Parallel itemset mining in massively distributed environments

Saber Salah

THÈSE

Pour obtenir le grade de Docteur

Délivré par l'**Université de Montpellier**

Préparée au sein de l'école doctorale **I2S**[*]
Et de l'unité de recherche **UMR 5506**

Spécialité: **Informatique**

Présentée par **Saber Salah**
saber.salah@inria.fr

Parallel Itemset Mining in Massively Distributed Environments

Soutenue le 20/04/2016 devant le jury composé de :

[*] **I2S:** ÉCOLE DOCTORALE INFORMATION STRUCTURES SYSTÈMES

Résumé

Le volume des données ne cesse de croître. À tel point qu'on parle aujourd'hui de "Big Data". La principale raison se trouve dans les progrès des outils informatique qui ont offert une grande flexibilité pour produire, mais aussi pour stocker des quantités toujours plus grandes.

Les méthodes d'analyse de données ont toujours été confrontées à des quantités qui mettent en difficulté les capacités de traitement, ou qui les dépassent. Pour franchir les verrous technologiques associés à ces questions d'analyse, la communauté peut se tourner vers les techniques de calcul distribué. En particulier, l'extraction de motifs, qui est un des problèmes les plus abordés en fouille de données, présente encore souvent de grandes difficultés dans le contexte de la distribution massive et du parallélisme.

Dans cette thèse, nous abordons deux sujets majeurs liés à l'extraction de motifs : les motifs fréquents, et les motifs informatifs (*i.e.*, de forte entropie).

Les algorithmes d'extraction des motifs fréquents peuvent montrer de mauvaises performances lors du traitement des grandes volumes des données. Ceci est particulièrement le cas lorsque i) les données tendent à être très grandes et/ou ii) le seuil de support minimum est très faible. Dans cette thèse, nous adressons ce problème en faisant appel à des techniques spécifiques de placement des données dans des environnements massivement distribués pour améliorer la performance des algorithmes d'extraction des motifs fréquents. Nous étudions soigneusement l'impact de la combinaison d'un algorithme d'extraction des motifs fréquents avec une stratégie particulière de placement des données. Dans un premier temps, nous montrons que le choix d'une stratégie de placement des données dans un environnement massivement distribué, associé à un algorithme spécifique d'extraction des motifs, a un très fort impact sur le processus d'extraction et peut aller jusqu'à le rendre inopérant. Nous proposons ODPR (Optimal Data-Process relationship) une solution pour l'extraction des motifs fréquents dans MapReduce. Notre méthode permet de découvrir des motifs fréquents dans des grandes bases des données, là où les solutions standard de la littérature ne passent pas à l'échelle. Notre proposition a été évaluée en utilisant des données du monde réel. Nos différents résultats illustrent la capacité de notre approche à passer à l'échelle, même avec un support minimum très faible, ce qui confirme l'efficacité de notre approche.

Sur la base de ce premier résultat, nous avons étendu ce travail en poussant encore un peu les possibilités apportées par le calcul distribué. Généralement, dans un environnement massivement distribué, la performance globale d'un processus est améliorée quand on peut minimiser le nombre de "jobs" (les "aller/retours" entre les machines distribuées).

Cela impacte le temps d'exécution, mais aussi le transfert de données, etc. Dans le cas de l'extraction des motifs fréquents, la découverte des motifs fréquents en un seul job simplifié serait donc préférable. Nous proposons Parallel Absolute Top Down (PATD), un algorithme parallèle d'extraction des motifs fréquents qui minimise ces échanges. PATD rend le processus d'extraction des motifs fréquents dans les grandes bases des données (au moins 1 Téraoctets de données) simple et compact. Son processus d'extraction est constitué d'un seul job, ce qui réduit considérablement le temps d'exécution, les coûts de communication et la consommation énergétique dans une plate-forme de calcul distribué. Faisant appel à une stratégie adaptée et efficace de partitionnement des données nommée IBDP (Item Based Data Partitioning), PATD est capable de fouiller chaque partition des données indépendamment, en se basant sur un seuil de support minimum absolu au lieu d'un seuil relatif. La performance de l'algorithme PATD a été évaluée avec des données du monde réel. Nos résultats expérimentaux suggèrent que PATD est largement plus efficace par rapport à d'autres approches.

Malgré les réponses que les algorithmes d'extraction des motifs fréquents fournissent concernant les données, certaines relations cachées ne peuvent pas être facilement détectées dans les données. Cela est particulièrement le cas lorsque les données sont extrêmement grandes et qu'il faut faire appel à une distribution massive. Dans ce cas, une analyse minutieuse de l'information contenue dans les motifs, mesurée grâce à l'entropie, peut donner plus d'explications et de détails sur les corrélations et les relations entre les données. Cependant, explorer de très grandes quantités des données pour déterminer des motifs informatifs présente un défi majeur dans la fouille des données. Ceci est particulièrement le cas lorsque la taille des motifs à découvrir devient très grande.

Dans un deuxième temps, nous adressons donc le problème de la découverte des motifs informatifs maximales de taille k (*miki* ou "maximally informative k-itemsets) dans les big data. Nous proposons PHIKS (Parallel Highly Informative K-ItemSet), un algorithme pour leur extraction en environnement distribué. PHIKS rend le processus d'extraction de *miki* dans des grandes bases de données simple et efficace. Son processus d'extraction se résume à deux jobs. Avec PHIKS, nous proposons un ensemble de techniques d'optimisation pour calculer l'entropie conjointe des motifs de différentes tailles. Ceci permet de réduire le temps d'exécution du processus d'extraction de manière significative. PHIKS a été évalué en utilisant des données massives de monde réel. Les résultats de nos expérimentations confirment l'efficacité de notre approche par le passage à l'échelle de notre approche sur des motifs de grande taille, à partir de très grandes volumes données.

Titre en français

Fouille de motifs en parallèle dans des environnements massivement distribués

Mots-clés

- Extraction des motifs
- Données distribuées

Abstract

Since few decades ago, the volume of data has been increasingly growing. The rapid advances that have been made in computer storage have offered a great flexibility in storing very large amounts of data.

The processing of these massive volumes of data have opened up new challenges in data mining. In particular, frequent itemset mining (FIM) algorithms have shown poor performances when processing large quantities of data. This is particularly the case when i) the data tends to be very large and/or ii) the minimum support threshold is very low.

Despite the answers that frequent itemset mining methods can provide about the data, some hidden relationships cannot be easily driven and detected inside the data. This is specifically the case when the data is very large and massively distributed. To this end, a careful analysis of the informativeness of the itemsets would give more explanation about the existing correlations and relationships inside the data. However, digging through very large amount of data to determine a set of maximally informative itemsets (of a given size k) presents a major challenge in data mining. This is particularly the case when the size k of the informative itemsets to be discovered is very high.

In this thesis, first we address the problem of frequent itemset mining in big data. We call for specific data placement techniques in massively distributed environments to improve the performance of parallel frequent itemset mining (PFIM) algorithms. We thoroughly study and investigate the impact of combining such a frequent itemset algorithm with a specific data placement strategy. We show that an adequate placement of the data in a massively distributed environment along with a specific frequent itemset mining algorithm can make a mining process either inoperative or completely significant. We propose ODPR (Optimal Data-Process Relationship) our solution for fast mining of frequent itemsets in MapReduce. Our method allows discovering itemsets from massive data sets, where standard solutions from the literature do not scale. Indeed, in a massively distributed environment, the arrangement of both the data and the different processes can make the global job either completely inoperative or very effective. Our proposal has been evaluated using real-world data sets and the results illustrate a significant scale-up obtained with very minimum support which confirms the effectiveness of our approach.

Generally, in a massively distributed environment (*e.g.,* MapReduce or Spark), minimizing the number of jobs results in a significant performance of the process being executed. In the case of frequent itemset mining problem, discovering frequent itemsets in just one simple job would be preferable. To this end, we propose a highly scalable, parallel frequent itemset mining algorithm, namely Parallel Absolute Top Down (PATD).

PATD algorithm renders the mining process of very large databases (up to Terabytes of data) simple and compact. Its mining process is made up of only one parallel job, which dramatically reduces the mining runtime, the communication cost and the energy power consumption overhead, in a distributed computational platform. Based on a clever and efficient data partitioning strategy, namely Item Based Data Partitioning (IBDP), the PATD algorithm mines each data partition independently, relying on an absolute minimum support instead of a relative one. PATD has been extensively evaluated using real-world data sets. Our experimental results suggest that PATD algorithm is significantly more efficient and scalable than alternative approaches.

The second problem which we address in this thesis is discovering maximally informative k-itemsets (*miki*) in big data based on joint entropy. We propose PHIKS (Parallel Highly Informative *K*-ItemSet) a highly scalable, parallel *miki* mining algorithm that renders the mining process of large scale databases (up to Terabytes of data) succinct and effective. Its mining process is made up of only two efficient parallel jobs. With PHIKS, we provide a set of significant optimizations for calculating the joint entropies of the *miki* having different sizes, which drastically reduces the execution time of the mining process. PHIKS has been extensively evaluated using massive real-world data sets. Our experimental results confirm the effectiveness of our proposal by the significant scale-up obtained with high itemsets length and over very large databases.

Title in English

Parallel Itemset Mining in Massively Distributed Environments

Keywords

- Pattern Mining
- Data distribution

Equipe de Recherche

Zenith Team, INRIA & LIRMM

Laboratoire

LIRMM - Laboratoire d'Informatique, Robotique et Micro-électronique de Montpellier

Adresse

Université Montpellier
Bâtiment 5
CC 05 018
Campus St Priest - 860 rue St Priest
34095 Montpellier cedex 5

Résumé étendu

Introduction

Depuis quelques années, il y a eu une forte augmentation dans le volume des données produites et stockées. Les données proviennent de plusieurs sources, telles que les réseaux des capteurs, les réseaux sociaux, etc. Le traitement de ces données, en particulier avec des méthodes de fouille de données, aide à comprendre, interpréter et tirer des conclusions à propos divers phénomènes du monde réel (e.g., en science naturelle, sociale, économique et politique, etc.).

Pour traiter des données aussi volumineuses, une solution consiste à les distribuer sur plusieurs machines et les traiter en parallèle. En fouille de données, cette solution exige une révision profonde des différents algorithmes, pour qu'ils deviennent capables de traiter les données massives en parallèle. La fouille des données représente un ensemble de techniques et méthodes pour analyser et explorer les données. L'extraction de motifs fréquents présente une variante de ces techniques. Il permet de déterminer les motifs qui se répètent fréquemment dans la base des données. La fréquence de co-occurrence des variables d'un motif présente une mesure d'information qui permet de déterminer l'utilité d'un tel motif en se basant sur sa fréquence d'apparition dans la base des données. L'extraction des motifs fréquents connaît de nombreux domaines d'applications. Par exemple, en fouille de texte [1] (comme ce sera illustré dans les chapitres 3 et 4), une technique d'extraction des motifs fréquent peut être utilisée pour déterminer les mots qui se répètent fréquemment dans une grande base des données. En commerce électronique, une telle technique d'extraction des motifs fréquents peut être utilisée pour recommander des produits comme des livres, des vêtements, etc.

Toutefois, dans certaines applications, analyser les données en se basant sur la fréquence de co-occurrences des variables comme une mesure d'information n'aboutit pas forcément à des résultats pertinents. La fréquence de co-occurrences des variables n'aide pas à capturer tous les motifs intéressants et informatifs dans la base des données. En particulier, c'est le cas quand les données sont creuses ou qu'elles répondent à une distribution large. Dans de tels cas, d'autres mesures d'information des motifs peuvent être prises en compte. une mesure d'information intéressante pour les motifs est l'entropie (plus précisément l'entropie pour une variable et l'entropie conjointe pour un motif, qui est un ensemble de variables). Le motif de taille k qui a une valeur d'entropie maximale parmi les autres motifs (de même taille k), serait considéré comme un motif discriminant de taille k (*miki*, ou maximally informative k-itemset). Les items qui composent un tel motif discriminant sont faiblement corrélés entre eux, mais si on les considère tous ensemble, ces items du motif divisent les enregistrements de la base des données de manière optimale. Les motifs discriminants ont des applications dans des domaines différents. Par exemple, en classification, ils peuvent être utilisés pour déterminer les attributs indépendants les plus pertinents dans la base d'apprentissage. Dans cet exemple, un motif discriminant serait un motif (ensemble d'attributs indépendants ou ensemble d'items) qui a une valeur d'entropie maximale par rapport aux autres motifs ayant la même taille k dans la base d'apprentissage.

Avec la disponibilité des modèles de programmation performants comme MapReduce [2] ou Spark [3], le traitement des données de masses devient une tache facile à accomplir. Cependant, la plupart des algorithmes parallèles de la fouille des données souffrent encore de plusieurs problèmes. En particulier, les algorithmes parallèles d'extraction des motifs fréquents souffrent des mêmes limitations que leurs implémentation séquentielle. Ces différentes limitations sont fortement liées à la logique et aux principes de fonctionnement de chaque algorithme. Par exemple, l'implémentation centralisée (i.e., séquentielle) de l'algorithme Apriori [4] demande plusieurs accès au disque. Une version parallèle de cet algorithme, avec une implémentation directe qui considère les jobs MapReduce comme une interface remplaçant les accès au disque, présenterait les mêmes inconvénients (*i.e.*, la multiplication des jobs pour valider les différentes générations d'itemsets serait un goulot d'étranglement). Enfin, bien que l'algorithme FP-Growth [5], a été considéré comme l'algorithme le plus efficace pour l'extraction des motifs fréquents, avec un très faible support minimum et très grand volume des données, sa version parallèle PFP-Growth [6] n'est pas capable de passer à l'échelle à cause de sa consommation en mémoire.

De la même manière, les algorithmes d'extraction des motifs discriminants n'échappent pas à cette difficulté d'adaptation du centralisé vers le parallèle. L'extraction des *miki* en parallèle n'est pas une tâche facile. Le calcul parallèle de l'entropie est coûteux en raison du grande nombre d'accès au disque dont il a besoin. Par exemple, considérons une version parallèle de l'algorithme *ForwardSelection* [7], pour déterminer les *miki*, *ForwardSelection* aurait besoin de k jobs en parallèle.

En plus des problèmes de traitement liés à la découverte de motifs, dans des environnements massivement distribués, la quantité de données transférées peut affecter la performance globale. La conception d'algorithmes d'extraction des motifs fréquents ou discriminants en parallèle doit alors considérer cette question, pour optimiser le coût de communication des données dans les environnements distribués.

Dans la suite, nous présentons des exemples de problèmes qu'un algorithme parallèle de fouille des données pourrait avoir quand il traite des grandes quantités des données. En particulier, nous nous concentrons sur les algorithmes d'extraction des motifs fréquents et les algorithmes d'extraction des *miki*.

Example 1. *Considérons un support minimum très petit, supposons que nous voulons déterminer les motifs fréquents dans une base de données $\mathcal{D}$ très large en utilisant une version parallèle de l'algorithme Apriori. Le nombre de jobs MapReduce serait proportionnel à la taille de motif candidat le plus long dans la base des données $\mathcal{D}$. En général,* *dans un environnement massivement distribué, cette approche qui consiste à un scan multiple de $\mathcal{D}$ aboutit à une mauvaise performance. En particulier, le nombre des données (les motifs candidats) transférées entre les mappers et les reducers serait très grand.*

Maintenant, considérons une version parallèle de l'algorithme FP-Growth pour extraire les motifs fréquents dans la base des données $\mathcal{D}$. Avec le même support (très petit) et une recherche exhaustive de motifs fréquents (le paramètre k prend une valeur infini), l'algorithme souffrirait de plusieurs limitations. La taille de l'arbre FP-Tree pourrait être très grande et donc dépasser la capacité de la mémoire. Si n'est pas le cas, la quantité des données transférées serait très grande ce qui affecte la performance globale du processus d'extraction des motifs fréquents.

Example 2. *Dans cet exemple, supposons que nous voulons déterminer les motifs informatifs maximales de taille k. Considérons une version parallèle de l'algorithme Forward-Selection. Selon la taille k des motifs à découvrir, l'algorithme s'exécuterait en k jobs de MapReduce, offrant une performance très pauvre. De plus, le nombre de motifs candidats serait très grand. Donc, l'extraction parallèle des motifs informatifs maximales de taille k tombe dans les mêmes limitations et restrictions de celles de l'extraction parallèle des motifs fréquents.*

État de L'art

Extraction Parallèle des Motifs Fréquents

Le problème d'extraction des motifs fréquents a été d'abord introduit dans [8]. Dans cette thèse, nous adoptons les notations utilisées dans [8].

Definition 1. *Soit $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ un ensemble qui contient des éléments appelés $items$. Un $Motif$ X est un ensemble d'$items$ de $\mathcal{I}$, i.e. $X \subseteq \mathcal{I}$. La taille $(size)$ de X est le nombre d'$items$ qu'il contient. Une $transaction$ T est un ensemble d'$items$ telle que $T \subseteq \mathcal{I}$ et $T \neq \emptyset$. Une transaction T supporte un $item$ $x \in \mathcal{I}$ si $x \in T$. Une $transaction$ T supporte un $motif$ $X \subseteq \mathcal{I}$ si elle supporte tous les $item$ $x \in X$, i.e. $X \subseteq T$. Une base des données $(database)$ $\mathcal{D}$ est un ensemble des $transactions$. Le $support$ d'un $motif$ X dans la base des données $\mathcal{D}$ est égal au nombre totale de $transactions$ $T \in \mathcal{D}$ qui contiennent X. Un $motif$ $X \subseteq \mathcal{I}$ est dit fréquent $(frequent)$ dans $\mathcal{D}$ si son $support$ est supérieur ou égal à un seuil de support minimum $(MinSup)$. Un $motif$ fréquent maximal est un motif fréquent qui n'est pas inclus dans aucun autre $motif$ fréquent.*

une approche naïve pour déterminer tous les motifs fréquents dans une base des données $\mathcal{D}$ consiste simplement à déterminer le support ($support$) de toutes les combinaisons des items dans $\mathcal{D}$. Ensuite, garder seulement les items/motifs qui satisfont un seuil de support minimum ($MinSup$). Cependant, cette approche est très coûteuse, car elle impose plusieurs accès à la base des données.

Example 3. *considérons la base de données $\mathcal{D}$ qui contient 7 transactions comme illustré par la Figure 2.1. Avec un seuil de support minimum égale à 7, il n'y a pas d'items*

TID	Transaction
1	C, D, E
2	B, C, E
3	A, B, C, E
4	B, E
5	A, B, D
6	A, B, C, E
7	B, C, D, E

FIGURE 1 – Base de données $\mathcal{D}$

fréquents (par conséquent, pas de motifs fréquents). Par contre, avec un seuil de support minimum égal à 5, il y a cinq motifs fréquents : $\{(B), (C), (E), (BE), (CE)\}$

Dans la littérature, plusieurs algorithmes ont été proposés pour résoudre le problème d'extraction des motifs fréquents [4], [9], [5] , [10], [11], [12], [13], etc. Malgré la différence entre leurs principes et logiques, l'objectif de ces algorithmes est d'extraire des motifs fréquents dans une base des données $\mathcal{D}$ en respectant un seuil de support minimum ($MinSup$).

Dans ce qui suit, nous focalisons notre étude sur les algorithmes parallèles d'extraction des motifs fréquents dans les environnements massivement distribués. En particulier, on limite notre étude sur les approches existantes qui seront en relation avec ce travail de thèse.

Apriori : Dans des environnements massivement distribués, une version parallèle de l'algorithme Apriori (i.e., Apriori Parallèle) [14] a montré une performance meilleure que sa version centralisée. Cependant, malgré le parallélisme et la disponibilité de plusieurs ressources, Apriori souffre des mêmes limitations trouvées dans sa version centralisée. Dans un environnement massivement distribué comme MapReduce, en utilisant Apriori, le nombre de jobs nécessaire pour extraire les motifs fréquents est proportionnel à la taille de motif le plus long dans la base des données. Ainsi, avec un seuil de support minimum très petit et de très grand volume des données, la performance d'Apriori en distribué n'est pas satisfaisante. C'est du au fait que le principe de fouille d'apriori est basé sur la génération des motifs candidats qui doivent ensuite être testés, ce qui demande des accès multiples à la base des données. En outre, dans un environnement massivement distribué, Apriori demande un grand nombre de transferts de données entre les mappers et les reducers, et particulièrement lorsque le seuil de support minimum est très petit.

SON : Contrairement à l'algorithme Apriori [4], l'algorithme SON [14], est plus adapté à la distribution. Par exemple, en MapReduce, une version parallèle de SON consiste en deux jobs. Dans le premier job, la base des données est divisée sous forme de partitions des données, chaque mapper fouille une partition en se basant sur un seuil de support minimum local et en utilisant un algorithme d'extraction des motifs fréquents spécifique (e.g., Apriori). Les mappers envoient ces résultats (i.e., les motifs qui sont localement fréquents) aux reducers. Les reducers agrègent les résultats et font la somme des valeurs de chaque clé (i.e., motif) puis écrivent les résultats dans HDFS (Hadoop distributed file system) le système de fichiers distribué de Hadoop. Dans le deuxième job, les motifs qui sont globalement fréquent sont distingués de ceux qui ne sont que localement fréquents. Pour cela, SON vérifie la fréquence d'occurrence de chaque motif qui est localement fréquent (résultat du premier job) dans l'ensemble total des données $\mathcal{D}$.

En comparant la version parallèle de SON avec celle de l'algorithme Apriori, nous remarquons une grande différence. Avec SON, la fouille des motifs fréquents se fait en deux accès à la base des données ce qui minimise considérablement les coûts d'accès à la base des données.

Suivant le principe et la logique de fouille de l'algorithme SON, la performance globale de sa version parallèle dépend fortement de son premier job. Par exemple, quand une partition (i.e., mapper) contient plusieurs transactions similaires (elles partagent de nombreux items en commun), alors le nombre de motifs qui sont fréquents localement dans cette partition est élevé. Dans ce cas, le temps d'exécution d'Apriori sur chaque mapper serait élevé (les fréquents sont longs et nombreux), ce qui impacte la performance globale de SON. En effet, pour passer au deuxième job, il faudra attendre que ce mapper très long finisse d'extraire les itemsets localement.

CDAR : Dans la littérature sur la fouille des données, nous ne trouvons pas de version parallèle de l'algorithme CDAR (Cluster Decomposition Association Rule Algorithm) [13]. Pourtant, une version parallèle de cet algorithme, associée à un partitionnement adapté, pourrait avoir une performance meilleure que d'autres alternatives comme les versions parallèles d'apriori et SON. Le principe de fouille de CDAR ne se base pas sur une approche de génération des motifs candidats comme Apriori par exemple. Avec un seuil de support minimum très petit et des partitions des données homogènes, CDAR donne de bonnes performances. C'est un point que nous étudierons dans cette thèse.

FP-Growth : PFP-Growth est une version parallèle de l'algorithme FP-Growth, proposée dans [6]. PFP-Growth est considéré comme un des algorithmes parallèles d'extraction des motifs fréquents les plus performants dans un environnement massivement distribué. Dans son premier job, PFP-Growth détermine les items qui sont fréquents dans la base des données, ce qui lui permet de construire la F-List (la liste des items fréquents). Dans le deuxième job, l'algorithme construit un arbre FP-tree à partir de la F-List obtenue dans le premier job. Cet arbre sera ensuite fouillé dans les reducers (chaque reducer étant chargé de travailler sur une branche). Le processus de fouille de PFP-Growth se déroule en mémoire, ce qui explique ses bonnes performances.

Malgré ses avantages, avec un seuil de support minimum très petit et de grands volumes des données, PFP-Growth ne passe pas à l'échelle. Ce comportement de PFP-Growth sera mieux illustré par nos expérimentations dans les chapitres 3 et 4. La raison de cette limitation de PFP-Growth est justement liée au fait qu'il travaille en mémoire, ce qui rend les performances dépendantes des capacités mémoire de la machine.

A	B	C	D
1	0	1	0
1	0	1	0
1	0	0	0
1	1	0	0
1	1	1	0
0	0	1	0
0	0	0	0
0	0	0	1

FIGURE 2 – Base des Données Binaire $\mathcal{D}'$

Features	Documents									
	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}
A	1	1	1	1	1	0	0	0	0	0
B	0	1	0	0	1	1	0	1	0	1
C	1	0	0	1	0	1	1	0	1	0
D	1	0	1	1	1	1	1	1	1	1
E	1	1	1	1	1	1	1	1	1	1

TABLE 1 – Features in The Documents

Extraction Parallèle des Motifs Informatifs

Dans un environnement massivement distribué et avec très grand volume des données, la découverte des motifs informatifs maximales de taille k (*miki*) est un défi conséquent. Les approches conventionnelles, qui ont été proposées pour les environnements centralisés, doivent être soigneusement adaptées quand on veut les paralléliser. Malheureusement, dans la littérature, on ne trouve pas de solutions pour l'extraction de *miki* en parallèle. Ainsi, dans cette section, nous limitons notre discussion à l'algorithme *ForwardSelection* [7].

Definition 2. *Soit $\mathcal{F} = \{f_1, f_2, \dots, f_n\}$ un ensemble qui contient des variables, également appelées $features$. Un $motif$ X est un ensemble de variable issues de $\mathcal{F}$, i.e., $X \subseteq \mathcal{F}$. La $taille$ d'un $motif$ X est égale au nombre de variables qu'il contient. Une $transaction$ T est un ensemble d'éléments tels que $T \subseteq \mathcal{F}$ et $T \neq \emptyset$. Une $base$ de $données$ $\mathcal{D}$ est un ensemble de $transactions$.*

Definition 3. *Dans une base de données $\mathcal{D}$, l'entropie [15] d'une variable i mesure la quantité attendue d'information nécessaire pour spécifier l'état d'incertitude ou de désordre de la variable i dans $\mathcal{D}$. Soit i une variable dans $\mathcal{D}$, et $P(i = n)$ la probabilité que i prenne la valeur n dans $\mathcal{D}$ (nous considérons le cas des données catégoriques i.e., la valeur est '1' si l'objet contient la variable et '0' sinon). L'entropie de la variable i est donnée par*

$$H(i) = -(P(i = 0)log(P(i = 0)) + P(i = 1)log(P(i = 1)))$$

où le logarithme est en base 2.

Definition 4. *La projection binaire d'un motif X dans une transaction T ($proj(X,T)$) est un ensemble de taille $|X|$ où chaque item (i.e., variable) de X est remplacé par '1' s'il apparaît dans T et '0' sinon. Le comptage de projection de X dans la base de données $\mathcal{D}$ est l'ensemble des projections de X dans chaque transaction de $\mathcal{D}$, où chaque projection est associée avec son nombre d'occurrences dans $\mathcal{D}$.*

Example 4. *Considérons le Tableau 2.1. La projection de (B,C,D) dans d_1 est $(0,1,1)$. Les projections de (D,E) dans la base de données de Tableau 2.1 sont $(1,1)$ avec 9 occurrences et $(0,1)$ avec une seule occurrence.*

Definition 5. *Soient un motif $X = \{x_1, x_2, \ldots, x_k\}$ et un tuple de valeurs binaires $\mathcal{B} = \{b_1, b_2, \ldots, b_k\} \in \{0\,1\}^k$. L'entropie conjointe de X est définie comme suit :*

$$H(X) = - \sum_{\mathcal{B} \in \{0,1\}^{|k|}} J \times log(J)$$

où J= $P(x_1 = b_1, \ldots, x_k = b_k)$ est la probabilité conjointe de $X = \{x_1, x_2, \ldots, x_k\}$.

Étant donnée une base de données $\mathcal{D}$, l'entropie conjointe $H(X)$ d'un motif X dans $\mathcal{D}$ est proportionnelle à sa taille k. *i.e.,* l'augmentation de la taille de X implique une augmentation dans sa valeur d'entropie conjointe $H(X)$. Une augmentation de la valeur de $H(X)$, l'entropie du motif X, correspond à une augmentation de l'information qu'il contient. Pour simplifier, dans la suite, nous utiliserons le terme entropie pour désigner l'entropie conjointe d'un motif X.

Example 5. *Considérons la base de données du Tableau 2.1. L'entropie conjointe de (D,E) est $H(D,E) = -\frac{9}{10}log(\frac{9}{10}) - \frac{1}{10}log(\frac{1}{10}) = 0.468$. Les quantités $\frac{9}{10}$ et $\frac{1}{10}$ représentent (respectivement) les probabilités jointes des projections $(1,1)$ and $(0,1)$ (respectivement) dans la base de données.*

Definition 6. *Soit un ensemble $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ de variables, un motif $X \subseteq \mathcal{F}$ de taille k est un motif informatif maximal de taille k (ou maximally informative k-itemsetmiki) si, pour tout motif $Y \subseteq \mathcal{F}$ de taille k, $H(Y) \leq H(X)$. Ainsi, un motif informatif maximal de taille k est un motif qui a la plus grande valeur d'entropie conjointe.*

Definition 7. *Étant donnée une base de données $\mathcal{D}$ qui contient un ensemble de n variables $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$. Soit k un entier naturel, le problème de la découverte du miki dans $\mathcal{D}$ consiste à déterminer un sous ensemble $F' \subseteq F$ de taille k, i.e., $|F'| = k$, ayant la plus grande valeur d'entropie conjointe dans $\mathcal{D}$, i.e., $\forall F'' \subseteq F \wedge |F''| = k, H(F'') \leq H(F')$.*

Parallel *ForwardSelection* Algorithm : L'algorithme *ForwardSelection* utilise une approche par niveaux (du type "générer-élaguer" où des candidats de taille $j+1$ sont proposés à partir des motifs de taille j) pour déterminer les *miki* de taille k. Ainsi, une version parallèle de cet algorithme aurait besoin de k jobs pour s'exécuter (un job par génération de candidat). Avec très grand volume des données et une grande valeur de k, la performance de *ForwardSelection* serait alors mauvaise. Comme pour l'extraction de motifs fréquents, cela s'explique par les accès multiples à la base des données, par la génération des *miki* candidats et par la phase de comparaison des valeurs des entropies à chaque étape (par exemple dans un reducer qui se chargerait de récolter et valider les résultats locaux). En outre, une version parallèle de *ForwardSelection* souffrirait d'autres limitations. En particulier, le taux des données échangées entre les mappers et les reducers serait très grand ce qui impacte la performance globale de processus d'extraction des *miki*.

Split	A	B	C	D
S_1	1	0	1	0
	1	0	1	0
	1	0	0	0
	1	1	0	0
	1	1	1	0
S_2	0	0	1	0
	0	0	0	0
	0	0	0	1

TABLE 2 – Partitions des Données

Example 6. *Considérons la base des données $\mathcal{D}'$ illustrée par la Figure 2.5. Supposons que nous voulons déterminer les miki de taille k égale à 2 en utilisant une version parallèle de ForwardSelection. Supposons que la base des données $\mathcal{D}'$ est divisée en deux partitions (splits) comme illustré par le Tableau 2.2. Chaque partition des données (respectivement S_1 et S_2) est traitée par un mapper (respectivement m_1 and m_2). Dans le premier job, chaque mapper traite sa partition des données et envoie chaque item comme clé et sa projection (i.e., combinaison des '0' et '1') comme valeur. Par exemple, m_1 envoie (A, 1) 5 fois au reducer. m_2 envoie (A, 0) 3 fois au reducer (ici, on peut utiliser une simple optimisation qui consiste à envoyer seulement les items qui apparaissent dans les transactions, avec projections des '1'). Puis, le reducer prend en charge le calcul des entropies des items et détermine l'item qui a l'entropie la plus forte. Dans un deuxième job, l'item avec l'entropie la plus forte est combiné avec chaque item restant dans la base des données pour construire des miki candidats de taille k égale à 2. Ensuite, la même processus est lancé pour déterminer le miki de taille 2. Dans cet exemple, le résultat de premier job sera {C} ($H(C) = 1$) et le résultat du second job sera {C A} ($H(CA) = 1.905$). Ce processus d'extraction des miki continue jusqu'à la détermination des miki de taille k. Dans un environnement MapReduce, ce processus utiliserait donc k jobs, ce qui le ren-*

drait très coûteux, en particulier quand k tend vers de grandes valeurs et que le volume des données est grand.

Contributions

L'objectif de cette thèse est de développer des nouvelles techniques pour l'extraction parallèle des motifs fréquents et *miki* dans des environnements massivement distribués. Nos contributions sont comme suivantes.

Optimisation de la relation données-processus pour accélérer l'extraction des motifs fréquents. Dans ce travail, nous étudions l'efficacité et l'influence d'un processus parallèle d'extraction des motifs fréquents avec des stratégies spécifique de placement des données dans MapReduce. En étudiant différent scénarios, nous proposons ODPR (Optimal Data-Process Relationship), notre solution pour l'extraction rapide des motifs fréquents dans MapReduce. Notre méthode, permet la découverte des motifs fréquents dans les données massives, tandis que les autres approches dans la littérature ne passent pas à l'échelle. En effet, dans un environnement massivement distribué, l'organisation des données avec des différents une combinaison mal étudiée d'un placement des données particulier et d'un processus spécifique pour l'extraction des motifs fréquents pourrait rendre le processus d'extraction très peu performant.

Partitionnement des données pour accélérer l'extraction des motifs fréquents. Dans ce travail, nous proposons PATD (Parallel Absolute Top Down), un algorithme parallèle pour l'extraction des motifs fréquents. PATD rend le processus d'extraction des motifs fréquents dans les données massives (au moins 1 Téraoctet de données) simple et compact. PATD fouille une telle base des données en un seul job, ce qui réduit significativement le temps d'exécution, le coût de communication des données et la consommation énergétique dans les plates-formes de calcul distribué. En se basant sur une méthode de partitionnement des données nommée IBDP (Item Based Data Partitioning), PATD fouille chaque partition des données d'une façon indépendante en utilisant une seuil de support minimum absolu à la place d'un support minimum relatif.

Extraction rapide des *miki* dans des données massivement distribuées. Dans ce travail, nous étudions le problème de l'extraction des *miki* en parallèle, en se basant sur le calcul d'entropie. Nous proposons PHIKS ($\underline{\text{P}}$arallel $\underline{\text{H}}$ighly $\underline{\text{I}}$nformative $\underline{K}$-Item$\underline{\text{S}}$et), un algorithme parallèle pour l'extraction des *miki*. Avec PHIKS, nous fournissons plusieurs techniques d'optimisations de calcul de l'entropie, pour améliorer l'extraction des *miki*. Ces différents techniques réduisent considérablement les temps d'exécution, les taux de communication des données, et la consommation énergétique dans les plates-formes de calcul distribué.

Pour toutes ces contributions, nos méthodes ont été testées sur des données massives issues du monde réel. Ces données brutes représente parfois plus d'un Téra-octet de texte à analyser (aucune image). En distribuant ces données de manière massive pour les analyser, nous montrons que :

- les méthodes de l'état de l'art ne passent pas à l'échelle (et quand l'état de l'art ne propose aucune méthode, nous comparons nos approches à une implémentation en parallèle directe des méthodes centralisées).
- nos approches permettent des gains considérables en temps de réponse, en communications et en consommation d'énergie.

Publications

- Saber Salah, Reza Akbarinia, and Florent Masseglia. A Highly Scalable Parallel Algorithm for Maximally Informative k-Itemset Mining. In KAIS : International Journal on Knowledge and Information Systems (accepté)
- Saber Salah, Reza Akbarinia, and Florent Masseglia. Fast Parallel Mining of Maximally Informative k-itemsets in Big Data. In Data Mining (ICDM), 2015 IEEE International Conference on, pages 359–368, Nov 2015.
- Saber Salah, Reza Akbarinia, and Florent Masseglia. Data Partitioning for Fast Mining of Frequent Itemsets in Massively Distributed Environments. In DEXA'2015 : 26^{th} International Conference on Database and Expert Systems Applications, Valencia, Spain, September 2015.
- Saber Salah, Reza Akbarinia, and Florent Masseglia. Optimizing the Data-Process Relationship for Fast Mining of Frequent Itemsets in MapReduce. In MLDM'2015 : International Conference on Machine Learning and Data Mining, volume 9166 of LNCS, pages 217–231, Hamburg, Germany, July 2015.

Organisation de la Thèse

La suite de cette thèse est organisé ainsi :

- Dans le chapitre 2, nous étudions l'état de l'art. Ce chapitre est divisé en trois sections : Dans la Section 2.1, nous étudions les techniques utilisées dans la littérature pour la découverte et l'exploration des connaissances dans les environements centralisés. En particulier, nous focalisons notre étude sur deux sujets : l'extraction des motifs fréquents et l'extraction des motifs informatifs maximaux de taille k. Dans la Section 2.2, nous introduisons les techniques et les méthodes récentes qui ont été proposées pour traiter les données massives. Dans la Section 2.3, nous étudions les approches parallèles qui ont été proposées dans la littérature pour l'extraction des motifs fréquents ce qui sera le sujet des chapitres 3 et 4. Enfin, la découverte des motifs informatifs maximales de taille k sera le thème de chapitre 5.

- Dans le chapitre 3, nous étudions le problème d'extraction des motifs fréquents dans des bases des données de grands volumes et proposons notre contribution pour ce sujet. Dans la Section 3.2, nous proposons P2S notre algorithme parallèle pour l'extraction des motifs fréquents. Dans la Section 3.3, nous évaluons l'efficacité de notre approche en utilisant des données massives de monde réel. Finalement, dans la Section 3.4, nous concluons cette partie du travail et nous discutons des améliorations potentielles.

- Dans le chapitre 4, nous adressons le problème d'extraction des motifs fréquents. Dans la Section 4.2, nous proposons l'algorithme PATD (Parallel Absolute Top Down) et nous expliquons son principe de fonctionnement. Dans la Section 4.3, nous présentons des expérimentations permettant d'évaluer notre approche en utilisant des données massives du monde réel. Finalement, dans la Section 4.4, nous résumons et concluons cette partie de notre travail.

- Dans le chapitre 5, nous étudions le problème d'extraction des motifs informatifs de taille k dans les données massives. Dans la Section 5.3, nous proposons PHIKS (Parallel Highly Informative $\underline{K}$-ItemSet), notre algorithme parallèle pour l'extraction des *miki*. Dans la Section 5.4, nous validons notre approche en utilisant des données massives du monde réel. Dans la Section 5.5, nous concluons cette partie de notre travail.

Conclusion

Dans cette thèse, nous avons abordé deux problème principaux : l'extraction parallèle des motifs fréquents et l'extraction parallèle des motifs informatifs maximales de taille k. Dans ce chapitre, nous avons discuté les problèmes reliés au processus d'extractions des motifs fréquents et des *miki*, en étudiant les approches proposées dans l'état de l'art. Les avantages et les limitations de ces processus d'extraction des motifs sont reliés particulièrement aux accès multiples à la base des données et la capacité de la mémoire. Typiquement, ces différents limitations présentent un défi majeur quand le volume des données est grand et que le support minimum est très petit ou que les motifs à découvrir sont de grande taille.

Contents

Chapter 1

Introduction

1.1 Context

Since few decades ago, there has been a high increase in the volumes of data. For instance, the web log data from social media sites such as Twitter produce over one hundred terabytes of raw data daily [16]. The data comes from everywhere such as sensor networks [17], social networks [18], etc. The storage of these large amounts of data is less challenging than their processing. The data can be distributed and stored in more than one commodity machine. Then, processing these data is carried out in parallel.

Data mining [19], [20] wrappers a set of methods and techniques that allow for analyzing and exploring these data. Frequent itemset mining (FIM for short) presents a variant of these techniques with the aim to determine the itemsets (or, features, patterns, or terms) that frequently co-occur together in the data. The co-occurrence frequency is a measure of informativeness (*i.e.,* interestingness) that helps to measure the utility of the itemsets based on their number of co-occurrences in the data.

FIM has a large range of applications in various domains. For instance, in text mining [1], as it will better illustrated in chapter 3 and chapter 4 of this thesis, FIM can be used to determine the co-occurrence number of words in a very large database. In e-commerce [21], FIM can be used to recommend products such as books, clothing, etc.

In the literature, there have been several different proposed approaches for mining frequent itemsets [22], [5] , [10], [11], [23], etc.

However, for some specific application domains, the co-occurrence frequency measure fails to capture and determine all interesting or informative itemsets in the data. This is particularly the case when the data is sparse and calling for large-scale distribution. To this end, other informativeness measures should be taken into account. One of these interesting measures is the joint entropy of the itemsets. In particular, the itemsets with the maximum joint entropy would be informative. *i.e.,* the items that constitute such an informative itemset have a weak relationship between each other, but together maximally shatter the data. The informative itemsets based on joint entropy measure are of significant use in various domains. For instance, in classification [24], among all available features (*i.e.,* independent attributes), we always prefer a small subset of features (featureset or itemset) that contains highly relevant items to the classification task. A maximally informative k-itemset (*miki* for short) is the informative itemset of size k that has maximum joint entropy.

Recently, with the availability of powerful programming models such as MapReduce [2] or Spark [3], the processing of massive amounts of data has become handy. However, most of the parallel data mining algorithms still suffer from several drawbacks.

Parallel frequent itemset mining (PFIM for short) algorithms have brought the same limitations as in their sequential implementations. These limitations have been primarily related to their core mining principle. For instance, the centralized (*i.e.,* sequential) implementation of the popular Apriori [4] algorithm for mining the frequent itemsets, requires high disc access. Likewise, a parallel version of the Apriori algorithm would require multiple scans of the database, thus a multiple parallel jobs. Although FP-Growth [5] algorithm has been considered as one of the most powerful algorithms for mining frequent itemsets, with very low minimum support and very large amount of data, its parallel version, PFP-Growth [6] cannot scale due to memory issues.

Similarly, the parallel mining of the itemsets based on the joint entropy as an informativeness measure, does not escape the rule from suffering from various drawbacks as for the frequent itemset mining. Mining the *miki* in parallel is not trivial. This is because the parallel computation of the joint entropy is costly due to the high access to the disc. For instance, a parallel version of the popular *ForwardSelection* [7] algorithm would require several parallel jobs to determine the *miki*.

In addition to the regular issues that a parallel data mining algorithm may have when processing massive amounts of data, in massively distributed environments, the quantity of transferred data may impact the whole mining process. Thus, a careful parallel design of these algorithms should be taken into account.

Lets illustrate the potential issues and problems that may happen for a parallel mining algorithm when processing very large amounts of data by using the following examples.

Example 7. *Suppose we are given a very low minimum support, and we want to determine the frequent itemsets in a very large database $\mathcal{D}$ using a parallel version of the popular Apriori algorithm. The required number of the MapReduce jobs would be proportional to the size of the most lengthy candidate itemset. In a massively distributed environment, this would result in a very poor performance since the transferred data (e.g., candidate itemsets) between the mappers and the reducers would be very high. Consider a parallel version of FP-Growth for mining the database $\mathcal{D}$. With very low minimum support and an exhaustive search of the frequent itemsets (i.e., the parameter k set to infinity), the algorithm would suffer from various limitations. First, the FP-Tree may not fit into the memory. Second, if it is not the case, the transferred data would be very high which would highly impact the mining process.*

Example 8. *Suppose we want to determine the maximally informative k-itemsets in parallel. Consider a parallel version of the popular ForwardSelection algorithm. Depending on the size k of the miki to be discovered, the algorithm would perform k MapReduce jobs. This would result in a very poor performance. Beside the multiple database scans,* *the number of itemset candidates would be very high. Thus, the parallel mining of miki falls in the same limitations and restrictions of those of the parallel mining of frequent itemsets.*

1.1.1 Contributions

The objective of this thesis is to develop new techniques for parallel mining of frequent itemsets and *miki* in massively distributed environments. Our main contributions are as follows.

Optimizing the Data-Process Relationship for Fast Mining of Frequent Itemsets in MapReduce. In this work, we study the effectiveness and leverage of specific data placement strategies for improving the parallel frequent itemset mining (PFIM) performance in MapReduce. By offering a clever data placement and an optimal organization of the extraction algorithms, we show that the itemset discovery effectiveness does not only depend on the deployed algorithms. We propose ODPR (Optimal Data-Process Relationship), a solution for fast mining of frequent itemsets in MapReduce. Our method allows discovering itemsets from massive data sets, where standard solutions from the literature do not scale. Indeed, in a massively distributed environment, the arrangement of both the data and the different processes can make the global job either completely inoperative or very effective. Our proposal has been evaluated using real-world data sets and the results illustrate a significant scale-up obtained with very low MinSup, which confirms the effectiveness of our approach.

Data Partitioning for Fast Mining of Frequent Itemsets in Massively Distributed Environments. In this work, we propose a highly scalable, parallel frequent itemset mining (PFIM) algorithm, namely Parallel Absolute Top Down (PATD). PATD algorithm renders the mining process of very large databases (up to Terabytes of data) simple and compact. Its mining process is made up of only one parallel job, which dramatically reduces the mining runtime, the communication cost and the energy power consumption overhead, in a distributed computational platform. Based on a clever and efficient data partitioning strategy, namely Item Based Data Partitioning (IBDP), PATD algorithm mines each data partition independently, relying on an absolute minimum support instead of a relative one. PATD has been extensively evaluated using real-world data sets. Our experimental results suggest that PATD algorithm is significantly more efficient and scalable than alternative approaches.

Fast Parallel Mining of Maximally Informative k-Itemsets in Big Data. In this work, we address the problem of parallel mining of maximally informative k-itemsets (*miki*) based on joint entropy. We propose PHIKS ($\underline{\text{P}}$arallel $\underline{\text{H}}$ighly $\underline{\text{I}}$nformative $\underline{K}$-Item$\underline{\text{S}}$et) a highly scalable, parallel *miki* mining algorithm. With PHIKS, we provide a set of significant optimizations for calculating the joint entropies of *miki* having different sizes, which drastically reduces the execution time, the communication cost and the energy consumption, in a distributed computational platform. PHIKS has been extensively evaluated using massive real-world data sets. Our experimental results confirm the effectiveness of our proposal by the significant scale-up obtained with high itemsets length and over very large databases.

1.1.2 Publications

- Saber Salah, Reza Akbarinia, and Florent Masseglia. A Highly Scalable Parallel Algorithm for Maximally Informative k-Itemset Mining. In KAIS: International Journal on Knowledge and Information Systems (accepted)
- Saber Salah, Reza Akbarinia, and Florent Masseglia. Fast Parallel Mining of Maximally Informative k-itemsets in Big Data. In Data Mining (ICDM), 2015 IEEE International Conference on, pages 359–368, Nov 2015.
- Saber Salah, Reza Akbarinia, and Florent Masseglia. Data Partitioning for Fast Mining of Frequent Itemsets in Massively Distributed Environments. In DEXA'2015: 26^{th} International Conference on Database and Expert Systems Applications, pages 303–318, Valencia, Spain, September 2015.
- Saber Salah, Reza Akbarinia, and Florent Masseglia. Optimizing the Data-Process Relationship for Fast Mining of Frequent Itemsets in MapReduce. In MLDM'2015: International Conference on Machine Learning and Data Mining, volume 9166 of LNCS, pages 217–231, Hamburg, Germany, July 2015.

1.1.3 Road Map

The rest of the thesis is organized as follows.

In chapter 2, we review the state of the art. It is divided into three main sections: In Section 2.1, we give a general overview on the main knowledge discovery techniques in centralized environment. In particular, we deal with two techniques: frequent itemset mining, maximally informative k-itemset. In Section 2.2, we introduce the cutting-edge solutions and techniques that have been used to process massive amounts of data. In Section 2.3, we deal with the basics, recently used parallel techniques for discovering knowledge from large databases. Specifically, we focus on three problems: Parallel frequent itemset mining, which will be the subject of chapter 3 and chapter 4. Parallel mining of maximally informative k-itemset, which will be the focus of chapter 5.

In chapter 3, we deal with the problem of frequent itemset mining in large databases. In Section 3.2, we propose our solution Parallel Two Steps (P2S) algorithm and we thoroughly depict its core mining process. In Section 3.3, we asses the efficiency of our proposed approach by carrying out extensive experiments with very large real-world data sets. Finally, in Section 3.4, we summarize our work and we discuss potential further improvements.

In chapter 4, we address the problem of frequent itemset mining in very large databases. In Section 4.2, we propose our algorithm Parallel Absolute Top Down (PATD) and we thoroughly explain its mining principle. In Section 4.3, we validate our proposal through extensive, different experiments using very large real-world data sets. Eventually, in Section 4.4, we conclude our work.

In chapter 5, we deal with the problem of mining maximally k-itemsets in big data. In Section 5.3, we propose our PHIKS algorithm for *miki* parallel discovery. We thoroughly detail its mining principle. In Section 5.4, we validate our approach by carrying out various, extensive experiments using very massive real-world data sets. Finally, in Section 5.5, we summarize our work.

Chapter 2

State Of The Art

In this chapter, we introduce the basics and the necessary background of this thesis. First, we present the general concept of knowledge discovery (KD) [25]. In particular, we introduce the problem of the frequent itemset mining (FIM) and we discuss the main recent, existing techniques and methods that have been proposed to solve them. In addition, we address the problem of mining maximally informative k-itemsets (*miki*) and we discuss the different approaches and techniques that have been proposed to handle them.

Second, we investigate and detail the different working processes of the recent, existing parallel and distributed mining algorithms. In particular, we address the problem of parallel mining of frequent itemsets and maximally informative k-itemsets in massively distributed environments.

2.1 Knowledge Discovery

Knowledge discovery [25] is the whole process of identifying new, potentially useful patterns in the data. Data mining (DM) [19], [20] presents a core step of a knowledge discovery process. It wrappers a set of techniques and methods that allow extracting new knowledge from the data.

In this Section, we discuss the state of the art of these different techniques. In particular, we present the problem of mining frequent itemsets and maximally informative k-itemsets and we discuss the main methods and techniques that have been proposed in the literature to solve them.

2.1.1 Frequent Itemset Mining

The problem of frequent itemset mining (FIM for short) was first introduced in [8]. In this thesis, we adopt the notations used in [8].

Definition 8. *Let $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ be a set of literals called $items$. An $Itemset$ X is a set of $items$ from $\mathcal{I}$, i.e. $X \subseteq \mathcal{I}$. The $size$ of the $itemset$ X is the number of $items$ in it. A $transaction$ T is a set of elements such that $T \subseteq \mathcal{I}$ and $T \neq \emptyset$. A transaction*

T supports the $item$ $x \in \mathcal{I}$ if $x \in T$. A $transaction$ T supports the $itemset$ $X \subseteq \mathcal{I}$ if it supports any $item$ $x \in X$, i.e. $X \subseteq T$. A $database$ $\mathcal{D}$ is a set of $transactions$. The $support$ of the $itemset$ X in the $database$ $\mathcal{D}$ is the number of $transactions$ $T \in \mathcal{D}$ that contain X. An $itemset$ $X \subseteq \mathcal{I}$ is $frequent$ in $\mathcal{D}$ if its $support$ is equal or higher than a given $MinSup$ threshold. A $maximal\ frequent\ itemset$ is a frequent itemset that has no frequent superset.

TID	Transaction
1	C, D, E
2	B, C, E
3	A, B, C, E
4	B, E
5	A, B, D
6	A, B, C, E
7	B, C, D, E

Figure 2.1 – Database $\mathcal{D}$

A naive approach to determine all frequent itemsets in a database $\mathcal{D}$ simply consists of determining the $support$ of all items combinations in $\mathcal{D}$. Then, retain only the items/itemsets that satisfy a given minimum support $MinSup$. However, this approach is very expensive, since it results in a high number of I/O disc access (*i.e.,* database scans).

Example 9. *Let us consider a database $\mathcal{D}$ with 7 transactions as shown in Figure 2.1. With a minimum support of 7, there will be no frequent items (and no frequent itemsets). With a minimum support of 5, there will be 5 frequents itemsets:* $\{(B), (C), (E), (BE), (CE)\}$

In the literature there have been various proposed algorithms to solve the problem of frequent itemset mining [22], [5], [10], [11], [23], etc. Despite their different logic and working principles, the main purpose of these algorithms is to extract all frequent itemsets from a database $\mathcal{D}$ with a minimum support $MinSup$ specified as a parameter. In the following, we discuss the main frequent itemset mining algorithms that have been proposed in the literature.

Apriori Algorithm: Apriori algorithm was first introduced in [4]. Its main motivation was to reduce the I/O disc access when mining frequent itemsets. To this end, Apriori algorithm relies on an anti-monotonicity criterion. *i.e.,* if an item/itemset is not frequent, then all of its super-sets cannot be frequent. To extract the frequent itemsets, Apriori scans the database $\mathcal{D}$ and determines a candidate list C_1 of frequent items of size one, then the algorithm filters C_1 and keeps only the items that satisfy the minimum support and stores them in a list L_1. From L_1, the algorithm generates candidate itemsets of size two in a list say C_2 and that by combining all pair of frequent items of size one in L_1. Then, Apriori scans $\mathcal{D}$ and determines all itemsets in C_2 that satisfy the minimum support, the result is stored in a list L_2. The mining process of Apriori is carried out until there is no more candidate itemsets in $\mathcal{D}$ to be checked.

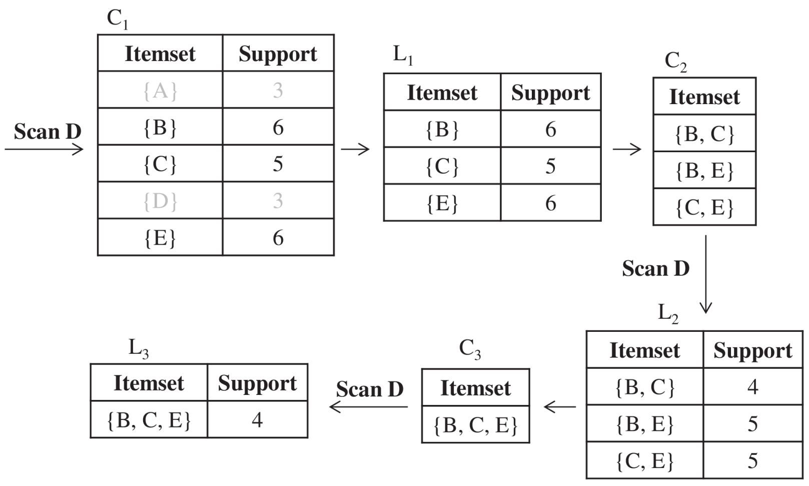


Figure 2.2 – Frequent itemset mining using Apriori

Despite the anti-monotonicity property, Apriori has shown several drawbacks. This is particularly the case when the minimum support is very low (which implies a huge number of frequent itemset candidates). In this case, Apriori algorithm would perform multiple database scans until determining the count of the most lengthy itemset in the database. This behaviour of Apriori algorithm would result in a poor performance. The performance of Apriori algorithm is proportional to its number of itemset candidates to be checked against the database.

Example 10. *Figure 2.2 shows a working example of Apriori algorithm over the database $\mathcal{D}$ of Figure 2.1. In this example, an itemset is frequent, if it occurs at least 4 times in the database $\mathcal{D}$. After the first database scan, we have three frequent items ({B}, {C}, {D}) in L_1. An itemset candidate generation step is carried out to build the candidate itemsets of size two ({B, C}, {B, E}, {C, E}) in C_2. From C_2 a list of frequent itemsets of size two L_2 is returned by filtering the itemset candidates in C_2 based on their support count. Then, from the list L_2, An itemset candidate step generation is executed to determine the candidate itemsets of size three ({B, C, E}) in C_3. Finally, a last database scan is performed to filter C_3 and keeps only the frequent itemsets of size three in L_3. Since, there is no more candidate itemsets, the algorithm stops.*

In example 10, with a support of 4, Apriori performs 3 database scans in total to determine all frequent itemsets of size 3. With a low minimum support, there would be more frequent itemsets in $\mathcal{D}$ which implies more database scans. For example, if there are 10^4 frequent items, then Apriori algorithm will need to generate more than 10^7

candidate itemsets of size 2 and the supports of all of these candidates would be checked in $\mathcal{D}$. Thus, the overall performance of Apriori algorithm is highly impacted when the minimum support is low (*i.e.,* high number of candidate itemsets).

SON Algorithm: This algorithm firstly was introduced in [9], to extract the set of frequent itemsets in two steps. The mining principle of SON is drawn from the fact that the set of all global frequent itemsets (*i.e.,* all frequent itemsets in $\mathcal{D}$) is included in the union of the set of all local frequent itemsets. To determine the set of frequent itemsets (*i.e.,* global frequent itemsets), SON proceeds by performing a mining process in two phases as following.
Phase 1: Divide the input database $\mathcal{D}$ into n data partitions, $\mathcal{D} = \{P_1, P_2, \dots, P_n\}$ in a way that each P_i in $\mathcal{D}$ fits into the available memory. Then, mine each data partition P_i in the memory based on a local minimum support $LMinSup$ (*i.e.,* the local minimum support is computed based on the number of transactions in P_i and the given global minimum support $GMinSup$) and a specific FIM algorithm (*e.g.,* Apriori algorithm or one of its improvements). Thus, SON algorithm first phase is devoted to determine a list of local frequent itemsets LFI.
Phase 2: This phase proceeds by filtering the local frequent itemsets in LFI list based on the global minimum support $GMinSup$. This step is carried out to validate the global frequency of the set of local frequent itemsets. SON algorithm scans the whole database $\mathcal{D}$ and checks the frequency of each local frequent itemset in LFI. Then, it returns a list of global frequent itemsets (GFI) which is a subset of LFI *i.e.,* $GFI \subseteq LFI$.

Since it performs two database scans, SON has shown better performance than Apriori algorithm. However, the main limitation of this algorithm is its first mining phase. *i.e.,* in the case when a data partition contains a high number of local frequent itemsets, in this case, the performance of the second phase would be impacted too.

Eclat Algorithm: To avoid the I/O disc access in mining frequent itemsets, Eclat algorithm [12] consists of performing a mining process in the memory without accessing the disc. The algorithm proceeds by storing a list of transactions identifiers (tid) in the memory for each item in the database. To determine the $support$ of an itemset X of any size k, Eclat intersects the tids of all items in X. To traverse the tids, it uses different techniques such as top down, bottom up and hybrid techniques.

Despite its efficiency (*i.e.,* speed up) in counting the support of the itemsets, the main bottleneck of Eclat algorithm is the size of the tids. When the size of the transaction identifiers is very large, then, they cannot fit into the memory.

Cluster Decomposition Association Rule Algorithm: Cluster Decomposition Association Rule (CDAR for short) [13] algorithm uses a simple, yet efficient principle for mining frequent itemsets. It performs the mining process in two steps as follow.

Step 1: CDAR divides the database $\mathcal{D}$ into $|P| = n$ data partitions $\mathcal{D} = \{P_1, P_2, \ldots, P_n\}$. Each data partition P_i in $\mathcal{D}$ only holds the transactions whose length is i, where the length of a transaction is the number of items in it.

Step 2: CDAR starts mining the data partitions according to the transaction lengths in decreasing order. A transaction in each data partition accounts for an itemset. If a transaction T is frequent in a data partition P_{i+1} then, it will be stored in a list L of frequent itemsets, otherwise, CDAR stores T in a temporary data structure Tmp. Then, after checking the frequency of T in P_{i+1}, CDAR generates the i-length subsets of all T in Tmp and adds them to the data partition P_i. The same mining process is carried out until visiting all partitions $P_i \subset \mathcal{D}$. Before counting the $support$ of a transaction T, CDAR checks its inclusion in L, and if it is included, then CDAR does not consider T, as it is already in L which means it is frequent.

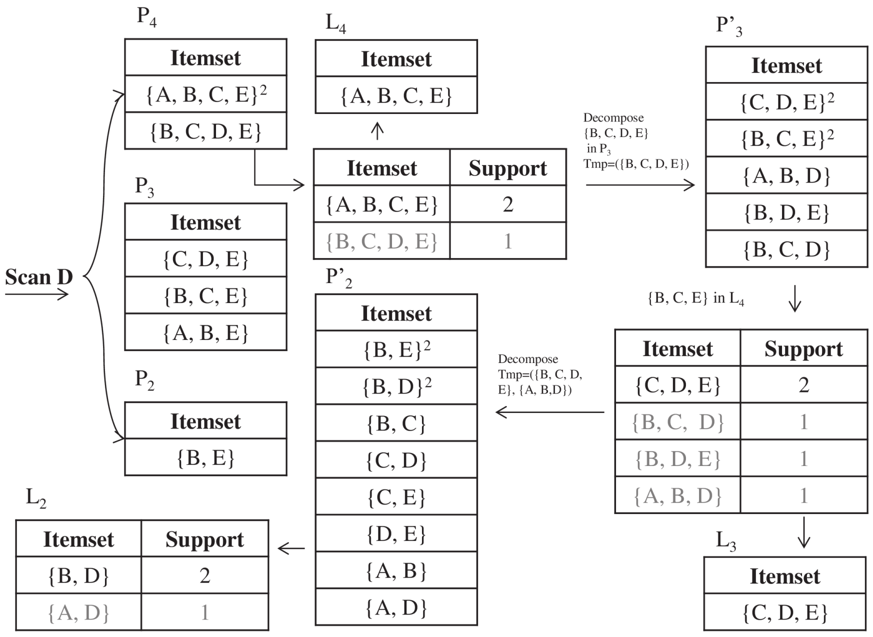
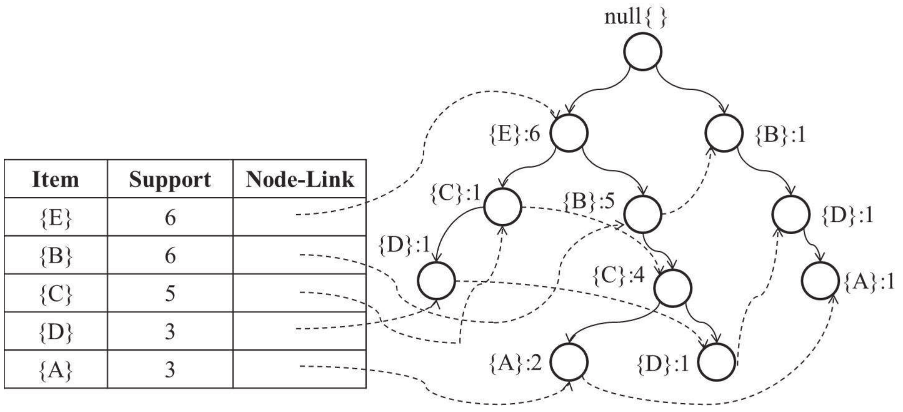


Figure 2.3 – Frequent itemset mining using CDAR

Example 11. *Given a database $\mathcal{D}$ as shown in Figure 2.1, in this example, we consider an itemset to be frequent, if it occurs at least 2 times in $\mathcal{D}$. Figure 2.3 illustrates the mining process of CDAR. The algorithm scans the database $\mathcal{D}$ and creates the data partitions (i.e., clusters or groups of transactions). In this example we have 3 different data partitions P_4, P_3 and P_2. Each one of these data partitions holds a set of transactions having the same length. CDAR starts mining the data partition having the most lengthy transaction, which is in this example P_4. The itemset (i.e., transaction) {A, B, C, E} is frequent because it appears two times in P_4. Thus, {A, B, C, E} is stored in a list L_4. on the other side, the itemset {B, C, D, E} is not frequent it occurs once in P_4. Hence, the itemset {B,* *C, D, E} is added to the temporary data structure Tmp and its decomposition to subsets of size 3 is added to P'_3. Since the itemset {B, C, E} is a subset of {A, B, C, E} which is already frequent in L_4, CDAR does not consider {B, C, E} when counting the supports of the itemsets of size 3. The algorithm determines {C, D, E} as a frequent itemset in P'_3 and stored it in L_3. The itemsets {B, C, D}, {B, D, E} and {A, B, D} are not frequent, however, since the itemsets {B, C, D} and {B, D, E} are already included in the itemset {B, C, D, E} in Tmp, CDAR adds only {A, B, D} to Tmp. Then, the algorithm generates the subsets of size two of each itemset in Tmp and adds them to P'_2. In this example, the itemsets {B, E}, {B, C}, {C, D}, {C, E}, {D, E} and {A, B} are already in $L_3 \cup L_4$, then CDAR does not consider their support count. Finally, the algorithm determines the itemset {B, D} as a frequent itemset. The algorithm stops because there is no more data partitions to visit.*



Item	Support	Node-Link
{E}	6	
{B}	6	
{C}	5	
{D}	3	
{A}	3	

Figure 2.4 – Frequent itemset mining using FP-Growth

FP-Growth Algorithm: FP-Growth (Frequent Pattern Growth) algorithm [5] has been considered as the most powerful technique in mining frequent itemsets. The popularity that FP-Growth algorithm has gained is related to its none-candidate generation feature. Unlike previously mentioned techniques, FP-Growth algorithm does not rely on any itemset candidate generation approach. To determine the frequent itemsets, FP-Growth accesses the database two times *i.e.,* to filter out the none-frequent items and compress the whole database in an FP-Tree structure. Once the FP-Tree is built, the algorithm uses a recursive divide and conquer approach to mine the frequent itemsets from the FP-Tree. Thus, FP-Growth algorithm performs two database scans. The following describes each pass over the database.

Pass 1: Same as Apriori algorithm, FP-Growth's first pass over the database $\mathcal{D}$ is devoted to determine the $support$ count of each item in $\mathcal{D}$. The algorithm retains only the frequent items in a list L. Then, FP-Growth sorts L in a descending order according to the $support$ counts of the items.

Pass 2: The algorithm creates the root of the tree, labeled with "null", then scans the database $\mathcal{D}$. The items in each transaction are processed in L order (*i.e.,* sorted according to descending $support$ count), and a branch is created for each transaction. Each node in the FP-Tree accounts for an item in L and each node is associated with a counter (*i.e.,* $support$ count initialized to 1). If a transaction shared a common *prefix* with another transaction, then the $support$ count of each visited node is incremented by 1. To facilitate the FP-Tree traversal, an item header table is built so that each item points to its occurrences in the tree via a chain of node-links.

Mining FP-Tree: The FP-Tree is mined as follows. Start from each frequent item (*i.e.,* pattern) of size 1 (as an initial suffix pattern), construct its conditional pattern base (a "sub-database," which consists of the set of prefix paths in the FP-Tree co-occurring with the suffix pattern), then construct its (conditional) FP-Tree, and perform the mining recursively on the tree. The pattern growth is achieved by the concatenation of the suffix pattern with the frequent patterns generated from a conditional FP-tree.

Example 12. *Let us consider the database $\mathcal{D}$ shown in Figure 2.1. In this example, we consider an itemset to be frequent, if it occurs at least 2 times in $\mathcal{D}$. After the first pass over the database $\mathcal{D}$, FP-Growth returns the list L of frequent items, in this example, we have* $L = \{\{A : 3\}, \{B : 6\}, \{C : 5\}, \{D : 3\}, \{E : 6\}\}$. *Then, it sorts the list L of frequent items in descending order according to their $support$ counts. Hence, the list L becomes* $L = \{\{E : 6\}, \{B : 6\}, \{C : 5\}, \{D : 3\}, \{A : 3\}\}$. *Then, by scanning the database $\mathcal{D}$, an FP-Tree is constructed according to the order of items in L. The left part of Figure 2.4 shows a header table that contains the information about each node in the constructed FP-Tree (right part of Figure 2.4) of our example. To mine the constructed FP-Tree, we consider the last item in L {A}. The item {A} occurs in two FP-Tree branches (Right side of Figure 2.4). The occurrences of {A} can easily be found by its chain of node links. The paths formed by these branches are < {E}, {B}, {C}, {A}: 2 > and < {B}, {D}, {A}: 1 >. Considering the item {A} as a suffix, its corresponding two prefix paths are < {E}, {B}, {C}: 2 > and < {B}, {D}: 1 > which form its conditional pattern base. Using this conditional pattern base as a transaction database, FP-Growth builds an {A}-conditional FP-Tree which contains only a single path < {E}, {B}, {C}: 2 >. Here the item {D} is not included because its $support$ count is 1 (not frequent). The single path < {E}, {B}, {C}: 2 > generates all frequent itemsets that involves the item {A} ({A, B, C, E}: 2, {A, B ,C}: 2, {A, B, E}: 2, {A, C, E}: 2, {A, B}: 3, {A, C}: 2, {A, E}: 2). Likewise, the item {D} occurs in 3 FP-Tree branches. The paths formed by these 3 branches are < {E}, {C}, {D}: 1 >, < {E}, {B}, {C}, {D}: 1 > and < {B}, {D}: 1 >. Considering the item {D} as a suffix, then its corresponding prefix paths are < {E}, {C}: 1 >, < {E}, {B}, {C}: 1 > and < {B}: 1 > which form its conditional pattern base. FP-Growth builds an {D}-conditional FP-Tree which contains two paths < {E}, {C}: 2 > and <{B}: 2 >. From these two paths, FP-Growth generates all frequent itemsets involving the item {D} ({C, D, E}: 2, {B, D}: 2). Same for th item {C}, its {C}-conditional FP-Tree contains two paths < {E}: 5 > and <{B}: 4 >, FP-Growth generates the frequent itemsets involving the item {C}, ({C, E}: 5, {B, C}: 4). For the item {B}, its {B}-conditional FP-Tree contains a single path < {E}: 5 >, thus the frequent itemset involves {B} is ({B, E}: 5). The item {E} does not have any prefix, then the algorithm stops.*

Features	Documents									
	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}
A	1	1	1	1	1	0	0	0	0	0
B	0	1	0	0	1	1	0	1	0	1
C	1	0	0	1	0	1	1	0	1	0
D	1	0	1	1	1	1	1	1	1	1
E	1	1	1	1	1	1	1	1	1	1

Table 2.1 – Features in The Documents

In the literature, there are many alternatives and extensions to the FP-growth approach, including depth-first generation of frequent itemsets [26], H-Mine [27]. Authors of [28] explore a hyper-structure mining of frequent patterns; building alternative trees; exploring top-down and bottom-up traversal of such trees in pattern-growth mining by Liu et al. (2002, 2003) and an array-based implementation of prefix-tree-structure for efficient pattern growth mining [29].

Generally, using very low minimum support results in a high number of frequent itemsets. Therefore, the number of rules generated from the set of frequent itemsets are very large and they are hard to be interpreted and analyzed by an expert. To overcome this problem, such a solution stands for mining only the closed frequent itemsets (CFI in short) [30]. The set of closed frequent itemsets presents a generator to all other frequent itemsets. In the literature, there have been several approaches [31], [32], [33], [34], proposed to mine the closed frequent itemsets. Existing algorithms for mining CFI flag out good performance when the input dataset is small or the support threshold is high. However, when the size of the database grows or the support threshold turns to be low, both memory usage and communication costs become hard to bear

2.1.2 Maximally Informative K-Itemsets Mining

The co-occurrence frequency of the itemsets (or featureset) in the database does not give much information about the hidden correlations between the itemsets. For instance, an itemset say $\{A, B\}$ can be frequent, but the items (or features) inside $\{A, B\}$ can be redundant, thus, if we know $\{A\}$, we may not need $\{B\}$. Therefore, beside the frequency criterion of the itemsets as a measure of informativeness or interestingness, other measures have been proposed such as the joint entropy. In the following discussion, we address the problem of maximally informative k-itemsets (*i.e., miki*) mining. First, we introduce the basic definitions and notations of the *miki* problem that will be used in the rest of this thesis. Second, we discuss the main existing methods and techniques that have been proposed in the literature to extract *miki*.

The following definitions introduce the basic requirements for mining maximally informative k-itemsets [7].

Definition 9. *Let $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ be a set of literals called $features$. An $itemset$ X is a set of $features$ from $\mathcal{F}$, i.e., $X \subseteq \mathcal{F}$. The $size$ or $length$ of the $itemset$ X is the number of $features$ in it. A $transaction$ T is a set of elements such that $T \subseteq \mathcal{F}$ and $T \neq \emptyset$. A $database$ $\mathcal{D}$ is a set of $transactions$.*

Definition 10. *The entropy [15] of a feature i in a database $\mathcal{D}$ measures the expected amount of information needed to specify the state of uncertainty or disorder for the feature i in $\mathcal{D}$. Let i be a feature in $\mathcal{D}$, and $P(i = n)$ be the probability that i has value n in $\mathcal{D}$ (we consider categorical data, where the value will be '1' if the object has the feature and '0' otherwise). The entropy of the feature i is given by*

$$H(i) = -(P(i = 0)log(P(i = 0)) + P(i = 1)log(P(i = 1)))$$

where the logarithm base is 2.

Definition 11. *The binary projection, or projection of an itemset X in a transaction T ($proj(X, T)$) is the set of size $|X|$ where each item (i.e., feature) of X is replaced by '1' if it occurs in T and by '0' otherwise. The projection counting of X in a database $\mathcal{D}$ is the set of projections of X in each transaction of $\mathcal{D}$, where each projection is associated with its number of occurrences in $\mathcal{D}$.*

Example 13. *Let us consider Table 2.1. The projection of (B, C, D) in d_1 is $(0, 1, 1)$. The projections of (D, E) on the database of Table 2.1 are $(1, 1)$ with nine occurrences and $(0, 1)$ with one occurrence.*

Definition 12. *Given an itemset $X = \{x_1, x_2, \ldots, x_k\}$ and a tuple of binary values $\mathcal{B} = \{b_1, b_2, \ldots, b_k\} \in \{0\,1\}^k$. The joint entropy of X is defined as:*

$$H(X) = -\sum_{\mathcal{B}\in\{0,1\}^{|k|}} J \times log(J)$$

Where J= $P(x_1 = b_1, \ldots, x_k = b_k)$ is the joint probability of $X = \{x_1, x_2, \ldots, x_k\}$.

Given a database $\mathcal{D}$, the joint entropy $H(X)$ of an itemset X in $\mathcal{D}$ is proportional to its size k *i.e.,* the increase in the size of X implies an increase in its joint entropy $H(X)$. The higher the value of $H(X)$, the more information the itemset X provides in $\mathcal{D}$. For simplicity, we use the term entropy of an itemset X to denote its joint entropy.

Example 14. *Let us consider the database of Table 2.1. The joint entropy of (D, E) is given by $H(D, E) = -\frac{9}{10}log(\frac{9}{10}) - \frac{1}{10}log(\frac{1}{10}) = 0.468$. Where the quantities $\frac{9}{10}$ and $\frac{1}{10}$ respectively represent the joint probabilities of the projection values $(1, 1)$ and $(0, 1)$ in the database.*

Definition 13. *Given a set $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ of features, an itemset $X \subseteq \mathcal{F}$ of length k is a maximally informative k-itemset, if for all itemsets $Y \subseteq \mathcal{F}$ of size k, $H(Y) \leq H(X)$. Hence, a maximally informative k-itemset is the itemset of size k with the highest joint entropy value.*

The problem of mining maximally informative k-itemsets presents a variant of itemset mining, it relies on the joint entropy measure for assessing the informativeness brought by an itemset.

Definition 14. *Given a database $\mathcal{D}$ which consists of a set of n attributes (features) $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$. Given a number k, the problem of miki mining is to return a subset $F' \subseteq F$ with size k, i.e., $|F'| = k$, having the highest joint entropy in $\mathcal{D}$, i.e., $\forall F'' \subseteq F \wedge |F''| = k, H(F'') \leq H(F')$.*

Example 15. *In this application, we would like to retrieve documents from Table 2.1, in which the columns d_1, d_{10} are documents, and the attributes A, B, C, D, E are some features (items, keywords) in the documents. The value "1" means that the feature occurs in the document, and "0" not. It is easy to observe that the itemset (D, E) is frequent, because features D and E occur together in almost every document. However, it provides little help for document retrieval. In other words, given a document d_x in our data set, one might look for the occurrence of the itemset (D, E) and, depending on whether it occurs or not, she will not be able to decide which document it is. By contrast, the itemset (A, B, C) is infrequent, as its member features rarely or never appear together in the data. And it is troublesome to summarize the value patterns of the itemset (A, B, C). Providing it with the values $< 1, 0, 0 >$ we could find the corresponding document O_3; similarly, given the values $< 0, 1, 1 >$ we will have the corresponding document O_6. Although (A, B, C) is infrequent, it contains lots of useful information which is hard to summarize. By looking at the values of each feature in the itemset (A, B, C), it is much easier to decide exactly which document they belong to. (A, B, C) is a maximally informative itemset of size $k = 3$.*

In data mining literature, several endeavors have been made to explore informative itemsets (or featuresets, or set of attributes) in databases [35] [36] [37] [7]. Different measures of itemset informativeness (*e.g.,* frequency of itemset co-occurrence in the database etc.) have been used to identify and distinguish informative itemsets from non-informative ones. For instance, by considering the itemsets co-occurrence, several conclusions can be drawn to explain interesting, hidden relationships between different itemsets in the data.

Mining itemsets based on the co-occurrence frequency (*e.g.,* frequent itemset mining) measure does not capture all dependencies and hidden relationships in the database, especially when the data is sparse [37]. Therefore, other measures must be taken into account. Low and high entropy measures of itemsets informativeness were proposed [37]. The authors of [37] have proposed the use of a tree based structure without specifying a length k of the informative itemsets to be discovered. However, as the authors of [37] mentioned, such an approach results in a very large output.

Beyond using a regular co-occurrence frequency measure to identify the itemsets informativeness, the authors of [38] have proposed an efficient technique that is more general. The main motivation is to get better insight and understanding of the data by figuring out other hidden relationships between the itemsets (*i.e.,* the inner correlation between the itemsets themselves), in particular when determining the itemsets' rules. To this end, the authors of [38] did not focus only on the analysis of the positive implications between the itemsets in the data (*i.e.,* implications between supported itemsets), but also they take into account the negative implications. To determine the significance of such itemsets implications, the authors of [38] have used a classic statistical chi-squared measure to efficiently figure out the interestingness of such itemsets rules.

Generally, in the itemset mining problem there is a trade-off between the itemset informativeness and the pattern explosion (*i.e.,* number of itemsets to be computed). Thus, some itemset informativeness measures (*e.g.,* the co-occurrence frequency measure with very low minimum support) would allow for a potential high number of useless patterns (*i.e.,* itemsets), and others would highly limit the number of patterns. The authors of [39] proposed an efficient approach that goes over regular used itemset informativeness measures, by developing a general framework of statistical models allowing the scoring of the itemsets in order to determine their informativeness. In particular, in [39], the initial focus is on the exponential models to score the itemsets. However these models are inefficient in terms of execution time, thus, the authors propose to use decomposable models. On the whole, the techniques proposed in [39] and [38] are mainly dedicated to mining in centralized environments, while our techniques are dedicated to parallel data mining in distributed environments.

The authors of [7] suggest to use a heuristic approach to extract informative itemsets of length k based on maximum joint entropy. Such maximally informative itemsets of size k is called *miki*. This approach captures the itemsets that have high joint entropies. An itemset is a *miki* if all of its constructing items shatter the data maximally. The items within a *miki* are not excluding, and do not depend on each other. [7] proposes a bunch of algorithms to extract *miki*. A brute force approach consists of performing an exhaustive search over the database to determine all *miki* of different sizes. However, this approach is not feasible due to the large number of itemsets to be determined, which results in multiple database scans. Another algorithm proposed in [7] namely *ForwardSelection*, consists of fixing a parameter k that denotes the size of the *miki* to be discovered. This algorithm proceeds by determining a top n *miki* of size 1 having highest joint entropies, then, the algorithm determines the combinations of *1-miki* of size 2 and returns the top n most informative itemsets. The process continues until it returns the top n *miki* of size k. Example 16 illustrates the mining process of this algorithm:

Example 16. *Given the binary database $\mathcal{D}'$ as shown in Figure 2.5. $\mathcal{D}'$ contains 4 items and 8 transactions. Suppose that we want to determine the miki of size $k = 3$ using ForwardSelection algorithm.*

The algorithm starts by determining the entropies of each item in $\mathcal{D}'$ as follow:

$$H(A) = -\tfrac{5}{8}log(\tfrac{5}{8}) - \tfrac{3}{8}log(\tfrac{3}{8}) = 0.954$$
$$H(B) = -\tfrac{6}{8}log(\tfrac{6}{8}) - \tfrac{2}{8}log(\tfrac{2}{8}) = 0.811$$
$$H(C) = -\tfrac{1}{2}log(\tfrac{1}{2}) - \tfrac{1}{2}log(\tfrac{1}{2}) = 1$$

A	B	C	D
1	0	1	0
1	0	1	0
1	0	0	0
1	1	0	0
1	1	1	0
0	0	1	0
0	0	0	0
0	0	0	1

Figure 2.5 – Binary Database $\mathcal{D}'$

$H(D) = -\frac{7}{8}log(\frac{7}{8}) - \frac{1}{8}log(\frac{1}{8}) = 0.543$

The item {C} has the highest entropy, thus {C} presents a seed. From this item seed, ForwardSelection generates the miki candidates of size two ({C A}, {C B}, {C D}). A scan to the database $\mathcal{D}'$ is performed to determine the entropy of each miki candidates of size two. Here, we have:

$$H(CA) = -\frac{3}{8}log(\frac{3}{8}) - \frac{1}{8}log(\frac{1}{8}) - \frac{2}{8}log(\frac{2}{8}) - \frac{2}{8}log(\frac{2}{8}) = 1.905$$
$$H(CB) = -\frac{3}{8}log(\frac{3}{8}) - \frac{1}{8}log(\frac{1}{8}) - \frac{3}{8}log(\frac{3}{8}) - \frac{1}{8}log(\frac{1}{8}) = 1.811$$
$$H(CD) = -\frac{4}{8}log(\frac{4}{8}) - \frac{3}{8}log(\frac{3}{8}) - \frac{1}{8}log(\frac{1}{8}) = 1.405$$

The itemset {C A} has the highest entropy, thus the miki of size 2 *in $\mathcal{D}'$ is {C A}. For $k = 3$, {C A} presents a seed to construct the miki candidates of size* 3 *({C A B}, {C A D}). The same procedure is carried out to determine the miki of size* 3 *by scanning the database $\mathcal{D}'$ and determine the entropy of each miki candidate ({C A B}, {C A D}).*

$$H(CAB) = -\frac{2}{8}log(\frac{2}{8}) - \frac{1}{8}log(\frac{1}{8}) - \frac{1}{8}log(\frac{1}{8}) - \frac{1}{8}log(\frac{1}{8}) - \frac{1}{8}log(\frac{1}{8}) - \frac{2}{8}log(\frac{2}{8}) = 2.5$$
$$H(CAD) = -\frac{2}{8}log(\frac{2}{8}) - \frac{2}{8}log(\frac{2}{8}) - \frac{1}{8}log(\frac{1}{8}) - \frac{1}{8}log(\frac{1}{8}) - \frac{1}{8}log(\frac{1}{8}) - \frac{1}{8}log(\frac{1}{8}) = 2.15$$

Since the itemset {C A B} has the highest entropy, thus, {C A B} is a miki of size 3.

Although *ForwardSelection* algorithm accounts for a simple and efficient method to determine the *miki* of size k, it has major drawbacks. When the size k of the itemset to be discovered tends to be very high, there would be a high number of database scans which impacts the overall performance of the algorithm.

The problem of extracting informative itemsets was not only proposed for mining static databases. There have been also interesting works in extracting informative itemsets in data streams [40] [41]. The authors of [42] proposed an efficient method for discovering maximally informative itemsets (*i.e.,* highly informative itemsets) from data streams based on sliding window.

Extracting informative itemsets has a prominent role in feature selection [43]. Various techniques and methods have been proposed in the literature to solve the problem of selecting relevant features to be used in classification [24] tasks. These methods fall into two different categories, namely *Filter* and *Wrapper* methods [44]. Filter methods serve to pre-process the data before being used for a learning purpose. They aim to determine a small set of relevant features. However, these methods capture only the correlations between each feature (*i.e.,* independent variable, attribute or item) and the target class (*i.e.,* predictor). They do not take into account the inter correlation between the selected features (*i.e.,* if the features are inter correlated then they are redundant). In the other hand, to determine an optimal set of relevant features, wrapper methods perform a feature's set search that maximizes an objective function (*i.e.,* classifier performance). However, these methods yield in heavy computations (*i.e.,* selecting each time a set of features and evaluate an objective function). To solve this problem, *Embedded* [43] methods have been proposed. The main goal is to incorporate the wrapper methods in the learning process.

2.2 Parallel and Distributed Computing

In this Section, we introduce the MapReduce programming model and we detail its working principle and its basic architecture.

2.2.1 MapReduce

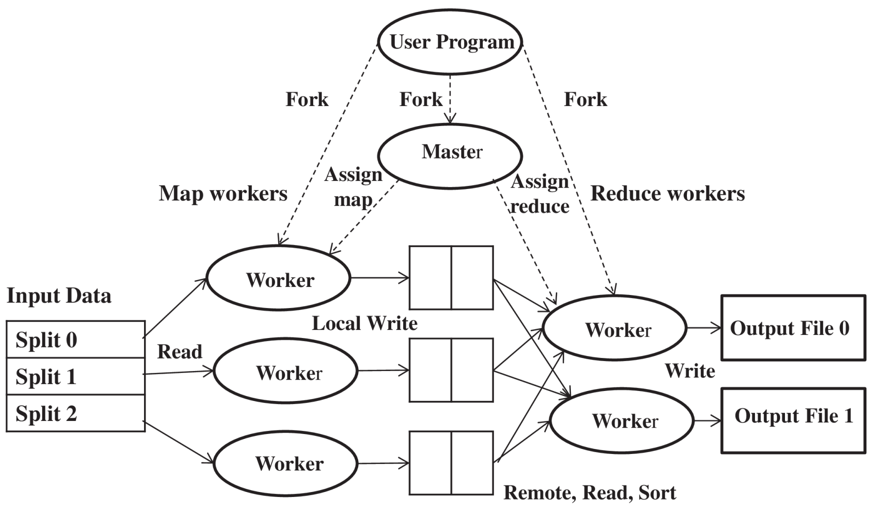


Figure 2.6 – MapReduce Architecture

What is MapReduce ? MapReduce [2] is a parallel framework for large scale data processing. It has gained increasing popularity, as shown by the tremendous success of Hadoop [45], an open-source implementation. Hadoop enables resilient, distributed processing of massive unstructured data sets across commodity computer clusters (*i.e.,* set of commodity machines), in which each node of the cluster includes its own storage. MapReduce serves two essential functions:

- **map():** The map function is applied to process the input data. Generally the input data is in the form of file or directory and is stored in the Hadoop file system (HDFS). The input file is passed to the mapper function line by line. The mapper processes the data and creates several small chunks of data
- **reduce():** The Reducer function is applied to process the data that comes from the mapper. After processing, it produces a new set of output, which will be stored in the HDFS.

MapReduce Architecture Figure 2.6 illustrates the architecture of MapReduce. The user sends its program to the master node. The master node assigns map and reduce tasks to the workers. The data is divided into data splits (*i.e.,* in a hadoop file system). Each map worker reads and executes a map task on its data split and writes the results on its disc (local write). After the execution of all map tasks, each map worker sends its results in the form of (key, value) pairs to the reduce workers. Between the map and the reduce phases, a sorting process is carried out where each key is associated with its list of values. Finally, the reducers perform their computing logic and output the final results to a hadoop distributed filesystem (*i.e.,* HDFS).

Example 17. *As an example, suppose we want to count the number of times every word appears in a novel. We can split the task among some people, so each takes a page, writes a word on a separate sheet of paper and takes a new page until they're finished. This is the map aspect of MapReduce. And if a person leaves, another person takes his place. This exemplifies MapReduce's fault-tolerant element.*

When all pages are processed, users sort their single-word pages into some boxes, which represent the first letter of each word. Each user takes a box and sorts each word in the stack alphabetically. The number of pages with the same word is an example of the reduce aspect of MapReduce.

2.3 Knowledge Discovery and Distributed Computing

With the explosive growth of data, a centralized knowledge discovery process becomes unable to process large volumes of data. The distribution of the computation over several machines has solved the problem. *i.e.,* the data is located in different commodity machines, and the computation process is distributed over these shared data and being executed in parallel.

In this Section, first we focus our study on the main parallel frequent itemset mining (PFIM for short) algorithms that have been used in the literature. Second, we discuss the problem of the parallel discovery of maximally informative k-itemsets.

2.3.1 Parallel Frequent Itemset Mining

In data mining literature, there have been several endeavours to improve the parallel discovery of frequent itemsets. In the following, we present the major methods and techniques that have been proposed in the literature.

Parallel Apriori Algorithm: In a massively distributed environment, the parallel version of Apriori (*i.e.,* Parallel Apriori) algorithm [14] has shown better performance than its sequential one. Although the parallelism setting and the availability of high number of resources, Apriori algorithm has brought regular issues and limitations as shown in its sequential implementation. In a massively distributed environment such as MapReduce, using Apriori algorithm, the number of jobs required to extract the frequent itemsets is proportional to the size of the lengthy itemset. Hence, with very small minimum support and large amount of data, the performance of Parallel Apriori is very poor. This is, because, the inner working process of Apriori algorithm is based on a candidate generation and testing approach which results in a high disc I/O access. In addition, in a massively distributed environment, Apriori algorithm allows for a high data communication between the mappers and the reducers, this is particularly the case when the minimum support tends to be very small.

Parallel SON Algorithm: Based on its mining principle, unlike Apriori algorithm [4], SON algorithm is more flexible and suitable to be parallelized in a massively distributed environment. For instance, in MapReduce, a parallel version of SON algorithm [14] is represented by two jobs. At the first job, each data partition (*i.e.,* data split) is given to a mapper to be mined based on a local minimum support and a specific FIM algorithm (*e.g.,* Apriori). Then each mapper emits its mining results (*i.e.,* local frequent itemset as a key and its occurrence number in the data partition as a value). The reducer aggregates the results and sums up the values of each key and writes the final result to the Hadoop distributed file system (HDFS). At the second MapReduce job, a filtering step is carried out to check the global frequency of all local frequent itemsets of the first MapReduce job. To this end, the count of each local frequent itemset is determined by scanning the database $\mathcal{D}$.

Comparing to the Parallel Apriori algorithm, the main advantage of the parallel SON is the low cost in terms of the database scan. In addition, in a massively distributed environment such as MapReduce, generally, less number of jobs is more adequate and natural than multiple jobs. This makes SON algorithm more suitable to the parallelism and results in a lower data communication cost compared to Parallel Apriori algorithm.

Regarding the core mining process of the Parallel SON algorithm, its overall performance highly depends on its first MapReduce job. For instance, when a data partition (*i.e.,* mapper) holds a high number of similar transactions, then the number of generated local frequent itemsets by that data partition is high. *i.e.,* the runtime process of the mapper is high which in turns would affect the overall mining process.

Parallel Eclat Algorithm: Parallel Eclat algorithm [12] has brought same regular issues and limitations of its sequential implementation. In particular, the high number of frequent items implies a high number of transaction identifiers to be stored. This can yields in a memory issue, *i.e.,* the list of the transaction identifiers cannot fit into the available memory.

Parallel CDAR Algorithm: In the data mining literature, there have been no proposed parallel version of CDAR algorithm [13]. However, a parallel approach would have better performances than other existing alternatives such as Parallel Apriori and Parallel SON algorithms. The main reason is that CDAR is not a candidate-based generation algorithm such as Apriori. Interestingly, with a very low minimum support and homogeneous data partitions, CDAR would give good performances.

Parallel FP-Growth Algorithm: Parallel FP-Growth (PFP-Growth for short) algorithm [6] has been successfully applied to efficiently extract the frequent itemsets from large databases. Its high performance in terms of processing time comes as the result of its core mining principle. At its first MapReduce job, PFP-Growth performs a simple counting process to determine a list of frequent items. The second MapReduce job is dedicated to construct an FP-tree to be mined later at the reducer phase. The mining process is carried out in the memory which explains the high performance runtime of PFP-Growth.

Although, PFP-Growth has been considered as a highly efficient mining technique, with very small minimum support, large amount of data and a top k (*top k itemsets to be returned by the algorithm*) equals to infinity, it does not scale. This behaviour of PFP-Growth will be better illustrated by our different experiments in chapters 3, 4. The main reason behind the limitations of PFP-Growth algorithm is the memory constraint.

PARMA Algorithm: Parallel PARMA algorithm [46] has shown better performances than PFP-Growth algorithm. However, PARMA does not determine the exhaustive list of frequent itemsets, instead the algorithm approximates them. To this end, despite the scalability and the high performance of its runtime process, in this thesis, we exclude this algorithm from being compared to our approaches.

To represent the frequent itemsets in a condensed schema that contains only the releavant and no redundunt itemsets, several parallel algorithms have been proposed. Some early efforts tried to speed up the mining algorithms by running them in parallel [47], using frameworks such as MapReduce [48] that allow to make powerful computing and storage units on top of ordinary machines. In [49], Wang et al. propose an approach for mining closed itemsets using MapReduce, but it suffers from the lack of scalability.

Many research efforts [50, 51] have been introduced to design parallel algorithms capable of working under multiple threads under a shared memory environment. Unfortunately, these approaches do not address the major problem of heavy memory requirement when processing large scale databases. To overcome the latter, MapReduce platform was designed to enable and facilitate the ability to distribute processing of large scale datasets on large computing clusters. In [52], the authors propose a parallel FP-Growth algorithm in MapReduce, which achieves quasi-linear speedups. However, the method presented so far suffers from either excessive amounts of data that need to be transferred and sorted and a high demand for main-memory at cluster nodes.

Moreover, having a large amount of transactional data, finding correlation between them highlights the necessity of discovering a condensed representations of items. Since the introduction of CFI in [30], numerous algorithms for mining it were proposed [53, 54]. In fact, these algorithms tried to reduce the problem of finding frequent itemsets to the problem of mining CFIs by limiting the search space to only CFIs rather than the the whole powerset lattice. Furthermore, they have good performance whenever the size of dataset is small or the support threshold is high. However, as far as the size of the datasets becomes large, both memory use and communication cost are unacceptable. Thus, parallel solutions are of a compelling need. But, research works on parallel mining of CFI are few. In [49] introduce a new algorithm based on the parallel FP-Growth algorithm PFP [52] that divides an entire mining task into independent parallel subtasks and achieves quasi-linear speedups. The algorithm mines CFI in four MapReduce jobs and introduces a redundancy filtering approach to deal with the problem of generating redundant itemsets. However, experiments on algorithm were on a small-scale dataset.

2.3.2 Parallel Maximally Informative K-Itemsets Mining

In a massively distributed environment and with very large volumes of data, the discovery of the maximally informative k-itemsets is very challenging. The conventional and sequential proposed approaches should be carefully designed to be parallelized. Unfortunately, in the literature, there has been no solutions for the problem of parallel discovery of maximally informative k-itemsets in massively distributed environments. In this Section, we limit our discussion to the popular *ForwardSelection* algorithm [7]. We depict a straightforward parallel solution for it.

Parallel *ForwardSelection* Algorithm: In a massively distributed environments, with large amount of data, extracting the *miki* of different sizes is not trivial. Since, *ForwardSelection* algorithm uses a level-wise approach to determine the *miki* of size k, its parallel version would perform several k jobs. Therefore, with very large volumes of data and very high size of the *miki* to be extracted, *ForwardSelection* algorithm would give a poor performance. This is due to the high disc I/O access, the candidate approach principle and the comparison step at the reducer phase to emit the *miki* having higher joint entropy. In fact, this is not only lead to a poor performance in terms of execution time but also in terms of data communication cost. *i.e.,* with a high *miki* size, the quantity of the data being transferred between the mappers and the reducers would be very high.

Split	A	B	C	D
S_1	1	0	1	0
	1	0	1	0
	1	0	0	0
	1	1	0	0
	1	1	1	0
S_2	0	0	1	0
	0	0	0	0
	0	0	0	1

Table 2.2 – Data Splits

Example 18. *Consider the database $\mathcal{D}'$ as shown in Figure 2.5. We want to determine the miki of size k equals to 2 in parallel, by using a parallel version of ForwardSelection algorithm. Suppose that the database $\mathcal{D}'$ is divided into two data splits as shown in Table 2.2. Each data split (respectively S_1 and S_2) is processed by a dedicated mapper (respectively m_1 and m_2). At a first MapReduce job, each mapper proceeds by emitting each itemset of size one as a key and its corresponding projection (i.e., combination of the '0s' and '1s') as a value. For instance, the mapper m_1 would emit (A, 1) 5 times to the reducer. m_2 would emit (A, 0) 3 times to the reducer (here a simple optimization can be used consists of emitting only the itemsets that appears in the transactions i.e., having '1s' projections). Then, the reducer is in charge of computing the joint entropy of each itemset and emitting the itemset with the highest value of joint entropy. At a second MapReduce job, the itemset with highest joint entropy is combined to each item in each split to generate the candidate miki list of size 2. After the candidate generation step, the joint entropy of each miki candidate of size two is computed similarly as in the first MapReduce job. For instance, in our example the first MapReduce job would return the item {C} as a miki of size one ($H(C) = 1$). The second MapReduce job would return the itemset {C A} as a miki of size two ($H(CA) = 1.905$) as it has the higher value of joint entropy. This should continue until reaching the miki with size k, i.e., using k MapReduce jobs. However, performing k MapReduce jobs does not lead to good performance results, particularly when k is not small, and when the database is very big.*

2.4 Conclusion

In this chapter, we have discussed the state of the art about parallel solutions for itemset mining. The main limitations of the existing parallel solutions are the multiple scan of the database and the memory related issues. Basically, these different limitations become more challenging when the data set is very large and / or the minimum support is very small or the size k of a *miki* is very high.

In this thesis, we address the problem of mining itemsets in parallel. In particular, we handle the problem of parallel mining of frequent itemsets and maximally informative k-itemsets. We carry out extensive theoretical and practical studies and proposed various solutions validated with real-world very large data sets.

Chapter 3

Data Placement for Fast Mining of Frequent Itemset

In this chapter, we address the problem of frequent itemset mining (FIM) in very large databases. Despite crucial recent advances, the problem of frequent itemset mining is still facing major challenges. This is particularly the case when: (i) the mining process must be massively distributed and; (ii) the minimum support (MinSup) is very low. In this chapter, we study the effectiveness and leverage of specific data placement strategies for improving parallel frequent itemset mining (PFIM) performance in MapReduce, a highly distributed computation framework. By offering a clever data placement and an optimal organization of the extraction algorithms, we show that the itemset discovery effectiveness does not only depend on the deployed algorithms. We propose ODPR (Optimal Data-Process Relationship), a solution for fast mining of frequent itemsets in MapReduce. Our method allows discovering itemsets from massive data sets, where standard solutions from the literature do not scale. Indeed, in a massively distributed environment, the arrangement of both the data and the different processes can make the global job either completely inoperative or very effective. Our proposal is thoroughly explained in Section 3.2. We evaluate our proposed approach with real-world massive data sets to show its effectiveness.

3.1 Motivation and Overview of the Proposal

With the availability of inexpensive storage and the progress that has been made in data capture technology, several organizations have set up very large databases, known as Big Data. This includes different data types, such as business or scientific data [55], and the trend in data proliferation is expected to grow, in particular with the progress in networking technology. The manipulation and processing of these massive data have opened up new challenges in data mining [14]. In particular, frequent itemset mining (FIM) algorithms have shown several flaws and deficiencies when processing large amounts of data. The problem of mining huge amounts of data is mainly related to the memory restrictions as well as the principles and logic behind FIM algorithms themselves [56].

In order to overcome the above issues and restrictions in mining large databases, several efficient solutions have been proposed. The most significant solution required to rebuild and design FIM algorithms in a parallel manner relying on a specific programming model such as MapReduce [57]. MapReduce is one of the most popular solutions for big data processing [58], in particular due to its automatic management of parallel execution in clusters of commodity machines. Initially proposed in [2], it has gained increasing popularity, as shown by the tremendous success of Hadoop [59], an open-source implementation.

The idea behind MapReduce is simple and elegant. Given an input file, and two map and reduce functions, each MapReduce job is executed in two main phases. In the first phase, called map, the input data is divided into a set of splits, and each split is processed by a map task in a given worker node. These tasks apply the map function on every key-value pair of their split and generate a set of intermediate pairs. In the second phase, called reduce, all the values of each intermediate key are grouped and assigned to a reduce task. Reduce tasks are also assigned to worker machines and apply the reduce function on the created groups to produce the final results.

Although MapReduce refers as an efficient setting for FIM implementations, most of parallel frequent itemset mining (PFIM) algorithms have brought same regular issues and challenges of their sequential implementations. For instance, invoking such best PFIM algorithm with very low minimum support($MinSup$) could exceed available memory. Unfortunately, dealing with massive data sets (up to Terabytes of data) implies working with very low supports since data variety lowers item frequencies. Furthermore, if we consider a FIM algorithm which relies on a candidate generation principle, its parallel version would remain carrying the same issues as in its sequential one. Therefore, covering the problem of FIM algorithms does not only involve the distribution of computations over data, but also should take into account other factors.

Interestingly and to the best of our knowledge, there has been no focus on studying data placement strategies for improving PFIM algorithms in MapReduce. However, as we highlight in this work, the data placement strategies have significant impacts on PFIM performance. In this work, we identify, investigate and elucidate the fundamental role of using such efficient strategies for improving PFIM in MapReduce. In particular, we take advantage of two data placement strategies: Random Transaction Data Placement (RTDP) and Similar Transaction Data Placement (STDP). In the context of RTDP, we use a random placement of data on a distributed computational environment without any data constraints, to be consumed by a particular PFIM algorithm. However, in STDP, we use a similarity-based placement for distributing the data around the nodes in the distributed environment. By leveraging the data placement strategies, we propose ODPR (Optimal Data-Process Relationship) as explained in Section 3.2, a new solution for optimizing the global extraction process. Our solution takes advantage of the best combination of data placement techniques and the extraction algorithm.

We have evaluated the performance of our solution through experiments over ClueWeb and Wikipedia data sets (the whole set of Wikipedia articles in English). Our results show that a careful management of the parallel processes along with adequate data placement, can dramatically improve the performance and make a big difference between an inoperative and a successful extraction.

3.2 Optimal Data-Process Relationship

Let us now introduce our PFIM architecture, called Parallel Two Steps (P2S), which is designed for data mining in MapReduce. From the mining point of view, P2S is inspired from SON [60] algorithm. The main reason behind opting SON as a reference to P2S is that a parallel version of the former algorithm does not require costly overhead between mappers and reducers. However, as illustrated by our experiments in Section 3.3, a straightforward implementation of SON in MapReduce would not be the best solution for our research problem. Therefore, with P2S, we propose new solutions for PFIM mining, within the "two steps" architecture.

The principle of P2S is drawn from the following observation. Dividing a database $\mathcal{D}$ into n partitions $p_1, p_2, ..., p_n$, where $\cup p_i = D$, $i = 1...n$

$$GFI \subseteq \cup LFI \tag{3.1}$$

where GFI denotes global frequent itemsets and LFI refers to local frequent itemsets. This particular design allows it to be easily parallelized in two steps as follow:

Job 1: Each mapper takes a data split, and performs particular FIM algorithm. Then, it emits a list of local frequent itemsets to the reducer

Job 2: Takes an entire database $\mathcal{D}$ as input, and filters the global frequent itemsets from the list of local frequent itemsets. Then, it writes the final results to the reducer.

P2S thus divides the mining process into two steps and uses the dividing principle mentioned above. As one may observe from its pseudo-code, given by Algorithm 1, P2S is very well suited for MapReduce.

The first MapReduce job of P2S consists of applying specific FIM algorithm at each mapper based on a local minimum support ($localMinSup$), where the latter is computed at each mapper based on $MinSup$ δ percentage and the number of transactions of the split being processed. At this stage of P2S, the job execution performance mainly depends on a particular data placement strategy (*i.e.,* RTDP or STDP). This step is done only once and the resulting placement remains the same whatever the new parameters given to the mining process (*e.g.,* $MinSup$ δ, local FIM algorithm, etc.). Then P2S determines a list of local frequent itemsets LFI. This list includes the local results of all data splits found by all mappers. The second step of P2S aims to deduce a global frequent itemset GFI. This step is carried out relying on a second MapReduce job. In order to deduce a GFI list, P2S filters the LFI list by performing a global test of each local frequent itemset. At this step, each mapper reads once the list of local frequent itemset stored in Hadoop Distributed Cache. Then, each mapper takes a transaction at a time and checks

```
Algorithm 1: P2S
   Input: Database $\mathcal{D}$ and $MinSup\ \delta$
   Output: Frequent Itemsets
1  //Map Task 1
2  map( key:Null : $\mathcal{K}_1$, $value$ = Whole Data Split: $\mathcal{V}_1$ )
3      - Determine a local $MinSup\ ls$ from $\mathcal{V}_1$ based on $\delta$
4      - Perform a complete FIM algorithm on $\mathcal{V}_1$ using $ls$
5      emit (key: local frequent itemset, value: Null)
6  //Reduce Task 1
7  reduce( key:local frequent itemset, $list(values)$ )
8      emit (key,Null)
9  //Map Task 2
10 Read the list of local frequent itemsets from Hadoop Distributed Cache $LFI$ once
11 map( key:line offset : $\mathcal{K}_1$, $value$ = Database Line: $\mathcal{V}_1$ )
12     if $an\ itemseti \in LFI\ and\ i \subseteq \mathcal{V}_1$ then
13         $key \leftarrow i$
14     emit (key:$i$, value: 1)
15 //Reduce Task 2
16 reduce( key:$i$, $list(values)$ )
17     $sum \leftarrow 0$ while $values.hasNext()$ do
18         $sum+ = values.next().get()$
19     if $sum >= \delta$ then
20         emit (key:$i$, value: Null)
```

the inclusion of its itemsets in the list of the local frequent itemset. Thus, at this map phase of P2S algorithm, each mapper emits all local frequent itemsets with their complete occurrences in the whole database (*i.e.,* key: itemset, value: 1). The reducer of the second P2S step, simply computes the sum of the count values of each key (*i.e.,* local frequent itemset) by iterating over the value list of each key. Then, the reducer compares the number of occurrences of each local frequent itemset to $MinSup\ \delta$, if it is greater or equal to δ, then, the local frequent itemset is considered as a global frequent itemset and it will be written to the Hadoop distributed file system. Otherwise, the reducer discards the key (*i.e.,* local frequent itemset).

Theoretically, based on the inner design principles of P2S algorithm, different data placements would have significant impacts on its performance behavior. In particular, the performance of P2S algorithm at its first MapReduce job, and specifically at the mapper phase, strongly depends on RTDP or STDP used techniques. That is due to the sensitivity of the FIM algorithm being used at the mappers towards its input data.

The goal of this work is to provide the best combination of both data placement and local algorithm choice in the proposed architecture. In Section 3.2.1, we develop two data placement strategies and explain more their role in the overall performances.

3.2.1 Data Placement Strategies

The performance of PFIM algorithms in MapReduce may strongly depend on the distribution of the data among the workers. In order to illustrate this issue, consider an example of a PFIM algorithm which is based on a candidate generation approach. Suppose that most of the workload including candidate generation is being done on the mappers. In this case, the data split or partition that holds most lengthy frequent itemsets would take more execution time. In the worst case, the job given to that specific mapper would not complete, making the global extraction process impossible. Thus, despite the fairly automatic data distribution by Hadoop, the computation would depend on the design logic of PFIM algorithm in MapReduce.

Actually, in general, FIM algorithms are highly susceptible to the data sets nature. Consider, for instance, the Apriori algorithm. If the itemsets to be extracted are very long, it will be difficult for this algorithm to perform the extraction. And in case of very long itemsets, it is even impossible. This is due to the fact that Apriori has to enumerate each subset of each itemset. The longer the final itemset, the larger the number of subsets (actually, the number of subsets grows exponentially). Now let us consider Job 1, mentioned above. If a mapper happens to contain a subset of D that will lead to lengthy local frequent itemsets, then it will be the bottleneck of the whole process and might even not be able to complete. Such a case would compromise the global process.

On the other hand, let us consider the same mapper, containing itemsets with the same size, and apply the CDAR algorithm to it. Then CDAR would rapidly converge since it is best suited for long itemsets. Actually, the working principle of CDAR is to first extract the longest patterns and try to find frequent subsets that have not been discovered yet. Intuitively, grouping similar transactions on mappers, and applying methods that perform best for long itemsets seems to be the best choice. This is why a placement strategy, along with the most appropriate algorithm, should dramatically improve the performances of the whole process.

From the observations above, we claim that optimal performances depend on a particular care of massive distribution requirements and characteristics, calling for particular data placement strategies. Therefore, in order to boost up the efficiency of some data sensitive PFIM algorithms, P2S uses different data placement strategies such as *Similar Transaction Data Placement (STDP)* and *Random Transaction Data Placement (RTDP)*, as presented in the rest of this Section.

RTDP Strategy: RTDP technique merely refers to a random process for choosing bunch of transactions from a database $\mathcal{D}$. Thus, using RTDP strategy, the database is divided into n data partitions $p_1, p_2, ..., p_n$ where $\cup p_i = D$, $i = 1...n$. This data placement strategy does not rely on any constraint for placing such bunch of transactions in same partition p.

TID	Transaction
T_1	a, b, c
T_2	a, b, d
T_3	e, f, g
T_4	d, e, f

Table 3.1 – Database $\mathcal{D}$

STDP Strategy: Unlike RTDP data placement strategy, STDP relies on the principle of similarity between chosen transactions. Each bucket of similar transactions is mapped to the same partition p. Therefore, the database $\mathcal{D}$ is split into n partitions and $\cup p_i = D$, $i = 1...n$.

In STDP, each data split would be more homogeneous, unlike the case of using RTDP. More precisely, by creating partitions that contain similar transactions, we increase the chance that each partition will contain frequent local itemset of high length.

3.2.2 Data Partitioning

In STDP, data partitioning using similarities is a complex problem. A clustering algorithm may seem appropriate for this task. However, we propose a graph data partitioning mechanism that will allow a fast execution of this step, thanks to existing efficient algorithms for graphs partitioning such as Min-Cut [61]. In the following, we describe how transaction data can be transformed into graph data for doing such partitioning.

- First, for each unique $item$ in D, we determine the list of transactions L that contain it. Let D' be the set of all transaction lists L.
- Second, we present D' as a graph $G = (V, E)$, where V denotes a set of vertices and E is a set of edges. Each transaction $T \in D$ refers to a vertex $v_i \in G$ where $i = 1...n$. The weight w of an edge that connects a pair of vertices $p = (v_i, v_j)$ in G equals to the number of common items between the transactions representing v_i and v_j.
- Then, after building the graph G, a Min-Cut algorithm is applied in order to partition D'.

In the above approach, the similarity of two transactions is considered as the number of their common items, *i.e.,* the size of their intersection. In order to illustrate our graph partitioning technique, let us consider a simple example as follows.

Example 19. *Let us consider $\mathcal{D}$, the database from Table 3.1. We start by mapping each item in D to its transactions holder. As illustrated in the table of Figure 3.2.2, T_1 and T_2 have 2 common items, likewise, T_3 and T_4 have 2 common items, while the intersection of T_2 and T_3 is one. The intersection of transactions in D' refers to the weight of their*

TID	Transaction
T_1	a, b, c
T_2	a, b, e
T_3	e, f, g
T_4	d, e, f

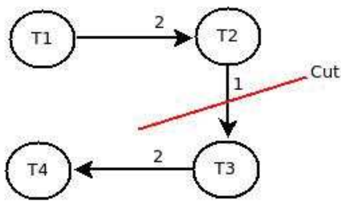


Figure 3.1 – Transactions of a database (left) & Graph representation of the database (right)

edges. In order to partition D', we first build a graph G from D' as shown in Figure 3.2.2. Then, the algorithm Min-Cut finds a minimum cut in G (red line in Figure 3.2.2), which refers to the minimum capacity in G. In our example, we created two partitions: $Partition_1 =< T_1, T_2 >$ and $Partition_2 =< T_3, T_4 >$.

We have used a particular graph partitioning tool namely PaToH [62] in order to generate data partitions. The reason behind opting for PatoH lies in its set of configurable properties, *e.g.,* the number of partitions and the partition load balance factor.

Based on the architecture of P2S and the data placement strategies we have developed and efficiently designed two FIM mining algorithms. Namely Parallel Two Steps CDAR (P2SC) and Parallel Two Steps Apriori (P2SA) depending on the itemset mining algorithm implemented for itemset mining on the mapper, in the first step of P2S. These two algorithms are highly data-sensitive PFIM algorithms.

For instance, if we consider P2SC as a P2S algorithm with STDP strategy, its performance would not be the same as we feed it with RTDP. Because relying on STDP, each split of data fed to such a mapper holds similar transactions, thus, there is less generation of transaction subsets. These expectations correspond to the intuition given in Section 3.2.1. The impact of different data placement strategies will be better observed and illustrated through out experimental results as shown in Section 3.3.

As shown by our experimental results in Section 3.3, P2S has given the best performance when instantiated with CDAR along with STDP strategy.

3.3 Experiments

To assess the performance of our proposed mining approach and investigate the impact of different data placement strategies, we have done an extensive experimental evaluation. In Section 3.3.1, we depict our experimental setup, and in Section 3.3.2, we investigate and discuss the results of our experiments.

3.3.1 Experimental Setup

We implemented our P2S principle and data placement strategies on top of Hadoop-MapReduce, using Java programming language. As mining algorithms on the mappers, we implemented Apriori as well as CDAR. For comparison with PFP-Growth [6], we adopted the default implementation provided in the Mahout [63] machine learning library (Version 0.7). We denote by P2Sx-R and P2Sx-S the use of our P2S principle with STDP (P2Sx-S) or RTDP (P2Sx-R) strategy for data placement, where local frequent itemsets are extracted by means of the 'x' algorithm. For instance, P2SA-S means that P2S is executed on data arranged according to STDP strategy, with Apriori executed on the mappers for extracting local frequent itemsets. MR-Apriori is the straightforward implementation of Apriori in MapReduce (one job for each length of candidates, and database scans for support counting are replaced by MapReduce jobs). PApriori does not use any particular data placement strategy. To this end, we just opted to test the algorithm with a RTDP data placement strategy for a comparison sake. Eventually, the instance of P2S architecture with Apriori exploited for local frequent itemset mining on the mappers and data arranged according to the RTDP strategy has to be considered as a straightforward implementation of SON. Therefore, we consider this version of P2S being the original version of SON in our experiments.

We carry out all our experiments based on the Grid5000 [64] platform, which is a platform for large scale data processing. We have used a cluster of 16 and 48 machines respectively for Wikipedia and ClueWeb data set experiments. Each machine is equipped with Linux operating system, 64 Gigabytes of main memory, Intel Xeon $X3440$ 4 core CPUs, and 320 Gigabytes SATA II hard disk.

To better evaluate the performance of ODPR and the impact of data placement strategies, we used two real-world data sets. The first one is the 2014 English Wikipedia articles [65] having a total size of 49 Gigabytes, and composed of 5 millions articles. The second one is a sample of ClueWeb English data set [66] with size of 240 Gigabytes and having 228 millions articles. For each data set we performed a data cleaning task, by removing all English stop words from all articles and obtained a data set where each article accounts for a transaction (where items are the corresponding words in the article) to each invoked PFIM algorithm in our experiments.

We performed our experiments by varying the $MinSup$ parameter value for each algorithm along with particular data placement strategy. We evaluate each algorithm based on its response time, in particular, when $MinSup$ is very low.

3.3.2 Performance Results

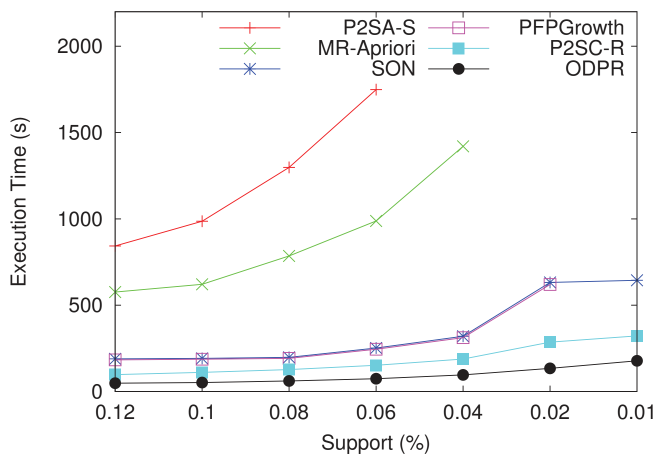


Figure 3.2 – All Algorithms Executed on the Whole Set of Wikipedia Articles in English

Figures 3.2 and 3.3 report our results on the whole set of Wikipedia articles in English. Figures 3.2 gives a complete view on algorithms performances for a support varying from 0.12% to 0.01%. We see that MR-Apriori runtime grows exponentially, and gets quickly very high compared to other presented PFIM algorithms. In particular, this exponential runtime growth reaches its highest value with 0.04% threshold. Below this threshold, MR-Apriori needs more resources (*e.g.,* memory) than what exists in our tested machines, so it is impossible to extract frequent patterns with this algorithm. Another interesting observation is that P2SA-S, *i.e.,* the two step algorithm that use Apriori as a local mining solution, is worse that MR-Apriori. This is an important result, since it confirms that a bad choice of data-process relationship compromises a complete analytics process and makes it inoperative. Let us now consider the set of four algorithms that scale. The less effective are PFP-Growth and P2SA-R. It is interesting to see that two very different algorithmic schemes (PFPGrowth is based on the pattern tree principle and P2SA-R is a two steps principle with Apriori as a local mining solution with no specific care to data placement) have similar performances. The main difference being that PFP-Growth exceeds the available memory below 0.02%. Eventually, P2SC-R and ODPR give the best performances, with an advantage for ODPR.

Figure 3.3 focuses on the differences between the three algorithms that scale in Figure 3.2. The first observation is that P2SA-R is not able to provide results below 0.008%. Regarding the algorithms based on the principle of P2S, we can observe a very good performance for ODPR thanks to its optimization between data and process relationship. These results illustrate the advantage of using a two steps principle where an adequate data placement favors similarity between transactions, and the local mining algorithm does better on long frequent itemsets.

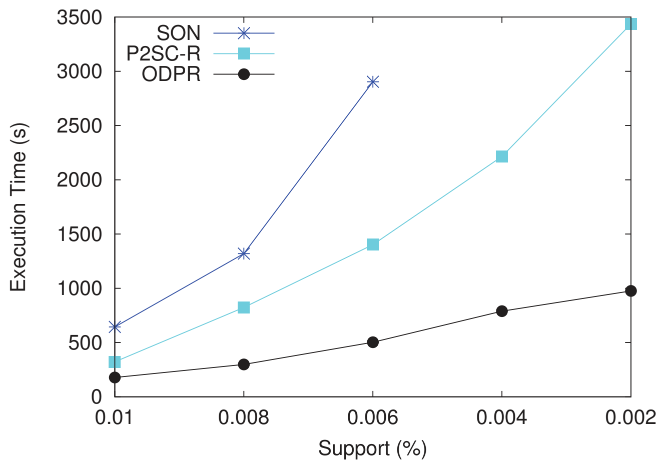


Figure 3.3 – A Focus on Algorithms that Scale on Wikipedia Articles in English

In Figure 3.4, similar experiments have been conducted on the ClueWeb data set. We observe that the same order between all algorithms is kept, compared to Figures 3.2 and 3.3. There are two bunches of algorithms. One, made of P2SA-S and MR-Apriori which cannot reasonably applied to this data set, whatever the minimum support. In the other bunch, we see that PFP-Growth suffers from the same limitations as could be observed on the Wikipedia data set in Figure 3.2, and it follows a behavior that is very similar to that of P2SA-R, until it becomes impossible to execute.

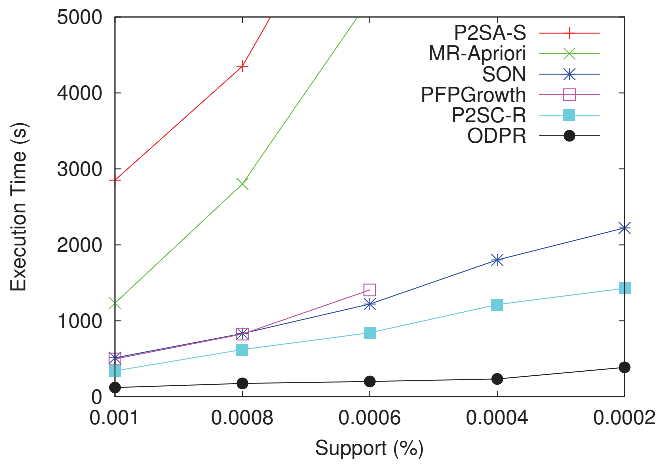


Figure 3.4 – Experiments on ClueWeb Data Set

On the other hand, P2SC-R and ODPR are the two best solutions, while ODPR is the optimal combination of data placement and algorithm choice for local extraction, providing the best relationship between data and process.

3.4 Conclusion

We have identified the impact of the relationship between data placement and process organization in a massively distributed environment such as MapReduce for frequent itemset mining. This relationship has not been investigated before this work, despite crucial consequences on the extraction time responses allowing the discovery to be done with very low minimum support. Such ability to use very low threshold is mandatory when dealing with Big Data and particularly hundreds of Gigabytes like we have done in our experiments. Our results show that a careful management of processes, along with adequate data placement, may dramatically improve performances and make the difference between an inoperative and a successful extraction.

This work opens interesting research avenues for PFIM in massively distributed environments. In general, we would like to deeply investigate a larger number of algorithms and the impact of data placement on them. More specifically, there are two main factors we want to study. Firstly, we need to better identify what algorithms can be implemented in MapReduce while avoiding to execute a large number of jobs (because the larger the number of jobs, the worse the response time). Secondly, we want to explore data placement alternatives to the ones proposed in this thesis.

Chapter 4

Data Partitioning for Fast Mining of Frequent Itemset

In this chapter, we address the problem of mining frequent itemset (FIM) in massively distributed environments. Frequent itemset mining is one of the fundamental cornerstones in data mining. While, the problem of FIM has been thoroughly studied, few of both standard and improved solutions scale. This is mainly the case when i) the amount of data tends to be very large and/or ii) the minimum support threshold is very low. We propose a highly scalable, parallel frequent itemset mining algorithm, namely Parallel Absolute Top Down (PATD). PATD algorithm renders the mining process of very large databases (up to Terabytes of data) simple and compact. Its mining process is made up of only one parallel job, which dramatically reduces the mining runtime, the communication cost and the energy power consumption overhead, in a distributed computational platform. Based on a clever and efficient data partitioning strategy, namely Item Based Data Partitioning (IBDP), PATD algorithm mines each data partition independently, relying on an absolute minimum support instead of a relative one.

All the details about our PATD algorithm and IBDP data partitioning strategy are given in Section 4.2. In Section 4.3, we evaluate our proposal by carrying out extensive various experiments. Our experimental results suggest that PATD algorithm is significantly more efficient and scalable than alternative approaches.

4.1 Motivation and Overview of the Proposal

Since a few decades, the amount of data in the world and our lives seems ever-increasing. Nowadays, we are completely overwhelmed with data, it comes from different sources, such as social networks, sensors, etc. With the availability of inexpensive storage and the progress that has been made in data capture technology, several organizations have set up very large databases, known as Big Data [67]. The processing of this massive amount of data, helps to leverage and uncover hidden relationships, and brings up new, and useful information. Itemsets are one of these tackled levers and consist in frequent correlations of features. Their discovery is known as Frequent itemset mining (FIM for short), and presents an essential and fundamental role in many domains. In business and e-commerce, for instance, FIM techniques can be applied to recommend new items, such as books and different other products. In science and engineering, FIM can be used to analyze such different scientific parameters (*e.g.,* based on their regularities). Finally, FIM methods can help to perform other data mining tasks such as text mining [1], for instance, and, as it will be better illustrated by our experiments in Section 4.3, FIM can be used to figure out frequent co-occurrences of words in a very large-scale text database. However, the manipulation and processing of large-scale databases have opened up new challenges in data mining [68]. First, the data is no longer located in one computer, instead, it is distributed over several machines. Thus, a parallel and efficient design of FIM algorithms must be taken into account. Second, parallel frequent itemset mining (PFIM for short) algorithms should scale with very large data and therefore very low $MinSup$ threshold. Fortunately, with the availability of powerful programming models, such as MapReduce [2] or Spark [3], the parallelism of most FIM algorithms can be elegantly achieved. They have gained increasing popularity, as shown by the tremendous success of Hadoop [45], an open-source implementation. Despite the robust parallelism setting that these solutions offer, PFIM algorithms remain holding major crucial challenges. With very low $MinSup$, and very large data, as will be illustrated by our experiments, most of standard PFIM algorithms do not scale. Hence, the problem of mining large-scale databases does not only depend on the parallelism design of FIM algorithms. In fact, PFIM algorithms have brought the same regular issues and challenges of their sequential implementations. For instance, given best FIM algorithm X and its parallel version X'. Consider a very low $MinSup$ δ and a database $\mathcal{D}$. If X runs out of memory in a local mode, then, with a large database $\mathcal{D}'$, X' might also exceed available memory in a distributed mode. Thus, the parallelism, all alone, does not guarantee a successful and exhaustive mining of large-scale databases and, to improve PFIM algorithms in MapReduce, other issues should be taken into account. Our claim is that the data placement is one of these issues. We investigate an efficient combination between a mining process (*i.e.,* a PFIM algorithm) and an efficient placement of data, and study its impact on the global mining process. We have designed and developed a powerful data partitioning technique, namely Item Based Data Partitioning (IBDP for short). One of the drawbacks of existing PFIM algorithms is to settle for a disjoint placement. IBDP allows, for a given item i to be placed in more than one mapper if necessary. Taking the advantages from this clever data partitioning strategy, we have designed and developed a MapReduce based PFIM algorithm, namely Parallel Absolute Top Down Algorithm (PATD for short), which is capable to mine a very large-scale database in just one simple and fast MapReduce job. We have evaluated the performance of PATD through extensive experiments over two massive data sets (up to one Terabyte and half a billion Web pages). Our results show that PATD scales very well on large databases with very low minimum support, compared to other PFIM alternative algorithms.

4.2 Parallel Absolute Top Down Algorithm

As briefly mentioned in Section 4.1, using an efficient data placement technique, could significantly improve the performance of PFIM algorithms in MapReduce. This is particularly the case, when the logic and the principle of a parallel mining process is highly sensitive to its data. For instance, let consider the case when most of the workload of a PFIM algorithm is being performed on the mappers. In this case, the way the data is exposed to the mappers, could contribute to the efficiency and the performance of the whole mining process (*i.e.,* invoked PFIM algorithm).

In this context, we point out to the data placement, as a custom placement of database transactions in MapReduce. To this end, we use different data partitioning methods. We illustrate the impact of data placement techniques on the performance of PFIM algorithms, by considering particular PFIM algorithms which are based on two MapReduce jobs schema (2-Jobs schema for short).

In this Section, first, we investigate the impact of partitioning data (*i.e.,* impact of data placement) on 2-Jobs schema. Second, we introduce our IBDP method for data partitioning, and then we detail its working logic and principle. Finally, we introduce PATD algorithm and elucidate its design and core mining process in MapReduce.

4.2.1 Impact of Partitioning Data on 2-Jobs Schema

Performing a mining process in two steps was first proposed in [9] and it was designated for centralized environments. SON [9] algorithm divides a mining process as follows:

- **Step 1:** Divide the input database $\mathcal{D}$ into n data chunks (*i.e.,* data splits), where $\mathcal{D} = \{P_1, P_2, \ldots, P_n\}$. Then, mine each data chunk (P_i) in the memory, based on a local minimum support ($LMinSup$), and a specific FIM algorithm. Thus, the first step of SON algorithm is to determine a list of local frequent itemsets (LFI).

- **Step 2:** From previous step result, proceed by filtering the local frequent itemsets in LFI list, based on a global minimum support $GMinSup$. This may be done with a scan on $\mathcal{D}$ and checking the frequency of each itemset is LFI. The main idea is that any frequent itemset on $\mathcal{D}$ will be frequent on at least one chunk P_i and will be found in LFI. Then, return a list of global frequent itemsets (GFI) which is a subset of LFI ($GFI \subseteq LFI$).

In a massively distributed environment, the main bottleneck of such 2-Jobs schema PFIM algorithm is its first execution phase, where an FIM algorithm has to be executed on the chunks. The choice of this algorithm is crucial. Relying on SON mining principle, we have implemented a parallel version of CDAR [69] and Apriori [8] algorithms on MapReduce, namely, Parallel Two Round CDAR (P2RC) and Parallel Two Round Apriori (P2RA) respectively. Each version makes use of CDAR or Apriori on the chunks in the first phase. P2RC divides the mining process into two MapReduce jobs as follows:

- **Job 1:** In the first phase, the principle of CDAR (see [69] for more details) is adapted to a distributed environment. A global minimum support $GMinSup$ Δ is passed to each mapper. The latter deduces a local minimum support $LMinSup$ δ from Δ and its input data split (*i.e.,* number of transaction in the input split). Then, each mapper divides its input data split ($\mathcal{S}$) into n data partitions, $\mathcal{S} = \{S_1, S_2, \dots, S_n\}$. Each partition S_i in $\mathcal{S}$ holds only transactions that have length i, where the length of a transaction is the number of items in it. Then, the mapper starts mining the data partitions $S_i ... S_n$ according to transaction lengths in decreasing order. A transaction in each partition accounts for an itemset. If a transaction T is frequent ($Support(T) \geq \delta$) in partition S_{i+1}, then it will be stored in a list of frequent itemsets L. Otherwise, T will be stored in a temporary data structure $Temp$. After checking the frequency of all transactions T in S_{i+1}, the process continues by generating i subsets of all T in $Temp$ and adds the i generated subsets to partition S_i. The same mining process is carried out until visiting all partitions S_i in $\mathcal{S}$. Before counting the $Support$ of a transaction T, an inclusion test of T in L is performed. If the test returns true, T will be not considered, as it is already in L which means frequent. Each mapper emits all its local frequent itemsets to the reducer. The reducer writes all local frequent itemsets to the distributed file system.

- **Job 2:** Each mapper takes a data split $\mathcal{S}$ and a list of local frequent itemsets LFI. Each mapper determines the inclusion of LFI elements in each transaction of $\mathcal{S}$. If there is an inclusion, then the mapper emits the itemset as a key and one as value (key: itemset, value: 1). A global minimum support $GMinSup$ Δ is passed to the reducer. The reducer simply iterates over the values of each received key, and sums them up in variable sum. If ($sum \geq \Delta$), then the itemset under consideration is globally frequent.

As illustrated above, the main workload of P2RC algorithm is done on the mappers independently. intuitively, the mapper that holds more homogeneous data (*i.e.,* homogeneous transactions) will be faster. Actually, by referring to the mining principle of CDAR, a mapper that holds homogeneous transactions (*i.e.,* similar transactions) allows for more itemset inclusions which in turn results in less subsets generation. Thus, placing each bucket of similar transactions (non-overlapping data partitions) on the mappers would improve the performance of P2RC algorithm. This data placement technique can be achieved by means of different data partitioning methods.

In contrast, the partitioning of data based on transaction similarities (STDP for short: Similar Transaction Data Partitioning), logically would not improve the performance of Parallel Two Round Apriori (P2RA), instead it should lower it. In this case, each mapper would hold a partition of data (*i.e.,* data split) of similar transactions which allows for a high number of frequent itemsets in each mapper. This results in a higher number of itemset candidates generation. Interestingly, using a simple Random Transaction Data Partitioning (RTDP for short) to randomly place data on the mappers, should give the best performance of P2RA. Our experiments given in Section 4.3 clearly illustrate this intuition.

P2RC performs two MapReduce jobs to determine all frequent itemsets. Thus, PFIM algorithms that depend on SON process design duplicate the mining results. Also, at their first mining step (*i.e.,* first MapReduce job), 2-Jobs schema PFIM algorithms output itemsets that are locally frequent, and there is no guarantee to be globally frequent. Hence, these algorithms amplify the number of transferred data (*i.e.,* itemsets) between mappers and reducers.

To cover the above-mentioned issues, our major challenge is to limit the mining process to one simple job. This would guarantee low data communications, less energy power consumption, and a fast mining process. In a distributed computational environment, we take the full advantage of the available massive storage space, CPU(s), etc.

4.2.2 IBDP: An Overlapping Data Partitioning Strategy

Our claim is that duplicating the data on the mappers allows for a better accuracy in the first job and therefore leads to less infrequent itemsets (meaning less communications and fast processing). Consider a data placement with a high overlap, with for instance 10 partitions, each holding 50% of the database. Obviously, there will be less globally infrequent itemsets in the first job (in other words, if an itemset is frequent on a mapper, then it is highly likely to be frequent on the whole database). Unfortunately, such an approach is not realistic. First, we still need a second job to filter the local frequent itemsets and check their global frequency. Furthermore, such a thoughtless placement is absolutely not plausible, given the massive data sets we are dealing with. However, we take advantage of this duplication opportunity and propose IBDP, an efficient strategy for partitioning the data over all mappers, with an optimal amount of duplicated data, allowing for an exhaustive mining in just one MapReduce job. The goal of IBDP is to replace part of the mining process by a clever placement strategy and optimal data duplication.

The main idea of IBDP is to consider the different groups of frequent itemsets that are usually extracted. Let us consider a minimum threshold Δ and X, a frequent itemset according to Δ on $\mathcal{D}$. Let S_X be the subset of $\mathcal{D}$ restricted to the transactions supporting X. The first expectation is to have $|S_X| \ll |\mathcal{D}|$ since we are working with very low minimum thresholds. The second expectation is that X can be extracted from S_X with Δ as a minimum threshold. The goal of IBDP is a follows: for each frequent itemset X, build S_X the subset from which the extraction of X can be done in one job. Fortunately, itemsets usually share a lot of items between each other. For instance, with Wikipedia articles, there will be a group of itemsets related to the *Olympic games*, another group of itemsets related to *Algorithms*, etc. IBDP exploits these affinities between itemsets. It divides the search space by building subsets of $\mathcal{D}$ that correspond to these groups of itemsets, optimizing the size of duplicated data.

More precisely, given a database of transactions $\mathcal{D}$, and its representation in the form of a set $\mathcal{S}$ of n non-overlapping data partitions $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$. Each one of these non-overlapping data partitions (*i.e.,* $\bigcap_{i=1}^{n} S_i = \emptyset$), holds a set of similar transactions (the union of all elements in $\mathcal{S}$ is $\mathcal{D}$, $\bigcup_{i=1}^{n} S_i = \mathcal{D}$). For each non-overlapping data partition S_i in $\mathcal{S}$, we extract a "centroid". The centroid of S_i contains the different items, and their number of occurrences, in S_i. Only the items having a maximum number of occurrences over the whole set of partitions, are kept for each centroid. Once the centroids are built, IBDP simply intercepts each centroid of S_i with each transaction in $\mathcal{D}$. If a transaction in $\mathcal{D}$ shares an item with a centroid of S_i, then the intersection of this transaction and the centroid will be placed in an overlapping data partition called S_i'. If we have n non-overlapping data partitions (*i.e.,* n centroids), IBDP generates n overlapping data partitions and distributes them on the mappers.

The core working process of IBDP data partitioning and its parallel design on MapReduce, are given in Algorithm 2, while its principle is illustrated by Example 20.

Algorithm 2: IBDP

```
1  //Job1
   Input: Non-overlapping data partitions $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ of a database $\mathcal{D}$
   Output: Centroids
2  //Map Task 1
3  map( key: Split Name: $\mathcal{K}_1$, value = Transaction (Text Line): $\mathcal{V}_1$ )
4      - Tokenize $\mathcal{V}_1$, to separate all items
5      emit (key: Item, value: Split Name)

6  //Reduce Task 1
7  reduce( key: Item, list(values) )
8      while values.hasNext() do
9          emit (key:(Split Name) values.next (Item))

10 //Job2
   Input: Database $\mathcal{D}$
   Output: Overlapping Data Partitions
11 //Map Task 2
12 - Read previous job1 result once in a (key, values) data structure (DS), where key:
   SplitName and values: Items
13 map( key: Null: $\mathcal{K}_1$, value = Transaction (Text Line): $\mathcal{V}_1$ )
14     for SplitName in DS do  if Items.Item $\cap \mathcal{V}_1 \neq \emptyset$ then
15         emit (key: SplitName, value: $\mathcal{V}_1$)
16

17 //Reduce Task 2
18 reduce( key: SplitName, list(values) )
19     while values.hasNext() do
20         emit (key: (SplitName), values.next: (Transaction))
```

- **Job 1** *Centroids*: Each mapper takes a transaction (line of text) from non-overlapping data partitions as a value, $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$, and the name of the split being processed as a key. Then, it tokenizes each transaction (value) to determine different items, and emits each item as a key coupled with its split name as a value. After mappers execution, the reducer aggregates over the keys (items), and emits each key (item) coupled with its different value (split name) in the list of values (split names).

- **Job 2** *Overlapping Partitions*: The format of the MapReduce output is set to "MultiFileOutput" in the driver class. In this case, the keys will denote the name of each overlapping data partition output (we override the "generateFileNameForKeyValue" function in MapReduce to return a string as a key). In the map function, first, we store (once) the previous MapReduce job (Centroids) in a (key, value) data structure (e.g.MultiHashMap, etc.). The key in the used data structure is the split name, and the value is a list of items. Then, each mapper takes a transaction (line of text) from the database $\mathcal{D}$, and for each key in the used data structure, if there is an intersection between the values(list of items) and the transaction being processed, then the mapper emits the key as the split name (in the used data structure) and value as the transaction of $\mathcal{D}$. The reducer simply aggregates over the keys (split names) and writes each transaction of $\mathcal{D}$ to an overlapping data partition file.

Example 20. *Figure 4.1 shows a transaction database $\mathcal{D}$ with 5 transactions. In this example, we have two non-overlapping data partitions at step (1) and thus two centroids at step (2). The centroids are filtered in order to keep only the items having the maximum number of occurrences (3). IBDP intercepts each one of these two centroids with all transactions in $\mathcal{D}$. This results in two overlapping data partitions in (4) where the intersections only are kept in (5). Finally, the maximal frequent itemsets are extracted in (6). Redundancy is used for the counting process of different itemsets. For instance, transaction efg is duplicated in both partitions in (5) where the upper version participates to the frequency counting of a and the lower version participates to the frequency counting of fg.*

4.2.3 1-Job Schema: Complete Approach

We take the full advantage from IBDP data partitioning strategy and propose a powerful and robust 1-Job Schema PFIM algorithm namely PATD. PATD algorithm limits the mining process of very large database to one simple MapReduce job and exploits the natural design of MapReduce framework. Given a set of overlapping data partitions ($\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$) of a database $\mathcal{D}$ and an absolute minimum support $AMinSup$ Δ, the PATD algorithm mines each overlapping data partition S_i independently. At each mapper m_i, $i = 1, ..., n$, PATD performs CDAR algorithm on S_i. The mining process is based on the same $AMinSup$ Δ for all mappers, *i.e.,* each overlapping data partition S_i

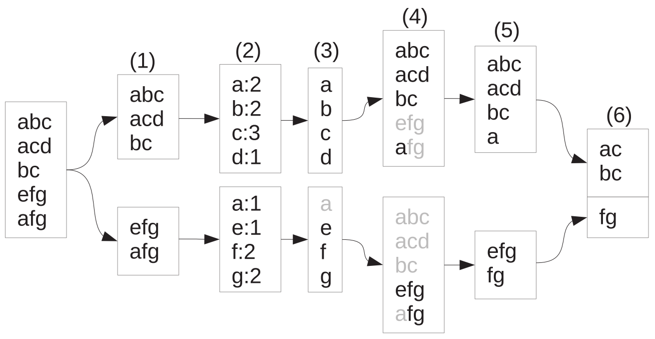


Figure 4.1 – Data Partitioning Process: (1) partitions of similar transactions are built; (2) centroids are extracted; (3) and filtered; (4) transaction are placed and filtered ; (5) to keep only the intersection of original transactions and centroids; (6) local frequent itemsets are also globally frequent.

is mined based on Δ. The mining process is carried out in parallel on all mappers. The mining result (*i.e.,* frequent itemsets) of each mapper m_i is sent to the reducer. The latter receives each frequent itemsets as its key and null as its value. The reducer aggregates over the keys (frequent itemsets) and writes the final result to a distributed file system.

The main activities of mappers and reducers in PATD algorithm are as follows:

- **Mapper:** Each mapper is given a S_i, $i = 1...m$ overlapping data partition, and a global minimum support (*i.e.,* $AMinSup$). The latter performs CDAR algorithm on S_i. Then, it emits each frequent itemset as a key and null for its value, to the reducer.

- **Reducer:** The reducer simply aggregates over the keys (frequent itemsets received from all mappers) and writes the final result to a distributed file system.

As illustrated in the mappers and reducers logic, PATD performs the mining process in one simple and efficient MapReduce job. These properties of PATD are drawn from the use of the robust data partitioning strategy IBDP. In fact, IBDP data partitioning strategy covers most of the mining complexities in PATD.

Example 21. *Lets take the example of Figure 4.1. Given an absolute minimum support $\Delta = 2$ (i.e., an itemset is considered frequent, if it appears at least in two transactions in $\mathcal{D}$). Following PATD mining principle, each mapper is given an overlapping data partition S_i as a value. In our example, we have two overlapping data partitions (5). We consider two mappers m_1 and m_2, each one of theses mappers performs a complete CDAR with $\Delta = 2$. In Figure 4.1 (5) from bottom-up : mapper m_1 mines first overlapping data partition and returns {fg} as a frequent itemset. Alike, mapper m_2 mines second overlapping data partition and returns {{ac}, {bc}}. All the results are sent to the reducer, the reducer aggregates over the keys (frequent itemsets) and outputs the final result to a distributed file system.*

4.2.4 Proof of Correctness

To prove the correctness of PATD algorithm, it is sufficient to prove that if an itemset x is frequent, then it is frequent in at least one of the partitions produced by IBDP. Since, each partition is locally mined by one mapper, then x will be found as frequent by one of the mappers. Thus, the correctness proof is done by the following lemma.

Lemma 1. *Given a database $\mathcal{D} = \{T_1, T_2, \ldots, T_n\}$, and an absolute minimum support Δ, then $\forall$ itemset x in $\mathcal{D}$ we have: $Support_{\mathcal{D}}(x) \geq \Delta \Leftrightarrow \exists\, \mathcal{P} \setminus Support_{\mathcal{P}}(x) \geq \Delta$ where $\mathcal{P}$ denotes one of the data partitions obtained by performing IBDP on $\mathcal{D}$.*

Proof.

We first prove that if $Support_{\mathcal{D}}(x) \geq \Delta$ then $\exists\, \mathcal{P} \setminus Support_{\mathcal{P}}(x) \geq \Delta$.
Let denote by $\mathcal{X}$, the set of all unique items of $\mathcal{D}$. The intersection of all transactions $\{T_1, T_2, \ldots, T_n\}$ with $\mathcal{X}$ is $\mathcal{D}$. Thus, in this particular case, $Support_{\mathcal{D}}(x) \geq \Delta \Rightarrow \exists\, \mathcal{D} \setminus Support_{\mathcal{D}}(x) \geq \Delta$. If the set of unique items $\mathcal{X}$ is partitioned into k partitions, then the intersection of each one of these k partitions with all $\{T_1, T_2, \ldots, T_n\}$ in $\mathcal{D}$, would result in a new data partition $\mathcal{P}$. Let denote by $\Pi = \{P_1, P_2, \ldots, P_k\}$, the set of all these new data partitions. For any given itemset x in $\mathcal{D}$, its total occurrence will be in one partition of Π, because, all items in $\mathcal{X}$ are shared among these partitions in Π. Therefore, $Support_{\mathcal{D}}(x) \geq \Delta \Rightarrow \exists\, I_P \setminus Support_{I_P}(x) \geq \Delta$

Next, we prove the inverse, i.e. if $\exists\, \mathcal{P} \setminus Support_{\mathcal{P}}(x) \geq \Delta$ then $Support_{\mathcal{D}}(x) \geq \Delta$.
This is done simply by using the fact that each partition $\mathcal{P}$ is a subset of $\mathcal{D}$. Hence, if the support of x in $\mathcal{P}$ is higher than Δ, then this will be the case in $\mathcal{D}$. Thus, we have: if $\exists\, \mathcal{P} \setminus Support_{\mathcal{P}}(x) \geq \Delta \Rightarrow Support_{\mathcal{D}}(x) \geq \Delta$.

Therefore, we conclude that: $Support_{\mathcal{D}}(x) \geq \Delta \Leftrightarrow \exists\, \mathcal{P} \setminus Support_{\mathcal{P}}(x) \geq \Delta$. □

4.3 Experiments

To assess the performance of PATD algorithm, we have carried out extensive experimental evaluations. In Section 4.3.1, we depict our experimental setup, and in Section 4.3.2 we investigate and discuss the results of our different experiments.

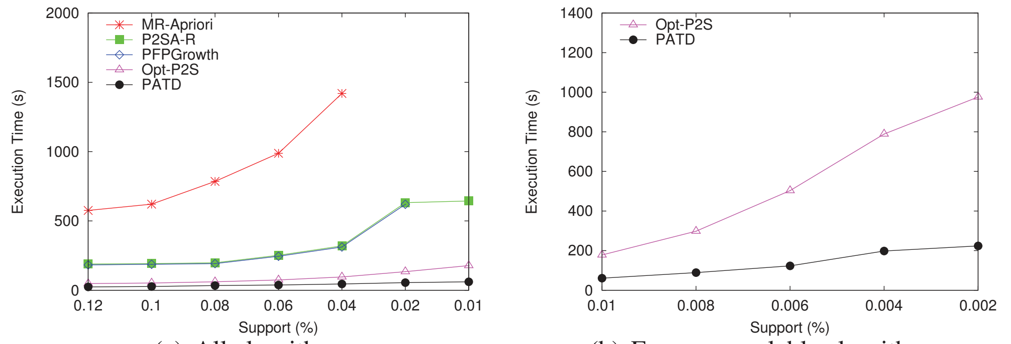
4.3.1 Experimental Setup

We implemented PATD, and all other presented algorithms on top of Hadoop-MapReduce, using Java programming language version 1.7 and Hadoop version 1.0.3. For comparing PATD performance with other PFIM alternatives, we implemented two bunches of algorithms. First, we followed SON algorithm design and implemented Parallel Two Round Apriori (P2RA) and Parallel Two Round CDAR (P2RC). These two PFIM algorithms are based on random transaction data partitioning (RTDP) and similar transaction data partitioning (STDP), respectively. Second, we designed and implemented a parallel version of standard Apriori [8] algorithm, namely Parallel Apriori (PA). For comparison with PFP-Growth [6], we adopted the default implementation provided in the Mahout [63] machine learning library (Version 0.7).

We carried out all our experiments based on the Grid5000 [64] platform, which is a platform for large-scale data processing. We have used a cluster of 16 and 48 machines respectively for Wikipedia and ClueWeb data set experiments. Each machine is equipped with Linux operating system, 64 Gigabytes of main memory, Intel Xeon $X3440$ 4 core CPUs, and 320 Gigabytes SATA hard disk.

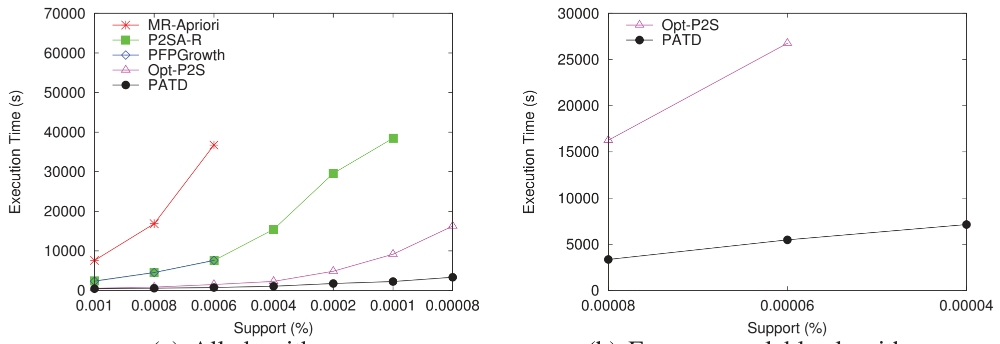
To better evaluate the performance of PATD algorithm, we used two real-world data sets. The first one is the 2014 English Wikipedia articles [65] having a total size of 49 Gigabytes, and composed of 5 million articles. The second one is a sample of ClueWeb English data set [66] with size of one Terabyte and having 632 million articles. For each data set, we performed a data cleaning task. We removed all English stop words from all articles, we obtained data sets where each article represents a transaction (items are the corresponding words in the article) to each invoked PFIM algorithm in our experiments.

In our experiments, we vary the $MinSup$ parameter value for each PFIM algorithm. We evaluate each algorithm based on its response time, its total amount of transferred data, and its energy power consumption. In particular, we consider these three different measurements, when the $MinSup$ is very low.



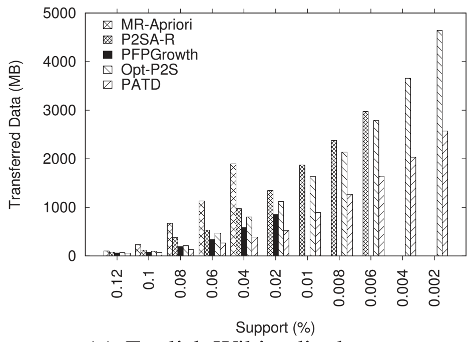
(a) All algorithms (b) Focus on scalable algorithms

Figure 4.2 – Runtime and Scalability on English Wikipedia Data Set

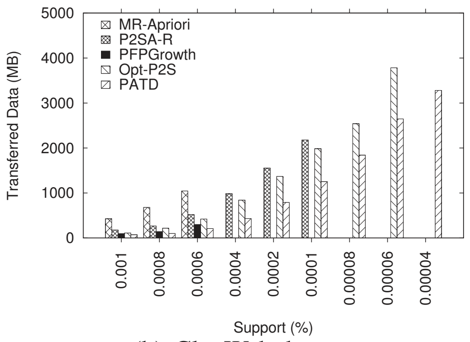


(a) All algorithms (b) Focus on scalable algorithms

Figure 4.3 – Runtime and Scalability on ClueWeb Data Set



(a) English Wikipedia data set



(b) ClueWeb data set

Figure 4.4 – Data communication

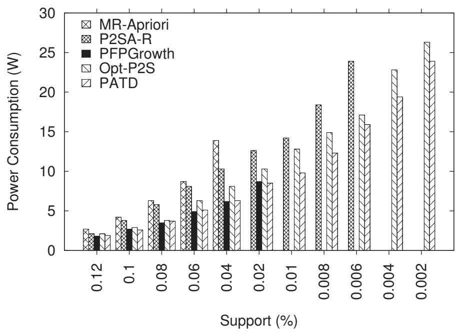


Figure 4.5 – Energy Consumption

4.3.2 Runtime and Scalability

Figures 4.2 and 4.3 give a complete view of our experiments on both English Wikipedia and ClueWeb data sets. Figures 4.2(a) and 4.2(b) report our experimental results on the whole English Wikipedia data set. Figure 4.2(a) gives an entire view on algorithms performances for a minimum support varying from 0.12% to 0.01%. We see that PA algorithm runtime grows exponentially, and gets quickly very high compared to other presented PFIM algorithms. This exponential run-time reaches its highest value with 0.04% threshold. Below this threshold, PA needs more resources (e.g. memory) than what exists in our tested machines, thus, it is impossible to extract frequent itemsets with this algorithm. Another interesting observation is that P2RA performance tends to be close to PFP-Growth until a minimum support of 0.02%. P2RA algorithm continues scaling with 0.01% while PFP-Growth does not. Although, P2RC scales with low minimum support values, PATD outperforms this algorithm in terms of running time. In particular, with a minimum support of 0.01%, PATD algorithm outperforms all other presented. This difference in the performance is better illustrated in Figure 4.2(b).

Figure 4.2(b) focuses on the differences between the four algorithms that scale in Figure 4.2(a). Although P2RC continues to scale with 0.002%, it is outperformed by

PATD in terms of running time. With 0.002% threshold, we observe a big difference in the response time between PATD and P2RC. This very good performance of PATD is due to its clever and simple mining principle, and its simple MapReduce job property that allows a low mining time.

In Figures 4.3(a) and 4.3(b), similar experiments have been conducted on the ClueWeb data set. We observe that the same order between all algorithms is kept, compared to Figures 4.2(a) and 4.2(b). There are three bunches of algorithms. One, made of PA which cannot reasonably applied to this data set, whatever the minimum support. In the second bunch, we see that PFP-Growth suffers from the same limitations as could be observed on the Wikipedia data set in Figure 4.2(a), and it follows a behavior that is very similar to that of P2RA, until it becomes impossible to execute. P2RA continues scaling until stops executing with a minimum support of 0.0001%. In the third bunch of algorithms, we see P2RC and PATD scale until 0.00008%. We decreased the minimum support parameter, and we zoom on these two algorithms. As shown in Figure 4.3(b), we observe a very good performance of PATD compared to P2RC. The P2RC algorithm becomes inoperative with a minimum support below 0.00006%, while PATD continues scaling very well. This big difference in the performance behavior between PATD and all other presented algorithms, shows the high capability of PATD in terms of scaling and response time. With both, Gigabytes and Terabytes of data, PATD gives a very good and significant performance. Whatever, the data size, the number of transactions, and the minimum support, PATD scales and achieves very good results.

4.3.3 Data Communication and Energy Consumption

Let's now study the amount of data transferred over the network for executing different PFIM algorithms. Figure 4.4(a) shows the transferred data (in mega bytes) of each presented algorithm on Wikipedia data set. We observe in this figure that PA has the highest peak, this is simply due to its several round of MapReduce executions. In other hand, we see that P2RA, P2RC and PFP-Growth represent smaller peaks. Among all the presented algorithms in Figure 4.4(a), we clearly distinguish PATD algorithm. We can see that whatever the used $MinSup$, PATD does not allow much data transfer compared to other algorithms. This is because PATD does not rely on chains of jobs like other presented alternatives. In addition, contrary to other PFIM algorithms, PATD limits the mappers from emitting non frequent itemsets. Therefore, PATD algorithm does not allow the transmission of useless data (itemsets).

In Figure 4.4(b), we report the results of the same experiment on ClueWeb data set. We observe that PATD algorithm always has the lowest peak in terms of transferred data comparing to other algorithms.

We also measured the energy consumption of the compared algorithms during their execution. For measuring the power consumption, we used the Grid5000 tools that measure the power consumption of the nodes during a job execution. Figure 4.5 shows the total amount of the power consumption of each presented PFIM algorithm. We observe in Figure 4.5, that the consumption increases when decreasing the minimum support for each algorithm. We see that PATD still gives a lower consumption comparing to other algorithms. Taking the advantage from its parallel design, PATD allows a high parallel computational execution. This, impacts the mining runtime to be fast, which is in turn, allows for a fast convergence of the algorithm and thus, a less consumption of the energy. PATD also transfers less data over the network, and this is another reason for its lower energy consumption.

4.4 Conclusion

We proposed a reliable and efficient MapReduce based parallel frequent itemset algorithm, namely PATD, that has shown significantly efficient in terms of; i) runtime and scalability; i) low data communication; and low energy consumption. PATD algorithm takes the advantage of an efficient data partitioning technique namely IBDP. IBDP data partitioning strategy allows for an optimized data placement on MapReduce. This placement technique has not been investigated before this work. It allows PATD algorithm to exhaustively and quickly mine very large databases. Such ability to use very low minimum supports is mandatory when dealing with Big Data and particularly hundreds of Gigabytes like what we have done in our experiments. Our results show that PATD algorithm dramatically outperforms other existing PFIM alternatives, and makes the difference between an inoperative and a successful extraction.

Chapter 5

Fast Parallel Mining of Maximally Informative K-Itemsets

In this chapter, we address the problem of mining maximally informative k-itemsets (*miki*) in big data. The discovery of informative itemsets is a fundamental building block in data analytics and information retrieval. While the problem has been widely studied, only few solutions scale. This is particularly the case when i) the data set is massive, calling for large-scale distribution, and/or ii) the length K of the informative itemset to be discovered is high. We propose PHIKS (Parallel Highly Informative K-itemSets) a highly scalable, parallel miki mining algorithm. PHIKS renders the mining process of large scale databases (up to Terabytes of data) succinct and effective. Its mining process is made up of only two compact, yet efficient parallel jobs. PHIKS uses a clever heuristic approach to efficiently estimates the joint entropies of miki having different sizes with very low upper bound error rate, which dramatically reduces the runtime process.

The *miki* problem is formally defined in Section 2.1.2 of chapter 2. We introduce our PHIKS algorithm in details, in Section 5.3. In Section 5.4, we evaluate our proposal with very large real-world data sets. Our different experimental results confirm the effectiveness of our proposal by the significant scale-up obtained with high featuresets length and hundreds of millions of objects.

5.1 Motivation and Overview of the Proposal

Featureset, or itemset, mining [70] is one of the fundamental building bricks for exploring informative patterns in databases. Features might be, for instance, the words occurring in a document, the score given by a user to a movie on a social network, or the characteristics of plants (growth, genotype, humidity, biomass, etc.) in a scientific study in agronomic. A large number of contributions in the literature has been proposed for itemset mining, exploring various measures according to the chosen relevance criteria. The most studied measure is probably the number of co-occurrences of a set of features, also known as frequent itemsets [35]. However, frequency does not give relevant results for a various range of applications, including information retrieval [71], since it does not give a complete overview of the hidden correlations between the itemsets in the database. This is particularly the case when the database is sparse [37]. Using other criteria to assess the informativeness of an itemset could result in discovering interesting new patterns that were not previously known. To this end, information theory [15] gives us strong supports for measuring the informativeness of itemsets. One of the most popular measures is the joint entropy [15] of an itemset. An itemset X that has higher joint entropy brings up more information about the objects in the database.

We study the problem of Maximally Informative k-Itemsets (*miki* for short) discovery in massive data sets, where informativeness is expressed by means of joint entropy and k is the size of the itemset [72, 7, 42]. *Miki* are itemsets of interest that better explain the correlations and relationships in the data. Example 15 gives an illustration of *miki* and its potential for real world applications such as information retrieval.

Miki mining is a key problem in data analytics with high potential impact on various tasks such as supervised learning [24], unsupervised learning [73] or information retrieval [71], to cite a few. A typical application is the discovery of discriminative sets of features, based on joint entropy [15], which allows distinguishing between different categories of objects. Unfortunately, it is very difficult to maintain good results, in terms of both response time and quality, when the number of objects becomes very large. Indeed, with massive amounts of data, computing the joint entropies of all itemsets in parallel is a very challenging task for many reasons. First, the data is no longer located in one computer, instead, it is distributed over several machines. Second, the number of iterations of parallel jobs would be linear to k (*i.e.,* the number of features in the itemset to be extracted [7]), which needs multiple database scans and in turn violates the parallel execution of the mining process. We believe that an efficient *miki* mining solution should scale up with the increase in the size of the itemsets, calling for cutting edge parallel algorithms and high performance evaluation of an itemset's joint entropy in massively distributed environments.

We propose a deep combination of both information theory and massive distribution by taking advantage of parallel programming frameworks such as MapReduce [74] or Spark [75]. To the best of our knowledge, there has been no prior work on parallel informative itemsets discovery based on joint entropy. We designed and developed an efficient parallel algorithm, namely Parallel Highly Informative K-itemSet (PHIKS in short), that renders the discovery of *miki* from a very large database (up to Terabytes of data) simple and effective. It performs the mining of *miki* in two parallel jobs. PHIKS cleverly exploits available data at each mapper to efficiently calculate the joint entropies of *miki* candidates. For more efficiency, we provide PHIKS with optimizations that allow for very significant improvements of the whole process of *miki* mining. The first technique estimates the upper bound of a given set of candidates and allows for a dramatic reduction of data communications, by filtering unpromising itemsets without having to perform any additional scan over the data. The second technique reduces significantly the number of scans over the input database of each mapper, *i.e.,* only one scan per step, by incrementally computing the joint entropy of candidate features. This reduces drastically the work that should be done by the mappers, and thereby the total execution time.

PHIKS has been extensively evaluated using massive real-world data sets. Our experimental results show that PHIKS significantly outperforms alternative approaches, and confirm the effectiveness of our proposal over large databases containing for example one Terabyte of data.

The rest of this chapter is structured as follows. Section 5.2 gives the necessary background. In Section 5.3, we propose our PHIKS algorithm, and depict its whole core mining process. Section 5.4 reports on our experimental validation over real-world data sets. Section 5.5 concludes.

5.2 Background

In this Section, we detail the *miki* discovery in a centralized environment.

5.2.1 Miki Discovery in a Centralized Environment

In [7], an effective approach is proposed for *miki* discovery in a centralized environment. Their *ForwardSelection* heuristic uses a "generating-pruning" approach, which is similar to the principle of *Apriori* [35]. i_1, the feature having the highest entropy is selected as a seed. Then, i_1 is combined with all the remaining features, in order to build candidates. In other words, there will be $|\mathcal{F}-1|$ candidates (*i.e.*, $(i_1, i_2), (i_1, i_3), \ldots, (i_1, i_{|\mathcal{F}-1|})$). The entropy of each candidate is given by a scan over the database, and the candidate having the highest entropy, say (i_1, i_2), is kept. A set of $|\mathcal{F}-2|$ candidates of size 3 is generated (*i.e.*, $(i_1, i_2, i_3), (i_1, i_2, i_4), \ldots, (i_1, i_2, i_{|\mathcal{F}-2|})$) and their entropy is given by a new scan over the database. This process is repeated until the size of the extracted itemset is k.

5.3 PHIKS Algorithm

In a massively distributed environment, a possible naive approach for *miki* mining would be a straightforward implementation of *ForwardSelection* [7] (see Section 5.2.1). However, given the "generating-pruning" principle of this heuristic, it is not suited for environments like Spark [75] or MapReduce [74] and would lead to very bad performances. The main reason is that each scan over the data set is done through a distributed job (i.e., there will be k jobs, one for each generation of candidates that must be tested over the database). Our experiments, in Section 5.4, give an illustration of the catastrophic response times of *ForwardSelection* in a straightforward implementation on MapReduce (the worst, for all of our settings). This is not surprising since most algorithms designed for a centralized itemset mining do not perform well in massively distributed environments in a direct implementation [76], [77], [78], and *miki* don't escape that rule.

Such an inadequacy calls for new distributed algorithmic principles. To the best of our knowledge, there is no previous work on distributed mining of *miki*. However, we may build on top of cutting edge studies in frequent itemset mining, while considering the very demanding characteristics of *miki*.

Interestingly, in the case of frequent itemsets in MapReduce, a mere algorithm consisting of two jobs outperforms most existing solutions [79] by using the principle of SON [80], a divide and conquer algorithm. Unfortunately, despite its similarities with frequent itemset mining, the discovery of *miki* is much more challenging. Indeed, the number of occurrences of an itemset X in a database $\mathcal{D}$ is additive and can be easily distributed (the global number of occurrences of X is simply the sum of its local numbers of occurrences on subsets of $\mathcal{D}$). Entropy is much more combinatorial since it is based on the the projection counting of X in $\mathcal{D}$ and calls for efficient algorithmic advances, deeply combined with the principles of distributed environments.

5.3.1 Distributed Projection Counting

Before presenting the details of our contribution, we need to provide tools for computing the projection of an itemset X on a database $\mathcal{D}$, when $\mathcal{D}$ is divided into subsets on different splits, in a distributed environment, and entropy has to be encoded in the key-value format. We have to count, for each projection p of X, its number of occurrences on $\mathcal{D}$. This can be solved with an association of the itemset as a key and the projection as a value. On a split, for each projection of an itemset X, X is sent to the reducer as the key coupled with its projection. The reducer then counts the number of occurrences, on all the splits, of each (key:value) couple and is therefore able to calculate the entropy of each itemset. Communications may be optimized by avoiding to emit a $key : val$ couple when the projection does not appear in the transaction and is only made of '0' (on the reducer, the number of times that a projection p of X does not appear in $\mathcal{D}$ is determined by subtracting the number projections of X in D from $|\mathcal{D}|$).

Example 22. *Let us consider $\mathcal{D}$, the database of Table 2.1, and the itemset $X = (D, E)$. Let us consider that $\mathcal{D}$ is divided into two splits $S_1 = \{d_1..d5\}$ and $S_2 = \{d_6..d_{10}\}$. With one simple MapReduce job, it is possible to calculate the entropy of X. The algorithm of a mapper would be the following: for each document d, emit a couple $(key : val)$ where $key = X$ and $val = proj(X, d)$. The first mapper (corresponding to S_1) will emit the following couples: $((D, E) : (1, 1))$ 4 times and $((D, E) : (0, 1))$ once. The second mapper will emit $((D, E) : (1, 1))$ 5 times. The reducers will do the sum and the final result will be $((D, E) : (1, 1))$ occurs 9 times and $(((D, E) : (0, 1))$ once.*

5.3.2 Discovering *miki* in Two Rounds

Our heuristic will use at most two MapReduce jobs in order to discover the k-itemset having the highest entropy. The goal of the first job is to extract locally, on the distributed subsets of $\mathcal{D}$, a set of candidate itemsets that are likely to have a high global entropy. To that end, we apply the principle of *ForwardSelection* locally, on each mapper, and grow an itemset by adding a new feature at each step. After the last scan, for each candidate itemset

Split	A	B	C	D	E
S_1	0	0	1	0	0
	0	1	0	0	0
	1	0	1	0	0
	1	1	0	0	0
	0	1	1	0	1
	1	0	0	0	0
S_2	0	0	0	0	1
	0	1	0	1	1
	1	0	0	0	1
	1	1	0	1	1
	0	1	0	0	0
	1	0	0	1	1

Table 5.1 – Local Vs. Global Entropy

X of size k we have the projection counting of X on the local data set. A straightforward approach would be to emit the candidate itemset having the highest local entropy. We denote by *local entropy*, the entropy of an itemset in a subset of the database that is read by a mapper (*i.e.,* by considering only the projections of X in the mapper). Then the reducers would collect the local *miki* and we would check their *global entropy* (*i.e.,* the entropy of the itemset X in the entire database $\mathcal{D}$) by means of a second MapReduce job. Unfortunately, this approach would not be correct, since an itemset might have the highest global entropy, while actually not having the highest entropy in each subset. Example 23 gives a possible case where a global *miki* does not appear as a local *miki* on any subset of the database.

Example 23. *Let us consider $\mathcal{D}$, the database given by Table 5.1, which is divided into two splits of six transactions. The global miki of size 3 in this database is (A, B, E). More precisely, the entropy of (A, B, E) on $\mathcal{D}$ is given by $-\frac{1}{12} \times log(\frac{1}{12}) \times 4 - \frac{2}{12} \times log(\frac{2}{12}) \times 4 = 2.92$. However, if we consider each split individually, (A, B, E) always has a lower entropy compared to at least one different itemset. For instance, on the split S_1, the projections of (A, B, E) are $(0, 0, 0)$, $(0, 1, 0)$, $(1, 1, 0)$ and $(0, 1, 1)$ with one occurrence each, and $(1, 0, 0)$ with two occurrences. Therefore the entropy of (A, B, E) on S_1 is 2.25 (i.e., $-\frac{1}{6} \times log(\frac{1}{6}) \times 4 - \frac{2}{6} \times log(\frac{2}{6}) = 2.25$). On the other hand, the projections of (A, B, C) on S_1 are $(0, 0, 1)$, $(0, 1, 0)$, $(1, 0, 1)$, $(1, 1, 0)$, $(0, 1, 1)$ and $(1, 0, 0)$ with one occurrence each, and the entropy of (A, B, C) on S_1 is 2.58 (i.e., $-\frac{1}{6} \times log(\frac{1}{6}) \times 6 = 2.58$). This is similar on S_2 where the entropy of (A, B, E) is 2.25 and the entropy of (A, B, D) is 2.58. However, (A, B, C) and (A, B, D) both have a global entropy of 2.62 on $\mathcal{D}$, which is lower than 2.92, the global entropy of (A, B, E) on $\mathcal{D}$.*

Since it is possible that a global *miki* is never found as a local *miki*, we need to consider a larger number of candidate itemsets. This can be done by exploiting the set of candidates that are built in the very last step of *ForwardSelection*. This step aims to calculate the projection counting of $\mathcal{F} - k$ candidates and then compute their local entropy. Instead of only emitting the itemset having the larger entropy, we will emit, for each candidate X, the projection counting of X on the split, as explained in Section 5.3.1. The reducers will then be provided with, for each local candidate X_i ($1 \leq i \leq m$, where m is the number of mappers, or splits), the projection counting of X on a subset of $\mathcal{D}$. The main idea is that the itemset having the highest entropy is highly likely to be in that set of candidates. For instance, in the database given by Table 5.1 and $k = 3$, the global *miki* is (A, B, E), while the local *miki* are (A, B, C) on S_1 and (A, B, D) on S_2. However, with the technique described above, the itemset (A, B, E) will be a local candidate, and will be sent to the reducers with the whole set of projections encountered so far in the splits. The reducer will then calculate its global entropy, compare it to the entropy of the other itemsets, and (A, B, E) will eventually be selected as the *miki* on this database.

Unfortunately, it is possible that X has not been generated as a candidate itemset on the entire set of splits (consider a biased data distribution, where a split contains some features with high entropies, and these features have low entropies on the other splits). Therefore, we have two possible cases at this step:

1. X is a candidate itemset on all the splits and we are able to calculate its exact projection counting on $\mathcal{D}$, by means of the technique given in Section 5.3.1.

2. There is (at least) one split where X has not been generated as a candidate and we are not able to calculate its exact projection counting on $\mathcal{D}$.

The first case does not need more discussion, since we have collected all the necessary information for calculating the entropy of X on $\mathcal{D}$. The second case is more difficult since X might be the *miki* but we cannot be sure, due to lack of information about its local entropy on (at least) one split. Therefore, we need to check the entropy of X on $\mathcal{D}$ with a second MapReduce job intended to calculate its exact projection counting. The goal of this second round is to check that no local candidate has been ignored at the global scale. At the end of this round, we have the entropy of all the promising candidate itemsets and we are able to pick the one with the highest entropy. This is the architecture of our approach, the raw version of which (without optimization) is called *Simple-PHIKS*. So far, we have designed a distributed architecture and a *miki* extraction algorithm that, in our experiments reported in Section 5.4 outperforms *ForwardSelection* by several orders of magnitude. However, by exploiting and improving some concepts of information theory, we may significantly optimize this algorithm and further accelerate its execution at different parts of the architecture, as explained in the following sections.

5.3.3 Candidate Reduction Using Entropy Upper Bound

One of the shortcomings of the basic version of our two rounds approach is that the number of candidate itemsets, which should be processed in the second job, may be high for large databases as it will be illustrated by our experiments in Section 5.4. This is particularly the case when the features are not uniformly distributed in the splits of mappers. These candidate itemsets are sent partially by the mappers (*i.e.,* not by all of them), thus we cannot compute their total entropy in the corresponding reducer. This is why, in the basic version of our approach, we compute their entropy in the second job by reading again the database.

Here, we propose an efficient technique for significantly reducing the number of candidates. The main idea is to compute an upper bound for the entropy of the partially sent itemsets, and discard them if they have no chance to be a global *miki*. For this, we exploit the available information about the *miki* candidates sent by the mappers to the corresponding reducer.

Let us describe formally our approach. Let X be a partially sent itemset, and m be a mapper that has not sent X and its projection frequencies to the reducer R that is responsible for computing the entropy of X. In the reducer R, the frequency of X projections for a part of the database is missing, *i.e.,* in the split of m. We call these frequencies as *missing* frequencies. We compute an upper bound for the entropy of X by estimating its missing frequencies. This is done in two steps. Firstly, finding the biggest subset of X, say Y, for which all frequencies are available and secondly, distributing the frequencies of Y among the projections of X in such a way that the entropy of X be the maximum.

Step 1: The idea behind the first step is that the frequencies of the projections of an itemset X can be derived from the projections of its subsets. For example, suppose two itemsets $X = \{A, B, C, D\}$ and $Y = \{A, B\}$, then the frequency of the projection $p = (1, 1)$ of Y is equal to the sum of the following projections in X: $p_1 = (1, 1, 0, 0)$, $p_2 = (1, 1, 0, 1)$, $p_3 = (1, 1, 1, 0)$ and $p_4 = (1, 1, 1, 1)$. The reason is that in all these four projections, the features A and B exist, thus the number of times that p occurs in the database is equal to the total number of times that the four projections p_1 to p_4 occur. This is stated by the following lemma.

Lemma 2. *Let the itemset Y be a subset of the itemset X, i.e., $Y \subseteq X$. Then, the frequency of any projection p of Y is equal to the sum of the frequencies of all projections of X which involve p.*

Proof. The proof can be easily done as in the above discussion.

In Step 1, among the *available subsets* of itemset X, *i.e.,* those for which we have all projection frequencies, we choose the one that has the highest size. The reason is that its intersection with X is the highest, thus our estimated upper bound about the entropy of X will be closer to the real one.

Step 2: let Y be the biggest available subset of X in reducer R. After choosing Y, we distribute the frequency of each projection p of Y among the projections of X that are derived from p. There may be many ways to distribute the frequencies. For instance, in the example of Step 1, if the frequency of p is 6, then the number of combinations for distributing 6 among the four projections p_1 to p_4 is equal to the solutions which can be found for the following equation: $x_1 + x_2 + x_3 + x_4 = 6$ when $x_i \geq 0$. In general, the number of ways for distributing a frequency f among n projections is equal to the number of solutions for the following equation:

$$x_1 + x_2 + ... + x_n = f \; for \; x_i \geq 0$$

Obviously, when f is higher than n, there is a lot of solutions for this equation. Among all these solutions, we choose a solution that maximizes the entropy of X. The following lemma shows how to choose such a solution.

Lemma 3. *Let $\mathcal{D}$ be a database, and X be an itemset. Then, the entropy of X over $\mathcal{D}$ is the maximum if the possible projections of X over $\mathcal{D}$ have the same frequency.*

Proof. The proof is done by implying the fact that in the entropy definition (see Definition 10), the maximum entropy is for the case where all possible combinations have the same probability. Since, the probability is proportional to the frequency, then the maximum entropy is obtained in the case where the frequencies are the same. □

The above lemma proposes that for finding an upper bound for the entropy of X (*i.e.,* finding its maximal possible entropy), we should distribute equally (or almost equally) the frequency of each projection in Y among the derived projections in X. Let f be the frequency of a projection in Y and n be the number of its derived projections, if (f modulo n) = 0 then we distribute equally the frequency, otherwise we first distribute the quotient among the projections, and then the rest randomly.

After computing the upper bound for entropy of X, we compare it with the maximum entropy of the itemsets for which we have received all projections (so we know their real entropy), and discard X if its upper bound is less than the maximum found entropy until now.

5.3.4 Prefix/Suffix

When calculating the local *miki* on a mapper, at each step we consider a set of candidates having size j that share a prefix of size $j-1$. For instance, with the database of Table 5.1 and the subset of split S_1, the corresponding mapper will extract (A, B) as the *miki* of size 2. Then, it will build 3 candidates: (A, B, C), (A, B, D) and (A, B, E). A straightforward approach for calculating the joint entropy of these candidates would be to calculate their projection counting by means of an exhaustive scan over the data of S_1 (*i.e.*, read the first transaction of S_1, compare it to each candidate in order to find their projections, and move to the next transaction). However, these candidates share a prefix of size 2: (A, B). Therefore, we store the candidates in a structure that contains the prefix itemset, of size $j-1$, and the set of $|\mathcal{F} - j|$ suffix features. Then, for a transaction T, we only need to i) calculate $proj(p, T)$ where p is the prefix and ii) for each suffix feature f, find the projection of f on T, append $proj(f,T)$ to $proj(p,T)$ and emit the result. Let us illustrate this principle with the example above (*i.e.*, first transaction of S_1 in Table 5.1). The structure is as follows: {prefix=(A,B):suffixes=C,D,E}. With this structure, instead of comparing (A,B,C), (A,B,D) and (A,B,E) to the transaction and find their respective projections, we calculate the projection of (A,B), their prefix, *i.e.*, $(0,0)$, and the projection of each suffix, *i.e.*, (1), (0) and (0) for C, D, and E respectively. Each suffix projection is then added to the prefix projection and emitted. In our case, we build three projections: $(0,0,1)$, $(0,0,0)$ and $(0,0,0)$, and the mapper will emit $((A,B,C):(0,0,1))$, $((A,B,D):(0,0,0))$ and $((A,B,E):(0,0,0))$.

5.3.5 Incremental Entropy Computation in Mappers

In the basic version of our two rounds approach, each mapper performs many scans over its split to compute the entropy of candidates and finally find the local miki. Given k as the size of the requested itemset, in each step j of the k steps in the local miki algorithm, the mapper uses the itemset of size $j-1$ discovered so far, and builds $|F|-j$ candidate itemsets before selecting the one having the highest entropy. For calculating each joint entropy, a scan of the input split is needed in order to compute the frequency (and thus the probability) of projections. Let $|F|$ be the number of features in the database, then the number of scans done by each mapper is $O(k*|F|)$. Although the input split is kept in memory, this high number of scans over the split is responsible for the main part of the time taken by the mappers.

In this Section, we propose an efficient approach to significantly reduce the number of scans. Our approach that incrementally computes the joint entropies, needs to do in each step just one scan of the input split. Thus, the number of scans done by this approach is $O(k)$.

To incrementally compute the entropy, our approach takes advantage of the following lemma.

Lemma 4. *Let X be an itemset, and suppose we make an itemset Y by adding a new feature i to X, i.e., $Y = X + \{i\}$. Then, for each projection p in X two projections $p_1 = p.0$, and $p_2 = p.1$ are generated in Y, and the sum of the frequency of p_1 and p_2 is equal to that of p, i.e., $f(p) = f(p_1) + f(p_2)$.*

proof. The projections of Y can be divided into two groups: 1) those that represent transactions containing i; 2) those representing the transactions that do not involve i. For each projection p_1 in the first group, there is a projection p_2 in the second group, such that p_1 and p_2 differ only in one bit, *i.e.,* the bit that represents the feature i. If we remove this bit from p_1 or p_2, then we obtain a projection in X, say p, that represents all transactions that are represented by p_1 or p_2. Thus, for each project p in X, there are two projections p_1 and p_2 in Y generated from p by adding one additional bit, and the frequency of p is equal to the sum of the frequencies of p_1 and p_2. $\square$

Our incremental approach for *miki* computing proceeds as follows. Let X be the miki in step j . Initially, we set $X = \{\}$, with a null projection whose frequency is equal to n, *i.e.,* the size of the database. Then, in each step j ($1 \leq j \leq k$), we do as follows. For each remaining feature $i \in F - X$, we create a hash map $h_{i,j}$ containing all projections of the itemset $X + \{i\}$, and we initiate the frequency of each projection to zero. Then, we scan the set of transactions in the input split of the mapper. For each transaction t, we obtain a set S that is the intersection of t and $F - X$, *i.e.,* $S = t \cap (F - X)$. For each feature $i \in S$, we obtain the projection of t over $X + \{i\}$, say p_2, and increment by one the frequency of the projection p_2 in the hash map $h_{i,j}$. After scanning all transactions of the split, we obtain the frequency of all projections ending with 1. For computing the projections ending with 0, we use Lemma 4 as follows. Let $p.0$ be a projection ending with 0, we find the projection $p.1$ (*i.e.,* the projection that differs only in the last bit), and set the frequency of $p.0$ equal to the frequency of p minus that of $p.1$, *i.e.,* $f(p.0) = f(p) - f(p.1)$. By this way, we compute the frequency of projections ending with 0.

After computing the frequencies, we can compute the entropy of itemset $X + \{i\}$, for each feature $i \in F - X$. At the end of each step, we add to X the feature i whose joint entropy with X is the highest. We keep the hash map of the selected itemset, and remove all other hash maps including that of the previous step. Then, we go to the next step until finishing step k. Notice that to obtain the frequency of p in step j, we use the hash map of the previous step, *i.e.,* $H_{i,j-1}$, this is why, at each step we keep the hash map of the selected miki.

Let us now prove the correctness of our approach using the following Theorem.

Theorem 5. *Given a database $\mathcal{D}$, and a value k as the size of requested miki. Then, our incremental approach computes correctly the entropy of the candidate itemsets in all steps.*

proof. To prove the correctness of our approach, it is sufficient to show that in each step the projection frequencies of $X + \{i\}$ are computed correctly. We show this by induction on the number of steps, *i.e.,* j for $1 \leq j \leq k$.

Base. In the first step, the itemset $X + \{i\} = \{i\}$ because initially $X = \{\}$. There are two projections for $\{i\} : p_1 = (0)$ and $p_2 = (1)$. The frequency of p_2 is equal to the number of transactions containing i. Thus during the scan of the split, we correctly set the frequency of p_2. Since there is no other projection for i, the frequency of p_1 is equal to $n - f(p_2)$, where n is the size of the database. This frequency is found correctly by our approach. Thus, for step $j = 1$ our approach finds correctly the projection frequencies of $X + \{i\}$.

Induction. we assume that our approach works correctly in step $j - 1$, then we prove that it will work correctly in step j. The proof can be done easily by using Lemma 4. According this lemma, for each projection p in step $j - 1$ there are two projections $p_1 = (p.0)$, and $p_2 = (p.1)$ in step j. The frequency of p_2 is computed correctly during the scan of the split. We assume that the frequency of p has been correctly computed in step $j - 1$. Then, Lemma 4 implies that the frequency of p_1 has been also well computed since we have $f(p) = f(p_1) + f(p_2)$. $\square$

5.3.6 Complete Approach

Our approach depicts the core mining process of Parallel Highly Informative K-itemSet Algorithm (PHIKS). The major steps of PHIKS algorithm for *miki* discovery are summarized in Algorithms 3 and 4. Algorithm 3 depicts the mining process of the first MapReduce job of PHIKS, while Algorithm 4 depicts the mining process of its second MapReduce job.

Algorithm 3: PHIKS: Job1

```
Input: $n$ data splits $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ of a database $\mathcal{D}$, $K$ the size of miki
Output: A miki of Size $K$
//Mapper Class 1
map( key: Line Offset: $\mathcal{K}_1$, $value$ = Whole $S_i$: $\mathcal{V}_1$ )
    - $\mathcal{F}_i \leftarrow$ the set of features in $S_i$
    - $\forall f \in \mathcal{F}_i$ compute $H(f)$ on $S_i$, using prefix/suffix
    - $n \leftarrow 1$ // current size of itemsets
    - HInFS $\leftarrow max(H(f)), \forall f \in \mathcal{F}_i$
    // HInFS is the itemset of size $n$
    // having the highest entropy
    while $i \neq k$ do
        - $n++$
        - $\mathcal{C}_n \leftarrow$ BuildCandidates(HInFS, $\mathcal{F}_i \backslash$HInFS)
        - $\forall c \in \mathcal{C}_p, H(c_i) \leftarrow$ ComputeJointEntropy($c$, $S_i$)
        - HInFS $\leftarrow max(H(c)), \forall c \in \mathcal{C}_n$
    // $\mathcal{C}_k$ contains all the candidate itemsets of size $k$
    // and $\forall c \in \mathcal{C}_k$, the joint entropy of $c$ is in $H(c_i)$
    for $c \in \mathcal{C}_k$ do
        - $\mathcal{P}_c \leftarrow$ projections($c$, $S_i$)
        for $p \in \mathcal{P}_c$ do
            - emit($key = c : value = p$)
//Reducer Class 1
reduce( key: itemset $c$,
      $list(values)$: projections($c$) )
    if $c$ has been emitted by all the mappers then
        // We have all the projections of $c$ on $\mathcal{D}$
        // we store its entropy in a file "complete"
        - $H(c) \leftarrow$ IncrJointEntropy($c$,projections($c$))
        - emit($c$,$H(c)$) in a file Complete
    else
        // Missing nformation. We have to estimate
        // the upper bound of c's joint entropy
        // and store it in a file "Incomplete"
        - $Est \leftarrow$ UpperBound($c$,projections($c$))
        - emit($c$, $Est$) in a file "Incomplete"
close( )
    - $C_{max} \leftarrow$ CandidateWithMaxEntropy("Complete")
    - emit($C_{max}$, $H(C_{max})$)
          in a file "CompleteMaxFromJob1"
    for $c \in$ "Incomplete" do
        if $Est(c) > H(C_{max})$ then
            // c is potentially a miki, it has
            // to be checked over $\mathcal{D}$
            - emit(c,Null) in a file "ToBeTested"
```

Algorithm 4: PHIKS: Job2

```
Input: Database $\mathcal{D}$, $K$ miki Size
Output: Tested miki of Size $K$
//Mapper Class 2
map( $key$: Line Offset: $\mathcal{K}_1$, $value$ = Transaction: $\mathcal{V}_1$ )
    - Read file 'ToBeTested' from Job1 (once) in the mapper
    - $\mathcal{F} \leftarrow$ set of itemsets in 'ToBeTested'
    for $f \in \mathcal{F}$ do
        - $p \leftarrow$ projections($f$, $\mathcal{V}_1$)
        emit ($key$: $f$, $value$: $p$)

//Reducer Class 2
reduce( $key$: itemset $f$,
        $list(values)$: projections($f$) )
    // we have all the projections of $f$ on $\mathcal{D}$ that come
    // from all mappers
    // we compute its joint entropy and we write the result to a file
    // "CompleteFromJob2"
    - $H(f) \leftarrow$ IncrJointEntropy($f$,projections($f$))
    - write($f$, $H(f)$) to a file "CompleteFromJob2" in HDFS
    // optional, we emit the result of use later, from the close() method
    - emit ($key$: $f$, $value$: $H(f)$)
close( )
    // emit miki having highest joint entropy
    - read file "CompleteMaxFromJob1"
    - read file "CompleteFromJob2"
    - Max $\leftarrow$ max("CompleteMaxFromJob1",
            "CompleteFromJob2")
    - emit(miki,Max)
```

Data Set	# of Transactions	# of Items	Size
Amazon Reviews	34 millions	31721	34 Gigabyte
English Wikipedia	5 millions	23805	49 Gigabytes
ClueWeb	632 millions	141826	1 Terabyte

Table 5.2 – Data Sets Description

5.4 Experiments

To evaluate the performance of PHIKS, we have carried out extensive experimental tests. In Section 5.4.1, we depict our experimental setup and its main configurations. In Section 5.4.2, we depict the different used data sets in our various experiments. Lastly, in Section 5.4.3, we thoroughly analyze and investigate our different experimental results.

5.4.1 Experimental Setup

We implemented PHIKS algorithm on top of Hadoop-MapReduce using Java programming language version 1.7 and Hadoop [81] version 1.0.3. For comparison, we implemented a parallel version of *ForwardSelection* [7] algorithm. To specify each presented algorithm, we adopt the notations as follow. We denote by 'PFWS' a parallel implementation of *ForwardSelection* algorithm, by 'Simple-PHIKS' an implementation of our basic two rounds algorithm without any optimization, and by 'Prefix' an extended version of Simple-PHIKS algorithm that uses the Prefix/Suffix method for accelerating the computations of the projection values. We denote by 'Upper-B' a version of our algorithm that reduces the number of candidates by estimating the joint entropies of *miki* based on an upper bound joint entropy. We denote by 'Upper-B-Prefix' an extended version of Upper-B algorithm that employs the technique of prefix/suffix. Lastly, we denote by 'PHIKS' an improved version of Upper-B-Prefix algorithm that uses the method of incremental entropy for reducing the number of data split scans at each mapper.

We carried out all our experiments on the Grid5000 [64] platform, which is a platform for large-scale data processing. In our experiments, we have used clusters of 16 and 48 nodes respectively for Amazon Reviews, Wikipedia data sets and ClueWeb data set. Each machine is equipped with Linux operating system, 64 Gigabytes of main memory, Intel Xeon X3440 4 core CPUs and 320 Gigabytes SATA hard disk.

In our experiments, we measured three metrics: 1) the response time of the compared algorithms, which is the time difference between the beginning and the end of a maximally informative k-itemsets mining process; 2) the quantity of transferred data (*i.e.,* between the mappers and the reducers) of each maximally informative k-itemsets mining process; 3) the energy consumption for each maximally informative k-itemsets mining process. To this end, we used the metrology API and Ganglia infrastructure of the Grid5000 platform that allow to measure the energy consumption of the nodes during an experiment.

Basically, in our experiments, we consider the different performance measurements when the size k of the itemset (*miki* to be discovered) is high.

5.4.2 Data Sets

To better evaluate the performance of PHIKS algorithm, we used three real-world data sets as described in Table 5.2. The first one is the whole 2013 Amazon Reviews data set [82], having a total size of 34 Gigabytes and composed of 35 million reviews. The second data set is the 2014 English Wikipedia data set [65], having a total size of 49 Gigabytes and composed of 5 million articles. The third data set is a sample of ClueWeb English data set [66] with size of around one Terabyte and having 632 million articles. For English Wikipedia and ClueWeb data sets, we performed a data cleaning task; we removed all English stop words from all articles, and obtained data sets where each article represents a transaction (features, items, or attributes are the corresponding words in the article). Likewise, for Amazon Reviews data set, we removed all English stop words from all reviews. Each review represents a transaction in our experiments on Amazon Reviews data set.

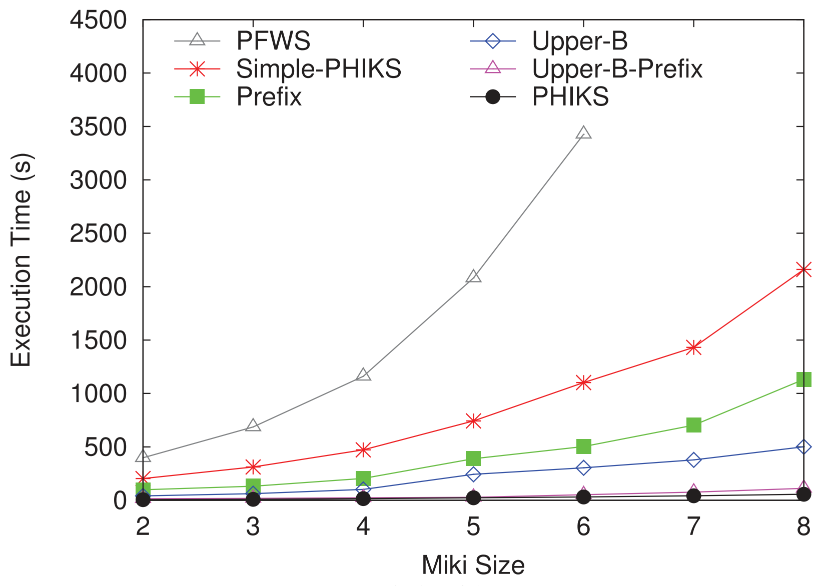
5.4.3 Results

In this Section, we report the results of our experimental evaluation.

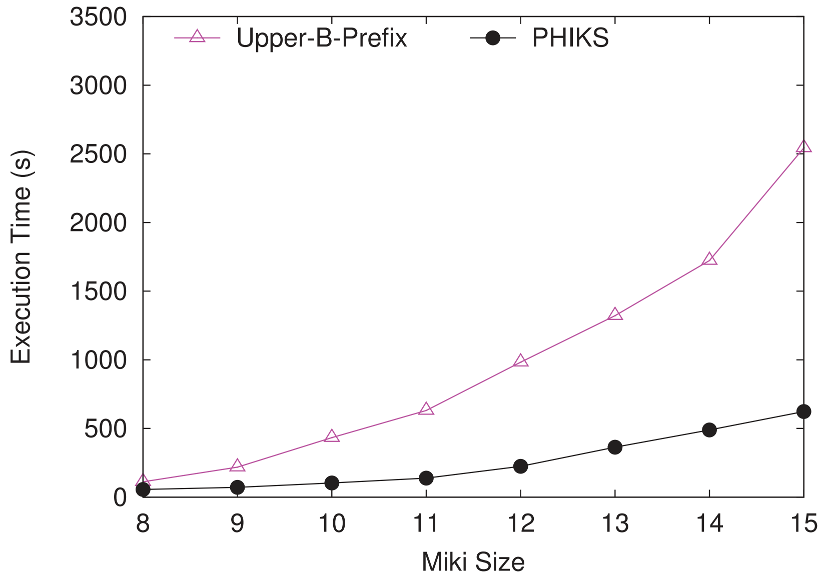
Runtime and Scalability: Figures 5.1, 5.2, and 5.3 show the results of our experiments on Amazon Reviews, English Wikipedia and ClueWeb data sets. Figures 5.1(a) and 5.1(b) give an overview on our experiments on the Amazon Reviews data set. Figure 5.1(a) illustrates the the performance of different algorithms when varying the itemset sizes from 2 to 8. We see that the response time of *ForwardSelection* algorithm (PFWS) grows exponentially and gets quickly very high compared to other algorithms. Above a size $k = 6$ of itemsets, PFWS cannot continue scaling. This is due to the multiple database scans that it performs to determine an itemset of size k (i.e, PFWS needs to perform k MapReduce jobs). In the other hand, the performance of Simple-PHIKS algorithm is better than PFWS; it continues scaling with higher k values. This difference in the performance between the two algorithms illustrates the significant impact of mining itemsets in the two rounds architecture.

Moreover, by using further optimizing techniques, we clearly see the improvements in the performance. In particular, with an itemset having size $k = 8$, we observe a good performance behavior of Prefix comparing to Simple-PHIKS. This performance gain in the runtime reflects the efficient usage of Prefix/Suffix technique for speeding up *miki* parallel extraction. Interestingly, by estimating *miki* at the first MapReduce job, we record a very good response time as shown by Upper-B algorithm. In particular, with $k = 8$ we see that Upper-B algorithm roughly outperforms Simple-PHIKS by a factor of 3. By coupling the Prefix/Suffix technique with Upper-B algorithm, we see very good improvements in the response time, which is achieved by Upper-B-Prefix. Finally, by taking advantage of our incremental entropy technique for reducing the number of data split scans, we record an outstanding improvement in the response time, as shown by PHIKS algorithm.

Figure 5.1(b) highlights the difference between the algorithms that scale in Figure

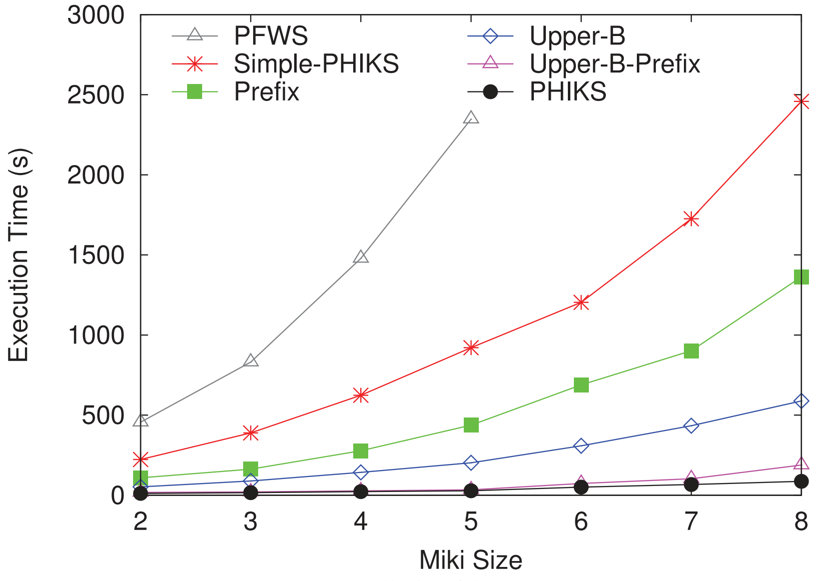


(a) All algorithms

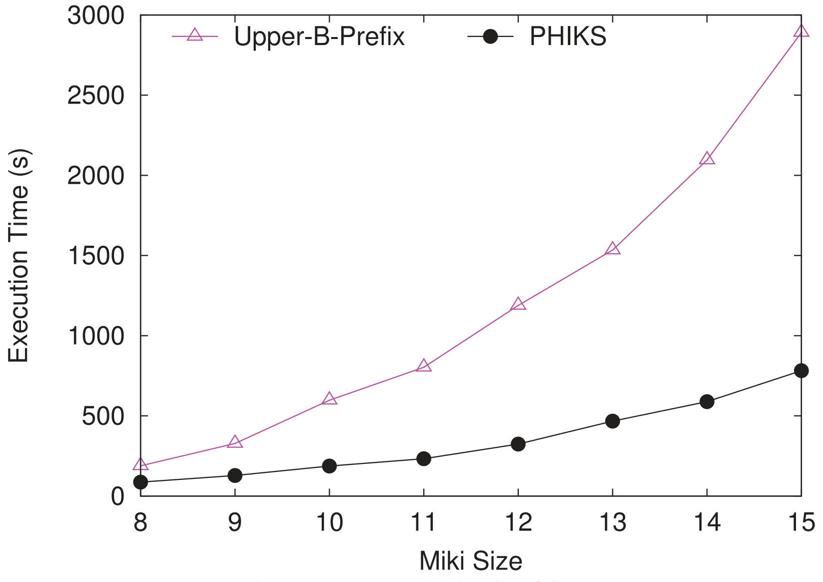


(b) Focus on scalable algorithms

Figure 5.1 – Runtime and Scalability on Amazon Reviews Data Set

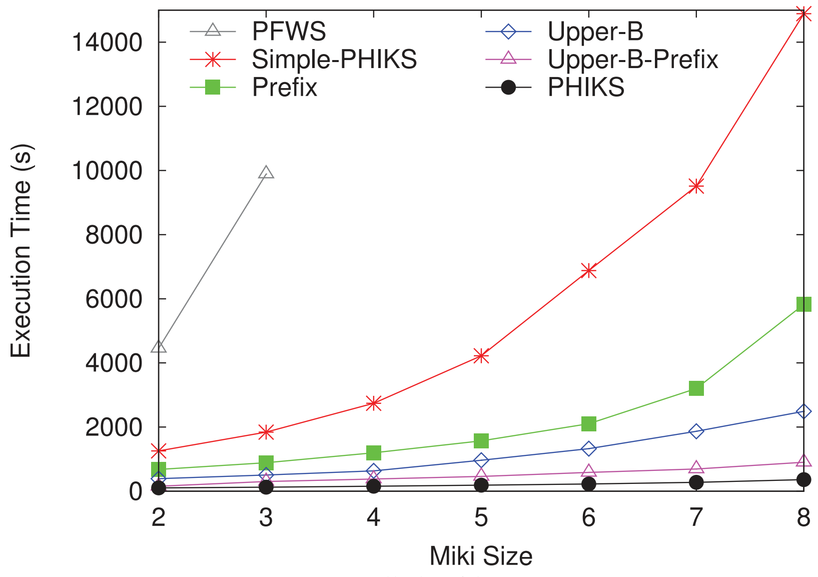


(a) All algorithms

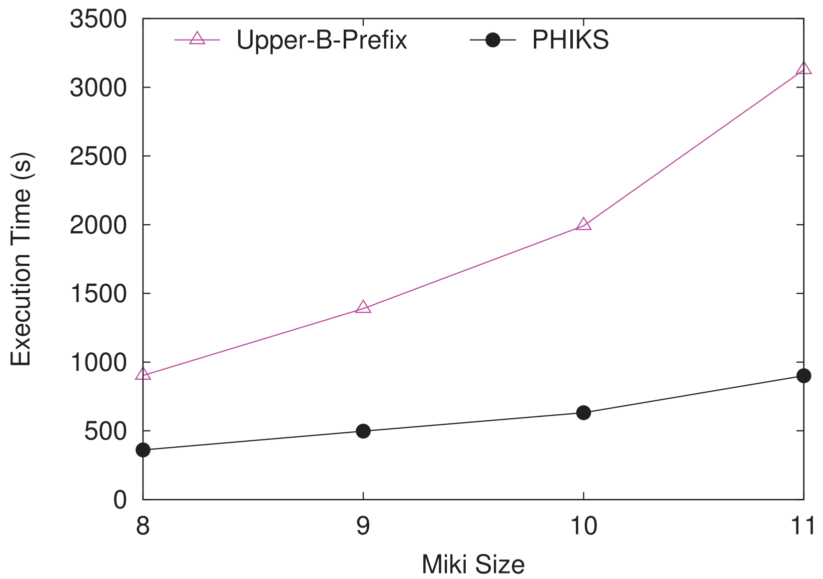


(b) Focus on scalable algorithms

Figure 5.2 – Runtime and Scalability on English Wikipedia Data Set



(a) All algorithms



(b) Focus on scalable algorithms

Figure 5.3 – Runtime and Scalability on ClueWeb Data Set

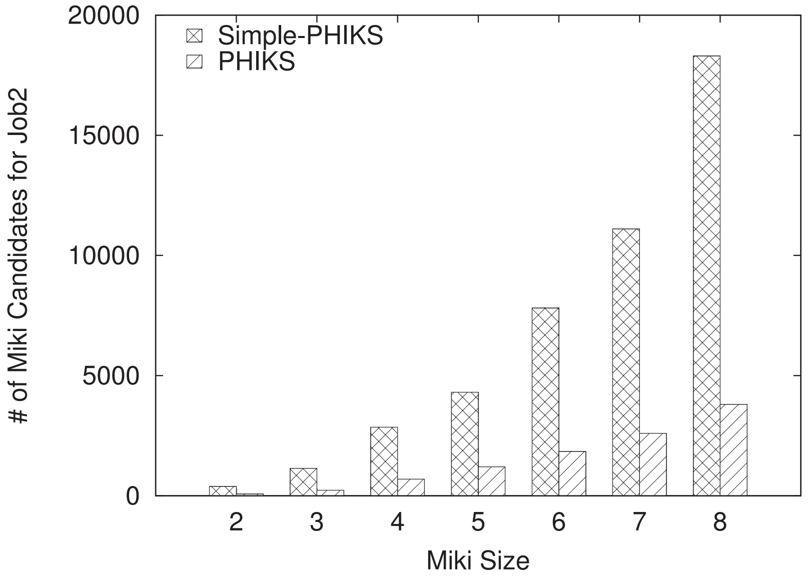
5.1(a). Although Upper-B-Prefix continues to scale with $k = 8$, it is outperformed by PHIKS algorithm. With itemsets of size $k = 15$, we clearly observe a big difference in the response time between Upper-B-Prefix and PHIKS. The significant performance of PHIKS algorithm illustrates its robust and efficient core mining process.

Figures 5.2(a) and 5.2(b) report our experiments on the English Wikipedia data set. Figure 5.2(a) gives a complete view on the the performance of different presented algorithms when varying the itemset sizes from 2 to 8. Similarly as in Figure 5.1(a), in Figure 5.2(a) we clearly see that the execution time of Forward Selection algorithm (PFWS) is very high compared to other presented alternatives. When the itemsets size reach values greater than $k = 5$, PFWS stops scaling. In the other side, we observe that Simple-PHIKS algorithm continues scaling and gives better performance than PFWS.

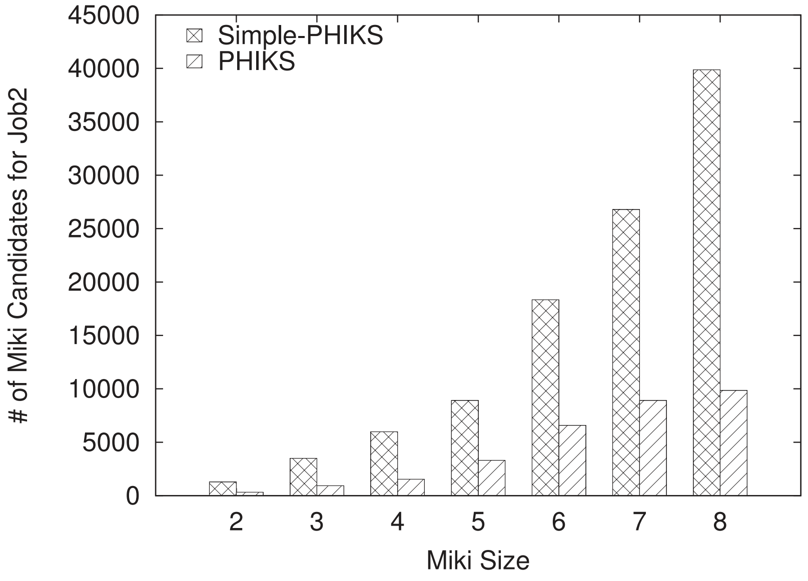
Performing more optimization, we significantly speed up the *miki* extraction. Specifically, with itemsets size $k = 8$, we see that the performance of Prefix is better than Simple-PHIKS. This difference in the performance behavior between the two algorithms explains the high impact of using Prefix/Suffix technique to speed up the whole mining process of the parallel *miki* extraction. By going on for further optimization using our efficient heuristic technique for estimating *miki* at the first MapReduce job, we get a significant improvement in the execution time as shown by Upper-B algorithm. Particularly, with itemsets size $k = 8$ we clearly see that Upper-B algorithm performance is better than Simple-PHIKS. By using Prefix/Suffix technique with Upper-B algorithm, we record a significant improvement in the performance as shown by Upper-B-Prefix. Eventually, based on our efficient technique of incremental entropy, we record a very significant performance improvement as shown by PHIKS algorithm.

Figure 5.2(b) illustrates the difference between the algorithms that scale in Figure 5.2(a). Despite the scalability recorded by Upper-B-Prefix when $k = 8$, Upper-B-Prefix gives very less performance compared to PHIKS algorithm. In particular, with higher itemsets size (*e.g.,* $k = 15$), we record a large difference in the execution time between Upper-B-Prefix and PHIKS algorithms. This difference in the performance between the two algorithms reflects the efficient and robust core mining process of PHIKS algorithm. In Figures 5.3(a) and 5.3(b), similar experiments have been conducted on the ClueWeb data set. We observe that the same order between all algorithms is kept compared to Figures 5.1(a), 5.1(b), 5.2(a) and 5.2(b). In particular, we see that PFWS algorithm suffers from the same limitations as could be observed on the Amazon Reviews and Wikipedia data sets in Figure 5.1(a) and Figure 5.2(a) . With an itemset size of $k = 8$, we clearly observe a significant difference between PHIKS algorithm performance and all other presented alternatives. This difference in the performance is better illustrated in Figure 5.3(b). By increasing the size k of *miki* from 8 to 11, we observe a very good performance of PHIKS algorithm. Although, Upper-B-Prefix algorithm scales with $k = 11$, it is outperformed by PHIKS.

***miki* Candidates Pruning:** Figure 5.4 gives a complete overview on the total number of *miki* candidates being tested at the second MapReduce job for both Simple-PHIKS and PHIKS algorithms. Figure 5.4(a) illustrates the number of *miki* candidates being



(a) Wikipedia data set



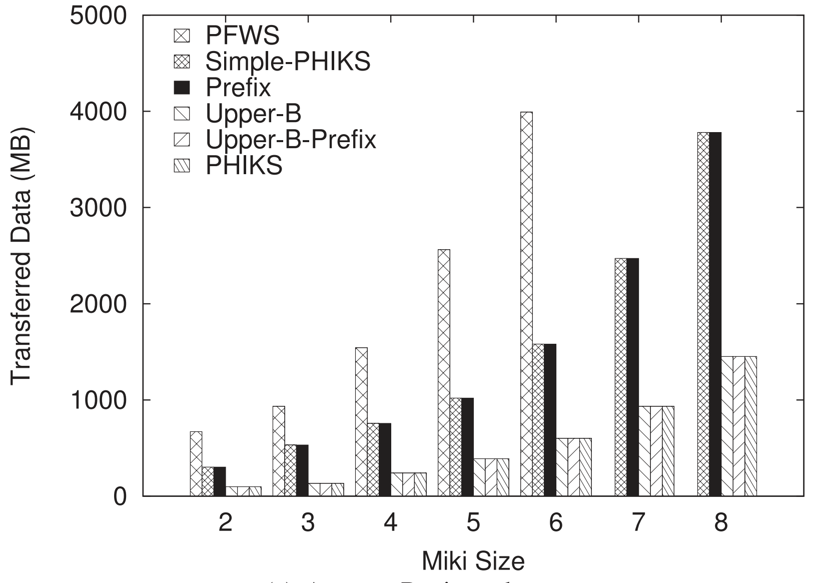
(b) ClueWeb data set

Figure 5.4 – Candidate Pruning

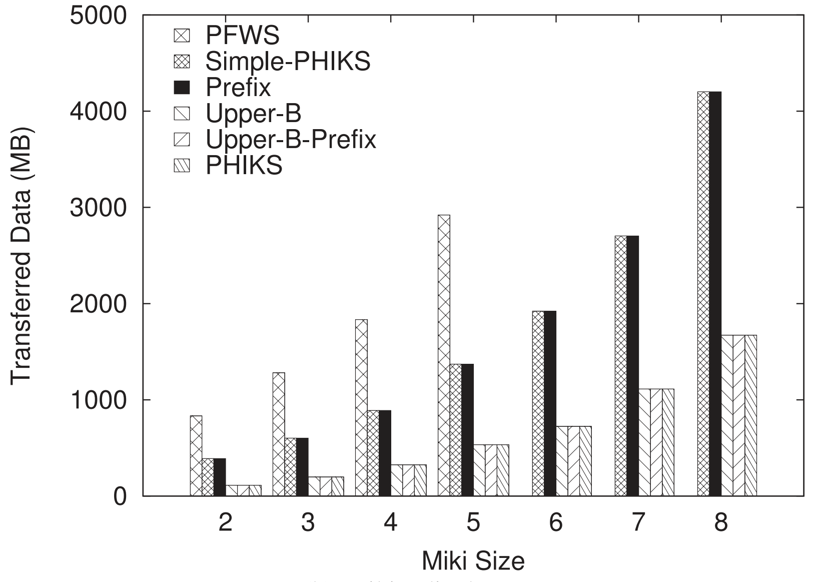
validated at the first MapReduce job on the Wikipedia data set. By varying the parameter size k of itemsets from 2 to 8, we observe a significant difference in the number of *miki* candidates being sent by each algorithm to its second MapReduce job. With $k = 8$, Simple-PHIKS algorithm sends to its second job roughly 6 times more candidates than PHIKS. This important reduction in the number of candidates to be tested in the second job is achieved due to our efficient technique for estimating the joint entropies of *miki* with very low upper bounds. Likewise, in Figure 5.4(b), we record a very good performance of PHIKS comparing to Simple-PHIKS. This outstanding performance of Simple-PHIKS algorithm reflects its high capability and its effectiveness for a very fast and successful *miki* extraction.

Data Communication and Energy Consumption: Figure 5.5 gives an entire view of the quantity of transferred data (in Megabyte) over the network by each presented algorithm on the three data sets. Respectively Figures 5.5(a), 5.5(b) and 5.5(c) show the performance of each presented maximally informative k-itemsets mining process on Amazon Reviews, English Wikipedia and ClueWeb data sets. In all figures, we observe that PFWS algorithm has the highest peak. This is due to its multiple MapReduce jobs executions. In the other hand, we see that Simple-PHIKS and Prefix algorithms have smaller peaks. This is because Simple-PHIKS and its optimized Prefix version algorithm (for fast computation of the local entropies at the mappers) rely on two MapReduce jobs whatever the *miki* size to be discovered. We see that Upper-B, Upper-B-Prefix and PHIKS outperform all other presented algorithms in terms of transferred data. This is due to the impact of estimating the joint entropies at their first MapReduce job which reduces the number of *miki* candidates (*i.e.,* data) being tested at their second MapReduce job.

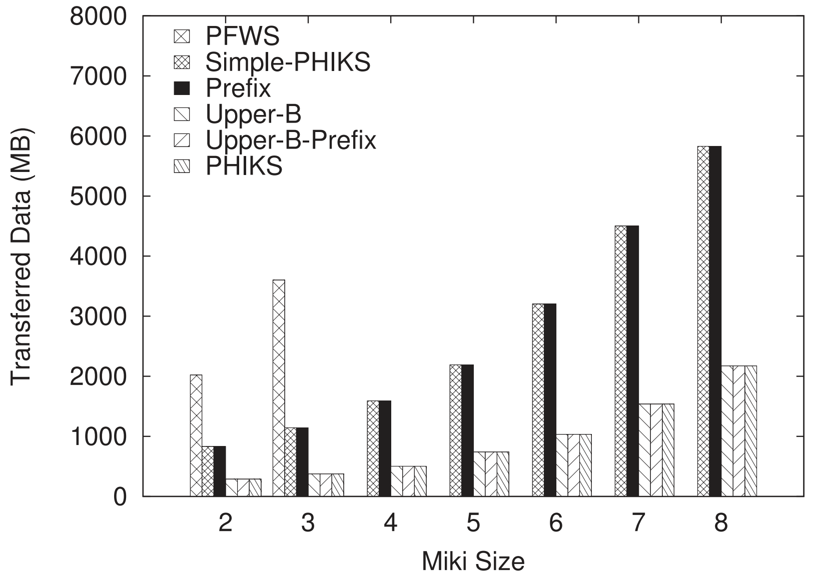
We also measured the energy consumption (in Watt) of the compared algorithms during their execution. We used the Grid5000 [64] tools that measure the power consumption of the nodes during a job execution. Figure 5.6 shows the total amount of the power consumption of each presented maximally informative k-itemsets mining process on Amazon Reviews, English Wikipedia and ClueWeb data sets. In Figures 5.6(a), 5.6(b) and 5.6(c) we observe that the energy consumption increases when increasing the size k of the *miki* to be discovered for each algorithm. We see that PHIKS still gives a lower consumption comparing to other presented algorithms. This is simply due to the higher optimizations in its core mining process. Actuelly the smaller number of candidates being tested during the second MapReduce job of PHIKS calls for a lower number of I/O access when computing the entropies. All of these different factors make PHIKS consumes less energy compared to other presented algorithms.



(a) Amazon Reviews data set

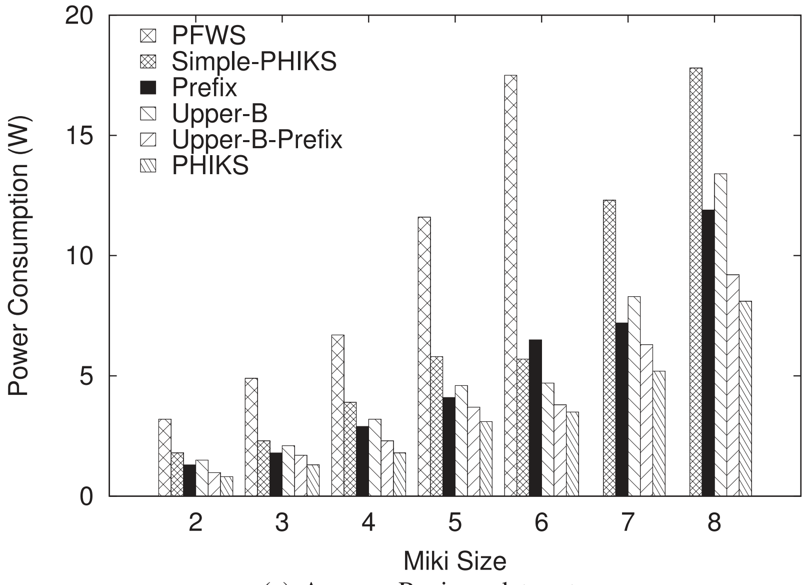


(b) Wikipedia data set

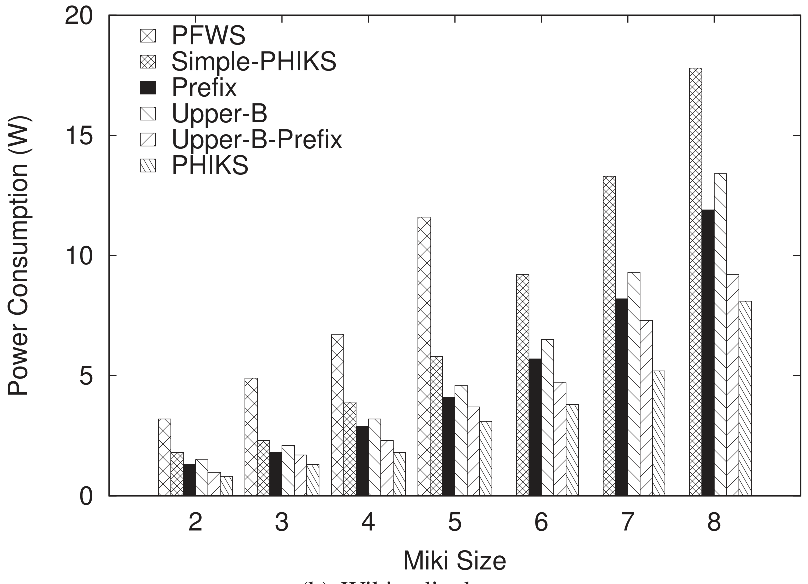


(c) ClueWeb data set

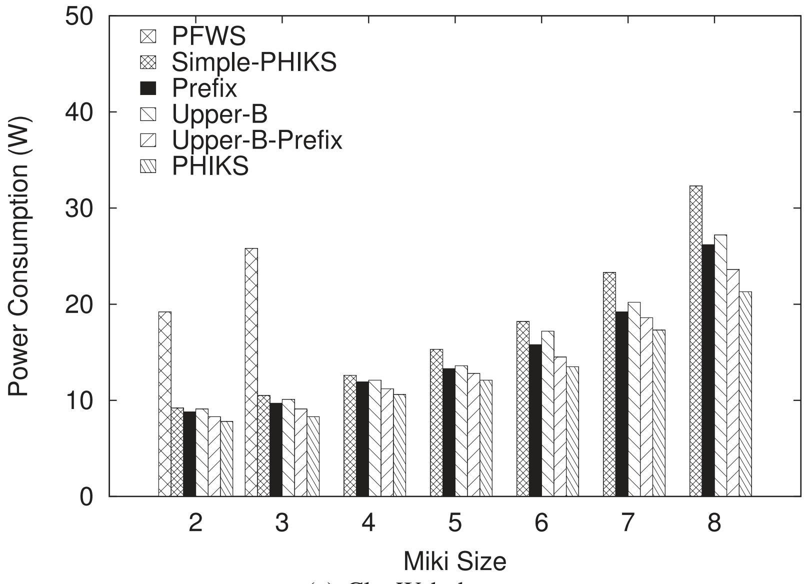
Figure 5.5 – Data Communication



(a) Amazon Reviews data set



(b) Wikipedia data set



(c) ClueWeb data set

Figure 5.6 – Energy Consumption

5.5 Conclusion

In this chapter, we proposed a reliable and efficient MapReduce based parallel maximally informative k-itemset algorithm namely PHIKS that has shown significant efficiency in terms of runtime, communication cost and energy consumption. PHIKS elegantly determines the *miki* in very large databases with at most two rounds. With PHIKS, we propose a bunch of optimizing techniques that renders the *miki* mining process very fast. These techniques concern the architecture at a global scale, but also the computation of entropy on distributed nodes, at a local scale. The result is a fast and efficient discovery of *miki* with high itemset size. Such ability to use high itemset size is mandatory when dealing with Big Data and particularly one Terabyte like what we have done in our experiments. Our results show that PHIKS algorithm outperforms other alternatives by several orders of magnitude, and makes the difference between an inoperative and a successful *miki* extraction.

As a future work, we plan to apply our technique of extracting *miki* to handle the problem of text classification. By using our PHIKS approach, we strongly believe that the extracted *miki* would highly discriminate the database, which is in turn would result in high classification accuracy and fast learning process.

Chapter 6

Conclusions

In this chapter, we summarize and discuss the main contributions made in the context of this thesis. Then, we give some research directions for future work.

6.1 Contributions

This thesis is in the context of the parallel mining of itemsets in massively distributed environments. We have focused on the itemset discovery problem in big data, aiming to improve and accelerate the itemset discovery processes which are of interest to various applications that deal with big data sets. In this thesis, we have made three contributions:

6.1.1 Data Placement for Fast Mining of Frequent Itemsets

We tackled the problem of mining frequent itemsets in massively distributed environments. Our main challenge was to allow for a fast discovery of frequent itemsets in large databases with very low minimum support.

Generally, we have showed that an optimal combination between a mining process with an adequate data placement in a massively distributed environment, highly improves the frequent itemsets extraction in very large databases. We have proposed our solution ODPR (Optimal Data-Process Relationship) for fast mining of frequent itemsets in massively distributed environments. Our proposal allows for extracting the frequent itemsets in just two simple, yet efficient MapReduce jobs. The first job is dedicated to extract a set of local frequent itemsets at each mapper based on CDAR algorithm along with STDP (Similar Transaction Data Placement). The second job is dedicated to validate the global frequency of the local frequent itemsets of the first job. Our approach has been extensively evaluated with very large real-world data sets and very low minimum support. The high scalability of our solution ODPR comparing to other alternatives confirms its effectiveness. We highly believe that our proposal would achieve very good performances in other massively distributed environments such as Spark.

6.1.2 Data Partitioning for Fast Mining of Frequent Itemsets

We addressed the problem of mining frequent itemsets in massively distributed environments. Our main challenge was to limit the itemset discovery to be done in just one simple, yet very efficient job.

Generally, mining the frequent itemsets in more than one job could impact the performance of the whole mining process. Although our proposal ODPR has achieved very good performance in extracting the frequent itemsets in large databases, it accounts for some limitations. The first job of ODPR algorithm could transfer candidate global frequent itemsets (*i.e.,* local frequent itemsets) of no use which impacts the whole mining process. In particular, this results in a high data (*i.e.,* local frequent itemsets) transfer between the mappers and the reducers which in turns degrade the performance of the second job. Thus, our main challenge is to omit the second job and perform a frequent itemset extraction in just one job instead of two. To this end, we proposed PATD (Parallel Absolute Top Down Algorithm). Based on a clever data partitioning technique strategy namely IBDP (Item Based Data Partitioning), PATD mines each data partition independently based on an absolute minimum support instead of a relative one. We have extensively evaluated our PATD algorithm using very large data sets (up to one Terabyte of data) and very low minimum support, the results confirms the efficiency and effectiveness of our approach.

6.1.3 Fast Parallel Mining of Maximally Informative K-itemset

We addressed the problem of mining maximally informative k-itemsets in big data. Our main challenge was to improve and accelerate the *miki* discovery in big data. To this end we proposed PHIKS (Parallel Highly Informative *K*-ItemSet), a highly, scalable algorithm that is capable to extract the *miki* of different sizes in just two simple, yet very efficient jobs. In a massively distributed environment such as MapReduce, at its first job, PHIKS determines a set of potential *miki* by applying a simple *ForwardSelection* algorithm at each mapper. By using a very clever technique to estimate the joint probabilities of the *miki* candidates at its first job, PHIKS reduces the computations load of its second job. With a bunch of computational optimizations, PHIKS renders the *miki* discovery in very large distastes (up to one Terabyte of data) simple and succinct. We evaluate the effectiveness and the capabilities of PHIKS algorithm by carrying out extensive, various experiments with very large real-world data sets. The results have shown an outstanding performance of PHIKS comparing to other alternatives.

6.2 Directions for Future Work

With the abundant and various researches that have been carried out to improve the performances of parallel data mining algorithms, one may wonder whether we have solved most of the critical problems related to both frequent and maximally informative k-itemsets mining. Particularly, with the challenges that we have been facing with big data, there would be several possible extensions and improvements of the work that we have achieved in this thesis. Some future directions of research are as follows.

- **Miki based classification:** With very large amounts of data, performing a classification task is not easy. This is due to the large numbers of attributes. It is likely that several attributes are of no relevance to the classification task.

  We plan to develop a new framework intended to improve any classification task by combining two simple, yet very efficient techniques:

  The first technique extends the ensemble classifier methods [83] by performing two parallel decision steps in a massively distributed environment. The main idea is to create several simple classifiers (*e.g.,* Naive Bayes classifiers) locally say at each mapper. This can be achieved by performing a simple sampling (based on the attributes) of the data at each mapper. Thus, each mapper would have a set of say k trained classifiers. To classify a new instance, a decision test would be made at each mapper (first decision step). Each mapper locally classifies the new instance based on a majority voting scheme. Then, the classified instance and its class label are sent to the reducer. The reducer receives each classified instance with its class label. The reducer further aggregates the results based on the class labels and outputs the final result *i.e.,* the classified instance (key) with its majority class (value) (second decision step).

  The second technique is to incorporate the *miki* with the two decision steps classification. We use the *miki* to improve the classification task. Since the miki highly discriminate the database, they would significantly improve the accuracy of the classification. Instead of using irrelevant attributes in the classification, we first apply our PHIKS algorithm to extract the relevant attributes and then use them with our two steps decision classification technique.

- **Using Spark for fast mining of frequent itemsets:** Since Spark [3] supports in-memory computations which is far faster than MapReduce, we highly believe that considering an adequate data placement strategy along with a specific frequent itemset mining algorithm in Spark, would improve the global mining process. For instance, our proposed 2-jobs schema architecture P2S as described in chapter 3, can be easily implemented in Spark.

  The first job consists of determining a set of local frequent itemsets *i.e.,* after applying a specific data placement strategy. Using a flatMap() function, each bunch of the database transactions is loaded into the memory as an RDD (Resilient Distributed data set) object. Then, a specific FIM algorithm is applied locally on each RDD (partition of the global data set). The results of the mappers (*i.e.,* flatMaps) are emitted to the reducers. A reduceByKey() function is applied to aggregate the different results.

The second job, consists of validating the global frequency of the local frequent itemsets. The results (local frequent itemsets) are shared between the different workers using an accumulator in Spark. A flatMap() function is applied on the RDDs to count the occurrence of each local frequent itemset in the whole database. Then, a reduceByKey() function is applied to aggregate the results received from the mappers. The reducer sums up the results and outputs the local frequent itemsets that are globally frequent.

Likewise, as in MapReduce, cleverly partitioning the data allows for a very fast mining of frequent itemsets. Our proposed PATD (Parallel Absolute Top Down) algorithm as described in chapter 4, is outstandingly capable to extract the frequent itemsets from very large databases and with very low minimum support in short time. Based on the Spark framework, we highly believe that the performance of PATD algorithm would be very significant. The design and the implementation of our proposed PATD algorithm in Spark would be very simple. The database partitions are loaded as RDDs to each mapper. The mapper applies CDAR algorithm locally on its RDD (*i.e.,* data partition) using an absolute minimum support. The results are emitted from the mapper to the reducer. The reducer receives the frequent itemsets from each mapper and outputs the final results.

Bibliography

[1] Michael Berry. *Survey of Text Mining Clustering, Classification, and Retrieval*. Springer New York, New York, NY, 2004.

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

[3] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[4] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceedings of International Conference on Very Large Data Bases (VLDB), Santiago de Chile, Chile*, pages 487–499, 1994.

[5] Han, Pei, and Yin. Mining frequent patterns without candidate generation. *SIGMODREC: ACM SIGMOD Record*, 29, 2000.

[6] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y. Chang. Pfp: parallel fp-growth for query recommendation. In Pearl Pu, Derek G. Bridge, Bamshad Mobasher, and Francesco Ricci, editors, *Proceedings of the ACM Conference on Recommender Systems (RecSys) Lausanne, Switzerland*, pages 107–114. ACM, 2008.

[7] Arno J. Knobbe and Eric K. Y. Ho. Maximally informative k-itemsets and their efficient discovery. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 237–244, 2006.

[8] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceedings of International Conference on Very Large Data Bases (VLDB), Santiago de Chile, Chile*, pages 487–499, 1994.

[9] Ashok Savasere, Edward Omiecinski, and Shamkant B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 432–444, 1995.

[10] Wei Song, Bingru Yang, and Zhangyan Xu. Index-bittablefi: An improved algorithm for mining frequent itemsets. *Knowl.-Based Syst.*, 21(6):507–513, 2008.

[11] N. Jayalakshmi, V. Vidhya, M. Krishnamurthy, and A. Kannan. Frequent itemset generation using double hashing technique. *Procedia Engineering*, 38(0):1467 – 1478, 2012.

[12] M.J. Zaki. Scalable algorithms for association mining. *Knowledge and Data Engineering, IEEE Transactions on*, 12(3):372–390, May 2000.

[13] Yuh-Jiuan Tsay and Ya-Wen Chang-Chien. An efficient cluster and decomposition algorithm for mining association rules. *Inf. Sci*, 160(1-4):161–171, 2004.

[14] Rajaraman Anand. *Mining of massive datasets*. Cambridge University Press, New York, N.Y. Cambridge, 2012.

[15] T. M. Cover. *Elements of information theory*. Wiley-Interscience, Hoboken, N.J, 2006.

[16] Jimmy Lin and Dmitriy Ryaboy. Scaling big data mining infrastructure: The twitter experience. *SIGKDD Explor. Newsl.*, 14(2):6–19, April 2013.

[17] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. Wireless sensor network survey. *Computer Networks*, 52(12):2292 – 2330, 2008.

[18] Alain Degenne. *Introducing social networks*. SAGE, London Thousand Oaks, 1999.

[19] D. J. Hand. *Principles of data mining*. MIT Press, Cambridge, Mass, 2001.

[20] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[21] Zheng Qin. *Introduction to e-commerce*. Springer, Berlin London, 2009.

[22] Bart Goethals. Survey on frequent pattern mining. *Univ. of Helsinki*, 2003.

[23] Benjamin Negrevergne, Alexandre Termier, Marie-Christine Rousset, and Jean-François Méhaut. Para miner: A generic pattern mining algorithm for multi-core architectures. *Data Min. Knowl. Discov.*, 28(3):593–633, May 2014.

[24] S. B. Kotsiantis. Supervised machine learning: A review of classification techniques. In *Proceedings of International Conference on Emerging Artificial Intelligence Applications in Computer Engineering*, pages 3–24, 2007.

[25] Usama M. Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. Advances in knowledge discovery and data mining. chapter From Data Mining to Knowledge Discovery: An Overview, pages 1–34. American Association for Artificial Intelligence, Menlo Park, CA, USA, 1996.

[26] Ramesh C. Agarwal, Charu C. Aggarwal, and V.V.V. Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing*, 61(3):350 – 371, 2001.

[27] Jian Pei, Jiawei Han, and L.V.S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 433–442, 2001.

[28] Junqiang Liu, Yunhe Pan, Ke Wang, and Jiawei Han. Mining frequent item sets by opportunistic projection. In *In Proc. 2002 Int. Conf. on Knowledge Discovery in Databases (KDD'02*, pages 229–238. ACM Press, 2002.

[29] Gösta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets, 2003.

[30] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *Proceedings of International Conference in Database Theory*, pages 398–416, Jerusalem, Israel, 1999.

[31] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Efficient mining of association rules using closed itemset lattices. *Inf. Syst.*, 24(1):25–46, March 1999.

[32] Jian Pei, Jiawei Han, Runying Mao, et al. Closet: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD workshop on research issues in data mining and knowledge discovery*, volume 4, pages 21–30, 2000.

[33] Mohammed Javeed Zaki and Ching-Jiu Hsiao. Charm: An efficient algorithm for closed itemset mining. In *SDM*, volume 2, pages 457–473. SIAM, 2002.

[34] Jianyong Wang, Jiawei Han, and Jian Pei. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 236–245, New York, NY, USA, 2003. ACM.

[35] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 487–499, 1994.

[36] Han, Pei, and Yin. Mining frequent patterns without candidate generation. *SIGMODREC: ACM SIGMOD Record*, 29, 2000.

[37] Hannes Heikinheimo, Eino Hinkkanen, Heikki Mannila, Taneli Mielikäinen, and Jouni K. Seppänen. Finding low-entropy sets and trees from binary data. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 350–359, 2007.

[38] Sergey Brin, Rajeev Motwani, and Craig Silverstein. Beyond market baskets: Generalizing association rules to correlations. *SIGMOD Rec.*, 26(2):265–276, June 1997.

[39] Nikolaj Tatti. Probably the best itemsets. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, July 25-28, 2010*, pages 293–302, 2010.

[40] Chris Giannella, Jiawei Han, Jian Pei, Xifeng Yan, and Philip S. Yu. Mining frequent patterns in data streams at multiple time granularities, 2002.

[41] Wei-Guang Teng, Ming-Syan Chen, and Philip S. Yu. A regression-based temporal pattern mining scheme for data streams. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 93–104, 2003.

[42] Chongsheng Zhang and Florent Masseglia. Discovering highly informative feature sets from data streams. In *Database and Expert Systems Applications*, pages 91–104. 2010.

[43] Girish Chandrashekar and Ferat Sahin. A survey on feature selection methods. *Computers and Electrical Engineering*, 40(1):16 – 28, 2014.

[44] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, mar 2003.

[45] Hadoop. `http://hadoop.apache.org`, 2014.

[46] Matteo Riondato, Justin A. DeBrabant, Rodrigo Fonseca, and Eli Upfal. Parma: a parallel randomized algorithm for approximate association rules mining in mapreduce. In *21st ACM International Conference on Information and Knowledge Management (CIKM), Maui, HI, USA*, pages 85–94. ACM, 2012.

[47] Ke Chen, Lijun Zhang, Sansi Li, and Wende Ke. Research on association rules parallel algorithm based on fp-growth. In *Proceedings of International Conference on Information Computing and Applications*, pages 249–256, Qinhuangdao, China, 2011.

[48] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, pages 107–113, 2008.

[49] Su-Qi Wang, Yu-Bin Yang, Yang Gao, Guang-Peng Chen, and Yao Zhang. Mapreduce-based closed frequent itemset mining with efficient redundancy filtering. In *Proceedings of IEEE International Conference on Data Mining*, pages 449–453, Brussels, Belgium, 2012.

[50] Osmar R. Zaïane, Mohammad El-Hajj, and Paul Lu. Fast parallel association rule mining without candidacy generation. In *Proceedings of IEEE International Conference on Data Mining*, pages 665–668, San Jose, California, USA, 2001.

[51] Eric Li and Li Liu. Optimization of frequent itemset mining on multiple-core processor. In *Proceedings of International Conference on Very Large Data Bases*, pages 1275–1285, Vienna, Austria, 2007.

[52] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y. Chang. Pfp: parallel fp-growth for query recommendation. In *Proceedings of International Conference on Recommender Systems*, pages 107–114, Lausanne, Switzerland, 2008.

[53] Jianyong Wang, Jiawei Han, and Jian Pei. CLOSET+: searching for the best strategies for mining frequent closed itemsets. In *Proceedings of International Conference on Knowledge Discovery and Data Mining*, pages 236–245, Washington, DC, USA, 2003.

[54] Claudio Lucchese, Salvatore Orlando, and Raffaele Perego. Fast and memory efficient mining of frequent closed itemsets. *IEEE*, 18:21–36, 2006.

[55] Hsinchun Chen, Roger H. L. Chiang, and Veda C. Storey. Business intelligence and analytics: From big data to big impact. *MIS Quarterly*, 36(4):1165–1188, December 2012.

[56] Bart Goethals. Memory issues in frequent itemset mining. In Hisham Haddad, Andrea Omicini, Roger L. Wainwright, and Lorie M. Liebrock, editors, *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC), Nicosia, Cyprus, March 14-17, 2004*, pages 530–534. ACM, 2004.

[57] Tom White. *Hadoop : the definitive guide*. O'Reilly, Beijing, 2012.

[58] Christian Bizer, Peter A. Boncz, Michael L. Brodie, and Orri Erling. The meaningful use of big data: four perspectives - four challenges. *SIGMOD Record*, 40(4):56–60, 2011.

[59] Hadoop. `http://hadoop.apache.org`, 2014.

[60] Ashok Savasere, Edward Omiecinski, and Shamkant B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 432–444, 1995.

[61] Shimon Even. *Graph algorithms*. Computer Science Press, Potomac, Md, 1979.

[62] Patoh. http://bmi.osu.edu/ umit/PaToH/manual.pdf, 2011.

[63] Sean Owen. *Mahout in action*. Manning Publications Co, Shelter Island, N.Y, 2012.

[64] Grid5000. `https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home`.

[65] English wikipedia articles. `http://dumps.wikimedia.org/enwiki/latest`, 2014.

[66] The clueweb09 dataset. `http://www.lemurproject.org/clueweb09.php/`, 2009.

[67] Alexandros Labrinidis and H. V. Jagadish. Challenges and opportunities with big data. *Proc. VLDB Endow.*, 5(12):2032–2033, August 2012.

[68] Wei Fan and Albert Bifet. Mining big data: Current status, and forecast to the future. *SIGKDD Explor. Newsl.*, 14(2):1–5, April 2013.

[69] Yuh-Jiuan Tsay and Ya-Wen Chang-Chien. An efficient cluster and decomposition algorithm for mining association rules. *Inf. Sci. Inf. Comput. Sci.*, 160(1-4):161–171, March 2004.

[70] Jiawei Han. *Data mining : concepts and techniques*. Elsevier/Morgan Kaufmann, 2012.

[71] Ed Greengrass. Information retrieval: A survey, 2000.

[72] Robert Gray. *Entropy and information theory*. Springer, New York, 2011.

[73] Zoubin Ghahramani. Unsupervised learning. In *Advanced Lectures on Machine Learning*, pages 72–112, 2004.

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

[75] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conf. on Hot Topics in Cloud Computing*, pages 10–10, 2010.

[76] S. Moens, E. Aksehirli, and B. Goethals. Frequent itemset mining for big data. In *IEEE International Conference on Big Data*, pages 111–118, 2013.

[77] Klaus Berberich and Srikanta Bedathur. Computing n-gram statistics in mapreduce. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT)*, pages 101–112, 2013.

[78] Iris Miliaraki, Klaus Berberich, Rainer Gemulla, and Spyros Zoupanos. Mind the gap: Large-scale frequent sequence mining. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 797–808, 2013.

[79] Rajaraman Anand. *Mining of massive datasets*. Cambridge University Press, New York, N.Y. Cambridge, 2012.

[80] Ashok Savasere, Edward Omiecinski, and Shamkant B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 432–444, 1995.

[81] Tom White. *Hadoop : the definitive guide*. O'Reilly, 2012.

[82] Amazon. `http://snap.stanford.edu/data/web-Amazon-links.html`.

[83] Anna Jurek, Yaxin Bi, Shengli Wu, and Chris Nugent. A survey of commonly used ensemble-based classification techniques. *The Knowledge Engineering Review*, 29:551–581, 11 2014.