim combine l'algorithme Apriori et l'algorithme Eclat afin

d'obtenir des résultats plus intéressants . Il existe aussi d'autres implémentations distribuées de l'algorithme Apriori qui sont présentées dans[40, 84, 126]. Tous les algorithmes ci-dessus sont développés sur Hadoop MapReduce soit en une seule étape , soit en plusieurs.

Dans tous les algorithmes parallèles basés sur Apriori, l'ensemble de données transactionnelles est soit représenté dans un format horizontal [84, 126], ou dans un format vertical [39, 68, 69]. Mais la plupart de ces implémentations utilisent une disposition horizontale de l'ensemble de données. Dans tous ces algorithmes, toutes les transactions sont scannées l'une après l'autre pendant chaque itération, afin de calculer le support d'un ensemble d'éléments. L'article [39, 69] a mis en œuvre l'algorithme Apriori basé sur MapReduce dans le format vertical. Dans cet algorithme, la génération d'un jeu d'éléments candidats nécessite une itération supplémentaire. L'algorithme produit également des ensembles d'éléments candidats insignifiants et inutiles. Le traitement de tous ces ensembles d'éléments est regroupés sur le reducer, ce qui augmente fortement l'ensemble des coûts de communication concernés. Pour résoudre les limites ci-dessus, nous proposons un algorithme efficace d'extraction de motifs fréquents, qui joint les dispositions verticales et horizontales des données, et calcul la cardinalité des différents ensembles d'éléments en profitant de la théorie d'intersection des ensembles [48]. Voici la liste de nos contributions :

• L'algorithme proposé est implémenté sous Hadoop.

• La disposition verticale de l'ensemble de données est utilisée pendant chaque itération afin de résoudre le problème du balayage de l'ensemble complet des données.

• L'ensemble de données horizontales est distribué et traité sur chaque machine afin de réduire le nombre de jeux de données candidats.

• Génération d'un nombre minimum de sous-ensembles des candidats afin d'accélérer le calcul de leurs supports à partir de la base verticale par le concept des intersections.

3.4 Algorithme proposé

3.4.1 Description

L'algorithme proposé se concentre sur le problème de l'analyse de l'ensemble des données à chaque itération qui entraîne un coût d'E/S et un espace disque élevé. De plus, a l'accès individuel des nœuds a tous l'ensemble des données pendant le processus d'extraction , ce qui nécessite une énorme capacité de stockage de la mémoire. A notre connaissance, aucune des implémentations distribuées d'Apriori sur hadoop n'a examiné la stratégie de partage des données verticales sur tous les nœuds avec une mise à jour des données à chaque k phase.

Afin d'accélérer le calcul des supports , nous avons utilisé le concept des intersections. L'ensemble de données vertical est constitué par la liste des éléments/ensembles suivie de leurs ID de transactions qui les contiennent. Il n'est pas nécessaire de balayer l'ensemble complet de données à chaque itération car l'ensemble de données vertical contient suffisamment d'informations pour calculer le support des ensembles candidats possibles.

Dans chaque enregistrement de données verticales, les ID sont triées par ordre croissant, ce qui facilite le calcul du support. Le support de k-élément candidat est calculé à travers l'intersection de leurs ID. Nous arrêtons l'intersection lorsque la condition de support minimum est remplie : Si le support minimum est de 10, pourquoi continuer à 11 alors que nous pouvons le valider à 10 ?

Pourtant, les données horizontales originales sont distribuées et traitées sur les nœuds du cluster pour générer les candidats en fonction de chaque transaction. L'algorithme complet est divisé en deux phases :

La phase 1 :La première phase de l'algorithme génère des éléments fréquents de 1-cardinalité dans une disposition verticale. Comme il s'agit de données volumineuses, l'ensemble de données peut contenir un grand nombre de transactions. Nous sauvegardons les données transactionnelles dans le système de fichiers distribués Hadoop (HDFS) du framework Hadoop, ce qui permet une meilleure utilisation de la mémoire du cluster et améliore la tolérance aux pannes. Plusieurs partitions de données sont réparties sur les nœuds de cluster.

L'ensemble de données verticales ne comprend que les éléments fréquents ainsi que les IDs des transactions qui les contiennent. Ensuite, la fonction de mappage est appliquée sur chaque élément pour produire une paire (clé, valeur), où clé est le motif et valeur est la liste des IDs des transactions. Le diagramme illustrant la

transformation d'une disposition horizontale à une disposition verticale est illustré à la figure 1. Après la génération des données verticales, on y applique l'étape d'élagage pour filtrer les éléments non fréquents. Par conséquent, à la fin de la phase 1, seuls les éléments fréquents font partie de l'ensemble de données verticales et tous les éléments non fréquents sont éliminés des données d'entrée horizontales originales, ce qui réduit la taille des données d'entrée. Cet ensemble de données est mis à jour afin de le préparer pour la phase suivante. L'algorithme proposé est illustré à la figure 3.2

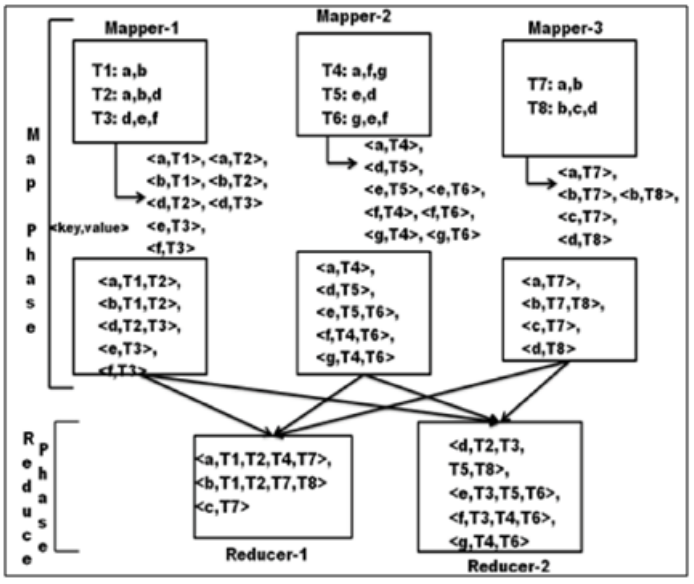


FIGURE 3.2: Diagramme illustrant la conversion de la disposition horizontale à la disposition verticale des données sur MapReduce.

Algorithm 4 for generating frequent itemsof 1 cardinalityin a vertical layout

input : Horizontal data-set
output : Vertical data-set
1. Each Mapper operates on its local database
2.Every Mapper generates local tid-listfor all items
3.Every Mapper exchanges tid-list with all other
4.To get total number of transaction associated with every items
5.Each Reducer regroups the tid-no related with an item from all Mapper output
6. to from global tid-list for each item

La phase 2 : Cette phase est une procédure itérative qui génère des ensembles d'items k-fréquents dans la k-ème itération. L'ensemble de données verticales, qui est généré lors du dernier passage, résout le problème du balayage de l'ensemble des données pour calculer le support des motifs candidats. Les données verticales sont destinées à conserver suffisamment d'informations afin de calculer rapidement

le support de chaque item, et générer tous les candidats potentiels. Tous les exécuteurs du cluster utilisés partagent les données verticales. Après la première phase, la base verticale contient les 1-éléments fréquents. Ces données ont des tailles plus petites que les données horizontales réelles, ce qui diminue l'utilisation de la mémoire. Les données horizontales révisées de la phase 1 sont réparties entre tous les exécuteurs afin de rendre l'algorithme utilisable en parallèle. Cependant, le fait que HFIMH génère les items candidats à partir de chaque transaction au lieu de générer tous les candidats possibles entraîne la réduction du nombre des items candidats dans chaque étape.
Les données horizontales révisées sont réparties sur les nœuds de cluster, et une liste d'items par rapport à chaque transaction est préparée. Nous stockons les données verticales dans la mémoire cachée distribuée sur hadoop afin de les partager par tous les exécuteurs. Ces données verticales partagées sont scannées et utilisées pour calculer le support de chaque ensemble des éléments candidats. Après chaque itération k, cet ensemble de données est mis à jour en ajoutant les nouveaux k -itemset fréquents dans la k-ème itération avec leurs listes des IDs afin de se préparer pour l'itération suivante. Tous les candidats de k-cardinalité sont produits dans la k-ème itération par rapport a chaque transaction.
Au lieu de générer tous les sous-ensembles de chaque item candidat $C_k = I_1 I_2 \dots I_k$,et afin d'obtenir son support, nous réduisons la recherche en joignant seulement les ID_sdes deux sous-ensembles S_{k-1} etS_k tels que $S_{k-1} = I_1 I_2 \dots I_{k-1}$ et $S_k = I_k$ sachant que $C_k = S_{k-1} S_k$, les ID_s des deux sous-ensembles S_{k-1} et Sk sont extraits à partir des données verticales partagées, et tous les ID_s de transactions communes des sous-ensembles sont enregistrés dans une liste commune. Un ensemble d'éléments est fréquent si la longueur de la liste commune est supérieure ou égale au minsup. Tous les ensembles d'éléments fréquents sont accumulés sous la forme (clé, valeur) où la clé est un ensemble d'éléments et la valeur est la liste des ID_s des transactions à ajouter à la base de données verticale partagée pour se préparer au passage suivant $(k + 1)$. L'algorithme proposé est illustré à la figure 3.

3.4.2 Analyse de complexité

Soit D l'ensemble de données contenant un total de k éléments dans n transactions et m le nombre d'éléments de la transaction la plus longue. Toutes les transactions sont scannées une seule fois pendant la première phase, et chaque item est

Algorithm 5 for generating frequent k- itemsof

input : Horizontal data-set with only frequent items
min sup= Minimum Support Threshold
$k \geqslant 2$: pass
output :$frequent\_itemset$:liste of frequent itemsets
1.For each transactin in $Revised\_horizontal\_data$
2.$map(trans\_ID, t)$
3.for each item I in t
4.$item\_list$=$\{I_1, I_2, I_j\}$
5.end for each
6. end map
7.end for each
8.for each item_list in t
9.combination=combination Generator($item\_list, k$)
10.for each combination C in combinations
11.$S_{k-1} = \{I_1, I_2, I_{k-1}\}, S_k = \{I_k\}$
12.Subset=$\{S_{k-1,S_k}\}$
13.ccommon=nul
14.for each itemset IS in subset
15.Search the itemset IS in Vertical _data
16.get the corresponding list_of_trans ID_S
17.If (common==nul)then
18.common=list_of_trans ID_s
19.end if
20.common=common interselect list_of_transID_S
21.end for each
22.if($length(common) \geqslant min\_sup$) then
23.frequen_itemset =frequent_ itemset List (C,common)
24.end if
25.end for each
26.end for each
27.Update the Vertical_data withnew frequent _itemsetfor the next pass

représenté dans une disposition verticale, ce qui prend un ordre de grandeur de $O(n \times m)$ temps dans le pire des cas. Tous les éléments passent par la phase d'élagage qui va scanner tous les k éléments, ce qui nécessitera un temps maximum $O(k)$. Ainsi, la complexité temporelle de la phase 1 est mesurée par $O(n \times m + k)$. Après la phase 1, l'ensemble de données original est révisé en supprimant tous les éléments non fréquents. Supposons que les données horizontales révisées de la phase 2 comprennent le total de K éléments, N transactions avec M la longueur de la transaction la plus longue, de sorte que $K \leq k, N \leq n$ *et* $M \leq m$. Toutes les transactions des données horizontales révisées sont scannées et une liste d'articles est créée, ce qui nécessite un temps d'ordre $O(N)$.

Ensuite, tous les candidats sont générés à partir des items de chaque liste, ce qui nécessite un temps $O({}_MC_i)$ égal à $O(Mmini, M-i)$ puisqu'après chaque passage, la valeur de i s'approche de M. Nous accédons à chaque candidat pour créer une liste de deux sous-ensembles. La complexité temporelle pour créer des sous-ensembles est égale à $O(2)$. Nous supposons que les données verticales contiennent v itemset et c nombre maximum d'ID de transaction pour un itemset, où $c \leq M$. Tous les sous-ensembles doivent scanner tous les itemsets à partir des données verticales et trouver les ID_s de transaction communs, ce qui nécessite un temps d'ordre maximal de $O(v \times c)$. Tous les candidats sont élagués en comparant la longueur des éléments de l'intersection au minsup, ce qui prendra du temps $O(1)$. Par conséquent, la limite supérieure du temps requis pour la phase 2 sera $O(N \times M^{mini,M-i} \times 2 \times v \times c)$. La complexité totale de l'HFIMH est la somme de la complexité des phases 1 et 2.

3.4.3 Analyse expérimentale et analyse des résultats

Pour exécuter l'algorithme HFIMH dans un environnement distribué, un cluster Hadoop de longueur variable est utilisé, où chaque nœud possède un processeur Intel® $Core^{TM}i5-3230MCPU$@ 2.60GHz et 6,00 Go de RAM avec le système d'exploitation Ubuntu 12.04 et Hadoop 2.2.0. Le système de fichier HDFS a été utilisé pour stocker les données transactionnelles. Les jeux de données T10I4D100K et Pumsb servent comme données expérimentales. Ils peuvent être téléchargés sur $http://fimi.ua.ac.be/dataJ$.

Dans l'expérience, nous avons comparé le temps d'exécution de notre algorithme

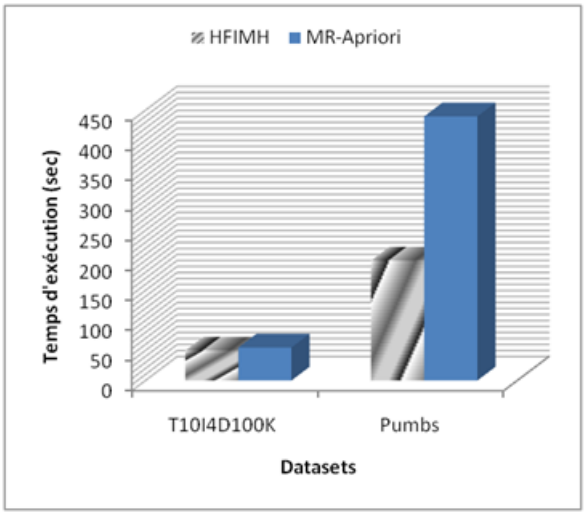


FIGURE 3.3: Comparaison du temps d'exécution

avec l'algorithme MR-Apriori[18]. Dans la figure 4 ci-dessus, l'axe vertical indique le temps d'exécution en secondes et l'axe horizontal indique la taille différente des ensembles de données. On peut voir que lorsque l'ensemble de données est de petite taille ($T10I4D100K$), la durée d'exécution des deux algorithmes est similaire. Mais, lorsqu'il s'agit d'un grand ensemble de données (Pumbs), la durée d'exécution de l'algorithme HFIMH sera plus courte que celle de MR-Apriori.

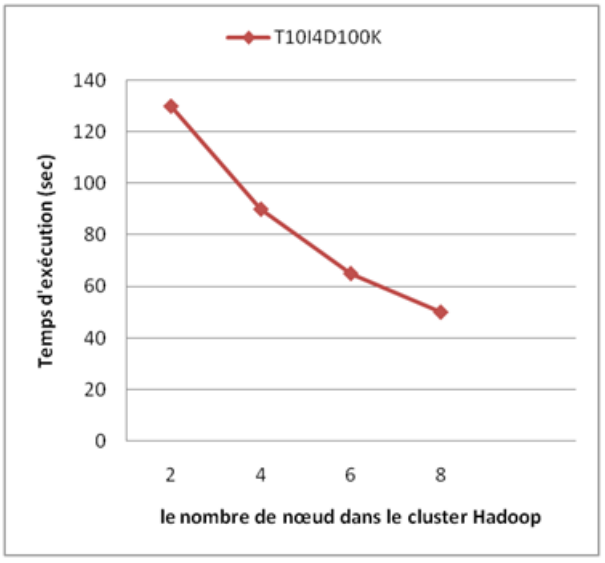


FIGURE 3.4: Temps d'exécution avec différents nombres de nœuds

Dans notre méthode, nous avons utilisé la disposition verticale des données et la théorie d'intersection des ensembles , qui ont simplifié le traitement des données, puisque on a produit des ensembles fréquents de 2-éléments à k- éléments en une seule étape à partir des ensembles de 1- élément. Le principal avantage du système ci-dessus est de réduire le temps total consacré à la recherche du support des éléments candidats. L'algorithme d'intersection nous a permis de calculer le support en comptant simplement les transactions communes dans chaque élément des ensembles candidats. Le temps écoulé lors de la recherche dans la base de données est considérablement réduit. De plus, le nombre de candidats est réduit, ce qui a amélioré le temps de parcours. Cela démontre que notre approche HFIMH donne de meilleures performances, et peut être utilisé efficacement pour le traitement de grands ensembles de données.

L'expérience suivante évalue l'évolutivité de l'HFIMH, qui est également mesurée par le temps de fonctionnement, en utilisant le jeu de données T10I4D100K. L'expérience est réalisée à condition que le nombre de nœuds d'ordinateurs du cluster varie entre 2 et 8 nœuds alors que la valeur du support est fixée à 0,5%.

Sur la figure 5, l'axe des abscisses indique le nombre de nœuds de calcul du cluster Hadoop, et l'axe des ordonnées indique la durée de l'algorithme HFIMH. La figure 5 montre ainsi le temps d'exécution de HFIMH avec différents nombres de nœuds du cluster : avec plus de nœuds, HFIMH nécessite moins de temps d'exécution, et la courbe de HFIMH a un déclin presque linéaire. L'algorithme HFIMH présente une caractéristique d'évolutivité quasi linéaire.

3.5 Conclusion

l'algorithme propose HFIMH utilise à la fois une disposition verticale et horizontale des données pour résoudre les déficiences mentionnées dans l'algorithme Apriori. En fait, l'algorithme HFIMH est une procédure à deux phases qui fonctionne efficacement dans un environnement distribué. Toutefois, pour mieux équilibrer la charge de travail, des recherches complémentaires sont nécessaires pour trouver un moyen de charger efficacement les données d'entrée dans les nœuds.

Chapitre 4

Amélioration des performances de l'algorithme PrePost sur Hadoop pour le traitement du Big Data

Avec la croissance exponentielle des données, il est devenu de plus en plus important d'intégrer des techniques améliorées d'extraction et des règles d'association, puis de tirer des informations utiles cachées. Plusieurs techniques d'extraction de données existantes permettent d'établir des règles d'association et d'obtenir des connaissances pertinentes. Mais avec l'arrivée rapide de l'ère des grandes bases de données, les algorithmes traditionnels d'extraction de données deviennent inefficace et demandent plus de ressources mémoires. Dernièrement, un nouvel algorithme d'extraction de motifs fréquents appelé PrePost a été suggéré. Prepost est basé sur l'idée des N-listes, et surpasse dans la plupart des cas les autres algorithmes de l'état de l'art. Dans ce chapitre, nous présentons HFIMH un nouvel algorithme distribué basé sur Prepost nommé HPrePostPlus qui est developpé sur Hadoop.Il utilise la structure HashMap pour parcourir efficacement l'arborescence de l'ensemble des données et améliorer le processus de création des N-listes liées aux 1-itemsets. L'expérience a démontré que l'algorithme HPrePostPlus est supérieur aux méthodes de l'état de l'art en termes de performances et d'évolutivité.

4.1 Introduction

L'explosion quantitative des données numériques a obligé les chercheurs à trouver de nouvelles manières de voir et d'analyser le monde. Il s'agit de découvrir de nouveaux ordres de grandeur concernant la capture, la recherche, le partage, le stockage, l'analyse et la présentation des données. Ainsi est né le « Big Data »[110]. Littéralement, ce terme signifie mégadonnées, grosses données ou encore données massives. Il désigne un ensemble très volumineux de données qu'aucun outil classique de gestion de base de données ou de gestion de l'information ne peut vraiment traiter.

Le Data Mining [62] est une composante essentielle des technologies Big Data et des techniques d'analyse de données volumineuses. Il s'agit là de la source des Big Data Analytics, des analyses prédictives et de l'exploitation des données. Le but de la fouille de données ou Data mining est d'examiner les données en recherchant et en interprétant des tendances ou des modèles imprévus, puis de vérifier les résultats avec les modèles détectés appliqués aux nouveaux sous-ensembles. Dans le domaine du data mining la recherche des règles d'association [55] est une méthode populaire étudiée d'une manière approfondie dont le but est de découvrir des relations ayant un intérêt pour le statisticien entre deux ou plusieurs variables stockées parmi d'importantes données. Traditionnellement, ce problème est décomposé en deux principaux sous-problèmes :

Premièrement, l'énumération de l'ensemble de motifs fréquents [7], i.e., les motifs dont la cooccurrence dans l'ensemble de données dépasse un seuil minimal de support fourni comme paramètre, puis la génération des règles d'association intéressantes à partir de ces motifs. En dépit de sa simplicité, la première phase, c'est-à-dire le calcul de la collection de motifs fréquents, constitue un problème complexe qui a été assez bien traité durant les deux dernières décennies.

La majorité des algorithmes d'extraction de motifs fréquents proposés peuvent être classés en deux groupes : méthode Apriori[8] et méthode FP-growth[61]. La méthode Apriori scanne la base de données pour trouver des ensembles d'éléments fréquents en générant un grand nombre de candidats. Alors que la méthode FP-growth effectue deux fois l'analyse pour extraire des ensembles d'éléments fréquents sans générer de candidat. Cette dernière utilise une arborescence comme structure de données pour stocker la base de données, et adopte la stratégie divide-and-conquer pour trouver des ensembles d'éléments fréquents.Ainsi elle est beaucoup plus efficace qu'Apriori.

Dernièrement, un algorithme efficace d'extraction des itemset fréquents nommé PrePost[34, 36] a été suggéré. PrePost utilise une nouvelle structure de données, N-Liste, pour représenter les éléments. N-List stocke toutes les informations importantes sur les items. En combinant l'approche de recherche par génération des candidats et l'approche de recherche de jeux de données fréquents directement sans génération de candidats, PrePost atteint une très grande efficacité pour extraire les motifs fréquents. L'efficacité de l'algorithme PrePost est atteinte pour les trois raisons suivantes. Premièrement, la structure N-List a le même degré de compression que l'arbre FP, et elle est donc beaucoup plus compacte que la structure verticale proposée précédemment. Deuxièment, pour le comptage des supports d'items, l'intersection de N-listes est beaucoup plus efficace que l'intersection des Ids de transactions tout en évitant les comparaisons inutiles. Troisièmement, PrePost peut extraire des ensembles d'éléments fréquents sans générer des éléments candidats dans certains cas, en utilisant la propriété du chemin unique de la liste N. La propriété du chemin unique de N-List permet de traiter efficacement le problème de génération d'un très grand nombre de candidats.

L'utilisation de la version originale de l'algorithme PrePost est devenue inefficace, du fait que la taille des données a fortement augmenté. Ainsi, le calcul traditionnel sur une seule machine n'est plus suffisant pour traiter des données volumineuses. L'utilisation de plusieurs machines est devenue une obligation pour stocker et traiter d'énormes données en mode distribué. Un calcul distribué, ou réparti ou encore partagé, est un calcul ou un traitement réparti sur plusieurs microprocesseurs et plus généralement toute unité centrale informatique. Le calcul distribué est souvent réalisé sur des clusters de calcul spécialisés, mais peut aussi être réalisé sur des stations informatiques individuelles à plusieurs cœurs. La distribution d'un calcul est un domaine de recherche des sciences mathématiques et informatiques. Elle implique notamment la notion de calcul parallèle.

MapReduce [31] un modèle (ou structure) de programmation disponible dans les environnements Hadoop est utilisé pour accéder aux big data stockées dans le Hadoop File System (HDFS). MapReduce entant qu'un élément essentiel fait partie intégrante du fonctionnement de l'environnement Hadoop.

MapReduce facilite les traitements concurrents en divisant les péta-octets de données en volumes plus petits et en les traitant en parallèle sur des serveurs standard dédiés à Hadoop. Pour résumer, MapReduce agrège les données de plusieurs serveurs et renvoie un résultat consolidé à l'application.

Un certain nombre de méthodes d'extraction d'ensembles d'éléments fréquents

distribués[18, 29, 81, 83, 91, 118], qui sont habituellement de simples extensions d'une méthode séquentielle utilisant des frameworks de traitement de données distribués, ont été proposées. Bien que les méthodes distribuées existantes puissent partiellement résoudre la limite de l'évolutivité, des problèmes persistent encore. Tout d'abord, elles n'ont pas une bonne extensibilité en raison de l'asymétrie de la charge de travail. Les méthodes distribuées existantes divisent l'espace de recherche des motifs (c.-à-d. l'arbre de dénombrement) à explorer en plusieurs branches (un sous-arbre) et assignent chacune à chaque machine. Chaque sous-arbre de l'arbre de dénombrement a tendance à avoir une taille différente, c.-à-d. une charge de travail différente. En particulier, les méthodes distribuées basées sur Eclat et FP-Growth présentent ce problème de façon remarquable. Par conséquent, les méthodes existantes ont tendance à ne pas améliorer les performances, proportionnellement au nombre de machines utilisées. Deuxièmement, n'ont pas une bonne extensibilité en raison de la surcharge de communication élevée du réseau. Les méthodes existantes effectuent généralement des fouilles fréquentes des ensembles d'éléments en redistribuant les données intermédiaires via le réseau. Cette approche pourrait largement dégrader les performances et l'évolutivité une fois que la quantité de données transférées entre les machines augmente. Dans cet article, nous proposons une version améliorée de PrePost, basée sur la méthode Hadoop, appelée HPrePostPlus. HPrePostPlus résout les problèmes ci-dessus et peut donc trouver des modèles fréquents sur des ensembles de données beaucoup plus importants que les méthodes distribuées existantes. Contrairement aux approches basées sur les arbres FP, l'algorithme HPrePostPlus ne construit pas d'arbres supplémentaires à chaque itération ; il extrait des ensembles d'éléments fréquents directement à l'aide du concept N-Liste.

L'efficacité de HPrePostPlus est confirmée parce que : (i) Les listes N sont beaucoup plus compactes que les structures verticales précédemment proposées, (ii) et le support d'un ensemble d'éléments fréquents candidat peut être déterminé par intersection des listes N. Ce processus est plus efficace pour de trouver l'intersection des listes TID, car il évite les comparaisons inutiles. Pour résoudre le problème de la surcharge de communication réseau, HPrePostPlus ne diffuse que des éléments fréquents via le réseau, dont la taille est beaucoup plus petite que celles des données intermédiaires. Par conséquent, HPrePostPlus présente des performances bien supérieures à celles des méthodes de l'état de l'art basées sur MapReduce.

Les principales contributions de ce document sont les suivantes :

(i)Nous proposons HPrePostPlus, une méthode évolutive basée sur Hadoop pour

l'extraction des motifs fréquents, sans données intermédiaires, et une petite communication réseau.
(ii)Nous utilisons la structure HashMap pour parcourir efficacement l'arborescence PPC, ainsi que pour accélérer le processus de création des N-listes associées aux 1-itemsets fréquents.
Les expériences montrent que HPrePostPlus surpasse les méthodes de pointe basées sur MapReduce en termes de vitesse et d'évolutivité. Le reste du chapitre est organisé comme suit. La section 4.2 présente les concepts de base. La section 4.3 donne un aperçu des travaux connexes. La section 4.4 présente l'approche proposée. Ensuite, la section 4.5 présente les résultats et les reflexions qui en découlent, ainsi que la discussion et l'exposé. Enfin, nous clôturons avec une conclusion dans la section 4.6 .

4.2 Préliminaires

4.2.1 Extraction des motifs fréquents

Les motifs fréquents sont des motifs ou patterns (tels que les ensembles d'items, les sous séquences, ou les sous structures) qui apparaissent fréquemment dans un ensemble de données. Par exemple, un ensemble d'items tel que le lait et le pain qui apparaissent souvent dans une base de transactions au supermarché, est un ensemble d'items fréquent. Une sous séquence telle que acheter premièrement un PC puis une caméra numérique ensuite une Carte mémoire qui se produit souvent dans la base historique des achats, est une séquence d'items fréquente. Les sous structures peuvent être des sous-graphes ou des sous-arbres qui peuvent être combinés avec des ensembles ou des séquences d'items. Trouver de tels motifs fréquents joue un rôle essentiel dans la fouille des associations et des corrélations, et représente une tâche importante en data mining. cela constitue toujours un thème remarquable de recherche.
Étant donné un ensemble fini non vide$A = \{a_1, a_2, \ldots, a_m\}$ de m éléments communément appelés items. Ces items peuvent désigner, comme déjà mentionné, des articles commandés d'un supermarché, des pages Web visitées en surfant, ou, en général, une collection de variables, d'attributs, ou d'évènements.

Un motif ou encore un itemset I est un sous-ensemble de A. Il est qualifié de k-motif lorsque son cardinal est égal à k. Dans le contexte du modèle du panier de la ménagère, un k-motif symbolise donc l'ensemble des produits commandés par un client lors de ses courses. De même, une transaction $t_{(i)}$ est un sous-ensemble non vide de A. Elle est identifiée par son identificateur unique i. Un ensemble (ou une base) de données D est un ensemble de n transactions, qu'on note comme un multiset $D = \{t_1, t_2, \ldots, t_n\}$.
Notons que dans ce problème, le nombre n de transactions est généralement supposé très large et que la taille (longueur) d'une transaction est nettement inférieure au nombre total d'items. Dans une base de données D, l'ensemble de transactions qui contiennent un motif x est appelé sa couverture. Le support d'un motif x, dénoté $sprt(x, D)$ est le nombre de transactions qui contiennent x, i.e., le cardinal de sa couverture.

$$sprt(x, D)_{abs} = |\{t_k \in D / x \subseteq t_k\}| \tag{4.1}$$

Précisons que l'équation donne la valeur absolue du support (un entier positif inférieur ou égal à la taille de la base de données), qui peut aussi être exprimée en relatif, c-à-d, par un réel compris entre zéro et un, représentant un pourcentage en le divisant par la taille de l'ensemble de données comme cela est montré dans la formule ci-dessous .

$$sprt(x, D)_{rel} = \frac{|\{t_k \in D / x \subseteq t_k\}|}{|D|} \tag{4.2}$$

Afin d'alléger l'écriture, nous le notons simplement $sprt(x)$ sans sous-mention abs ni référence à une base de transactions qui seront déduits du contexte. En effet, ceci est actuellement une tendance largement répondue dans la communauté fouille de données [19, 20, 46, 78].

4.2.1.1 Définition

(Motif fréquent) Un motif x est fréquent si son support dépasse un seuil minimal s spécifié d'avance. Formellement, x est fréquent dans D relativement au seuil s ssi : $sprt(x) \geqslant s$
Le problème de la fouille de motifs consiste à énumérer l'ensemble $F_D(s)$ de tous les motifs fréquents dérivés de A présents dans la base de transactions D au regard du seuil minimal du support s. Pour alléger la notation, ce dernier ensemble sera,

désormais, noté F tout court.

$$F_D(S) = \{x \in /sprt(x) \geqslant s\} \tag{4.3}$$

4.2.2 Hadoop Mapreduce

Nous sommes actuellement dans l'ère de la production massive de données (Big-Data) dont une définition implique trois dimensions (3Vs) : Volume, Variété et Vélocité (fréquence). Les sources de données sont nombreuses. D'une part les applications génèrent des données issues des logs, des réseaux de capteurs, des rapports de transactions, des traces de GPS, etc. et d'autre part, les individus produisent des données telles que des photographies, des vidéos, des musiques ou encore des données sur l'état de santé (rythme cardiaque, pression ou poids).
Un problème se pose alors quant au stockage et à l'analyse des données. La capacité de stockage des disques durs augmente mais le temps de lecture croît également. Il devient alors nécessaire de paralléliser les traitements en stockant sur plusieurs unités de disques durs. Toutefois, cela soulève forcément le problème de fiabilité des disques durs qui engendre la panne matérielle. La solution envisagée est la duplication des données comme le ferait un système RAID.
Apache Hadoop (High-availability distributed object-oriented platform) est un système distribué qui répond à ces problématiques. D'une part, il propose un système de stockage distribué via son système de fichier HDFS (Hadoop Distributed File System) et ce dernier offre la possibilité de stocker la donnée en la dupliquant, un cluster Hadoop n'a donc pas besoin d'être configuré avec un système RAID qui devient inutile. D'autre part, Hadoop fournit un système d'analyse des données appelé MapReduce. Ce dernier officie sur le système de fichiers HDFS pour réaliser des traitements sur des gros volumes de données.
Hadoop n'a d'intérêt que s'il est utilisé dans un environnement composé de plusieurs machines. Utiliser Hadoop dans un environnement monomachine, comme nous allons le faire dans le prochain tutoriel, n'a de sens que pour tester la configuration de l'installation ou fournir un environnement de développement MapReduce (prochain article). Hadoop n'a également pas d'intérêt pour les données de petite taille. C'est même l'effet inverse qui pourrait se produire. Si les données ne dépassent pas les limites de la mémoire ou des disques durs du marché (actuellement la limite est aux alentours de cinq téraoctets), posez-vous la question sur l'utilité d'utiliser Hadoop face à des solutions utilisant des bases de données relationnelles.

Vous trouverez sur ce lien une analyse simple mais efficace.

HDFS

HDFS (Hadoop Distributed File System) reprend de nombreux concepts proposés par des systèmes de fichiers classiques comme ext2 pour Linux ou FAT pour Windows. Nous retrouvons donc la notion de blocs (la plus petite unité que l'unité de stockage peut gérer), les métadonnées qui permettent de retrouver les blocs à partir d'un nom de fichier, les droits ou encore l'arborescence des répertoires.

Toutefois, HDFS se démarque d'un système de fichiers classique pour les principales raisons suivantes.

• HDFS n'est pas solidaire du noyau du système d'exploitation. Il assure une portabilité et peut être déployé sur différents systèmes d'exploitation. Un des inconvénients est de devoir solliciter une application externe pour monter une unité de disque HDFS.

• HDFS est un système distribué. Sur un système classique, la taille du disque est généralement considérée comme la limite globale d'utilisation. Dans un système distribué comme HDFS, chaque nœud d'un cluster correspond à un sous-ensemble du volume global de données du cluster. Pour augmenter ce volume global, il suffira d'ajouter de nouveaux nœuds. On retrouvera également dans HDFS, un service central appelé Namenode qui aura la tâche de gérer les métadonnées.

• HDFS utilise des tailles de blocs largement supérieures à ceux des systèmes classiques. Par défaut, la taille est fixée à 64 Mo. Il est toutefois possible de monter à 128 Mo, 256 Mo, 512 Mo voire 1 Go. Alors que sur des systèmes classiques, la taille est généralement de 4 Ko, l'intérêt de fournir des tailles plus grandes permet de réduire le temps d'accès à un bloc. Notez que si la taille du fichier est inférieure à la taille d'un bloc, le fichier n'occupera pas la taille totale de ce bloc.

• HDFS fournit un système de réplication des blocs dont le nombre de réplications est configurable. Pendant la phase d'écriture, chaque bloc correspondant au fichier est répliqué sur plusieurs nœuds. Pour la phase de lecture, si un bloc est indisponible sur un nœud, des copies de ce bloc seront disponibles sur d'autres nœuds

MapReduce

MapReduce adresse deux choses. La première concerne le modèle de programmation MapReduce, étudié dans cette section. La seconde concerne le framework d'implémentation MapReduce, étudié dans le prochain article. Pour ce dernier, nous nous focaliserons sur les différentes API proposées par Apache Hadoop pour développer un programme MapReduce à partir du langage Java.

Le modèle de programmation fournit un cadre à un développeur afin d'écrire une

fonction de Map et de Reduce. Tout l'intérêt de ce modèle de programmation est de simplifier la vie du développeur. Ainsi, ce développeur n'a pas à se soucier du travail de parallélisation et de distribution du travail, de récupération des données sur HDFS, de développements spécifiques à la couche réseaux pour la communication entre les nœuds, ou d'adapter son développement en fonction de l'évolution de la montée en charge (scalabilité horizontale, par exemple). Ainsi, le modèle de programmation permet au développeur de ne s'intéresser qu'à la partie algorithmique. Il transmet alors son programme MapReduce développé dans un langage de programmation au framework Hadoop pour l'exécution.
Autre chose avant de continuer, le terme de « job » MapReduce est couramment utilisé dans la littérature. Celui-ci concerne une unité de travail que le client souhaite réaliser. Cette unité est constituée de données d'entrée (contenues dans HDFS), d'un programme MapReduce (implémentation des fonctions map et reduce) et de paramètres d'exécution. Hadoop exécute ce job en le subdivisant en deux tâches : les tâches de Map et les tâches de Reduce.
Voyons maintenant le principe général de MapReduce, puis nous détaillerons son fonctionnement distribué dans Hadoop.
Les concepts de map et de reduce ne sont pas nouveaux puisqu'ils ont été empruntés aux langages fonctionnels, sauf que Google les a efficacement propulsés dans l'univers du calcul distribué et du grand volume de données. Ils sont utilisés pour implémenter des opérations de base sur les données comme le tri, le filtrage, la projection, l'agrégation ou le regroupement.

4.2.3 Algorithme Prepost

4.2.3.1 Définition de l'arbre PPC

Avant d'aborder la structure N-List, nous commençons d'abord par l'arbre PPC, qui est la base de la structure N-List. Nous définissons un arbre PPC comme suit.
Définition 1
L'arbre PPC est une structure arborescente :
1.Elle se compose d'une racine appelée "null", et d'un ensemble de sous-arbres constitué de préfixes des différents éléments comme étant les enfants de la racine.
2.Chaque nœud du sous-arbre de préfixe d'élément se compose de cinq champs :

item-name, count, children-list, pre-order et post-order. Item-name enregistre l'élément représentant le noeud, count enregistre le nombre des transactions présentées par la partie du chemin accédant à ce nœud, children-list enregistre tous les enfants du nœud, pre-order est le rang du pre-order du nœud et post-order est le rang du post-ordre du nœud. Selon la définition 1, l'arbre PPC ressemble à un arbre FP [1,15]. Notons que l'arbre FP est appelé arbre PC dans [15]. Cependant, il y a trois différences importantes entre eux.

Tout d'abord, FP-tree a un champ de lien de noeud dans chaque noeud et une structure de table pour maintenir la connexion des noeuds dont les noms d'élément sont égaux dans l'arbre, Quant à PPC-tree , il ne possède pas de telle structure, Car PPC-tree c'est un arbre de préfixes plus simples.

Deuxièmement, chaque nœud de l'arbre PPC possède des champs de pré-ordre et de post-ordre qu'on ne trouve pas dans l'arbre FP qui n'en possède pas. Le pré-ordre d'un nœud est déterminé par une traversée pre-order de l'arbre. Dans une traversée pre-order, un nœud N est visité et affecté au rang pré-ordre avant que tous ses enfants ne soient parcourus récursivement de gauche à droite. En d'autres termes, le pré-ordre enregistre le moment où l'on accède au nœud N pendant la traversée du pré-ordre. De la même manière, le post-order d'un nœud est déterminé par une traversée post-ordre de l'arbre. Dans une traversée post-order, un nœud N est visité et affecté à son rang post-ordre après que tous ses enfants ont été parcourus récursivement de gauche à droite.

Troisièmement, une fois l'arbre FP contruit, il sera utilisé pour l'extraction des motifs fréquents pendant tout le déroulement de l'algorithme FP-growth, qui est un algorithme récursif et complexe. Toutefois, l'arbre PPC n'est utilisé que pour générer le code Pre-Post de chaque nœud. Plus tard, nous verrons qu'après avoir collecté le code Pré-Post de chaque élément fréquent en premier, l'arbre PPC peut être supprimé après avoir terminé cette tache complètement .

Pour mieux comprendre le concept et l'algorithme de construction de l'arbre PPC, nous allons examiner l'exemple suivant. Soit la liste des transactions présente dans le tableau ci-dessous. Pour un minsup =0.4. l'ensemble des 1-itemsets fréquents $F1 = \{a.b, c, e, f\}$. La figure 4.1 montre l'arbre PPC résultant de la liste du tableau 4.1.

TABLEAU 4.1: Liste des transactions

ID	Items	Ordered frequent items
1	a, c, g, f	c ,f ,a
2	e, a, c, b	b ,c ,e ,a
3	e, c, b, i	b ,c ,e
4	b ,f, h	b ,f
5	b, f, e, c, d	b ,c ,e ,f

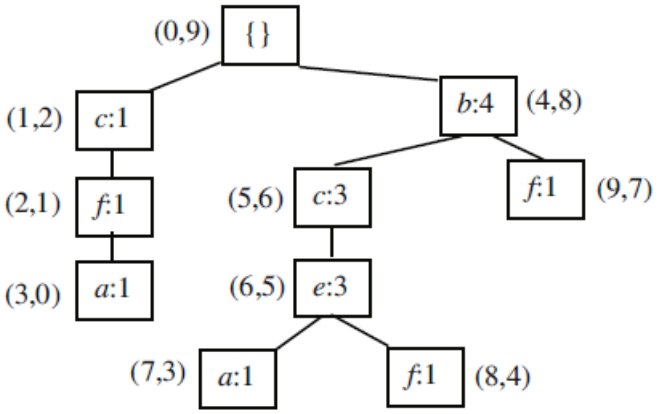


FIGURE 4.1: Arbre PPC résultant à partir du tableau x

4.2.3.2 N-List : définitions et propriétés

Dans cette section, nous donnerons la définition de N-liste et présenterons quelques propriétés importantes de N-liste, qui déterminent l'efficacité de notre nouvel algorithme proposé pour la fouille d'éléments fréquents de la liste. Nous définissons d'abord le code PP, qui est le composant de base de N-List.

Définition 2 (code PP). Pour chaque nœud N dans un arbre PPC, nous appelons ((N.pre-order, N.post-order) : count), le code PP de N. En fait, le but de la construction de l'arbre PPC est de générer les codes PP des éléments fréquents, puisque les codes PP peuvent refléter la structure de l'arbre PPC comme suit.

Propriété 1. Pour deux nœuds N1 et N2 différents, N1 est un ancêtre de N2, si et seulement si N1pre-order $<$ N2.pre-order et N1.post-order $>$ N2.post-order.

Pour la preuve de la Propriété 1, se référer à[16]. La propriété 1 montre également que les nœuds et leur code PP sont des cartographies 1-1. En d'autres termes, un nœud détermine de façon unique un code PP et un code PP détermine également de façon unique un nœud. Par conséquent, nous avons la définition suivante.

Définition 3(Relation ancêtre-descendant des codes PP). Étant donné deux codes

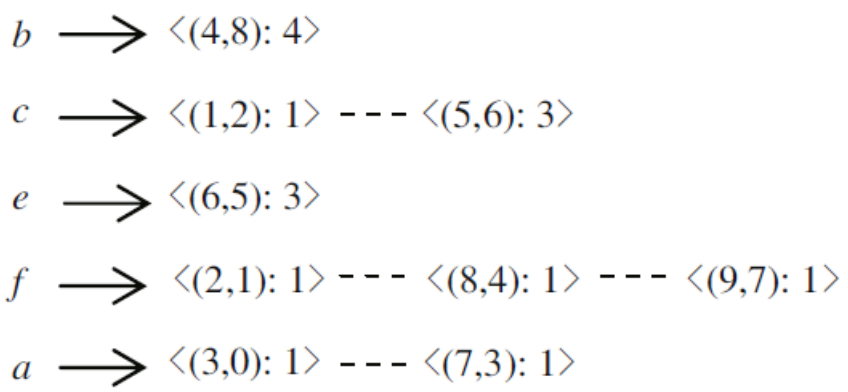


FIGURE 4.2: N-lists correspondantes au tableau 1

PP X_1 et X_2, X_1 est un ancêtre deX_2 si et seulement si le nœud représenté parX_1 est un ancêtre du nœud représenté par X_2. Soit$X_1(X_1, y_1) : z_1$ et $X_2(X_2, y_2) : z_2$, Définition 3 est équivalent au fait que X_1 est un ancêtre de X_2 si et seulement si$X_1 < X_2$ et$y_l > y_2$. Nous appelons aussi Y un descendant de X si X est un ancêtre de Y.

Définition 4 (N-liste des éléments fréquents). Avec un arbre PPC, N-liste d'un élément fréquent est une séquence de tous les codes PP des nœuds qui enregistrent l'élément dans l'arbre PPC. Les codes PP sont classés dans l'ordre croissant de leurs valeurs de pre-order. Chaque code PP de la N-liste est noté par (x, y) : z, où x est son pre-order, y son post-ordre et z son count. Et la N-liste d'un élément fréquent est notée par $(x_1, y_1) : z_1, (x_2, y_2) : z_2, ..., .., (x_1, y_l) : z_l$, où$x_1 < x_2 < - - - < x_1$. Par exemple, la N-liste de f contient trois nœuds, qui sont (2, 1) : 1, (8, 4) : 1, et (9, 7) : 1. La figure 2 montre les N-listes de tous les éléments fréquents de l'exemple 1.

Propriété 2. Pour deux nœuds N_1 etN_2 différents qui représentent le même élément (N_1.item-name = N_2.item-name), si N1.pre-order < $N_2$2.pre-order, puis N1.post-order < N2.post-order. Lorsque N_1 . pré-ordre < $N_2$2. pré-ordre, cela signifie que N_1 est parcouru plus tôt queN_2 pendant la traversée pré-ordre. Puisque N_1 ne peut pas être un ancêtre de N_2 parce qu'ils enregistrent tous les deux le même élément, N_1 doit être sur la branche gauche de l'arbre PPC par rapport à N_2. Pendant la traversée post-ordre, la branche gauche sera également traversée avant N_2, donc N_1 .post-ordre < $N_2$2.post-ordre. Dans la N-liste de l'élément i qui est désignée par $(x_1, y_1) : z_1, (x_2, y_2) : z_2, ..., .., (x_1, y_l) : z_l$, puisque nous

arrangeons le code PP dans l'ordre auquel on accède lors du passage en pre-order, nous avons $x_1 < x_2 < - - - < x_1$. Selon la propriété 2, nous avons aussi $y_1 < y_2 < y_2 < - - - < y_l$. Par exemple, dans la figure 2, la N-liste de l'élément c est (1, 2) : 1,(5, 6) : 3 et la N-liste de l'élément f est (2, 1) : 2,(8, 4) : 1,(9, 7) : 1. Ils confirment tous les deux cette propriété.

Propriété 3. Dans la N-liste de l'élément i, qui est désignée par $(x_1, y_1) : z_1, (x_2, y_2) : z_2, ..., .., (x_m, y_m) : z_m$, le support de l'élément i est $z_1 + z_2 + - - - + z_m$.

C'est une conséquence de la définition du code PP. Puisque chaque code PP correspond à un nœud de l'arbre PPC, dont le nombre de transactions incluant l' élément i est enregistré, la somme des counts de nœuds qui enregistrent l'élément i est supporté.

Précisons que, dans ce document, nous désignons par L1 l'ensemble des éléments fréquents, dans lequel les éléments fréquents sont triés par ordre décroissant de support. Sur la base de L_1, nous définissons la relation de deux éléments comme suit.

Définition 5 (relation>). Pour deux éléments fréquents i_1 et$i_2(i_1, i_2 \in L_1).i_1 i_2$ si et seulement si i_1 est avant i_2 dansL_1.

Pour simplifier la description, tout ensemble d'éléments P dans ce document est désigné par$i_1 i_2 - - - i_k$, où$i_1 i_2 - - - i_k$. Nous définissons la N-liste d'un ensemble de 2 postes, qui ne contient que deux postes différents, comme suit.

Définition 6(N-liste de 2 articles). Pour deux éléments fréquents différents i_1 et i_2 (i_1 est avant i_2 dans L_1), dont les N-listes sont$((x_{11}, y_{11}) : z_{11}), ((x_{12}, y_{12}) : z_{12}), ..., ((x_{1m}, y_{1m}) : z_{1m})$ et $\{((x_{21}, y_{21}) : z_{21}), ((x_{22}, y_{22}) : z_{22}), ..., ((x_{2n}, y_{2n})$ respectivement $z_{2n})\}$. La N-liste de 2 ensembles d'éléments $i_1 i_2$ est une suite de codes PP selon l'ordre croissant pré-ordre et est générée en croisant les N-listes de i_1 et i_2, qui suit la règle suivante :

1)Pour $((x_{1p}, y_{1p}) : z_{1p}) \in$ la N-liste de $i_1(1 \leq p \leq m)$ et $((x_{2q}, y_{2q}) : z_{2q)} \in$ la N-liste de $i_2(1 \leq q \leq n)$, si $((x_{1p}, y_{1p}) : z_{1p})$ est un ancêtre de $((x_{2q}, y_{2q}) : z_{2q})$, puis $((x_{1p}, y_{1p}) : z_{2q})$est ajouté à la N-liste de $i_1 i_2$. Ensuite, on obtient une première liste N de i1$i_1 i_2$.

2) vérifier à nouveau la N-liste initiale de i1$i_1 i_2$. Fusionner les nœuds sous la forme $((x_{1b}, y_{1b}) : z_{1b})((x_{1b}, y_{1b}) : z_{2b}) - - - ((x_{1b}, y_{1b}) : z_{rb})$ pour obtenir un nouveau

nœud $((x_{1b}, y_{1b}) : (z_{1b} + z_{2b} - - - + z_{rb}))$.
Dans la figure 2, la N-liste de b est ((4, 8) : 4) et la N-liste de f est((2, 1) : 1),((8, 4) : 1), ((9, 7) : 1). On verra comment générer la N-liste de bf en associant la N-liste de b et la N-liste de f. Selon la définition 6, on doit vérifier une relation ancêtre-descendant de ((4, 8) : 4) avec chaque code PP de la N-liste de f. Parce que 4, le préordre de ((4, 8) : 4), est supérieur à 2, le préordre de((2, 1) : 1), ((4, 8) : 4) ne peut être un ancêtre de((2, 1) : 1) au sens de la définition 3. Comparons ((4, 8) : 4) et ((8, 4) : 1)Puisque 4, le pré-ordre de ((4, 8) : 4), est inférieur à 8, le pré-ordre de ((8, 4) : 1), et 8, le post-ordre de ((4, 8) : 4), est supérieur à 4, le post-ordre de ((8, 4) : 1). Selon la définition 3,((4, 8) : 4) est un ancêtre de ((8, 4) : 1). au sens de la définition 6 (première règle), ((4, 8) : 1), qui est composée du préordre et du post-ordre de ((4, 8) : 4) et le count de ((8, 4) : 1), est ajouté à la N-liste des N de bf. De même, nous savons que ((4, 8) : 4)est un ancêtre de ((9, 7) : 1) Donc,((4, 8) : 1), qui est composé du pre-ordre et post ordre ((4, 8) : 4)et le count ((9, 7) : 1) est ajouté à la N-liste du bf. En vérifiant les éléments de la N-liste de bf, nous constatons que les deux seuls éléments qui ont le même pré-ordre et post-ordre, sont respectivement de 4 et 8. Selon la définition 6 (deuxième règle), nous devrons combiner ((4, 8) : 1) et ((4, 8) : 1) pour former ((4, 8) : 2). Ainsi, la N-liste de bf est ((4, 8) : 2). Parce qu'il n'y a pas d'autres éléments sauf((4, 8) : 4) dans la N-liste de b, le traitement est arrêté. La figure 3 montre l'ensemble de la procédure de traitement. De même, nous savons que la N-liste des cf est((1, 2) : 1), ((5, 6) : 1) . Basé sur la définition 6, généralisons-la au concept de la N-liste d'un ensemble de k éléments($k \geqslant 3$).

Définition 7(N-liste de l 'ensemble de k éléments). Soit $P = i_x i_y i_1 i_2 i_2 ... i_{(k-2)}$un ensemble des éléments ($k \geqslant 3$) et chaque élément dans P est un élément fréquent), la N-liste $P_1 = x_{11}, y_{11}) : z_{11}, (x_{12}, y_{12}) : z_{12}, ..., (x_{1m}, y_{1m}) : z_{1m}$, et la N-liste $P_2 = i_y i_1 i_2 i_{(k-2)} b_e(x_{21}, y_{21}) : z_{21}, (x_{22}, y_{22}) : z_{22}, ..., (x_{2n}, y_{2n}) : z_{2n}$. La N-liste de P est une suite de codes PP selon l'ordre croissant pré-ordre et générés en croisant les N-listes deP_1 et P_2, qui suit la règle ci-dessous :
pour $(x_{1p}, y_{1p}) : z_{1p} \in$ la N-liste de $P_1(1 \leq p \leq m)$ et$(x_{2q}, y_{2q}) : z_{2q} \in$ la N-liste de $P_2(1 \leq q \leq n)$, si $(x_{1p}, y_{1p}) : z_{1p}$ est un ancêtre de $(x_{2q}, y_{2q}) : z_{2q}, puis(x_{1p}, y_{1p}) :: z_{2q}$ est ajoutée à la N-liste de $i_x i_y i_1 i_2 i_2 ... i_{(k-2)}$. ensuite, nous obtenons une première N-liste de P.
On vérifie à nouveau la N-liste initiale de P et on fusionne les nœuds sous la forme $((x_{1b}, y_{1b}) : z_{1b}))((x_{1b}, y_{1b}) : z_{2b}) - - - ((x_{1b}, y_{1b}) : z_{rb})$pour obtenir un nouveau

nœud $((x_{1b}, y_{1b}) : (z_{1b} + z_{2b} - - - + z_{rb}))$.

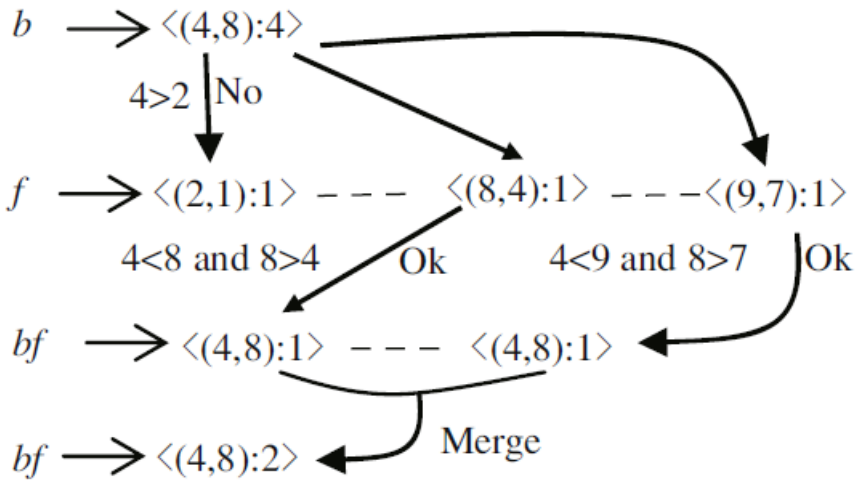


FIGURE 4.3: N-lists correspondante à l'item bf du tableau 1

Par exemple, nous avons connu la N-liste de bf et cf sont ((4, 8) : 2) et ((1, 2) : 1),((5, 6) : 1). Selon à la définition 7, la N-liste de bcf peut être construite comme suit. Tout d'abord, on compare ((4, 8) : 2) avec ((1, 2) : 1). On a trouvé que ((4, 8) : 2) ne peut pas être un ancêtre de ((1, 2) : 1) parce que 4, le pré-ordre des premiers, est plus grand que 1, le pré-ordre de ce dernier. Ensuite, on continue à comparer ((4, 8) : 2) avec ((5, 6) : 1). Évidemment, ((4, 8) : 2) est un ancêtre de ((5, 6) : 1) parce que 4, le pre-order du premier, est inférieur à 5, le pre-order du dernier, et 8, le post-ordre du premier, est supérieur à 6, le post-ordre du second. Alors, ((4, 8) : 1) sont ajoutés à la N-liste de bcf. Parce qu'il n'y a pas d'autres éléments que ((4, 8) : 2) dans la N-liste de bf, le traitement est arrêté. Enfin, on sait que la N-liste de bcf est ((4, 8) : 1). Figure 4 montre l'ensemble de la procédure de traitement.

conclusions : définitions 4, 6 et 7,on a les propriétés suivantes.

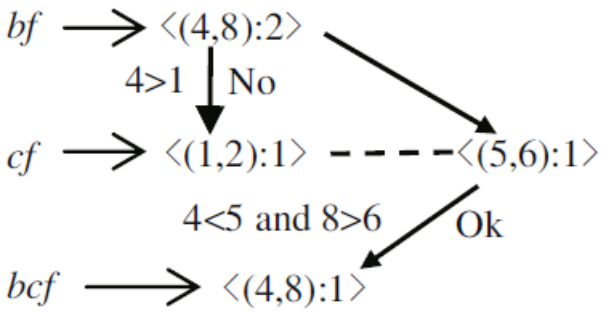


FIGURE 4.4: : N-lists correspondante à l'item bcf du tableau 1

Propriété 4. Soit (x, y) : z est un code PP dans la N-liste d'ensembles de k éléments $i_1 i_2 - - - i_k$. on a les conclusions suivantes :
1) Il doit avoir un nœud d'enregistrement i1 ayant le code PP avec le même (x, y) ;
2) z est le nombre de tous les d'ensembles de k éléments $i_1 i_2 - - - i_k$ dans le sous-arbre avec le nœud correspondant qui enregistre i1 comme racine.
Preuve. On démontrera cette propriété par induction mathématique.
k = 1. Selon la définition 4, on sait que chaque code PP dans la N-liste d'un élément fréquent i représente un nœud qui enregistre i. La propriété 4 est donc vraie pour k = 1. k = 2. D'une part, selon la définition 6, pour tout code PP dans la N-liste de i1 $i_1 i_2$, il doit avoir un code PP dans la N-liste de $i_1$avec le même code (x, y). Et selon la définition 4, chaque code PP dans la N-Liste de i_1 correspond à un nœud qui enregistre i_1. Il doit donc y avoir un nœud d'enregistrement i_1 ayant le code PP avec le même code (x, y). La conclusion 1) est donc vraie. D'un autre côté, selon le processus de fusion de la définition 6, on peut clairement trouver que z de chaque code PP de la N-liste de $i_1 i_2$ est la somme du counts de tous les ensembles d'éléments i1 $i_1 i_2$2 dans le sous-arbre avec le nœud correspondant qui enregistre i1 comme racine. Ensuite, on sait que la Conclusion 2) est également vraie. Par conséquent, la propriété 4 s'applique à k = 2.
Propriété 5.Une N-liste de n'importe quel ensemble k élément $P = i_1 i_2 - - - i_k$, qui est désigné par$\{(x_1, y_1) : z_1, (x_2, y_2) : z_2, ..., .., (x_m, y_m) : z_m\}$, le support des ensembles d élément P est $z_1 + z_2 + - - - + z_m$

Propriété 6. Soit $P = i_1 i_2 - - - i_k$ est un ensemble de k éléments et la N-liste de P est $(x_1, y_1) : z_1, (x_2, y_2) : z_2, ..., .., (x_m, y_m) : z_m$. Ensuite, on a $x_1 < x_2 < - - - < x_m$et $y_1 < y_2 < - - - < y_m$.

Preuve. Dans les définitions 4, 6 et 7, la N-liste de P est une suite de codes PP selon l'ordre croissant de préordre. Par conséquent, on a $x_1 < x_2 < - - - < x_m$. De plus, chaque $(x_j, y_j) : z_j$correspond à un nœud avec item-name = i_1 selon la propriété 4. Par propriété 2, on a $y_1 < y_2 < - - - < y_m$.

4.3 Travaux connexes

On pourrait classer les algorithmes d'extraction de motifs fréquents précédemment proposés en trois groupes : algorithmes avec génération de candidats, algorithmes sans génération de candidats et approches hybrides. Récemment, trois types de structure ont été suggérés afin de représenter l : Node-list[35], N-list[36], et Node set[15, 33], pour faciliter l'extraction d'éléments fréquents. Ils sont basés sur une arborescence de codage des préfixes, qui sauvegardent le minimum d'informations suffisantes pour traiter les données.
Node-List et N-List sont basés sur un arbre PPC, qui est un arbre de préfixes avec chaque nœud codé par son numéro de pré-ordre et son numéro de post-ordre. La N-List (ou Node-List) d'un itemsets est un ensemble de nœuds dans l'arbre PPC. La seule différence entre N-List et Node-List réside dans le fait que la Node-List d'un itemset est constituée de nœuds descendants tandis que N-List est constituée de nœuds ancêtres.
Les N-listes (ou Node-list) ont deux spécifications importantes : Tout d'abord, le support d'un itemset est la somme des compteurs enregistrés dans les nœuds de sa N-liste (ou Node-list). Deuxièmement, la N-list (ou Node-list) d'un ensemble d'éléments (k + 1) peut être formée en joignant les N-lists (ou Node-lists) de son sous-ensemble avec une longueur de k avec une complexité de calcul linéaire. Par rapport aux représentations verticales des données, la taille de la N-list ou de la Node-list est beaucoup plus petite. Par ailleurs, ils sont plus simples et plus flexibles par rapport à l'arbre FP[61]. Par conséquent, les algorithmes basés sur N-List ou Node-List sont très efficaces et surpassent les algorithmes classiques existants, tels que Eclat et FP-growth. Les structures Node-list et N-list ont deux avantages : Le premier est que la longueur de la liste N d'un ensemble d'éléments est beaucoup plus petite que la longueur de sa Node-List. L'autre est que les N-listes ont une propriété appelée propriété à chemin unique, qui peut être utilisée pour extraire directement des ensembles d'éléments fréquents sans générer des ensembles d'éléments candidats dans quelques cas. Celles-ci prouvent l'efficacité de l'algorithme PrePost[36].
Récemment, l'algorithme PrePost a été amélioré en utilisant diverses techniques[34]. Bien que N-list et Node-list soient des structures efficaces pour l'extraction des motifs fréquents, elles doivent inclure des numéros d'ordres, ce qui exige beaucoup de mémoire lorsqu'il s'agit du traitement des grandes bases de données.

Avec l'apparition de grandes données durant ces dernières années, le système mono-machine s'avère incapable de traiter de grandes données. Un grand nombre de recherches ont été réalisées pour l'exploration fréquente de modèles dans un environnement multi-machine, c'est-à-dire un environnement informatique distribué[14]. Hadoop est l'un des principaux cadres de calcul distribué, qui est adopté par de nombreux chercheurs pour l'exploration fréquente de modèles dans les grandes données.

Tableau 4.2: Caractéristiques des méthodes principales de l'état de l'art basées sur Mapreduce

Méthodes	Taille de données intermédiaires	Vitesse de calcul du support	Evolutivité
SPC	Small	Slow	Good
BigFIM	Large	Fast	Bad
PFP	Large	fast	Bad

De nombreuses méthodes basées sur MapReduce ont été suggérées pour trouver des ensembles d'éléments fréquents sur des grandes bases de données. Le tableau 4.2 résume les propriétés des principales méthodes existantes basées sur MapReduce[[108] : SPC, BigFim et PFP. Il existe différentes méthodes distribuées basées sur Apriori sous MapReduce. Lin et al [16] ont proposé trois méthodes Apriori distribuées sur MapReduce : SPC, FPC et DPC. Le SPC exécute itérativement les étapes de génération et de test des candidats sous la forme d'un cycle MapReduce. Lors de la k-ème itération, chaque mappeur lit une partition de la base de données, puis génère les éléments candidats et calcule leur nombre associé à cette partition. Ensuite, l'étape de réduction permet de calculer le support d'un ensemble d'éléments candidats et de le comparer à minsup. Le résultat de l'étape de réduction est diffusé pour être utilisé dans la prochaine itération. FPC réduit le nombre de passages de MapReduce en utilisant la fonction map qui traite les k-itemsets candidats, (k+1)-itemsets, et (k+2)-itemsets en un seul passage MapReduce. DPC rassemble dynamiquement des ensembles d'éléments candidats de longueurs multiples consécutives qui seront traités par les mappeurs en une seule phase MapReduce en fonction de leur nombre. En comparant ces méthodes basées sur Apriori, HPrePostPlus effectue le calcul de support beaucoup plus rapidement à partir de l'intersection des N-listes, en évitant ainsi des comparaisons inutiles.
Moens et al[91] ont proposé BigFIM, qui est une approche hybride entre Apriori et Eclat. Il trouve d'abord des ensembles fréquents d'éléments de courtes longueurs à l'aide de de l'approche Apriori, puis il génère des bases de données conditionnelles

sous forme de classes d'équivalence dont les préfixes sont les ensembles d'éléments trouvés précédemment. Ensuite, il exécute l'algorithme séquentiel Eclat sur chaque base de données conditionnelle dans chaque machine indépendamment. Par rapport au méthodes basées sur Apriori , son calcul de support est rapide grâce à l'utilisation de l'algorithme séquentiel efficace Eclat. Cependant, comme les tailles des bases de données conditionnelles sont très différentes les unes des autres, il provoque une asymétrie de la charge de travail, sans oublier que les tâches de fouille sont souvent interrompues par un manque de mémoire dans une machine donnée, ou prennent une très longue durée en raison de la machine ayant le plus grand temps de réponse. De plus, BigFim génère une grande quantité de données intermédiaires et engendre d'importantes surcharges de communication réseau lors de la génération de bases de données conditionnelles. BigFIM a tendance à montrer une mauvaise évolutivité à mesure que le nombre de machines augmente.
L'algorithme PFP[79] et ses variations[125] sont des méthodes distribuées basées sur l'approche Fp-growth. Ils projettent d'abord une base de données d'entrée et construisent des arbres indépendants, qui sont en quelque sorte des bases de données conditionnelles, en utilisant les bases de données projetées. Ensuite, ils exécutent des opérations d'extraction des motifs fréquents sur chaque arbre FP dans chaque machine indépendamment. Comme BigFIM, PFP et ses variations peuvent trouver des ensembles fréquents d'items à partir des FP-Trees en utilisant un algorithme séquentiel efficace, FP-Growth. Cependant, de la même manière que BigFIM, l'algorithme PFP et ses variantes présentent plusieurs inconvénients tels que l'asymétrie de la charge de travail, la taille importante des données intermédiaires et la surcharge de communication réseau importante. Par conséquent, ils ont tendance à échouer à cause d'un manque de mémoire, et montrent une mauvaise évolutivité. Comparé à BigFIM et PFP, HPrePostPlus montre une meilleure évolutivité à mesure que le nombre de machines augmente, puisqu'il ne contient pas de données intermédiaires et qu'il a une faible surcharge réseau.
Liao et al[81]ont présenté un algorithme MRPrePost, un algorithme parallèle base sur l'algorithme PrePost sur la plate-forme Hadoop. L'algorithme utilise une structure de donnée N-liste en ajoutant un motif de préfixes. Un autre algorithme distribue base sur PrePost sur la plate-forme hadoop a été suggéré par Thakare et al[118].
Comparé aux versions paralleles précédentes de PrePost base sur hadoop, et qui adoptent la méthode d'arborescence générale en utlisant une structure de liste qui est une implémentation de l'interface Liste. Cette structure fournit un accès

séquentiel et efficace pour insérer et supprimer des éléments dans la liste. Cependant, il est devenu moins efficace en accédant aux éléments de la liste. Dans l'algorithme HPrePostPlus, la méthode générale d'arborescence est implémentée avec HashMap qui est une implémentation de l'interface Map, et qui fournit un moyen efficace et rapide pour localiser la valeur en fonction de la clé. Il offre un moyen facile d'accéder et de supprimer des éléments. L'algorithme HPrePostPlus utilise également un HashMap pour améliorer le processus de création des N-listes associées aux ensembles de 1-élément et combine les fonctionnalités de Hadoop afin de traiter des données volumineuses.

4.4 Algorithme HPrePostPlus

4.4.1 Le Design de HPrePostPlus

L'algorithme HPrePostPlus est un algorithme d'extraction de motifs fréquents à partir des données massives ou Big Data, qui utilise la structure de données N-liste pour représenter les données transactionnelles. Toutes les informations requises sur les données doivent être sauvegardées par N-liste. L'efficacité de l'algorithme HPrePostPlus est obtenue en utilisant la méthode de génération d'items fréquents sans génération des éléments candidats. L'algorithme HPrePostPlus utilise également la structure HashMap pour améliorer le processus de création des N-listes associées aux 1-itemsets de l'arbre PPC. L'algorithme HPrePostPlus est implémenté avec Hadoop afin d'améliorer ses performances. Nous stockons les grandes données transactionnelles dans le système de fichiers distribués HDFS du framework Hadoop, et plusieurs partitions de données sont réparties sur les nœuds du cluster. L'algorithme HPrePostPlus est divisé en trois phases, qui sont décrites comme suit :

Phase 1 : Le fichier de données est fourni en entrée au Hadoop, qui divise l'ensemble du fichier d'entrée en blocs de taille fixe appelés shard, et le mappe aux différents DataNode dans le cluster Hadoop. DataNode compte le nombre d'éléments dans chaque bloc. Ensuite,il applique le compte de soutien et il range tous les éléments dans l'ordre décroissant. Ensuite, le réducteur combine les données de tous les DataNode et il génère une liste appelée liste F1. La liste F1 est mappée

à différents DataNode avec le cache distribué. Le fichier d'entrée principal est réorganisé selon la liste F1. Ici, utilise le concept de cache distribué pour comparer deux fichiers avec Map. Ensuite, il génère la liste des 1-itemset fréquents par ordre décroissant appelé liste FL1. Le pseudo-code pour l'ensemble du processus de la phase 1 est présenté dans l'algorithme suivant.

Algorithm 6 Algorithm of parallel statistical 1-frequent itemsets and sort them

input :D = Transactional Dataset, minsup= Minimum Support Threshold, I = item
output :FL_1 = the set of frequent l -itemsets by descending order
1. Procedure Mapper(key,value=T)
2. for each item I in T do
3. Output (key=I, value=1)
4. End
5. End Procedure
6. Procedure Reduce (key=I,value=S(I))
7. Sum=0
8. For each 1 in S(I) do
9. Sum=Sum + 1
10. End
11. If (sum>=minsup) Output(key=I,value=Sum)
12. Then Call function Sort(Fim1)
13. End if
14. Output (FL_1)
15. End Procedure

Ensuite, l'arborescence est parcouru pour déterminer la post-order et pret-prder , puis il utilise le HashMap créé pour accélérer le processus de création des N-listes associées aux éléments 1-fréquent. Le pseudo-code est représenté dans les algorithmes suivantss .

Phase 3 : Les N-listes des 1-items fréquents NL1 sont réparties efficacement sur les nœuds du cluster sous la forme d'un groupe de listes pour avoir un équilibrage de charge sur le cluster. Par exemple à partir de l'arbre PPC de la Fig. 2, on va avoir les groupes suivants :

$$NL1G1 = \{b \rightarrow \{< (4,8) : 4 >\}, f \rightarrow \{< (2,1) : 1 >, < (8,4) : 1 >, < (9,7) : 1 >\}\}, \tag{4.4}$$

Algorithm 7 Algorithm of constructing PPC-Tree and corresponding HashMap

1. Output : PPC –Tree, H_1 the HashMap of FL_1
2. CreateH_1
3. Procedure Mapper(key,value=T)
4. for each Transaction T in D do
5. select the frequent item in T and sort out them according to the order of FL_1

Let the sorted frequent-item list in T be a path [p|P] as the value to output <key, [p|P]>

where p is the first element and P is the remaining list.

6. End for
7. Procedure Reduce (key, [p|P])
8. Create root of a PPC –tree, and label it as "null"
9. For each [p|P]
10. Call $insert_tree$([p|P],T)
11. End for
12. Scan PPC-tree to generate the post-order of each node
13. Return H_1
14. Function $insert_tree$([p|P],T)
15. if T has a child N such that N.item-name = p.item-name
16. then increase N's count by 1 ;
17. else create a new node N, with its count initialized to 1, and add it to T's children-list ;
18. if P is nonempty then call insert tree(P,N) recursively.
19. end if
20. end if

Algorithm 8 Algorithm of generating N-List of 1-frequent itemsets from the HasMap

Input : PPC-tree and $FL_1$the set of frequent 1-itemsets,

H_1 the HashMap of FL_1

Output :NL_1 , the set of the N-lists of frequent 1-itemsets.

1. Procedure N-lists construction (R, H_1)

2. Let C=(R.pre-order,R.post-order,R.count)

3. Add C to H_1 [R,name] count by C.count

4. For each child in R.children do

5. N-lists construction(child)

6. End for

$$NL1G2 = \{c \rightarrow \{< (1,2) : 1 >, < (5,6) : 3 >\}, a \rightarrow \{< (3,0) : 1 >, < (7,3) : 1 >\}\}, , \tag{4.5}$$

$$NL1G3 = \{e \rightarrow \{< (6,5) : 3 >\}\}. \tag{4.6}$$

Nous stockons donc la liste N des 1-itemsets fréquents dans un cache distribué, qui est partagé entre tous les nœuds. Chaque nœud parcourt indépendamment en profondeur chaque élément fréquent du groupe assigné, jusqu'à ce que tous les ensembles d'éléments fréquents avec le sous-arbre de préfixes courant soient identifiés. Par exemple, pour l'élément b dans le groupe 1, le préfixe courant est b, lorsque c et e sont ajoutés à la sous-arborescence du préfixe pour générer 2 ensembles fréquents $\{bc, be\}$(bf et ba ne sont pas des ensembles fréquents). Les ensembles d'éléments fréquents sur b sont alors $\{b, bc, bc, be, bce\}$.
Dans le processus de fusion de sous-arbre de préfixes, normalement quand b et c sont combinés, l'algorithme original génère un PPCode <(b.préordre, b.post ordre) : c.count> , quand la condition est b.préordre <c. pré ordre et b.post ordre> c.post ordre est satisfaite.. Mais, dans notre approche on va générer PP-Code comme suit : <c.preorder, c.postorder) : c.count>. En raison de la profondeur et de la stratégie de sous-arbre de préfixe, nous devons promettre le nouvel élément ajouté et le sous-arbre de préfixe courant sur le même chemin. C'est pourquoi nous générons PPCode comme suit <(b.preorder, b.postorder) : c.count». Le pseudo-code de la phase 3 est présenté dans l'algorithme suivant.

4.5 Expérience

Dans cette section, l'algorithme HPrePostPlus a été comparé avec sa version originale PrePost[36], et trois algorithmes récents de l'état de l'art : negFin[15], MRPrePost[81] et le fameux algorithme PFP[79]. Nous avons évalué la performance de la vitesse en analysant le temps d'exécution et l'évolutivité. Les expériences ont été exécutées sur un cluster Hadoop de trois nœuds, où chaque nœud possède un processeur Intel® Core ™ i5- 3230M CPU@2.60GHz et 12,00 Go de RAM avec le système d'exploitation Ubuntu 12.04 et Hadoop 2.2.0, le système de

Algorithm 9 Algorithm of mining frequent itemsets

Input :$NL_1G_{[i]}$=group i ofNL_1 and shared theNL_1 to be saved in distributed cache
Output : FL_k =frequent k-itemsets F
1. for each mapper do
2. for each NL_l of $NL_1G_{[i]}$ do
3. call $mining_fim_k(NL_l, FL_k, NL_1, minsup)$
4. end for
5. end for
6. Function $mining_fim_k(NL_k, NL_1, minsup)$
7. For i = 0 to NL_1 do
8. if $(NL_k.count >= |DBI| * minsup)$
9.$F = F \cup L_k$
10. if $(NL_kcount >= NL_1[i].count)$
11. Assume$L_k = x_1x_2....x_k, L[i].item = x_{k+1}, supp(x_k) > supp(x_{k+1})$
12.$FL_{k+1} = FL_k + FL_1[i]//FL_{k+1} = x_1x_2....x_kx_{k+1}$
13.$FL_k = FL_{k+1}$
14. compare N-list of$NL_k$with N-list of$NL_1[i]$
15. if $(NL_k.preorder < NL_1[i].postorder \;\&\&\; NL_k.postorder > NL_1[i].preorder|)$
16. $NL_k + l$.N-list.add $(NL_1[i].prepost, NL_1[i].postorder.count) : NL_1[i].count)$
17. end if
18. end If.
19. end if
20. end for

fichier HDFS a été utilisé pour stocker les données transactionnelles.

Les jeux de données T10I4D100K et T40l10D100K servent comme données expé-

Tableau 4.3: Propriétés des ensembles de données de l'expérience

Ensemble de données	Taille	Nombre de Transactions	Nombre d'Items	Longueur moyenne
T10I4D100K	3.8MB	100.000	870	10
T40l10D100K	14 MB	100.000	1000	40

rimentales. Ces deux ensembles de données réelles ont été présentés lors du premier atelier de l'IEEE ICDM sur l'extraction des motifs fréquents (FIMI' 03)[53]. Le tableau 4.3 présente le détail des deux ensembles de données. Les Fig. 4.5 et 4.6 montrent séparément la durée de fonctionnement avec différentes valeurs de support pour les jeux de données T10l4D100K et T40l10D100K. L'axe des abscisses indique la valeur du support et l'axe des ordonnées représente le temps de fonctionnement. Le degré du support passe de 0,1% à 0,5%. La figure 4.5 montre que la performance des algorithmes parallèles HPrePostPlus, MRPrePost et PFP sur de petits ensembles de données n'est pas aussi bonne que celle des algorithmes negFIN et PrePost. La raison c'est que chaque nœud doit envoyer des données

aux autres nœuds du cluster, ce qui rend le retard de la bande passante réseau imprévisible, et affecte le temps d'exécution principal et la performance de l'algorithme. A l'inverse, negFIN et PrePost ont l'avantage de la localisation des données. Lorsque l'ensemble de données est important, les méthodes séquentielles negFIN et PrePost ne peuvent pas compléter l'exécution lorsqu'ils arrivent à un seuil de support, en raison d'une surcharge mémoire. D'autre part, les algorithmes distribués, continuent l'extraction des éléments fréquents, ce qui est le but principal de la parallélisation qui permet de traiter de grands ensembles de données.

Nous pouvons également voir sur la Fig. 4.6 que HPrePostPlus, MRPrePost et

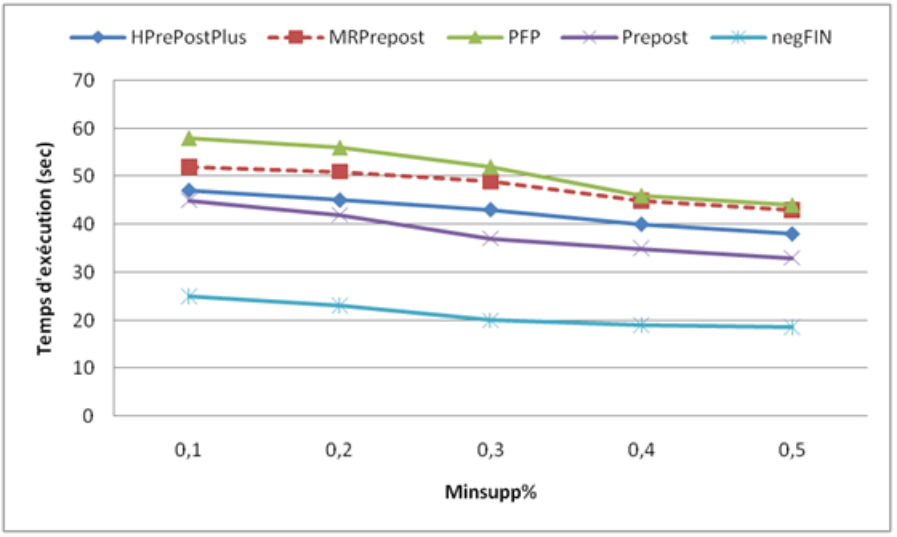


FIGURE 4.5: Temps d'exécution pour l'ensemble T10I4D100K

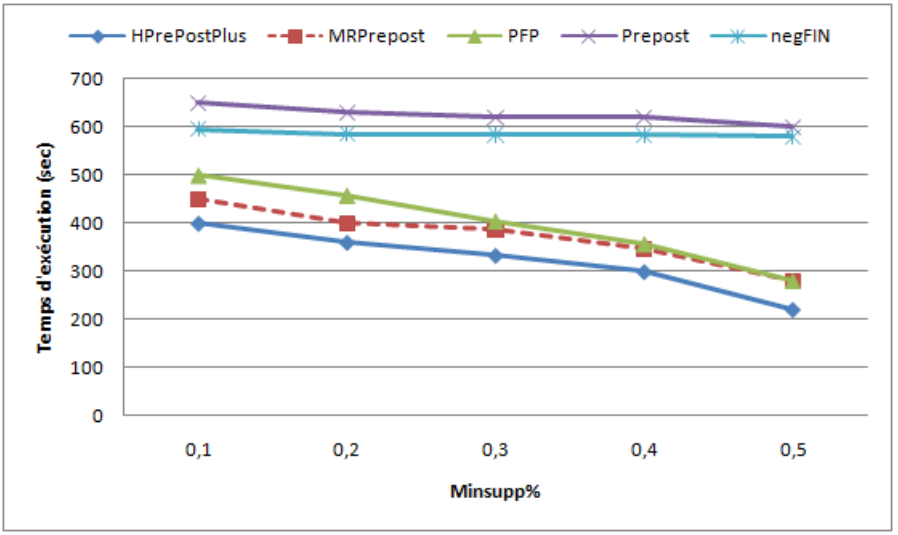


FIGURE 4.6: Temps d'exécution pour l'ensemble T10I4D100K

PFP sont significativement supérieurs à negFIN et PrePost, et ont réalisé une très

bonne performance. La parallélisation consiste à utiliser plusieurs processeurs indépendamment pour traiter des grandes bases de données de sorte que l'algorithme offre des performances supérieures à celles d'un environnement autonome. Les résultats reflètent également, que ce soit sur un grand ou petit ensemble de donnée que la durée d'exécution de HPrePostPlus est plus courte que MRPrePost et PFP, en raison du partage de la mémoire cache sur Hadoop lorsque HPrePostPlus mène une stratégie de profondeur en premier, ce qui réduit le temps de communication. Cependant, l'utilisation de la structure HasMap pour accélérer le processus de création des N-listes associées aux éléments fréquents de l'arbre PPC est très efficace.

Dans la Fig. 4.7, l'axe des abscisses représente le nombre de nœuds du cluster Ha-

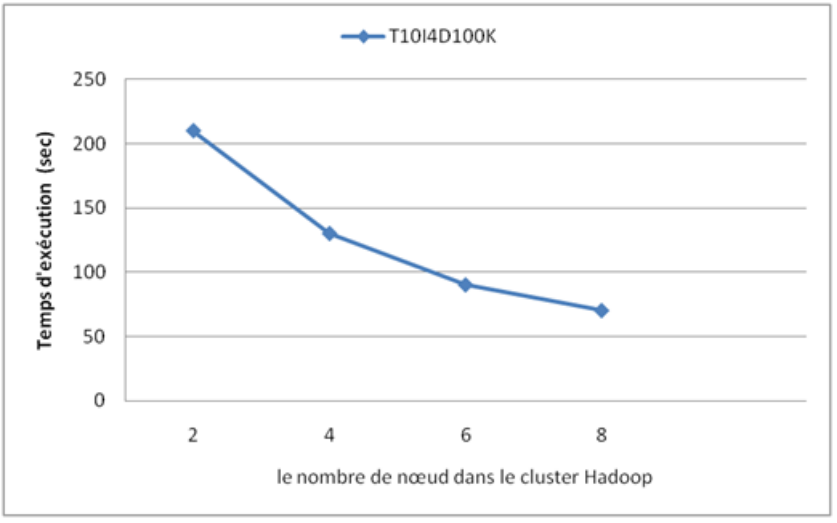


FIGURE 4.7: Temps d'exécution par rapport au nombre de nœuds

doop et l'axe des ordonnées représente la durée de fonctionnement de l'algorithme HPrePostPlus. La figure 4.7 illustre le temps de fonctionnement de HprePostPlus avec différents nombres de nœuds de calcul. Avec plus de nœuds informatiques, HPrePostPlus nécessite moins de temps d'exécution, et la courbe de HPrePostPlus a un déclin presque linéaire. HPrePostPlus présente une caractéristique d'évolutivité quasi linéaire.

4.6 Conclusion

Ce chapitre a proposé l'algorithme HPrePostPlus comme un algorithme efficace pour l'extraction d'ensembles d'éléments fréquents. En premier, nous avons proposé plusieurs améliorations sur l'algorithme PrePost publié précédemment : (i) l'utilisation de la structure HasMap pour améliorer le processus de création des N-listes associées aux 1 éléments fréquents à partir de l'arbre PPC et (ii) la mise en œuvre d'une méthode évolutive basée sur Hadoop pour l'extraction des éléments fréquents sans données intermédiaires et une communications réseau très réduite. HPrePostPlus ne fait pas mieux que negFIN et PrePost en ce qui concerne les petits ensembles de données, mais avec un intervalle de temps qui n'est pas très important .Mais en ce qui concerne les grands ensembles de données, HPrePostPlus est beaucoup plus rapide que ces méthodes séquentielles. Ensuite, la durée d'exécution de HPrePostPlus est toujours plus rapide que MRPrePost et PFP. De plus, les résultats expérimentaux indiquent que l'algorithme proposé est plus évolutif. Pour les travaux futurs, nous nous concentrerons sur l'application de notre approche pour l'extraction des ensembles fréquents fermés et maximaux.

4.7 Conclusion Générale

Cette thèse porte sur l'analyse et l'enrichissement de l'état de l'art de l'une des techniques d'extraction de données les plus adoptées : l'extraction des motifs fréquents. Cette méthode consiste en une analyse exploratoire des données utilisée pour découvrir la co-occurrence fréquente parmi les éléments d'un jeu de données transactionnel. L'extraction des motifs fréquents est une technique très exigeante en termes de ressources. Le problème est dû à la structure de données interne exploitée par les différents algorithmes afin d' explorer l'espace de recherche. L'application de l'extraction fréquente d'éléments dans un environnement Big Data s'avère encore plus complexe, car de plus grandes quantités de données conduisent à de plus grandes structures de données à manipuler et à analyser. Depuis que les techniques centralisées se sont révélées inefficaces, au cours des dernières années, certaines ont été conçues de nouveau pour exploiter des cadres distribués tels que Hadoop MapReduce et Spark. L'une des principales contributions de la thèse est l'analyse de ces techniques. Au moyen d'une analyse théorique et expérimentale structurée, nous avons classifié les travaux proposés en deux familles : les approches

de la répartition de l'espace de recherche et les approches de la répartition des données. Ensuite, nous avons évalué de manière approfondie le comportement et les performances de tous les algorithmes à l'aide d'un ensemble complet et étendu d'expériences. Grâce à cet inspection , nous avons pu identifier les aspects clés du problème et les questions en suspens. Parmi lesquelles et la plus importante posent un manque d'algorithmes conçus pour les données massives -Big Data-. Cela nous amène a concevoir deux approches algorithmiques : HFIMH et HPrePostPlus. Ces algorithmes, qui représentent la deuxième et troisième contribution principales de cette thèse. L'algorithme HFIMH (Hybrid Frequent Itemset Mining on Hadoop) est une approche distribue base sur hadoop mapreduce pour résoudre le problème de traitement itérative des ensembles de données effectuées par les algorithmes parallèles basées sur l'algorithme Apriori, et qui utilise une combinaison entre la disposition verticale et la disposition horizontale des ensembles de données. Afin de faciliter le calcul du support des différents motifs HFIMH adopte une méthode d'intersection des données des ensembles contenant les informations sur les supports des motifs. La troisième contribution présente l'algorithme HPrePostPlus , une version distribuée et améliorée du fameux algorithme PrePost , dont on a changé la structure de donnees en HashMap au lieu du Arraylist, afin d'accélérer le processus de création des listes qui génèrent âpres tous les motifs fréquents en adoptant une méthode d'intersection améliorée. Les algorithmes HFIMH et HPrePostPlus se sont avérés être des solutions fiables pour traiter des ensembles de données massives -big Data- caractérisés par un grand nombre de transactions et d'éléments par transaction.

Dans l'évaluation expérimentale du chapitre 4, nous avons soulignes l'importance des coûts de communication pour l'extraction des motifs fréquents, même dans un environnement avec une grande base de données. Des coûts de communication élevés sont pris en considération afin d'arriver de réaliser une répartition équilibrée de la charge entre les machines indépendantes. La raison est liée à la nature du problème, dans lequel la tâche la plus exigeante est liée à l'exploration de la structure interne des données. Cela implique que les coûts de lecture dominent à peine les performances globales et qu'une priorité plus élevée devrait être accordée à la manipulation des structures internes exploitées pour l'extraction des données. Il pourrait être très intéressant dans le futur de modéliser analytiquement l'échange entre les coûts de communication et d'E/S et la charge assignée à une seule tâche dans le contexte très spécifique de l'extraction des motifs fréquents. L'application de ses approches pour l'extraction des motifs fermes et motifs maximaux est aussi

un sujet très intéressant a développer dans le futur proche.

Bibliographie

[1] Cloudera. `http://www.cloudera.com`, 2015.

[2] Ramesh C Agarwal, Charu C Aggarwal, and VVV Prasad. Depth first generation of long patterns. In *KDD*, pages 108–118, 2000.

[3] Charu C Aggarwal, Mansurul A Bhuiyan, and Mohammad Al Hasan. Frequent pattern mining algorithms : A survey. In *Frequent pattern mining*, pages 19–64. Springer, 2014.

[4] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Database mining : A performance perspective. *IEEE transactions on knowledge and data engineering*, 5(6) :914–925, 1993.

[5] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *Acm sigmod record*, volume 22, pages 207–216. ACM, 1993.

[6] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, A Inkeri Verkamo, et al. Fast discovery of association rules. *Advances in knowledge discovery and data mining*, 12(1) :307–328, 1996.

[7] Rakesh Agrawal and John C Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge & Data Engineering*, (6) :962–969, 1996.

[8] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[9] Rakesh Agrawal, Ramakrishnan Srikant, et al. Mining sequential patterns. In *icde*, volume 95, pages 3–14, 1995.

[10] George S Almasi and Allan Gottlieb. Highly parallel computing. 1989, 1988.

[11] Fabrizio Angiulli, Giovambattista Ianni, and Luigi Palopoli. On the complexity of mining association rules. In *SEBD*, pages 177–184, 2001.

[12] Dario Antonelli, Elena Baralis, Giulia Bruno, Luca Cagliero, Tania Cerquitelli, Silvia Chiusano, Paolo Garza, and Naeem A Mahoto. Meta : Characterization of medical treatments at different abstraction levels. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(4) :57, 2015.

[13] Maria-Luiza Antonie, Osmar R Zaiane, and Alexandru Coman. Application of data mining techniques for medical image classification. In *Proceedings of the Second International Conference on Multimedia Data Mining*, pages 94–101. Springer-Verlag, 2001.

[14] Daniele Apiletti, Elena Baralis, Tania Cerquitelli, Paolo Garza, Fabio Pulvirenti, and Luca Venturini. Frequent itemsets mining for big data : a comparative analysis. *Big data research*, 9 :67–83, 2017.

[15] Nader Aryabarzan, Behrouz Minaei-Bidgoli, and Mohammad Teshnehlab. negfin : An efficient algorithm for fast mining frequent itemsets. *Expert Systems with Applications*, 105 :129–143, 2018.

[16] Marc Barbut. Ordre et classification. 1970.

[17] Roberto J Bayardo Jr. Efficiently mining long patterns from databases. In *ACM Sigmod Record*, volume 27, pages 85–93. ACM, 1998.

[18] C Bhat and CK Bhendadia. Mining big data using modified induction tree approach. *International Journal of Intelligent Engineering and Systems*, 9(2) :14–20, 2016.

[19] Ferenc Bodon. A fast apriori implementation. In *FIMI*, volume 3, page 63, 2003.

[20] Christian Borgelt. Frequent item set mining. *Wiley Interdisciplinary Reviews : Data Mining and Knowledge Discovery*, 2(6) :437–456, 2012.

[21] Dhruba Borthakur. The hadoop distributed file system : Architecture and design. *Hadoop Project Website*, 11(2007) :21, 2007.

[22] Sergey Brin, Rajeev Motwani, and Craig Silverstein. Beyond market baskets : Generalizing association rules to correlations. *Acm Sigmod Record*, 26(2) :265–276, 1997.

[23] Sergey Brin, Rajeev Motwani, Jeffrey D Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. *Acm Sigmod Record*, 26(2) :255–264, 1997.

[24] Douglas Burdick, Manuel Calimlim, and Johannes Gehrke. Mafia : A maximal frequent itemset algorithm for transactional databases. In *ICDE*, volume 1, pages 443–452, 2001.

[25] Toon Calders, Christophe Rigotti, and Jean-François Boulicaut. A survey on condensed representations for frequent sets. In *Constraint-based mining and inductive databases*, pages 64–80. Springer, 2006.

[26] Tania Cerquitelli and Evelina Di Corso. Characterizing thermal energy consumption through exploratory data mining algorithms. In *EDBT/ICDT Workshops*, 2016.

[27] David W Cheung, Jiawei Han, Vincent T Ng, and CY Wong. Maintenance of discovered association rules in large databases : An incremental updating technique. In *Proceedings of the twelfth international conference on data engineering*, pages 106–114. IEEE, 1996.

[28] David W Cheung, Sau Dan Lee, and Benjamin Kao. A general incremental technique for maintaining discovered association rules. In *Database Systems For Advanced Applications' 97*, pages 185–194. World Scientific, 1997.

[29] Kang-Wook Chon and Min-Soo Kim. Bigminer : a fast and scalable distributed frequent pattern miner for big data. *Cluster Computing*, 21(3) :1507–1520, 2018.

[30] Gao Cong, Anthony KH Tung, Xin Xu, Feng Pan, and Jiong Yang. Farmer : Finding interesting rule groups in microarray datasets. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 143–154. ACM, 2004.

[31] Jeffrey Dean and Sanjay Ghemawat. Mapreduce : simplified data processing on large clusters. *Communications of the ACM*, 51(1) :107–113, 2008.

[32] Jeffrey Dean and Sanjay Ghemawat. Mapreduce : a flexible data processing tool. *Communications of the ACM*, 53(1) :72–77, 2010.

[33] Zhi-Hong Deng. Diffnodesets : An efficient structure for fast mining frequent itemsets. *Applied Soft Computing*, 41 :214–223, 2016.

[34] Zhi-Hong Deng and Sheng-Long Lv. Prepost+ : An efficient n-lists-based algorithm for mining frequent itemsets via children–parent equivalence pruning. *Expert Systems with Applications*, 42(13) :5424–5432, 2015.

[35] Zhihong Deng and Zhonghui Wang. A new fast vertical method for mining frequent patterns. *International Journal of Computational Intelligence Systems*, 3(6) :733–744, 2010.

[36] ZhiHong Deng, ZhongHui Wang, and JiaJian Jiang. A new algorithm for fast mining frequent itemsets using n-lists. *Science China Information Sciences*, 55(9) :2008–2030, 2012.

[37] Nele Dexters, Paul Purdom, and Dirk Van Gucht. Analysis of candidate-based frequent itemset algorithms. In *Proc. ACM Symposium on Applied Computing, Data Mining Track*, 2006.

[38] Clarisse Dhaenens and Laetitia Jourdan. *Metaheuristics for big data*. John Wiley & Sons, 2016.

[39] S Dhanya, M Vysaakan, and AS Mahesh. An enhancement of the mapreduce apriori algorithm using vertical data layout and set theory concept of intersection. In *Intelligent Systems Technologies and Applications*, pages 225–233. Springer, 2016.

[40] Ramesh Dharavath, Vikas Kumar, Chiranjeev Kumar, and Amit Kumar. An apriori-based vertical fragmentation technique for heterogeneous distributed database transactions. In *Intelligent Computing, Networking, and Informatics*, pages 687–695. Springer, 2014.

[41] Youcef Djenouri, Habiba Drias, Zineb Habbas, and Hadia Mosteghanemi. Bees swarm optimization for web association rule mining. In *2012 IEEE/-WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, volume 3, pages 142–146. IEEE, 2012.

[42] Wei Fan. and bifet, a. 2012. *Mining Big Data : Current Status, and Forecast to the Future. SIGKDD Explorations*, 14(2) :1–5.

[43] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. *AI magazine*, 17(3) :37–37, 1996.

[44] Bodon Ferenc. A survey on frequent itemset mining. *Technical Report*, 2006.

[45] Alberto Fernández, Sara del Río, Victoria López, Abdullah Bawakid, María J del Jesus, José M Benítez, and Francisco Herrera. Big data with cloud computing : an insight on the computing environment, mapreduce, and programming frameworks. *Wiley Interdisciplinary Reviews : Data Mining and Knowledge Discovery*, 4(5) :380–409, 2014.

[46] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Bay Vo, Tin Truong Chi, Ji Zhang, and Hoai Bac Le. A survey of itemset mining. *Wiley Interdisciplinary Reviews : Data Mining and Knowledge Discovery*, 7(4) :e1207, 2017.

[47] Alex A Freitas. *Data mining and knowledge discovery with evolutionary algorithms*. Springer Science & Business Media, 2013.

[48] K Geetha and Sk Mohiddin. An efficient data mining technique for generating frequent item sets. *International Journal of Advanced Research in Computer Science and Software Engineering*, 3(4) :571–575, 2013.

[49] Liqiang Geng and Howard J Hamilton. Interestingness measures for data mining : A survey. *ACM Computing Surveys (CSUR)*, 38(3) :9, 2006.

[50] Ashish Ghosh and Bhabesh Nath. Multi-objective rule mining using genetic algorithms. *Information Sciences*, 163(1-3) :123–133, 2004.

[51] Arnaud Giacometti, Dominique H Li, Patrick Marcel, and Arnaud Soulet. 20 years of pattern mining : a bibliometric survey. *ACM SIGKDD Explorations Newsletter*, 15(1) :41–50, 2014.

[52] Bart Goethals. Survey on frequent pattern mining. *Univ. of Helsinki*, 19 :840–852, 2003.

[53] Bart Goethals and Mohammed J Zaki. Fimi'03 : Workshop on frequent itemset mining implementations. In *Third IEEE International Conference on Data Mining Workshop on Frequent Itemset Mining Implementations*, pages 1–13, 2003.

[54] Gösta Grahne and Jianfei Zhu. High performance mining of maximal frequent itemsets. In *6th International Workshop on High Performance Data Mining*, volume 16, page 34, 2003.

[55] Gösta Grahne and Jianfei Zhu. Fast algorithms for frequent itemset mining using fp-trees. *IEEE transactions on knowledge and data engineering*, 17(10) :1347–1362, 2005.

[56] Dimitrios Gunopulos, Roni Khardon, Heikki Mannila, Sanjeev Saluja, Hannu Toivonen, and Ram Sewak Sharma. Discovering all most specific sentences. *ACM Transactions on Database Systems (TODS)*, 28(2) :140–174, 2003.

[57] Apache Hadoop. The apache hadoop machine learning library. `http://hadoop.apache.org`, 2013.

[58] Mohammad El Hajj and Osmar R Zaiane. Yafima : Yet another frequent itemset mining algorithm. *Journal of Digital Information Management*, 3(4) :243, 2005.

[59] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. Frequent pattern mining : current status and future directions. *Data mining and knowledge discovery*, 15(1) :55–86, 2007.

[60] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining : concepts and techniques*. Elsevier, 2011.

[61] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *ACM sigmod record*, volume 29, pages 1–12. ACM, 2000.

[62] Liangxiu Han and Hwee Yong Ong. Parallel data intensive applications using mapreduce : a data mining case study in biomedical sciences. *Cluster Computing*, 18(1) :403–418, 2015.

[63] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. The rise of "big data" on cloud computing : Review and open research issues. *Information systems*, 47 :98–115, 2015.

[64] Kamel Eddine Heraguemi, Nadjet Kamel, and Habiba Drias. Multi-swarm bat algorithm for association rule mining using multiple cooperative strategies. *Applied Intelligence*, 45(4) :1021–1033, 2016.

[65] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[66] Jochen Hipp, Ulrich Güntzer, and Gholamreza Nakhaeizadeh. Algorithms for association rule mining- a general survey and comparison. *SIGKDD explorations*, 2(1) :58–64, 2000.

[67] Jochen Hipp, Ulrich Güntzer, and Gholamreza Nakhaeizadeh. Mining association rules : Deriving a superior algorithm by analyzing today's approaches. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 159–168. Springer, 2000.

[68] Hosny M Ibrahim, M Marghny, and NM Abdelaziz. Fast vertical mining using boolean algebra. *Pref*, 6(1), 2015.

[69] Abdullah Imran and Prabhat Ranjan. Improved apriori algorithm using power set on hadoop. In *Proceedings of the First International Conference on Computational Intelligence and Informatics*, pages 245–254. Springer, 2017.

[70] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *European conference on principles of data mining and knowledge discovery*, pages 13–23. Springer, 2000.

[71] Enric Junqué de Fortuny, David Martens, and Foster Provost. Predictive modeling with big data : is bigger really better ? *Big Data*, 1(4) :215–226, 2013.

[72] Karthik Kambatla, Giorgos Kollias, Vipin Kumar, and Ananth Grama. Trends in big data analytics. *Journal of Parallel and Distributed Computing*, 74(7) :2561–2573, 2014.

[73] Mehmet Kaya. Multi-objective genetic algorithm based approaches for mining optimized fuzzy association rules. *Soft computing*, 10(7) :578–586, 2006.

[74] Mohammed Khabzaoui, Clarisse Dhaenens, and El-Ghazali Talbi. Combining evolutionary algorithms and exact approaches for multi-objective knowledge discovery. *RAIRO-Operations Research*, 42(1) :69–83, 2008.

[75] Mika Klemettinen, Heikki Mannila, Pirjo Ronkainen, Hannu Toivonen, and A Inkeri Verkamo. Finding interesting rules from large sets of discovered association rules. In *Proceedings of the third international conference on Information and knowledge management*, pages 401–407. ACM, 1994.

[76] Walter A Kosters and Wim Pijls. Apriori, a depth first implementation. In *FIMI*, volume 3, page 63. Citeseer, 2003.

[77] Chan Man Kuok, Ada Fu, and Man Hon Wong. Mining fuzzy association rules in databases. *ACM Sigmod Record*, 27(1) :41–46, 1998.

[78] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge university press, 2014.

[79] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y Chang. Pfp : parallel fp-growth for query recommendation. In *Proceedings of the 2008 ACM conference on Recommender systems*, pages 107–114. ACM, 2008.

[80] Ning Li, Li Zeng, Qing He, and Zhongzhi Shi. Parallel implementation of apriori algorithm based on mapreduce. In *2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pages 236–241. IEEE, 2012.

[81] Jinggui Liao, Yuelong Zhao, and Saiqin Long. Mrprepost—a parallel algorithm adapted for mining big data. In *2014 IEEE Workshop on Electronics, Computer and Applications*, pages 564–568. IEEE, 2014.

[82] Dao-I Lin and Zvi M Kedem. Pincer-search : A new algorithm for discovering the maximum frequent set. In *International conference on Extending database technology*, pages 103–119. Springer, 1998.

[83] Ming-Yen Lin, Pei-Yu Lee, and Sue-Chen Hsueh. Apriori-based frequent itemset mining algorithms on mapreduce. In *Proceedings of the 6th international conference on ubiquitous information management and communication*, page 76. ACM, 2012.

[84] Xueyan Lin. Mr-apriori : Association rules algorithm based on mapreduce. In *2014 IEEE 5th International Conference on Software Engineering and Service Science*, pages 141–144. IEEE, 2014.

[85] Heikki Mannila, Hannu Toivonen, and A Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data mining and knowledge discovery*, 1(3) :259–289, 1997.

[86] María Martínez-Ballesteros, Alicia Troncoso, Francisco Martínez-Álvarez, and José C Riquelme. Obtaining optimal quality measures for quantitative association rules. *Neurocomputing*, 176 :36–47, 2016.

[87] Viktor Mayer-Schönberger and Kenneth Cukier. *Big data : A revolution that will transform how we live, work, and think*. Houghton Mifflin Harcourt, 2013.

[88] Dinesh P Mehta and Sartaj Sahni. *Handbook of data structures and applications*. Chapman and Hall/CRC, 2004.

[89] Taneli Mielikäinen. Transaction databases, frequent itemsets, and their condensed representations. In *International Workshop on Knowledge Discovery in Inductive Databases*, pages 139–164. Springer, 2005.

[90] Data Mining. Practical machine learning tools and techniques. *URL http ://www. cs. waikato. ac. nz/ml/weka*, 2005.

[91] Sandy Moens, Emin Aksehirli, and Bart Goethals. Frequent itemset mining for big data. In *2013 IEEE International Conference on Big Data*, pages 111–118. IEEE, 2013.

[92] Benjamin Negrevergne, Alexandre Termier, Jean-François Méhaut, and Takeaki Uno. Discovering closed frequent itemsets on multicore : Parallelizing computations and optimizing memory accesses. In *2010 International Conference on High Performance Computing & Simulation*, pages 521–528. IEEE, 2010.

[93] Ibrahim H Osman and Gilbert Laporte. Metaheuristics : A bibliography, 1996.

[94] Matthew Eric Otey, Srinivasan Parthasarathy, Chao Wang, Adriano Veloso, and Wagner Meira. Parallel and distributed methods for incremental frequent itemset mining. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34(6) :2439–2450, 2004.

[95] Valtchev P. A partition-based approach towards building galois (concept) lattices. *Discrete Mathematics*, 256 :801, 2002.

[96] Christos H Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

[97] Jong Soo Park, Ming-Syan Chen, and Philip S Yu. *An effective hash-based algorithm for mining association rules*, volume 24. ACM, 1995.

[98] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Pruning closed itemset lattices for association rules. In *BDA'1998 international conference on Advanced Databases*, pages 177–196, 1998.

[99] Jian Pei, Jiawei Han, Hongjun Lu, Shojiro Nishio, Shiwei Tang, and Dongqing Yang. H-mine : Hyper-structure mining of frequent patterns in large databases. In *Proceedings 2001 IEEE International Conference on Data Mining*, pages 441–448. IEEE, 2001.

[100] Gregory Piateski and William Frawley. *Knowledge discovery in databases*. MIT press, 1991.

[101] Hongjian Qiu, Rong Gu, Chunfeng Yuan, and Yihua Huang. Yafim : a parallel frequent itemset mining algorithm with spark. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 1664–1671. IEEE, 2014.

[102] Balazs Racz. An fp-growth variation without rebuilding the fp-tree. *Budapest : Computer and Automation Research Institute of the Hungarian Academy of Sciences, IEEE ICDM*, 4.

[103] Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2011.

[104] Jorge L Reyes-Ortiz, Luca Oneto, and Davide Anguita. Big data analytics in the cloud : Spark on hadoop vs mpi/openmp on beowulf. *Procedia Computer Science*, 53 :121–130, 2015.

[105] Yassir Rochd and Imad Hafidi. An enhanced apriori algorithm using hybrid data layout based on hadoop for big data processing. *INTERNATIONAL JOURNAL OF COMPUTER SCIENCE AND NETWORK SECURITY*, 18(6) :161–167, 2018.

[106] Yassir Rochd and Imad Hafidi. Parallel implementation of prepost algorithm based on hadoop for big data. In *2018 IEEE 5th International Congress on Information Science and Technology (CiSt)*, pages 24–28. IEEE, 2018.

[107] Yassir Rochd and Imad Hafidi. Performance improvement of prepost algorithm based on hadoop for big data. *The International Journal of Intelligent Engineering and Systems*, 11(5) :226–235, 2018.

[108] Yassir Rochd, Imad Hafidi, and Bajil Ouartassi. A review of scalable algorithms for frequent itemset mining for big data using hadoop and spark. In *First International Conference on Real Time Intelligent Systems*, pages 90–99. Springer, 2017.

[109] Ron Rymon. Search through systematic set enumeration. 1992.

[110] Rajinder Sandhu and Sandeep K. Sood. Scheduling of big data applications on distributed cloud based on qos parameters. *Cluster Computing*, 18(2) :817–828, Jun 2015.

[111] Ashok Savasere, Edward Robert Omiecinski, and Shamkant B Navathe. An efficient algorithm for mining association rules in large databases. Technical report, Georgia Institute of Technology, 1995.

[112] Apache Spark. The apache spark scalable machine learning library. `http://spark.apache.org/mlib`, 2016.

[113] Michael Steinbach, Pang-Ning Tan, Hui Xiong, and Vipin Kumar. Objective measures for association pattern analysis. *Contemporary Mathematics*, 443(2007) :205, 2007.

[114] Yudho Giri Sucahyo and Raj P Gopalan. Ct-pro : A bottom-up non recursive frequent itemset mining algorithm using compressed fp-tree data structure. In *FIMI*, volume 4, pages 212–223, 2004.

[115] Laszlo Szathmary. *Symbolic data mining methods with the Coron platform*. PhD thesis, Université Henri Poincaré-Nancy 1, 2006.

[116] E-G Talbi. A taxonomy of hybrid metaheuristics. *Journal of heuristics*, 8(5) :541–564, 2002.

[117] Joseph Tan. *E-health care information systems : an introduction for students and professionals*. John Wiley & Sons, 2005.

[118] Sanket Thakare, Sheetal Rathi, and RR Sedamkar. An improved prepost algorithm for frequent pattern mining with hadoop on cloud. *Procedia computer science*, 79 :207–214, 2016.

[119] Hannu Toivonen et al. Sampling large databases for association rules. In *VLDB*, volume 96, pages 134–145, 1996.

[120] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. Lcm ver. 2 : Efficient mining algorithms for frequent/closed/maximal itemsets. In *Fimi*, volume 126, 2004.

[121] A.A. Veloso, W. Meira Jr, M.B. de Carvalho, B. Pôssas, S. Parthasarathy, and M. Javeed Zaki. *Mining Frequent Itemsets in Evolving Databases*, pages 494–510.

[122] Wei Wang, Jiong Yang, and S Yu Philip. *Efficient mining of weighted association rules (WAR)*. IBM Thomas J. Watson Research Division, 2000.

[123] Rudolf Wille. Restructuring lattice theory : an approach based on hierarchies of concepts. In *International Conference on Formal Concept Analysis*, pages 314–339. Springer, 2009.

[124] Xindong Wu, Xingquan Zhu, Gong-Qing Wu, and Wei Ding. Data mining with big data. *IEEE transactions on knowledge and data engineering*, 26(1) :97–107, 2014.

[125] Yaling Xun, Jifu Zhang, Xiao Qin, and Xujun Zhao. Fidoop-dp : Data partitioning in frequent itemset mining on hadoop clusters. *IEEE transactions on parallel and distributed systems*, 28(1) :101–114, 2017.

[126] Xin Yue Yang, Zhen Liu, and Yan Fu. Mapreduce as a programming model for association rules algorithm on hadoop. In *The 3rd International Conference on Information Sciences and Interaction Sciences*, pages 99–102. IEEE, 2010.

[127] Run-Ming Yu, Ming-Gong Lee, Yuan-Shao Huang, and Shi-Xuan Chen. An efficient frequent patterns mining algorithm based on mapreduce framework. 2014.

[128] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets : A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[129] Mohammed J Zaki and Karam Gouda. Fast vertical mining using diffsets. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 326–335. ACM, 2003.

[130] Mohammed J Zaki and Ching-Jui Hsiao. Charm : An efficient algorithm for closed itemset mining. In *Proceedings of the 2002 SIAM international conference on data mining*, pages 457–473. SIAM, 2002.

[131] Mohammed Javeed Zaki. Scalable algorithms for association mining. *IEEE transactions on knowledge and data engineering*, 12(3) :372–390, 2000.

[132] Mohammed Javeed Zaki and Mitsunori Ogihara. Theoretical foundations of association rules. In *3rd ACM SIGMOD workshop on research issues in data mining and knowledge discovery*, pages 71–78, 1998.

[133] Zhi-Hua Zhou, Nitesh V Chawla, Yaochu Jin, and Graham J Williams. Big data opportunities and challenges : Discussions from data analytics perspectives. *IEEE Computational Intelligence Magazine*, 9(4) :62–74, 2014.

Publication

• Y.Rochd, I.Hafidi, and B.Ouartassi, " A Review of Scalable Algorithms for Frequent Itemset Mining for Big Data Using Hadoop and Spark", In :Real-Time Intelligent Systems , Advances in Intelligent Systems and Computing, Vol .756, pp.90-99, 2017

• Y.Rochd, I.Hafidi, "An Enhanced Apriori Algorithm Using Hybrid Data Layout Based on Hadoop for Big Data Processing", International Journal of Computer Science and Network Security, Vol .18, No.6, pp.161-176, June 2018.

• Y.Rochd, and I.Hafidi, "Parallel Implementation of PrePost Algorithm Based on Hadoop for Big Data," 5th Edition IEEE CiSt'18, International IEEE Congress on Information Science and Technology , octobre 2018.

• Y.Rochd, and I.Hafidi, "Performance Improvement of PrePost Algorithm Based on Hadoop for Big Data," International Journal of Intelligent Engineering and Systems, vol.11, no.5, pp.226-235, 2018.

• Y.Rochd, I.Hafidi, and B.Ouartassi, "Parallel Implementation of PrePost Algorithm Based on Spark for Big Data", In :International Conference on Big Data and Smart Digital Environment, Vol .53, pp.322-332, 2019.

• Y.Rochd, and I.Hafidi, "An Efficient Distributed Frequent Itemset Mining Algorithm Based on Spark for Big Data," International Journal of Intelligent Engineering and Systems, vol.12, no.4, pp.367-377, 2019.