UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

U.F.R. D'I.E.E.A.

LABORATOIRE D'INFORMATIQUE FONDAMENETALE L.I.F.L.

———

UNIVERSITE DE MONS-HAINAULT

INSTITUT D'INFORMATIQUE

THESE

en vue de l'obtention du grade de

DOCTEUR

DISCIPLINE : INFORMATIQUE

Auteur :

VALERIE FIOLET

Titre :

ALGORITHMES DISTRIBUES D'EXTRACTION DE CONNAISSANCES.

2006

Président :	Pierre Boulet	Professeur, Université de Lille 1
Rapporteurs :	Hervé Guyennet	Professeur, Université de Franche-Comté
	Bernard Lécussan	Professeur, CNAM, Paris
Examinateurs :	Isaac Scherson	Professeur, University of California (USA)
	Jozef Wijsen	Professeur, Université de Mons-Hainault (BELGIQUE)
	Richard Olejnik	Ingénieur de Recherche C.N.R.S.
Directeur :	Bernard Toursel	Professeur, Université de Lille 1

L.I.F.L. Cité Scientifique, 59655 Villeneuve d'Ascq (FRANCE)
Faculté des sciences, Plaine de Nimy , 7000 Mons (BELGIQUE)

Résumé

La disponibilité de plus en plus importante de clusters et de grilles de stations fournit des ressources puissantes et de faible coût pour des traitements de calculs hautes performances distribués. Les tâches de data mining (ou extraction de connaissances) sont des traitements coûteux, et les volumes de données disponibles augmentent avec les possibilités de stockage, imposant le recours au parallélisme.
La plupart des méthodes parallèles existantes de data mining étant développées pour exécution sur des super-calculateurs avec mémoire partagée, elles ne peuvent être utilisées sur une grille de stations.

Afin d'exploiter au mieux les ressources disponibles de type grille de calcul, il apparaît nécessaire de concevoir de nouveaux algorithmes de data mining spécialement adaptés à ce type d'architecture, et prenant en compte les spécifités d'exécution distribuée.
La distribution des problèmes de data mining passe par un partitionnement des données et l'utilisation de la fragmentation obtenue sur la grille via des traitements parallèles.

Le projet **DisDaMin** (Distributed Data Mining) développé dans cette thèse vise à proposer des solutions pour certains problèmes de data mining, tels que le problème de génération de règles d'association ou le problème de clustering (classification non supervisée).
Pour le problème spécifique de génération de règles d'association, nous suggérons l'utilisation d'un partitionnement "intelligent" des données. Ce partitionnement intelligent peut être obtenu par clustering.

Nous présentons donc un nouvel algorithme de clustering, appelé **Clustering Distribué Progressif**, qui exécute un clustering de manière progressive distribuée et efficace respectant les contraintes d'exécution sur grille de calculs.
Les clusters de données issus de ce clustering sont par la suite utilisés pour des tâches de data mining, en particulier pour le problème de génération de règles d'association, afin d'en réduire la complexité de traitement.
Nous introduisons un algorithme distribué pour le problème des règles d'association, appelé **DIC-Coop** (DIC Coopératif) et basé sur l'utilisation du partitionnement "intelligent".

Chacun des algorithmes présentés est suivi d'un résumé des expérimentations qui ont permis de les valider comme heuristiques de data mining. Enfin, une synthèse des concepts distribués exploités dans les deux méthodes présentées conclut ce rapport.

Mots clés :
data mining, règles d'association et clustering, traitements distribués et traitements sur grille, méthodes heuristiques.

Abstract

The increasingly availability of clusters and grids of stations provides powerful resources of low cost for high performances distributed computings.
Data Mining tasks (or knowledge extraction) are expensive treatments, and volumes of available data increase with the storage possibilities, imposing to have recourse to parallelism.
Most of the existing parallel data mining methods have been developed for execution on super-computers with shared memory. They cannot be used in an acceptable way on a grid of stations.

In order to exploit the available treatment resources as well as possible, it appears necessary to design new data mining algorithms especially adapted to architecture like grids, and taking under account specifities of distributed execution.

Data Mining problems distribution requiers a data partitioning and the use of the obtained fragmentation on the grid using parallel treatments.

The **DisDaMin** project (Distributed Dated Mining) presented here aims at proposing solutions to some data mining problems, such as the association rules generation problem or the clustering problem (not supervised classification).
For the specific association rules problem, we suggest the use of an "intelligent" data partitioning. This intelligent partitioning can be obtained by clustering.

We thus present a new clustering algorithm, called **Distributed Progressive Clustering**, which carries out a distributed clustering in a progressive and effective way, respecting the constraints of execution on grid.
Data clusters resulting from this clustering are used thereafter for data mining tasks, in particular for the association rules problem, in order to reduce the treatment complexity of it.
We introduce a distributed algorithm for the association rules problem, called **DICCoop** (DIC Co-operative), based on the use of "intelligent" partitioning.

Each presented algorithm is followed of a summary of the experiments which permit to validate them as data mining heuristics. At the end, a synthesis of the distributed concepts used in the two presented methods concludes this report.

Keywords :

data mining, association rules and clustering, distributed and GRID computings, heuristics methods.

Table des matières

Remerciements

Je remercie tout d'abord le Professeur Bernard Toursel, directeur de cette thèse et responsable de l'équipe PALOMA du Laboratoire d'Informatique Fondamentale de Lille pour m'avoir initié à la recherche au sein de son équipe et qui a su me prodiguer de nombreux conseils et me rassurer dans les moments de doute.

Je remercie les membres des jurys :

Monsieur Pierre Boulet, Professeur à l'université des sciences et technologies de Lille, pour avoir présidé le jury.

Messieurs Hervé Guyennet, Professeur à l'université de Franche Comté et Bernard Lécussan, Professeur au CNAM, pour avoir accepté d'être rapporteurs de cette thèse.

Messieurs Isaac Scherson, Professeur à l'Université de Californie, Jozef Wijsen, Professeur à l'université de Mons-Hainault et Richard Olejnik, Ingénieur de recherche C.N.R.S. qui m'ont fait l'honneur de participer au jury lillois.

Je tiens également à remercier les membres du Laboratoire d'Informatique Fondamentale de Lille, plus particulièrement ceux de l'équipe PALOMA, ainsi que les membres de l'institut d'informatique de l'Université de Mons-Hainault pour leur soutien durant ces années de travail.

Merci également aux différents chercheurs rencontrés en conférence et autres pour les discussions constructives autour de ce travail.

Merci aux différents étudiants de DEA ou DESS de l'USTL qui ont contribué à ce travail.

Merci enfin à mes proches qui m'ont apporté leur soutien durant l'élaboration de cette thèse.

Introduction

La découverte de connaissances incluses dans les données (KDD : Knowledge Discovery in Databases), et tout particulièrement les techniques de Fouille de données (Data Mining) sont de plus en plus utilisées. Le but de ces techniques d'analyse est d'extraire des informations utiles depuis de grandes bases de données.

Les algorithmes de Data Mining nécessitent généralement de grandes capacités de traitement qui peuvent être fournies par le recours à des traitements parallèles et distribués.

Le projet **DisDaMin** (Distributed Data Mining), contexte des travaux présentés ici, aborde des problèmes de Data Mining (tels que les règles d'association, le clustering - ou classification non supervisée, ...) pour une exécution distribuée.

Le but du projet DisDaMin est de développer des solutions parallèles et distribuées pour des problèmes de Data Mining, fournissant ainsi plusieurs sortes de gains pour les temps d'exécution :

- diminution du temps d'exécution grâce à une distribution intelligente des données ;
- diminution du temps d'exécution grâce à l'exploitation du parallélisme.

Dans des environnements parallèles et distribués tels que les grilles ou les clusters de stations de travail, les contraintes inhérentes à la plateforme d'exécution doivent être prises en considération dans les algorithmes (ces contraintes sont résumées Section 1.5).

L'absence de mémoire centrale partagée oblige à distribuer la base de données en fragments et à traiter ces fragments de manière parallèle. Du fait du coût élevé des communications dans ce type d'environnement, les traitements parallèles doivent être les plus indépendants possible pour éviter des communications et des synchronisations coûteuses.

Pour les problèmes de Data Mining distribué, nous proposons une approche reposant sur une **distribution intelligente des données**, de manière à obtenir des fragments très indépendants. Le principal problème consiste alors à obtenir cette fragmentation "intelligente".

Pour le problème des règles d'association, par exemple, le critère principal pour une fragmentation intelligente est d'avoir une similarité maximale des données au sein d'un fragment (similarité par rapport aux valeurs de chacun des attributs), ainsi qu'une dissimilarité maximale des données entre les fragments.
L'objectif est ici de permettre une parallélisation du problème de recherche de règles d'association (qui nécessite normalement un accès à la base de données complète) et d'obtenir

une diminution du temps d'exécution.

Le critère de fragmentation des données apparaît similaire à l'objectif des algorithmes de clustering (ou classification non supervisée), cette fragmentation peut dériver du clustering.
Cette fragmentation intelligente obtenue par clustering pour le problème des règles d'association permet d'obtenir un bon facteur d'accélération provenant d'une part de l'exploitation du parallélisme sur les fragments d'autre part d'une diminution de l'espace de recherche visité du fait de la distribution intelligente (voir Section 2.1.2).
La fragmentation intelligente pour le problème des règles d'association une fois vérifiée adéquate par utilisation d'un clustering classique, il apparaît nécessaire de s'intéresser de manière plus précise à la parallélisation de cette phase de fragmentation par clustering.

Dans une première partie, nous rappelerons les principes de traitements distribués et la problématique de Data Mining dans un contexte de grille de calculs.

Dans une seconde partie, nous détaillerons une nouvelle méthode appelée **CDP** (Clustering Distribué Progressif - voir Section 3.2). Les algorithmes parallèles de clustering existants ne respectant pas les critères des traitements distribués, il a été nécessaire de revisiter le problème. La phase de clustering doit être exécutée de manière distribuée et rapide de manière à ne pas ralentir le temps global de traitement du problème des règles d'association La méthode présentée utilise des résultats de clustering partiels sur des sous-ensembles d'attributs de la base de données pour construire des clusters sur la base entière.

La phase de fragmentation ayant été accomplie en respectant les contraintes du support d'exécution, reste alors à s'intéresser à la façon d'exploiter celle-ci pour la recherche de règles d'association.

Dans une troisième partie, nous proposons donc un nouvel algorithme de recherche de règles d'association, appelé **DIC-Coop** (pour DIC Coopératif - voir Section 4.2). Cet algorithme, inspiré de l'algorithme de DIC (voir Brin et Al. [14]) permet de faire collaborer des traitements parallèles des fragments, tout en respectant les critères d'exécution (pas de mémoire partagée, pas de synchronisation).
Afin de permettre une distribution optimale, une méta version de cet algorithme est proposée (voir Section 4.3), permettant de ne pas être limité par la granularité limite des clusters (ou fragments) identifiés (voir Section 2.3.7). Les détails de fonctionnement des deux méthodes (version classique et méta version) sont décrits, ainsi que les gains apportés.

Enfin, une quatrième partie permettra d'effectuer un bilan de la méthode générale présentée, avant de conclure.

Première partie

Data Mining Distribué

Chapitre 1

Définitions et problématiques

Le Data Mining (Fouille de données) est une technique précieuse pour l'analyse des données disponibles (voir Section 1.2). Parallèlement à l'intérêt de plus en plus développé pour les techniques d'extraction de connaissances (domaines bancaires, d'assurance, du commerce, médecine...), les quantités de données à disposition pour les analyses ne cessent d'augmenter, atteignant des Méga, Giga voir Téra octets pour certaines applications spécifiques.

L'augmentation des quantités de données à prendre en compte nécessite de posséder des capacités de traitement suffisantes qui peuvent être apportées par l'utilisation de ressources distribuées disponibles (voir Grid Computing Section 1.3).

Ainsi le projet **DisDaMin** (Distributed Data Mining) a pour but de présenter des solutions algorithmiques de déploiement des traitements d'analyse de données en environnement distribué respectant les contraintes associées à ce type de plateforme.

Les spécificités de support et de traitements seront exposés, plus particulièrement concernant le problème des règles d'association (problématique initiale du projet). Ensuite, les limitations actuelles seront présentées avant d'introduire l'approche proposée pour le projet DisDaMin pour le problème des règles d'association, et le schéma général de traitement adopté.

1.1 Conjonction des problématiques de Data Mining et de traitements distribués sur Grille de calcul : Projet DisDaMin

La problématique à la base du projet **DisDaMin** consistait à étudier une base de données médicale (DiabCare), pour le problème de recherche de règles d'association.
La base de données DiabCare est constituée de données concernant le diabète, elle est

considérée dans une forme brute (telle que fournie par les médecins).

Les algorithmes (parallèles ou non) existants pour résoudre le problème des règles d'association ne paraissant pas adaptable aux contraintes d'un environnement distribué, il a été décidé de s'intéresser à la création d'algorithmes, en prenant en compte dès la conception des méthodes :

- les spécificités du problème de data mining (voir Section 1.5.2) ;
- les spécificités du support d'exécution (voir Section 1.3).

La nécessité de disposer d'une méthode de clustering distribuée étant apparue également, nous avons été amenés à nous intéresser à cet autre problème de data mining qu'est le clustering, toujours en considérant à la fois les contraintes du problème et les contraintes de l'environnement d'exécution.

1.2 Le Data Mining ou la Fouille de données

Le Data Mining est un processus d'extraction de connaissances qui s'inscrit dans un contexte d'aide à la décision, utilisable dans des secteurs très variés. Il s'inscrit totalement dans le cadre de performance, dans une société où la détention d'informations apparaît comme un atout majeur pour les entreprises.
On distinguera le terme "data mining" (fouille de données) et le terme "extraction de connaissance", même si par abus de langage ces termes sont utilisés pour définir la découverte (ou extraction) de connaissances : **KDD** (Knowledge Discovery in Databases).
L'extraction de connaissances est réalisée grâce au processus particulier qu'est la fouille de données qui apparaît donc comme une méthode d'extraction parmi d'autres.

Nous rappelons ci-dessous plusieurs définitions extraites de la littérature abondante concernant ce domaine.

1.2.1 Plusieurs Définitions de Data Mining

- Terme général désignant l'ensemble des procédures mathématiques qui consistent à extraire des informations significatives d'un ensemble de données pouvant être soit de grande dimension, soit de grande complexité.
- Processus de recherche d'information dans de grandes quantités de données.
- Exploration et analyse des données par des moyens automatiques et semi-automatiques pour la découverte de modèles de données ou la découverte de connaissances.
- Un processus itératif et interactif de découverte de modèles valides, nouveaux, utiles et compréhensibles dans une base de données massive.
- Une étape dans le processus d'extraction de connaissances.
- Une méthode pour "torturer l'information jusqu'à ce qu'elle avoue".

1.2.2 Le Data Mining : Pour quoi faire ?

Les évolutions technologiques réalisées ces dernières années ont permis de diminuer de façon considérable les coûts de collecte et de stockage de données. De nombreuses entreprises ont saisi l'opportunité d'archiver nombre d'informations (informations clients, informations produits...) dans l'espoir de pouvoir les exploiter.
Ces sources d'informations potentielles jouent un rôle important dans le domaine de la concurrence, il reste à les utiliser au mieux.

Les techniques traditionnelles de statistiques ne sont pas utilisables sur d'aussi grandes quantités de données sans utilisation de techniques d'échantillonnage. Reste alors à pouvoir déterminer une méthode valide et performante d'échantillonage.
Le Data Mining apparaît alors comme la solution pour extirper des connaissances de cet amas de données disponibles.
Le Data Mining peut parfois également représenter une solution pour réduire les quantités de données, pour concentrer celles-ci (cataloguages, classification, segmentation...).

Le Data Mining apparaît ainsi comme un enjeu majeur dans la prise de décisions, il constitue un support d'aide à la décision important, tant dans des secteurs concurrentiels (domaine commercial, domaine bancaire, assurances...) que dans des secteurs tels que la santé, l'environnement...

1.2.3 Les différentes tâches du Data Mining

Le Data Mining désigne en réalité un ensemble de traitements très différents, menant à la découverte de connaissances variées. Ces traitements sont :

- **La classification** (tâche de prédiction) : affecter une classe à chaque instance.
- **Le clustering (ou classification non supervisée)** (tâche descriptive) : identifier des groupes d'instances.
- **La découverte de règles d'associations** (tâche descriptive) : rechercher des implications entre attributs.
- **La découverte de séquences** : similaire à la recherche de règles d'association avec insertion d'une notion de temps.
- **La détection de déviation / la détection d'écart** : identifier des valeurs exceptionnelles.
- **La recherche de similitudes.** : identifier des séquences communes entre instances (domaine de la bioinformatique).

Le Data Mining est une appellation qui regroupe donc plusieurs techniques très différentes les unes des autres. Le domaine est vaste, nous avons choisi de porter plus particulièrement notre attention sur la recherche de règles d'association.

1.3 Grille de calcul (GRID Computing)

Le projet DisDaMin propose d'offrir des solutions aux problèmes de Data Mining permettant le traitement de grandes quantités de données. Du fait de la grande quantité de données à traiter, le recours au parallélisme s'avère indispensable. Ainsi des méthodes de

calcul haute performance pour le Data Mining sont proposées.

On pourra ainsi considérer deux types de traitements parallèles :

- le parallélisme de type SMP/CCNUMA ;
- le parallélisme distribué pour l'utilisation de réseaux de stations, de grappes de PC ou à plus grande échelle de grilles de calcul (Grid Computing).

Ainsi les spécificités de ces types de support doivent être prises en considération.

1.3.1 Parallélisme de type SMP/CCNUMA

Les principes de base des architectures de type SMP ou CCNUMA sont (voir Figure 1.1) :

- l'utilisation simultanée de plusieurs processeurs ;
- l'existence d'une mémoire commune ou partagée par les processeurs (SMP : Shared Memory Processors, CCNUMA : Cache Coherent Non Uniform Memory Access) ;
- la disponibilité d'un réseau d'interconnexion interne rapide.
- l'homogénéité d'un système d'exploitation unique.

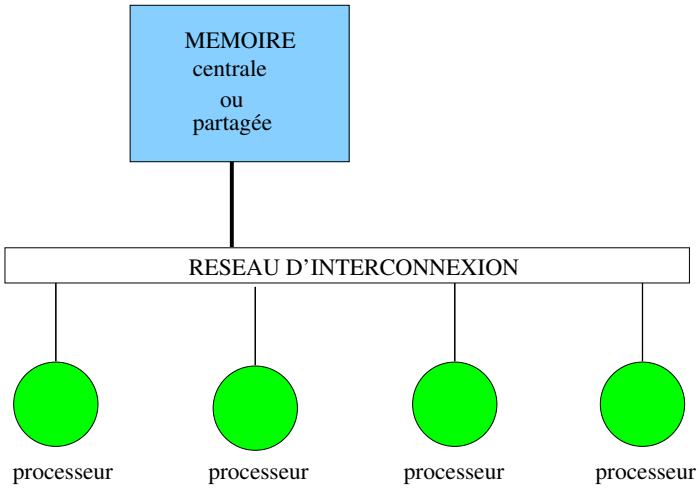


FIG. 1.1 – Architecture SMP.

Le problème des supercalculateurs de ce type réside dans leur coût, les rendant indisponibles en dehors de sociétés ou de centres spécifiques.

1.3.2 Parallélisme sur réseaux de stations et grille de calcul

Les principes de base de ce type de support sont (voir Figure 1.2) :

- l'utilisation simultanée de plusieurs stations de travail ;
- l'absence de mémoire commune ;
- la présence d'un réseau d'interconnexion lent par rapport à la puissance des machines (type Internet).
- l'hétérogénéité d'un système composé d'une grappes de stations ou d'une grappe de grappes.

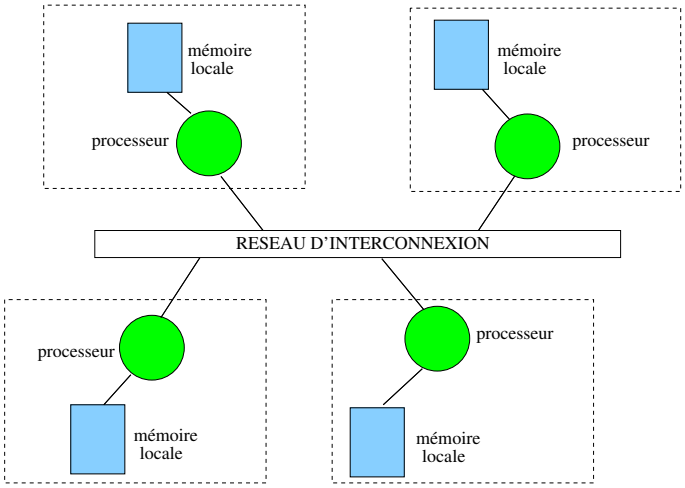


FIG. 1.2 – Architecture de type réseau de stations et grille de calcul.

Ces plateformes d'exécution proviennent de la mise en commun de ressources de traitement et de stockage disponibles dans le réseau et s'avèrent plus abordables financièrement (disponibilité d'un réseau de stations de travail sous-exploitées au sein d'une entreprise de taille moyenne).
Les utilisateurs se partagent les ressources du réseau (ou de la la grille) et chacun d'entre eux peut accèder à la totalité des ressources. C'est en fait une "globalisation virtuelle" d'infrastructures informatiques qui peut être composée de tout type d'unités de traitement et de stockage.
La distinction entre réseau de stations et grille réside dans la taille de l'infrastructure :

- un réseau de stations consiste à mettre en commun des stations de travail ou PC (exemple : les stations disponibles au sein d'une entreprise, d'un laboratoire d'université, d'une salle de travaux pratiques).
  On désigne par ce terme des grappes de stations spécialisées avec un middleware disponible ou un réseau homogène basique.
- à plus grande échelle, une grille de calcul permet de mettre en commun des réseaux de stations et éventuellement des supercalculateurs en une hiérarchie de clusters (grappe de grappes) (exemple : GRID'5000 - voir Annexe D et Figure 1.3, un exemple d'architecture de type grille). Chaque sous-réseau de stations peut être appelé sous-grappe.

Le développement de logiciels d'exploitation de plateformes de type réseau de stations de travail ou grille de calcul est un des défis de la recherche actuelle.

1.3.3 Contraintes à prendre en compte dans le projet DisDaMin

Le projet DisDaMin se concentre sur l'exploitation de ressources disponibles au sein d'un réseau de stations de travail ou d'une grille de calcul pour traiter des bases de données, avec la volonté d'exploiter au maximum les ressources à disposition sans nécessiter de structures spécifiques pour l'exécution.

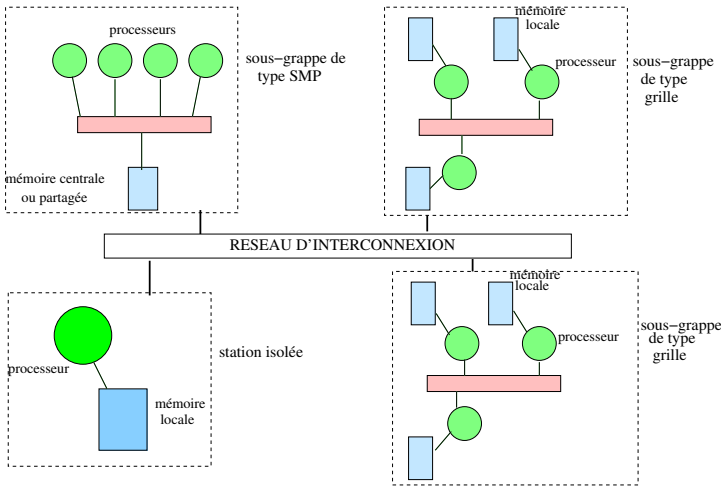
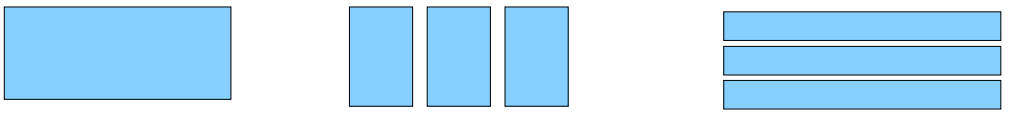


FIG. 1.3 – Exemple d'architecture de grille de calcul par sous-grappes.

Les solutions parallèles existantes utilisent l'existence d'une mémoire commune (ou partagée) avec accès rapide aux données. Dès lors ces solutions ne peuvent être utilisées dans un contexte de réseau de stations ou de grille puisqu'elles sont développées pour une exécution sur un système homogène (Parallel Data Miner pour IBM-SP3 par exemple).

Dans un système hétérogène de type grille de calcul, **la base de données** peut être (voir Figure 1.4) :

- centralisée ;
- distribuée par attributs (fragmentation verticale) : multi-base
- distribuée par instances (fragmentation horizontale) : base distribuée



base centralisée | fragmentation verticale de la base | fragmentation horizontale de la base

FIG. 1.4 – Possibilité de distribution de la base de données.

Dans ce type d'infrastructure réseau, on ne dispose pas de mémoire centrale.
Pour assurer un traitement optimal, il faudra s'assurer de distribuer les données de manière adéquate pour permettre la distribution du traitement et l'exploitation des ressources de stockage locales.
Une **fragmentation** de la structure de données à exploiter pré-existe ou s'avère indispensable. Une fragmentation existante peut nécessiter une adaptation : base distribuée au départ et nécessité d'un contexte multibase, ou multibase au départ et nécessité d'une base distribuée.

Une fois les données distribuées dans les mémoires locales et du fait de la non-existence d'une mémoire commune, le traitement devra s'effectuer sur les vues partielles disponibles localement sur chaque noeud, nécessitant ainsi des **communications** dans le schéma

de résolution.

Le **réseau** utilisé dans une plateforme d'exécution distribuée étant relativement **lent** par rapport à la puissance des machines, l'efficacité du parallélisme distribué pourra être amélioré par :

- un **recouvrement des temps de communications** par l'exécution des instructions CPU pour remédier à la lenteur de ces communications ;
- l'utilisation de l'**asynchronisme** de manière à utiliser les temps d'attente inactifs (pour une exploitation maximale de la puissance de calcul) et à éviter les temps d'attente inactive (lors de la synchronisation des tâches) ;
- l'utilisation d'un principe de pipeline (data-flow par disponibilité) pour anticiper des calculs et ainsi exploiter la puissance de calcul.

On prendra ainsi en compte les contraintes du support d'exécution (voir Toursel [63] pour plus de détails) :

- exploiter au maximum la puissance de calcul et les capacités de stockage disponibles ;
- utiliser au maximum le parallélisme aux différents niveaux du schéma de résolution ;
- effectuer des traitements parralléles les plus indépendants possibles ;
- limiter les communications et interdire les phases de synchronisations (du fait de la lenteur et de la non-fiabilité du réseau) ;
- exploiter au maximum les approches pipeline, le recouvrement des communications par des instructions CPU.
- l'utilisation de Java RMI permettra d'assurer l'interopérabilité entre les noeuds de la grappe ou de la grille.

Les trois éléments majeurs à prendre en compte sont :

1. la distribution des données ;
2. la distribution des traitements ;
3. les collaborations entre les traitements.

Les points 1 et 2 visent à assurer une adéquation entre la distribution des données et des traitements. Le point 3 devra exploiter le recours à l'asynchronisme.

1.4 Le Problème de la Recherche de Règles d'Association

Une des principales applications de la recherche de règles d'association appartient au secteur de la distribution dans le cadre de "l'analyse du panier de la ménagère" (Market-Basket Analysis), c'est-à-dire la recherche d'associations entre produits sur les tickets de caisse (on recherche des produits qui tendent à être achetés ensemble).
Bien que cette application soit la plus répandue, la recherche de règles d'association peut s'appliquer à bien d'autres secteurs d'activités : banques, communications, télécommuni-

cations, secteur médical...

L'attrait principal de cette méthode est la clarté des résultats qu'elle fournit. Ceux-ci sont simples à exploiter par les professionnels concernés.

1.4.1 Définition du Problème

Dans les traitements classiques de bases de données, on s'intéresse à des bases de données B composées d'un ensemble d'instances (ou enregistrements) et d'attributs en général continus.
Chaque instance d'une base B associe une valeur à chacun des attributs continus (et est donc un ensemble de valeurs continues).

Dans le problème de recherche de règles d'association, on suppose la base de données discrétisée de manière à travailler sur une base D composée d'un ensemble d'instances et d'un ensemble d'attributs discrets (ou nominaux) appelés **items**.

Remarque : Dans la suite du document, on considérera des attributs continus (tels que ceux utilisés dans B) pour les phases de discrétisation et de fragmentation, puis des attributs discrets (items tels que ceux utilisés dans D) pour la phase de génération de règles d'association.

On appelle **itemset** un ensemble d'items ; le nombre d'items d'un itemset constitue sa longueur (un itemset contenant k items est appelé un k-itemset).
Chaque instance d'une base D est un itemset.

La recherche de règles d'association consiste à produire des règles de dépendance, des relations entre les items d'une base D, et ceci afin de prédire l'occurrence d'autres items.

Soit **I={**i_1**,** i_2**, ...,** i_m**}** un ensemble de ***m*** items et soit **D** un ensemble d'instances (constituées des items i_i, i_j, ..., i_k de **I**).

Une règle d'association est une implication de la forme **X =>** **Y**, où X et Y sont inclus dans I et X ∩ Y= Ø[1].

- X est la condition ou l'antécédent,
- Y est la conclusion ou la conséquence.

La recherche de règles s'appuie sur des mesures statistiques qui permettent de générer les règles les plus pertinentes et de limiter l'espace de recherche :

- A chaque itemset, on associe une mesure appelée **support** : le support d'un itemset est le pourcentage d'instances de D qui contiennent l'itemset. (le support permet de mesurer l'intérêt de l'itemset, sa fréquence dans le jeu de données).
- A chaque règle d'association, on associe une mesure appelée **confiance** : $confiance(X \rightarrow Y) = \frac{support(X \cup Y)}{support(X)}$.
  (la confiance permet de mesurer une réelle causalité entre la condition et la conclusion).

[1] Formulation du problème proposée par Agrawal et al. [1] et [3]

	les instances
SALES	{lait, bière, beurre}
	{pain, lait, bière}
	{pain, bière, fromage}
	{pain, lait, bière, fromage}

TAB. 1.1 – Exemple de base D pour les règles d'association sur données issues du panier de la ménagère

- De nombreuses autres mesures d'intérêt des règles existent : la conviction, le lift, le coefficient de corrélation, la JMeasure (voir [7]).

La recherche de règles d'association consiste à trouver les règles de support et de confiance (ou autre mesure) supérieurs à certains seuils fixés au départ.

Notations :

On désignera par **k-itemset** u un itemset de longueur k, auquel on associera un support $support(u)$.

Exemple :

Soit la base D décrite par la Table 1.1 :

La base est composée de 4 instances. L'ensemble I des items est {*pain*, *lait*, *bière*, *fromage*, *beurre*}.

33% des instances qui contiennent *lait* et *bière* contiennent également *fromage*.
25% des instances contiennent à la fois *lait*, *bière* et *fromage*.

La règle {lait, bière} $=>$ {fromage} a pour *support* 25% et pour *confiance* 33%.

Dans l'ensemble des travaux existants, l'extraction de règles d'association est décomposée en deux sous-problèmes :

- la recherche des ensembles fréquents d'items (i.e. les itemsets fréquents),
- la génération des règles d'association à partir de ces itemsets fréquents.

Dans ces travaux, l'accent a été mis sur la recherche des itemsets fréquents, celle-ci représentant le principal coût de traitement dans le cas de grandes bases de données (en particulier celles comportant un grand nombre d'attributs), à cause de la génération de candidats (voir Section 2.2.2).

1.4.2 La Recherche des Itemsets Fréquents

La recherche des itemsets fréquents est un problème non trivial car le nombre d'itemsets potentiellement fréquents est exponentiel par rapport au nombre d'items considérés dans la base de données.

On appelle itemset **fréquent** un itemset dont le support est supérieur à un seuil ***minsup*** fixé.
Si le support n'est pas atteint, on dit que l'itemset est **infréquent**.

Soit **I**, l'ensemble des items, et *minsup* un seuil minimal de support.
L'ensemble **F** des itemsets fréquents est : F= $\{l \subset I \mid l \neq \emptyset$ et support(l) $\geq$ *minsup*$\}$
Si $Card(I)$ = M, le nombre d'itemsets possibles est 2^M.

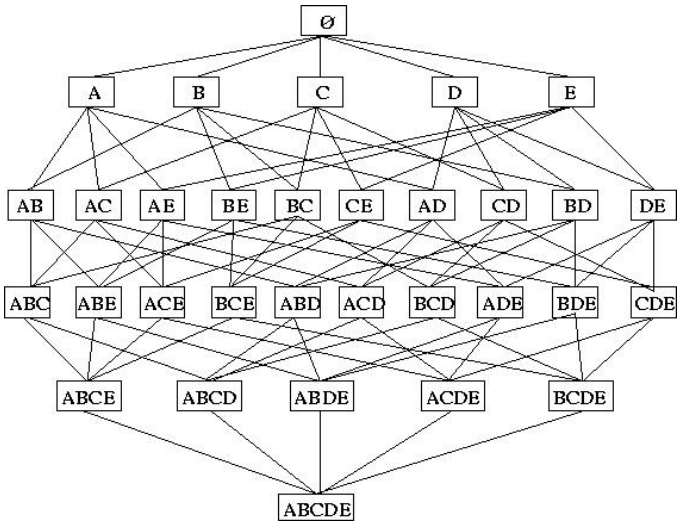


FIG. 1.5 – Treillis d'itemsets.

La figure 1.5 présente le treillis des itemsets pour l'ensemble **I={A, B, C , D, E}**. Les itemsets forment un ensemble ordonné, avec l'inclusion comme relation d'ordre. La borne supérieure est définie par l'union et la borne supérieure est définie par l'intersection.

Dans l'exemple, pour seulement 5 items dans **I**, l'ensemble des itemsets possibles comporte 32 (= 2^5) éléments (avec l'itemset vide en haut de la figure).

La phase de recherche de ces itemsets fréquents est la phase la plus coûteuse de l'extraction de règles d'association du fait de la taille exponentielle de l'espace de recherche et du nombre élevé nécessaire de balayages complets du jeu de données.
Ces balayages sont nécessaires afin d'évaluer les supports des itemsets (qui permettent de déterminer lesquels sont fréquents) puis de calculer les confiances des règles d'association, mais constituent des opérations très coûteuses en temps d'exécution.

Remarque : Ces deux critères (le nombre de balayages des données et le nombre d'itemsets générés) sont les deux facteurs d'efficacité ou d'inefficacité d'un algorithme de re-

cherche de règles d'association.

L'ensemble des k-itemsets fréquents constituent un treillis partiel (voir Figure 1.6). Les supports des itemsets fréquents diminuent de $\emptyset$ vers I, avec $sup(X) > sup(XY)$.

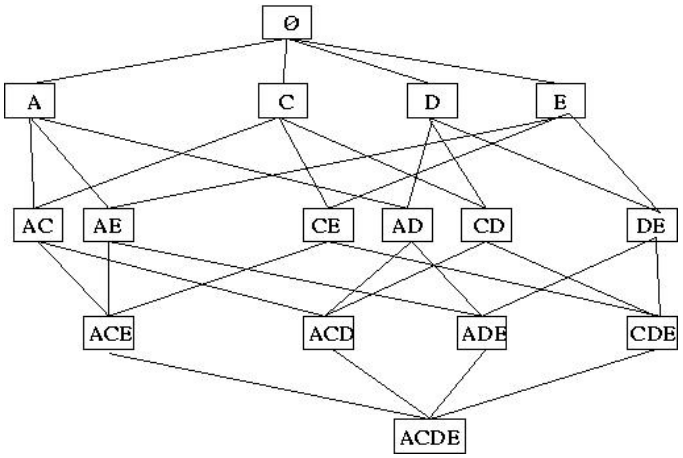


FIG. 1.6 – Treillis partiel composé des itemsets fréquents.

Une approche simpliste consisterait à tester le support de chaque itemset possible. Les cas réels d'application présentent au minimum une centaine d'items et un nombre d'instances en millions (auquel est lié le nombre d'accès disque ou de communications pour calculer les supports) ! De plus, le nombre d'itemsets fréquents est faible par rapport au nombre total d'itemsets.
Il convient donc d'utiliser des approches plus complexes, on peut distinguer deux types d'algorithmes :

- des algorithmes qui proposent une réduction de l'espace de recherche (par exemple l'algorithme de base Apriori, voir Section 4.1.2),
- des algorithmes qui proposent une réduction du nombre de balayages nécessaires du jeu de données (par exemple l'algorithme DIC, voir Section 4.1.2).

1.4.3 La Génération des Règles d'Association

Pour chacun des itemsets fréquents obtenus, on va rechercher les règles d'association qui possèdent une confiance supérieure au seuil minimal de confiance fourni par l'utilisateur.

On regarde les différentes combinaisons condition-conclusion possibles pour chaque itemset. On garde la combinaison la plus pertinente (celle de confiance la plus élevée), si elle répond au critère de seuil minimal de confiance.

A partir d'un k-itemset F_k fréquent, on cherche les règles de type : $F_i => F_j$ de support et de confiance suffisants.
F_i et F_j sont des itemsets de tailles respectives i et j, tels que : i + j = k, $F_i \subset F_k$, $F_j \subset F_k$ et $F_i \cap F_j$ = Ø.

Exemple :

Soit le 3-itemset {A B C} déterminé fréquent (les supports des différents itemsets sont indiqués par le Tableau 1.2).
Une règle d'association, qui contiendra les trois éléments A, B, C, aura un support de 5%.

itemset	support
{ A }	45%
{ B }	42.5%
{ C }	40%
{ A B }	25%
{ A C }	20%
{ B C }	15%
{ A B C }	5%

TAB. 1.2 – Exemple : les supports des itemsets.

On calcule la confiance des différentes combinaisons possibles. Pour plus de clarté, on ne s'intéresse qu'aux règles dont la conclusion ne comporte qu'un item. Cette limitation est habituelle dans ce type de traitement et permet à la fois d'assurer la lisibilité des résultats obtenus et de limiter l'espace de recherche. Les confiances des combinaisons possibles sont données par le tableau 1.3.
La règle apparaissant en dernière ligne est celle de confiance la plus élevée. C'est la plus intéressante des règles générées à partir de l'itemset {A B C}.
Si on fixe le support minimal de confiance à 30%, seule cette règle sera retournée à l'utilisateur (c'est la seule à posséder une confiance suffisante, i.e. supérieure au seuil).
Si par contre, on fixe le seuil de confiance minimal à 40%, aucune règle ne sera renvoyée. Cette méthode permet de limiter le nombre de règles renvoyées : on ne fournit que les plus pertinentes.

condition	conclusion	confiance
{ A B }	{ C }	20%
{ A C }	{ B }	25%
{ B C }	{ A }	33%

TAB. 1.3 – Exemple : les confiances des différentes règles.

1.5 Verrous technologiques et limitations actuelles

La recherche de règles d'association est un problème pour lequel de nombreux algorithmes séquentiels et parallèles existent. Nous présentons ici les principes des algorithmes et les verrous existants de manière synthétique afin d'introduire le schéma général DisDaMin pour la recherche de règles d'association. Les détails algorithmiques et les limitations d'utilisation induites seront détaillés dans la Partie III consacré à la recherche distribuée de règles d'association qui présente l'algorithme DICCoop.

1.5.1 Principe des algorithmes existants pour la recherche de règles

Les algorithmes d'extraction des itemsets fréquents procèdent de **façon itérative**, c'est à dire que l'ensemble des itemsets fréquents est parcouru par niveau (parcours en largeur du haut vers le bas sur la Figure 1.5). Lors de la k^{eme} itération, on recherche les k-itemsets fréquents, c'est-à dire les itemsets de longueur k qui sont fréquents. A l'itération suivante, la $(k+1)^{eme}$, on recherchera les (k+1)-itemsets fréquents et ainsi de suite.
Les premiers algorithmes d'extraction des itemsets fréquents "par niveaux", **AIS** [2], **Apriori** ([3], [60]) et **SETM** [37] ont été proposés en 1993.

Plusieurs autres algorithmes permettant de réduire le temps d'extraction des itemsets fréquents sont ensuite apparus, ainsi que plusieurs optimisations et structures de données permettant d'améliorer l'efficacité de l'algorithme Apriori :

- **AprioriTid** par Agrawal et Srikant - 1994 [3],
- **AprioriHybrid** par Agrawal et Srikant - 1994 [3],
- **DHP** (Direct Hashing and Pruning) par Park et al. - 1995 [52],
- **Partition** par Savasere et al.- 1995 [59],
- **Sampling** par Toivonen -1996 [62].
- **DIC** (Dynamic Itemsets Counting) par Brin et al. - 1997 [14],

Les détails de ces méthodes sont présentées dans le Chapitre III.

Afin de limiter le nombre de candidats considérés à chaque itération, l'algorithme Apriori (et ses dérivés) se base sur les deux propriétés suivantes :

Propriété 1 : Tous les sous-ensembles d'un itemset fréquent sont fréquents (voir justification de cette propriété Section 1.5.1).

La propriété 1 permet une limitation du nombre de candidats générés lors de la k^{eme} itération par une jointure conditionnelle des itemsets fréquents, de taille k-1 ,découverts précédemment.

Propriété 2 : Tous les sur-ensembles d'un itemset infréquent sont infréquents (voir justification de cette propriété Section 1.5.1).

La propriété 2 permet la suppression d'un candidat de taille k lorsqu'au moins un de ses sous-ensembles de taille k-1 ne fait pas partie des itemsets fréquents découverts précédemment.

Des détails de fonctionnement de ces algorithmes sont donnés en Section 4.1.2.

Principe des parallélisations existantes

Les algorithmes parallèles existants, utilisent différents principes pour la parallélisation :

- Duplication des itemsets candidats (CD 1999 [65], Parallel Partition 1995 [59], PDM 1995 [53])
- Partitionnement des itemsets (DD 1999 [65], IDD 1999 [65], HPA 1996 [57])
- Approche Hybride : Réplication partielle des itemsets candidats (HD 1999 [65], HPA-ELD 1996 [57])

Les principes de ces algorithmes sont détaillés en Section 4.1.2.

Ces parallélisations nécessitent d'introduire une nouvelle propriété.

Propriété 3 : Pour qu'un itemset soit globalement fréquent il faut qu'il soit localement fréquent sur au moins un site (voir justification de cette propriété Section 1.5.1).

Quel que soit le principe de l'algorithme parallèle choisi, il nécessite de nombreuses communications pour transiter tantôt les itemsets et leurs informations, tantôt les données, la plupart de ces communications devant se faire de manière plus ou moins synchronisée (voir la Section 4.1.2 pour plus de détails concernant les algorithmes parallèles).

Justification des propriétés énoncées

Soit B une base de n instances.

Soient le $3-$itemset ABC, $\mathcal{I}_{ABC}$ l'ensemble des instances de la base B qui contiennent le triplet d'items (A,B,C) et $Card(\mathcal{I}_{ABC})$ la cardinalité de l'ensemble $\mathcal{I}_{ABC}$, et donc $support(ABC)$ étant le support du $3-$itemset ABC, $support(ABC) = \frac{Card(\mathcal{I}_{ABC})}{n}$.

Le $3-$itemset ABC possède trois sous-ensembles de taille 2 ($2-$itemsets) : AB, AC, BC, d'ensembles d'instances associés $\mathcal{I}_{AB}$, $\mathcal{I}_{AC}$ et $\mathcal{I}_{BC}$ de cardinalités respectives $Card(\mathcal{I}_{AB})$, $Card(\mathcal{I}_{AC})$ et $Card(\mathcal{I}_{BC})$ (avec $support(AB) = \frac{Card(\mathcal{I}_{AB})}{n}$, $support(AC) = \frac{Card(\mathcal{I}_{AC})}{n}$ et $support(BC) = \frac{Card(\mathcal{I}_{BC})}{n}$).

Justification de la Propriété 1 :

Si le $3-$itemset ABC est fréquent alors *minsup*$\leq support(ABC)$.
$\mathcal{I}_{ABC}$ est un sous-ensemble de $\mathcal{I}_{AB}$: toutes les instances contenant le triplet d'items (A,B,C) contiennent obligatoirement le couple d'items (A,B). Par contre, toutes les instances contenant le couple d'items (A,B) ne contiennent pas forcément l'item C, et donc ne contiennent pas forcément le triplet d'items (A,B,C).

Donc $\mathcal{I}_{ABC} \subset \mathcal{I}_{AB}$,
d'où $Card(\mathcal{I}_{ABC}) \leq Card(\mathcal{I}_{AB})$ et par association, n étant positif $\frac{Card(\mathcal{I}_{ABC})}{n} \leq \frac{Card(\mathcal{I}_{AB})}{n}$
on a donc : $support(ABC) \leq support(AB)$.

Puisque *minsup*$< support(ABC)$, on a finalement *minsup*$\leq support(AB)$ et donc AB est fréquent.

De même, on peut montrer que AC et BC, ainsi que tous les autres sous-ensembles de ABC (A, B et C) sont fréquents.
On a donc bien : **Tous les sous-ensembles d'un itemset fréquent sont fréquents**.

Justification de la Propriété 2 :

Si le $2-$itemset AB est infréquent alors $support(AB) <$*minsup*.

Le $3-$itemset ABC est un sur-ensemble du $2-$itemset AB, on a donc : $support(ABC) \leq support(AB)$ (voir ci-dessus).

Puisque $support(AB) <$ *minsup*, on a finalement $support(ABC) <$*minsup* et donc ABC est fréquent.

De même, on peut montrer tous les autres sur-ensembles de AB (ABC, ABD, ABE, ... ABCD, ...) sont infréquents.
On a donc bien : **Tous les sur-ensembles d'un itemset infréquent sont infréquents**.

Justification de la Propriété 3 :

Soit $\mathcal{I}_{AB_G}$ un sous-ensemble d'instances de la base contenant l'itemset AB de cardinalité $Card(\mathcal{I}_{AB_G})$, $\frac{Card(\mathcal{I}_{AB_G})}{n}$ est donc le support global du $2-$itemset AB.

Supposons les n instances de la base distribuées de manière uniforme sur k sites (chaque site gère $\frac{n}{k}$ instances), avec sur chaque site i un sous-ensemble d'instances $Card(\mathcal{I}_{AB_i})$ contenant l'itemset AB de cardinalité $Card(\mathcal{I}_{AB_i})$.

$\mathcal{I}_{AB_G} = \mathcal{I}_{AB_1} \cup \mathcal{I}_{AB_2} \cup \ldots \cup \mathcal{I}_{k_{AB}}$
$\mathcal{I}_{AB_G} = \bigcup_{i=1}^{h} \mathcal{I}_{AB_k}$.

Les $\mathcal{I}_{AB_i}$ représente une distribution sans dupplication de $\mathcal{I}_{AB_G}$, donc $\mathcal{I}_{AB_i} \cap \mathcal{I}_{AB_j} = \emptyset$

$\forall i, j$.

On a donc
$Card(\mathcal{I}_{AB_G}) = Card(\mathcal{I}_{AB_1}) + \ldots + Card(\mathcal{I}_{AB_k})$.
$Card(\mathcal{I}_{AB_G}) = \sum_{i=1}^{h} Card(\mathcal{I}_{AB_k})$.

Un itemset AB est globalement fréquent si
minsup $\leq support_G(AB)$, donc *minsup*$n \leq Card(\mathcal{I}_{AB_G})$.

Un itemset AB est localement fréquent sur un site i si
minsup $\leq support_i(AB)$, donc $\frac{n}{k}$*minsup*$\leq Card(\mathcal{I}_{AB_i})$.

Si AB est infréquent sur chaque site i alors $Card(\mathcal{I}_{AB_i}) < \frac{n}{k}$*minsup* $\forall i$
donc $\sum_{i=1}^{k} Card(\mathcal{I}_{AB_i}) < \frac{n}{k}k$*minsup*
et donc $Card(\mathcal{I}_{AB_G}) < n$*minsup*, c'est-à-dire AB est globalement infréquent.

Il est donc nécessaire, pour que AB soit globalement fréquent, qu'au moins l'un de ses sous-ensembles soit fréquent.
On a donc bien : **Pour qu'un itemset soit globalement fréquent il faut qu'il soit localement fréquent sur au moins un site**.

Principe des projets distribués existants pour le data mining et la recherche de règles

Un certain nombre de projets de Data Mining dits distribués existent parmi lesquels on peut distinguer tris catégories :

- Des projets qui proposent des middleware applicatifs et des méthodes d'organisation des traitements.
  Ces projets proposent des outils et des services appliqués au data mining pour des environnements de type grille. Ils incluent des mécanismes d'intégration et de déploiement des algorithmes classiques de data mining sur grille de calculs (avec essentiellement la mise en commun des ressources) :
  Datamining Grid ([17] 2003-...), Discovery Net ([18] 2002-2005), Knowledge Grid ([43] 2000-2003), Grid Miner ([33] 2003-...), Grai (Grid Computing and Artificial Intelligence [32] 2005-2007), ...
- Des projets qui proposent des versions distribuées des algorithmes existants mais avec des synchronisations ([50], [51], ...)
- Les nouvaux projets européens en cours de démarrage...

1.5.2 Détails des verrous technologiques du problème de recherche de règles d'association

Le problème de génération des règles d'association nécessite un accès à la totalité des données à traiter (toutes les instances et tous les attributs).
Cette nécessité de "disposer" de la totalité des données et l'augmentation de la taille des bases de données à traiter pose problème :

- problème de stockage pour les versions séquentielles
- problème de la distribution et du nombre de communications de synchronisation nécessaires dans les versions parallèles et distribuées existantes.

Nous proposons donc d'utiliser le principe du **divide and conquer** afin de contourner les verrous existants de deux manières :

- tenter de diminuer l'espace de recherche effectivement visité afin de diminuer le temps de traitement.
- diminuer le temps de traitement par le recours au parallélisme, avec exécution optimale en contexte distribué en maximisant le degré de parallélisme, le recouvrement des communications et l'utilisation de l'asynchronisme.

Verrous liés à la compléxité du problème

La génération des itemsets fréquents (phase préalable et essentielle au problème des règles d'association) possède une complexité liée essentiellement au nombre d'items à considérer pour la recherche et au nombre d'instances.
La complexité du problème est de l'ordre de $O(n2^m)$ (pour n le nombre d'instances et m le nombre d'items).
Ainsi, pour 1 Tera de données et 200 items, on a une complexité de l'ordre de $10^{12}2^{200}$!
Cette complexité du problème apparaît clairement comme un verrou à l'utilisation de la méthode à grande échelle.

Les optimisations apportées ont essentiellement portées sur la limitation du nombre de parcours des données et la réduction du nombre d'accès à ces données.
Or le nombre de données disponibles augmente considérablement en longueur (nombre d'instances), mais également en largeur (nombre d'attributs). Les algorithmes de règles d'association se doivent de fournir des méthodes adéquates sur des bases de données vastes reprenant des attributs variés qui une fois croisés peuvent amener à des savoirs inédits (des règles d'association inédites).

Le recours au parallélisme ne peut pas être à lui seul la solution à cette problèmatique de complexité. Le recours à une distribution spécifique des données peut permettre une diminution de la taille de l'espace de recherche visité.

Verrous liés au parallélisme dans un contexte de Grille de calcul

Les spécificités du problème à résoudre (recherche de règles d'association) associées à celles du support visé pour l'exécution (clusters de stations de travail ou à plus grande échelle Grille de calcul) amène à devoir assurer les points suivants :

- Le but est de pouvoir manipuler de grandes quantités de données ;
- La méthode est basée sur des critères globaux (support, fréquence, ...) alors que l'on dispose de vues locales (partielles) de données du fait de la distribution du problème ;
- Il faut limiter le nombre de communications et de synchronisations ;
- Il faut exploiter au maximum la puissance de calcul disponible
  - effectuer des traitements les plus indépendants possibles ;

– utiliser le parallélisme autant que possible ;
– utiliser une approche pipeline entre les différentes étapes.

Les spécificités des méthodes d'extraction de connaissances doivent être prise en compte, à savoir :

- le traitement doit s'effectuer sur la base de données entière ;
- la comparaison de chacune des parties de la base avec toutes les autres doit être "possible" pour obtenir des informations globales sur la base de données (support, fréquence)

Le projet **DisDaMin** propose de considérer sous un aspect distribué la recherche de règles d'association.

Ainsi les contraintes des infrastructures distribuées doivent également être prises en compte dans la proposition de nouvelles méthodes.

Les possibilités de distribution des données

Les données sont considérées en tant que base de données multidimensionnelle (à plusieurs attributs).
Une distinction doit être faite entre une distribution homogène et une distribution hétérogène des données issues de bases de données multidimensionnelle, entre un découpage vertical ou horizontal de la base (voir Figure 1.7) :

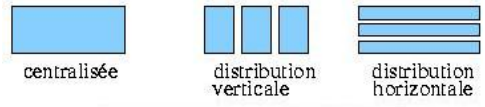


FIG. 1.7 – Possibilités de distribution d'une base.

- une distribution **homogène :** consiste en un découpage horizontal de la base de données. Les instances sont distribuées entre les stations de manière à ce qu'une instance complète se trouve sur une station (tous les attributs pour l'instance considérée). Ce type de distribution sera considérée comme distribution horizontale (ou découpage horizontal) dans la suite.
- une distribution **hétérogène :** consiste en un découpage vertical de la base de données. Les attributs sont distribués entre les stations de manière à ce que toutes les informations relatives à un attribut (pour chaque instance de la base de données) puissent être trouver sur une station (toutes les valeurs pour l'attribut considéré). Ce type de distribution sera considéré comme distribution verticale (ou découpage vertical) dans la suite
- un troisième type de distribution peut être envisagée, il s'agit d'une distribution horizontale et verticale.

Nous considérons ici une distribution (qu'elle soit verticale ou horizontale) de la base D relative à une partition P telle que $P = \{P_i\}, \cup_i P_i = D$ and $P_i \cap P_j = \emptyset, \forall i, j$.
Nous ne discuterons pas des possibilités de distribution avec recouvrement et donc réplication des données. Ce type de distribution nécessite en effet un marquage des données

dupliquées pour assurer une cohérence des traitements, et est a priori mal adapté à des bases qui sont déja de trés grande taille à l'origine.

1.6 Conclusions

La problématique de data mining distribué sur grille de calcul une fois présentée, nous pouvons maintenant présenter le schéma général du projet DisDaMin pour la résolution du problème de recherche de règles d'association.
Ce schéma se base sur un principe de distribution intelligente qui permet de prendre en compte les spécificités du support d'exécution visé (grille de calculs) et de diminuer la taille de l'espace de recherche effectivement visité (et donc le temps d'exécution nécessaire à la résolution du problème).

Chapitre 2

Approche proposée : répartition intelligente des données

Les algorithmes parallèles pour le problème de la recherche de règles d'association (voir détails Section 4.1.2), nécessitent un grand nombre de communications afin d'obtenir un résultat valide sur l'ensemble des données. De plus, ces communications sont soumises à des synchronisations, nécessaires entre chaque itération des algorithmes.

Ces synchronisations sont très pénalisantes dans un environnement distribué de type grille de calcul. La quantité d'information à échanger entre les processeurs est plus adaptée à un environnement incluant une mémoire commune plutôt qu'à un environnement utilisant un réseau de communication relativement lent. Ainsi, il apparaît clairement que ces algorithmes parallèles sont plutôt adaptés à des ordinateurs de type SMP ou CC-NUMA avec mémoire commune et réseau d'interconnexion rapide.

Du fait de la possibilité alternative offerte par les infrastructures de type clusters de stations ou grille de calcul, il est nécessaire de concevoir des méthodes destinées à ces supports.

2.1 Principe général proposé de l'approche par clustering

2.1.1 Introduction

Le schéma général de la solution DisDaMin pour la recherche de règles d'association (voir plus bas) suggère un découpage horizontal de la base de données qui sera détaillé plus loin dans ce rapport.
Ainsi la phase de distribution fournira une fragmentation horizontale des données.

Sur la base des critères nécessaires à cette distribution (voir Section 2.1.2) pour le pro-

blème visé (génération de règles d'association), nous recherchons à effectuer une distribution **intelligente** des données.

Nous proposons la conjecture suivante :

Conjecture :
Une fragmentation "intelligente" de la base de données permet de réduire la taille de l'espace de recherche visité pour la résolution du problème de recherche de règles d'association et d'améliorer l'efficacité de l'exécution distribuée :

- les résultats locaux sont riches sur chaque fragment (nombreuses informations disponibles, qui sont spécifiques aux données du fragment) ;
- les résultats globaux peuvent être obtenus à moindre coût à partir des résultats locaux.

Nous définissons comme critère de distribution intelligente, l'obtention de fragments de données tels que l'on distribue les instances (découpage horizontal de la base) de manière à :

- regrouper sur un même site les instances qui possèdent les même items ;
- regrouper sur des sites distincts des instances qui contiennent des items distincts.

Remarque : Ces critères d'intelligence sont détaillés plus loin (voir Section 2.1.2, et avantages attendus en terme de taille de l'espace de recherche visité 2.1.2)
Ainsi une première phase du traitement consiste à effectuer une **fragmentation par clustering**.
La base de données une fois fragmentée, les fragments de données identifiés doivent être traités en parallèle de la manière la plus indépendante possible, en respectant les contraintes d'un support d'exécution distribué : en limitant les communications, ou tout du moins en interdisant les synchronisations (trop *handicapantes* sur des supports de type grille de calcul). Ainsi, une deuxième phase du schéma général DisDaMin pour la recherche de règles d'association consiste à traiter les fragments de données en parallèles.

2.1.2 Distribution de données

La distribution des données doit se faire non seulement pour assurer un stockage local et éviter les communications d'échange de données mais également de manière cohérente pour permettre à des traitements locaux d'approcher la solution optimale.

Prenant en compte la complexité de la méthode basée sur le nombre d'items, il est décidé d'orienter la distribution sur une méthode permettant de diminuer le nombre d'items à considérer sur chaque site de traitement.

Un découpage vertical de la base de données (voir Possibilités de distribution - Section 1.5.2) pour les traitements liés à la recherche de règles d'association (en particulier l'iden-

tification des itemsets fréquents) amènerait à ne pas pouvoir évaluer un grand nombre de sous-ensembles d'items sur les fragments. En particulier les itemsets composés d'items distribués sur des sites distincts ne seraient pas évalués.

La recherche de règles d'association doit prendre en compte tous les items de la base. La complexité des algorithmes est fortement liée au nombre d'items considérés et d'itemsets "visités". Un découpage vertical permettrait de réduire la taille de l'espace de recherche visité (nombre d'itemsets évalués), mais provoquerait la perte de trés nombreux résultats.

Comme le montre la Figure 2.1, seule une partie des itemsets possibles seraient évaluée. Par exemple, les itemsets {A D} et {A D H} ne seraient évalués. La distribution du traitement doit permettre d'accélérer celui-ci tout en conservant *suffisamment* de résultats, il est clair que cette distribution ne peut pas se faire sur le nombre d'items à considérer et donc pas de manière verticale.

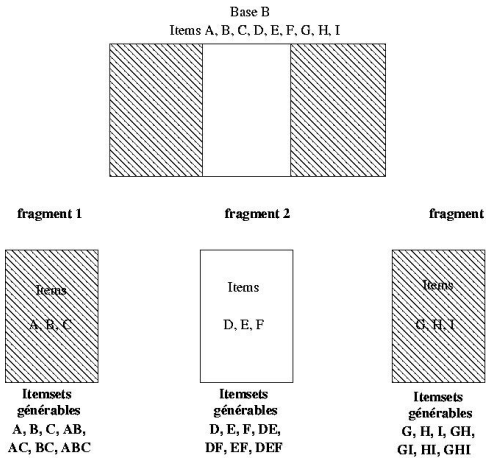


FIG. 2.1 – Découpage vertical de la base.

D'un autre côté, un découpage horizontal quelconque (aléatoire ou cyclique tel qu'utilisé dans les algorithmes parallèles utilisant une distribution horizontale, voir Section 4.1.2) ne permettrait pas d'obtenir une diminuation de l'espace de recherche visité (nombre d'itemsets), en rapport au nombre d'items à considérer (fondement de la complexité du problème de génération des règles d'association, voir Section 2.2.2) et nécessiterait de nombreuses communications.

On considére donc que la problématique revient à distribuer les instances (découpage horizontal) de manière à regrouper sur un même site les instances qui possèdent les même items (attributs discrets) et sur des sites distincts des instances qui contiennent des items distincts (voir plus loin avantages attendus en terme de taille de l'espace de recherche visité 2.1.2) : utilisation d'une **distribution intelligente** (plutôt qu'aléatoire).

L'identification de groupes d'enregistrements similaires peut en effet permettre une telle diminution. Des enregistrements sont similaires s'ils ont un maximum d'items communs. Après le calcul des 1-itemsets fréquents sur un groupe, de nombreux 1-itemsets non fréquents seront éliminés (ceux composés des items qui n'apparaissent que peu dans ce

groupe). La génération des k-itemsets qui suivra en sera facilitée, la taille de l'espace de recherche à visiter diminuant avec le nombre d'items fréquents. La distribution des enregistrements en groupes "cohérents" implique, indirectement, une distribution des items.

Dans la pratique, on ne pourra pas effectuer une séparation parfaite en fonction des items présents dans les fragments, les ensembles d'items dans les instances n'étant pas disjoints. (Une situation idéale correspondrait à la possibilité de répartir les instances telles que les ensembles d'items présents dans les fragments soient disjoints.)

Digression sur Clustering

Le problème revient donc à distribuer les instances en fonction des similarités d'items.
On cherche à former des groupes d'instances tels que :

- dans un groupe, le nombre d'items communs aux instances du groupe soit maximal ;
- entre deux groupes, le nombre d'items communs aux instances soit minimal.

On se ramène ici à une problématique de regroupement par clustering (voir Section 3.1) basé sur une similarité en terme d'items contenus dans les instances.

La méthode de distribution des données va donc être basée sur des méthodes de **clustering** (voir Section 3.1).

Les groupes identifiés lors de la phase de clustering de distribution pourront être considérés comme "profils" d'instances : dans un même groupe (un même profil), on aura des instances très similaires qui représentent potentiellement une sous-population avec des propriétés particulières (première propriété commune, un nombre d'items communs aux instances du profil).

Bénéficiant de l'expérience des travaux de Gupta [35], Lent [44] et Toivonen[61] (qui considéraient le problème inverse, à savoir la construction d'un clustering sur la base des résultats de règles d'association), nous proposons cette méthode de distribution par clustering.

Le principe du clustering et les algorithmes existants seront détaillés plus loin dans le Chapitre II consacré au clustering en contexte distribué qui présente l'algorithme CDP.

Avantages attendus en terme de taille de l'espace de recherche visité à partir d'une distribution par clustering

Les items communs aux instances d'un groupe fourniront une base non négligeable pour assurer la présence d'un certain nombre d'ensembles d'items fréquents (items présents dans un "grand" nombre d'instances).
En contrepartie, des items absents dans les instances d'un groupe (ou très faiblement présents) permettront rapidement de diminuer l'espace de recherche des ensembles fréquents d'items.
Des attributs ou items absents permettront de diminuer la taille potentielle de l'espace de

recherche dès le départ, en ne considérant qu'une partie des attributs sur les fragments. Ainsi, le traitement effectué sur chacun des fragments sera spécialisé aux items présents dans ce fragment d'où un gain au niveau du temps d'exécution (lié aux items absents sur le fragment et donc à la diminution de la taille de l'espace de recherche, voir fondement de la complexité Section 2.2.2).

Avantages attendus en terme de résultats obtenus à partir d'une distribution par clustering

On souhaite générer les itemsets fréquents (et les règles d'association liées) pour la base de données entières. Cette génération est effectuée sur base de fréquence d'itemsets dans des fragments d'instance.
On possède donc, en plus de l'information globale (itemsets globaux, fréquents sur toutes les instances de la base), des informations concernant l'ensemble des itemsets fréquents localement sur chaque groupes d'instances. On peut donc générer les règles d'association locales à chaque fragment d'instances.
Ces fragments d'instances résultant d'une distribution par similarité (clustering), on a des groupes correspondant potentiellement à des profils d'instances (profils de clients d'un supermarché, profils de pathologie sur données médicales ...).

Ces profils identifiés constituent en eux-mêmes un savoir important, il n'est pas trivial de fragmenter un ensemble d'instances en groupes homogènes et d'identifier des sous-populations à caractéristiques communes.

2.2 Complexité de la génération des règles d'association

2.2.1 Définitions

Soit *n* le nombre d'instances, *m* le nombre d'items (attributs discrets).

Un k-itemset est un itemset de taille k, où k peut varier de 1 à *m*.

2.2.2 Complexité de base de la génération d'itemsets fréquents

Le principe des algorithmes de base de génération des itemsets fréquents (voir algorithme APriori, Section 4.1.2) est itératif et consiste à répéter les actions :

- utiliser les $(k-1)$-itemsets pour générer les k-itemsets candidats ;
- scanner les instances pour calculer les supports de candidats.

La complexité de la génération de ces itemsets peut se décomposer en deux composantes :

- le nombre d'itemsets à générer ;
- le calcul de support pour les k-itemsets.

Nombre d'itemsets générés

Le nombre d'itemsets possibles est 2^m.

Le principe de l'algorithme itératif permet de ne pas générer la totalité des itemsets, on n'examine donc pas 2^m itemsets, mais beaucoup moins.
Pour chaque niveau k, on ne génère donc pas tous les k-itemsets mais seulement un sous-ensemble, basé sur le nombre de $(k-1)$-itemsets fréquents identifiés au niveau précédent (voir l'exemple présenté plus loin).

Pour m items, on identifie m 1-itemsets candidats. Le calcul des supports de ces candidats via un examen de l'ensemble des données permet d'identifier m' 1-itemsets fréquents (avec $m' < m$).

Le nombre de $2-$itemsets possibles est $C_m^2 = \frac{m!}{(m-2)!2!} = \frac{m \times (m-1)}{2}$.
Or, on ne génére qu'un sous-ensemble des 2-itemsets en ce basant sur la propriété : **Tous les sous-ensembles d'un itemset fréquent sont fréquents**.

Ainsi, on génére les candidats de taille 2 à partir des m' 1-itemsets fréquents, soit $\frac{m' \times (m'-1)}{2}$ 2-itemsets candidats (voir exemple plus loin) :

$\frac{m' \times (m'-1)}{2} < \frac{m \times (m-1)}{2}$ puisque $m' < m$.

De plus, on ne poursuit pas obligatoirement la génération jusqu'au itemsets de taille m, on s'arrête à la génération des l-itemsets (où $l < m$). Le critère d'arrêt de l'algorithme est lié à l'impossibilité de générer des candidats de taille $l+1$ (dans l'exemple présenté plus loin, on s'arrête à $l = 4$).

Nombre de scans des données

Le nombre de scans nécessaires des données dépend de l'implémentation réalisée. Pour calculer le support des 2^k-itemsets possibles, on doit réaliser :

- 2^k scans de la base, un pour chacun des itemsets, dans la version de base de l'algorithme (on effectue un scan des données pour chaque itemset dont on veut calculer le support) ;
- $k-1$ scans de la base, un pour chaque itération sur k dans une version utilisant une structure de treillis pour stocker les itemsets (on effectue un scan des instances à chaque itération et on fait parcourir la structure à chacune des instances) ;
- moins de $k-1$ scans de la base dans des algorithmes visant à optimiser encore cette étape (voir algorithme DIC, Section 4.1.2)

En considérant, la version de base, nécessitant un scan des données pour chaque itemset, la complexité de la phase de génération des itemsets fréquents est donc de l'ordre de $O(n2^m)$ (où n est le nombre d'instances, m le nombre d'items).

Exemple :

Soit les items A, B, C, D et E et les instances présentées dans la table 2.1.
Le nombre d'itemsets possibles est $2^5 = 32$.

Base D

Identifiant d'instance	Items contenus
1	A C D
2	B C E
3	A B C E
4	B E
5	A B C E
6	B C E

TAB. 2.1 – Exemple : les instances d'une base D.

On effectue le calcul de support pour les 1-itemsets A, B, C, D, E, on obtient les supports présentés table 2.2.

supports pour C_1

1-itemsets	Support
$\langle$ A $\rangle$	3/6 = 50%
$\langle$ B $\rangle$	5/6 $\simeq$ 85%
$\langle$ C $\rangle$	5/6 $\simeq$ 85%
$\langle$ D $\rangle$	1/6 $\simeq$ 15%
$\langle$ E $\rangle$	5/6 $\simeq$ 85%

TAB. 2.2 – Exemple : les supports des 1-itemsets générés sur D.

Basé sur un support minimum nécessaire à un itemset pour être fréquent de $2/6 \simeq 33\%$, parmi les 5 1-itemsets, seuls 4 sont fréquents : A, B, C et E.

A partir des 1-itemsets fréquents, on génére 6 2-itemsets candidats (au lieu de 9 2-itemsets possibles).
Ces 2-itemsets candidats sont présentés table 2.3 avec leurs supports respectifs.

Les 6 2-itemsets candidats sont fréquents (possèdent un support suffisant).

A partir des 2-itemsets fréquents, on génére 4 3-itemsets candidats (au lieu de 9 3-itemsets possibles).
Ces 3-itemsets candidats sont présentés table 2.4 avec leurs supports respectifs.

Les 4 3-itemsets candidats sont fréquents (possèdent un support suffisant).

supports pour C_2

2-itemsets	Support
$\langle$ A B $\rangle$	2/6 $\simeq$ 33%
$\langle$ A C $\rangle$	3/6 $=$ 50%
$\langle$ A E $\rangle$	2/6 $\simeq$ 33%
$\langle$ B C $\rangle$	4/6 $\simeq$ 66%
$\langle$ B E $\rangle$	5/6 $\simeq$ 85%
$\langle$ C E $\rangle$	4/6 $\simeq$ 66%

TAB. 2.3 – Exemple :les supports des 2-itemsets générés sur D.

supports pour C_3

3-itemsets	Support
$\langle$ A B C $\rangle$	2/6 $\simeq$ 33%
$\langle$ A B E $\rangle$	3/6 $=$ 50%
$\langle$ A C E $\rangle$	2/6 $\simeq$ 33%
$\langle$ B C E $\rangle$	4/6 $\simeq$ 66%

TAB. 2.4 – Exemple : les supports des 3-itemsets générés sur D.

A partir des 3-itemsets fréquents, on génére 1 4-itemset candidat (au lieu de 5 4-itemsets possibles).
Le 4-itemset candidat est présenté table 2.5 avec son support.

supports pour C_4

3-itemsets	Support
$\langle$ A B C E $\rangle$	2/6 $=$ 50%

TAB. 2.5 – Exemple : les supports des 4-itemsets générés sur D.

Le 4-itemset candidat est fréquent, mais on ne peut pas générer de candidat de taille 5, la méthode s'arrête.

Dans cet exemple, seuls 16 itemsets ont été générés et il a donc été nécessaire de calculer leur support, au lieu des 31 possibles (32 itemsets moins l'ensemble vide).

2.2.3 Gains issus de l'approche distribuée

Le projet DisDaMin vise à apporter deux types de gains au problème de la recherche de règles d'association. Les premiers gains sont issus de l'utilisation du parallélisme et de la distribution nécessaire aux traitements parallèles.
La méthode de distribution utilisée, et tout particulièrement le principe d'une distribution

intelligente, vise également à obtenir des gains en terme de taille de l'espace de recherche visité.

Gain issu du parallélisme et de la distribution horizontale

La distribution horizontale des données (distribution par instances) amène à considérer localement des fragments de taille $\frac{n}{p}$, pour p fragments identifés et en supposant une répartition équilibrée dans les fragments.

La complexité de la phase de génération des itemsets fréquents sur chaque fragment est donc de l'ordre de $O(\frac{n}{p}2^m)$, soit pour cette phase de génération sur l'ensemble des données une complexité de $\sum_{i=0}^{p} O(\frac{n}{p}2^m) = \mathbf{O(n2^m)}$.
On a donc une conservation de la complexité de la version classique. Le gain réside ici dans l'utilisation du parallélisme.

Gain issu de la distribution intelligente

La distribution intelligente des données (basée sur un clustering des instances) amène à considérer localement des fragments comportant potentiellement les m items présents dans la base.

Cependant, le critère d'intelligence utilisé (similarité en terme d'items contenus), permet de considérer que les instances d'un même fragment contiennent les mêmes items et que les instances de deux fragments contiennent des items différents.
Donc au sein de chaque fragment F_i, l'ensemble I des items est fragmenté en deux sous-ensembles $Util_{F_i}(I)$ et $Infreq_{F_i}(I)$.

Avec $Util_{F_i}(I) \cup Infreq_{F_i}(I) = I$, et $Util_{F_i}(I) \cap Infreq_{F_i}(I) = \emptyset$.

$Util_{F_i}(I)$ représentant un sous-ensemble d'items de I communs aux instances du fragment (donc utiles) donc de supports élevés et donc fréquents.

$Infreq_{F_i}(I)$ représentant un sous-ensemble d'items de I absents dans les instances du fragment donc de supports faibles et donc infréquents.

Ainsi, dès la fin de l'itération d'identification des 1-itemsets, il ne faudra considérer pour la génération de candidats que les 1-itemsets fréquents, soit ceux constitués à partir de $Util_{F_i}(I)$.

Soit $m_i = Card(Util_{F_i}(I))$, le rapport entre m_i et m dépend des données utilisées et du seuil de support considéré pour la fréquence.

La complexité de la phase de génération des itemsets fréquents sur chaque fragment est donc relative à $O(2^{m_i})$.

Soit une complexité sur chaque fragment F_i (pour phase de génération des itemsets fréquents) de l'ordre de $O(\frac{n}{p}2^{m_i})$.

Avec $O(\frac{n}{p}2^{m_i}) < O(\frac{n}{p}2^{m})$ puisque $m_i < m$.

De plus, si les supports des 1-itemsets constitués à partir de $Util(I)$ sont trés élevés, on peut inclure dans la méthode de génération des itemsets fréquents une notion de trivialité pour certains items (des items présents pour toutes ou presque toutes les instances du fragment).
Distinguons donc $Util_{F_i}(I)$ et $Triv_{F_i}(I)$ l'ensemble des items triviaux sur le fragment F_i.

On a : $Util_{F_i}(I) \cup Triv_{F_i}(I) \cup Infreq_{F_i}(I) = I$, et $Util_{F_i}(I) \cap Triv_{F_i}(I) = \emptyset$.

Ainsi, les items considérés triviaux (et identifiés dès la fin de l'itération d'identification des 1-itemsets) peuvent être écartés de la génération et ajoutés à tous les itemsets fréquents identifés sur $Triv_{F_i}(I)$, en fin de traitement, diminuant encore la taille de l'espace de recherche effectivement visité.

La complexité de la phase de génération des itemsets fréquents sur chaque fragment est donc relative à $O(2^{f_i})$.
Soit une complexité de traitement sur chaque fragment F_i (pour phase de génération des itemsets fréquents) de l'ordre de $O(\frac{n}{p}2^{f_i})$.

Avec $O(\frac{n}{p}2^{f_i}) < O(\frac{n}{p}2^{m_i})$ puisque $f_i < m_i$, et donc $O(\frac{n}{p}2^{f_i}) < O(\frac{n}{p}2^{m})$ puisque $f_i < m_i < m$.

Globalement, on se ramène donc à une complexité de traitement de l'ordre de $O(n \times 2^{f_i})$ au lieu d'une complexité d'ordre $O(n \times 2^{m})$ dans la version de base, soit un gain sur le facteur exponentiel de complexité.
Remarque : A noter qu'il faut ajouter à cette complexité de traitement le surcoût engendré par la distribution intelligente.

La distribution intelligente permet ainsi d'obtenir une diminution de la complexité de traitement (et donc du temps d'exécution) sur chacun des fragments à traiter (avec un minimum de communications pour assurer la cohérence). Cependant, il ne faudra pas oublier de s'intéresser aux itemsets présents dans plusieurs profils d'instances (information importante ou non discriminante pour la distribution, voir Section 4.2).

2.3 Exploitation d'une distribution intelligente

Nous présentons ici, les premiers résultats obtenus et leur incidence sur la suite des travaux effectués, à savoir la recherche d'une méthode distribuée de clustering.

2.3.1 Résultats d'expérimentations préalables du Schéma Général de recherche de règles d'association sur la base d'une distribution par clustering centralisé

De premiers travaux exploratoires réalisés (voir [27], [28]) sur la base d'une distribution par clustering centralisé, avaient permis de valider la théorie de diminution du temps d'exécution à partir d'une distribution intelligente.

Les expériences ont été menées sur des données réelles issues de la base Diabcare (182 attributs continus avant phase de discrétisation et 30 000 enregistrements).

Les programmes d'expérimentation ont été réalisés sur base de Java RMI, en environnement linux hétérogène, avec implémentation des mécanismes de pipeline pour les différentes phases du schéma.

Première expérience

Le but de cette expérience est de juger de la pertinence des fragments obtenus par la phase de fragmentation par distribution intelligente, pertinence en terme d'items contenus dans les fragments.
La recherche d'itemsets fréquents s'effectue sur base de traitement par l'algorithme Apriori de manière indépendante sur chaque fragment.
Dans cette expérience, la génération des itemsets fréquents a été arrêtée dès la fin de la première itération (1-itemsets fréquents) du fait de la complexité de la méthode et pour permettre une comparaison à une exécution séquentielle classique sur l'ensemble de la base.

La méthode de fragmentation utilisée correspond à une partition des instances en fonction de la distance des instances à un modèle (voir Annexe A ceci afin de limiter la taille des données à traiter de manière centralisée).
Des intervalles de valeurs pour la distance sont fixés et les instances attribuées à tel ou tel intervalle en fonction de leur distance au modèle.
On traite ici les instances discrétisées.
Une des instances de la base est aléatoirement sélectionnée comme modèle. La distance de chaque instance à ce modèle est calculée. La pertinence de plusieurs fonctions de distance binaires ont été comparée pour le problème (binaires puisqu'on travaille sur une base discrétisée représentable sous forme de matrice binaire, chaque colonne correspondant à un item et la valeur associée pour chaque colonne à chaque instance représentant le fait que l'instance contient ou non l'item considéré).

Les fragments de données obtenus sont évalués par rapport aux critères fixés pour une bonne fragmentation :

- les instances les plus similaires se trouvent dans un même fragment ;
- les instances les moins similaires se trouvent dans des fragments distincts ;

Dans cette première évaluation, c'est la distance de Hamming qui a été utilisée (voir An-

nexe A - Section A.1.2,).

Une évaluation complète a été réalisée pour cette distance, en utilisant une instance sélectionnée aléatoirement en tant que modèle. La pertinence de la distribution des instances obtenue est calculée selon les critères mentionnés plus haut.

L'observation des différentes étapes de l'évaluation et les critères de distribution permettent de mettre en évidence certaines déficiences de la fonction de distance (la distance de Hamming), à corriger dans les travaux suivants.
Une observation des fragments obtenus (base binaire) permet de constater qu'une distribution utilisant la distance de Hamming ne fournit pas une distribution respectant les critères énoncés. La fonction de distance est utilisée afin de calculer une évaluation de ces critères de distribution (similarité/ dissimilarité). La distance de Hamming s'avère inappropriée au problème posé. Il n'est pas simple de rechercher des critères de clustering sur des données binaires.
De plus, un déséquilibre de charge trés fort est observé dans la fragmentation obtenue. Une comparaison des itemsets fréquents générés n'est pas significative puisque la fragmentation ne fournit pas les résultats attendus.

D'autres expériences ont été réalisées en utilisant d'autres fonctions de distance binaire, et en utilisant d'autre choix pour le modèles de base : modèle généré aléatoirement, modèle choisis aléatoirement parmi les instances, modèle de valeur majoritaire pour chaque item et modèle de pourcentage d'instances associées (voir Annexe A - Section A.1.3).
Un réel problème de répartition de charge apparaît du fait du choix arbitraire des intervalles. Pour 10 intervalles fixés arbitrairement, une majorité de fragments vides sont obtenus. Pour certaines fonctions de distance (tel que la distance de Jaccard par exemple), un seul fragment est obtenu, et ceci pour les différents choix de modèle. Pour la plupart des fonctions de distance, seulement quatre ou cinq fragments sont obtenus alors que dix étaient souhaités.

Seconde expérience

Toujours du fait de la complexité du traitement à effectuer sur l'ensemble des itemsets et pour comparer les résultats distribués à une solution globale séquentielle, les différentes fonctions de distance et les différents choix de modèles ont été testés sur des données synthétiques, puis sur un petit fragment de la base DiabCare (20 attributs continus, 600 instances). 23 items ont été obtenus après discrétisation et élimination globale anticipée des items infréquents.
Les itemsets fréquents peuvent ainsi être générés sur la base entière de manière séquentielle centralisée et les résultats comparés aux différentes solutions distribuées (distinctes par la fonction de distance utilisée et par le choix du modèle de référence pour les distances).

Des intervalles sont utilisés pour la fragmentation (pour répartir les instances en fonction de leur similarité/dissimilarité sur base des distances au modèle de référence), mais ces

intervalles ne peuvent être fixés arbitrairement (voir Section 2.3.1).
Les valeurs arbitraires amènent à un déséquilibre de charge trés important, en particulier on se ramène souvent à un seul fragment ou à un faible nombre de fragments. Les valeurs d'intervalle doivent être adaptées à la base de données traitée en utilisant les informations de discrétisation. Pour contourner ce problème, et permettre malgré tout de justifier de l'intérêt d'une distribution basée sur les critères de similarité/dissimilarité, une méthode de synchronisation a été utilisée pendant la phase de fragmentation.
Un algorithme de clustering (voir Partie II) a été appliqué aux distances calculées (distances des instances au modèle de référence).
Des groupes (ou clusters) de valeurs ont été obtenus par ce clustering et les données fragmentées selon la répartition obtenue dans ces clusters (voir Annexe A pour les détails de répartition des instances).

Evaluation de la distribution : Plusieurs critères sont nécessaires pour évaluer la pertinence de la fragmentation obtenue pour le problème de génération des itemsets fréquents. Nous utilisons les trois critères suivants :

1. Le taux de conservation des itemsets fréquents (par rapport à une version centralisée séquentielle) doit être maximisé (le taux de perte doit être minimisé, la perte d'itemsets fréquents consiste à avoir un itemset globalement fréquent et qui ne soit pas identifié fréquent dans le traitement distribué.
2. Le taux d'itemsets fréquents "false positive" doit être minimisé (un itemset fréquent "false positive" est un itemset identifié fréquent dans le traitement distribué et qui ne l'est pas globalement).
3. La complexité de traitement de la solution distribuée doit être inférieure ou égale à la complexité du traitement centralisé séquentiel.

Résultats : Pour les fonctions de distance fournissant une discrimination suffisante des instances (un nombre suffisant de clusters et un nombre suffisant d'instances dans chaque cluster), un taux de conservation des itemsets fréquents de près de 100% est obtenu , et un taux de "false positive" de 15 à 20 % selon les fragments. La somme des complexités de traitement sur chaque fragment est inférieure à la compexité globale grâce à la possibilité d'éliminer dès la fin de la première itération les items localement infréquents sur chaque fragment (principe de l'algorithme APriori voir Section 1.5.1).
De plus un certain nombre d'items s'avèrent "triviaux" sur un fragment (items communs à toutes les instances du fragment) et pourraient être écartés de la génération itérative des itemsets fréquents pour être réinsérés par la suite (voir ci-dessous).

Items triviaux : La fragmentation étant basée sur les similarité/dissimilarité d'items au sein des fragments, le premier avantage de la méthode proposée réside dans la proportion d'items absents ou faiblement présents dans un fragment, et donc dans la possibilité d'écarter ces items infréquents dès le début de la génération des itemsets fréquents (écartant dès lors tous leurs sur-ensembles).
Un deuxième avantage découle de cette fragmentation, c'est la forte fréquence de certains items dans les fragments.
Le cas idéal est bien entendu de considérer que certains items sont présents pour toutes

les instances d'un fragment. Dans ce cas, ils peuvent être écartés des traitements et réintroduits dans les résultats à la fin du traitement. En effet si leur support est de 100%, le fait de les réinsérer dans les itemsets fréquents identifiés ne fausse pas les mesures de support effectuées.
On pourrait également admettre de fixer un seuil de trivialité trés haut (mais pas obligatoirement 100%) et d'écarter tous les items dont le support est supérieur au seuil. Ceci permet de réduire d'autant plus la complexité de traitement sur chaque fragment, celle-ci étant liée au nombre d'items à prendre en compte.
Voir Annexe A Section A.2 pour les détails de répartition des items triviaux entre les sites.

Ainsi, on traite moins d'items sur chaque fragment que dans la version centralisée séquentielle (qui doit prendre en compte tous les items), et donc la somme des complexités de traitement sur chaque fragment est inférieure à la complexité de traitement globale.

Problèmes résiduels : Certains problèmes subsistent dans cette expérience, même si elle permet de justifier du bien fondé de la fragmentation intelligente pour le problème posé.

1. Des problèmes liés à la répartition non uniforme des données : certaines fonctions de distance utilisées ne permettent pas de fragmenter la base de données telles que Jaccard, Salton, Sommation (voir Annexe A), elles peuvent donc être d'office écartées.
2. Des problèmes liés à la méthode d'évaluation : les itemsets fréquents générés sont comparés entre une version globale centralisée et un traitement distribué. Les itemsets sont considérés générés dans la solution distribuée, s'ils sont générés sur au moins un des sites de traitement, sans porter attention au support obtenu. Le support utilisé sur les fragments est relatif à la taille de chaque fragment. Un problème existe particulièrement sur les fragments de petite taille. Une bonne évaluation ne peut pas être réalisée sans une méthode corrective ou une phase de validation a posteriori. Cette méthode de validation nécessite l'accès aux données pour obtenir des mesures de supports correctes et éventuellement écarter les éléments ne satisfaisant pas au seuil de support fixé. Cette validation doit permettre de supprimer les "false positive" obtenus dans cette expérience.

Des propositions de résolutions de ces problèmes sont présentés dans les deux sections suivantes.

2.3.2 Utilisation du clustering pour la distribution intelligente

L'intérêt de l'utilisation d'un clustering pour effectuer la distribution intelligente ayant été validé par les premiers travaux (voir Section 2.3.1), il est nécessaire de s'assurer que la méthode de clustering utilisée pour la distribution correspond aux critères d'exécution distribuée définis.

Nous présentons dans la Section 3.2), un algorithme de clustering respectant les contraintes de l'infrastructure distribuée visée pour l'exécution. Cet algorithme de Clustering Progressif appelé **CDP** (voir Section 3.2) doit remplacer le clustering centralisé utilisé dans les travaux cités ci-dessus, en tant que phase de distribution dans le problème de génération des règles d'association.

2.3.3 Validation a posteriori ou collaboration dans les traitements

Une phase de validation corrective s'avérant nécessaire sur base des observations effectuées dans les expériences présentées ci-dessus, celle-ci peut être réalisée a posteriori, sur les résultats de traitements indépendants. Ceci nécessite de pouvoir accéder à l'ensemble des données, afin de recalculer les supports et s'avère problématique dans le contexte de grille (support visé pour l'exécution).
Cependant, on peut également imaginer d'effectuer cette validation durant le traitement, sur base d'une collaboration entre les traitements. Cette collaboration doit respecter les critères d'exécution sur une plateforme de type grille, à savoir pas de synchronisation et un nombre limité de communications. Ce principe de collaboration a été développé dans l'algorithme DICCoop (DIC coopératif) présenté en Section 4.2.

2.3.4 Spécificités possibles de la base

Les bases de données à traiter peuvent être de différents types :

- bases de données formatées pour le problème de recherche de règles de type D (voir Section 1.4.1) ;
- bases à données réelles et continues de type B (voir Section 1.4.1) ;
- bases spécialisées de données images, textes ...

Rappel : Pour la résolution du problème de règles d'association, on distingue deux types de bases de données B et D.
Une base réelle B est composée d'un ensemble d'instances (ou enregistrements) et d'attributs continus. Chaque instance de B associe une valeur à chacun des attributs continus (et est donc un ensemble de valeurs continues).
Une base discrétisée D composée d'un ensemble d'instances et d'un ensemble d'items (attributs discrets). Chaque instance de D est un sous-ensemble d'items appelé itemset.

La base de données initialement visée par l'application étant constituée de données médicales (symptômes et conséquences du diabète), on se situe au départ dans un contexte de base de type B. Un formatage des données est nécessaire. En particulier les attributs continus de la base B sont de tout type, avec une prépondérance d'attributs à valeurs réelles. La base B étant totalement brute, une premier travail consiste à la discrétiser en identifiant des classes de valeurs qui seront ensuite considérées comme les items d'une

base discrétisée D pour la génération des règles d'association.
La phase de pré-traitement consiste en une discrétisation de chaque attribut continu de B , de manière à travailler sur des attributs nominaux. Cette discrétisation a pour but de classifier les valeurs sur chaque attribut continu et ainsi permettre de générer des règles d'association intéressantes (pour lesquels les items n'ont pas une granularité trop fine).
La discrétisation de chaque attribut permettra d'obtenir un certain nombre d'items (attributs nominaux ou discrets) sur lesquels se basera la génération de règles d'association (voir Section 2.4 phase de pré-traitement). Cette discrétisation vise à identifier sur chaque attribut des groupes de valeurs à considérer ensemble.

Il existe 3 types d'attributs de la base B à traiter :

1. des attributs nominaux (ou catégoriques), i.e. qui prennent des valeurs dans un sous-ensemble.
   exemple : un attribut *couleur* qui prend ses valeurs dans {bleu, rouge, vert}.
   couleur est l'attribut dans la base B. Dans la base D, on considérera 3 items associés à cet attribut : *(couleur=bleu)*, *(couleur=rouge)* et *(couleur=vert)*.
2. des attributs numériques discrets, i.e. ayant **peu** de valeurs possibles.
   exemple : un attribut *nbVoisins* qui prend ses valeurs dans {1, 2, 3, 4}.
   nbVoisins est l'attribut dans la base B. Dans la base D, on considérera 4 items associés à cet attribut : *(nbVoisins=1)*, *(nbVoisins=2)*, *(nbVoisins=3)* et *(nbVoisins=4)*.
3. des attributs numériques continus, i.e. défini sur un intervalle de valeurs large.
   exemple : un attribut *age* qui prend ses valeurs dans l'intervalle $[0, 120]$.
   age est l'attribut dans la base B. Dans la base D, on considérera des items associés à cet attribut, comme par exemple : *age* $\in [20, 40]$*)*

Dans les deux premiers cas, le passage d'une base B à une base D est automatique en utilisant des associations attribut-item telles que celles présentées dans les exemples.
Dans le troisième cas (attributs numériques), le choix des intervalles de valeurs à associer aux items pose problème. Ce choix peut être réalisé de plusieurs manières :

- choix des intervalles par connaissances antérieures sur les données ;
- choix des intervalles par découpage en k intervalles de largeur uniforme ;
- choix des intervalles par découpage en k intervalles de fréquence uniforme ;
- génération des intervalles par fouille de données.

Le choix d'intervalles basé sur des connaissances antérieures est limité, le but de la méthode étant d'extraire des informations sur les données, une connaissance a priori n'existe pas dans la majorité des cas. Le résultat de la discrétisation réalisée constituera d'ailleurs une information importante indépendamment du contexte de génération de règles d'association.

Le choix d'intervalles de largeur uniforme pose le problème de l'arbitraire.

Exemple :

Soit une répartition des valeurs telle que celle présentée par la figure 2.2.
On identifie visuellement 4 groupes de valeurs liés aux intervalles $[0, 15[$, $[20, 30[$, $[40, 55[$,

$[90, 100[$. Un choix arbitraire de 4 intervalles aurait fourni les intervalles $[0, 25[$, $[25, 50[$, $[50, 75[$, $[75, 100[$.
De même, un choix des intervalles basé sur une fréquence uniforme aurait fourni des intervalles faussés de type $[0, 25[$, $[25, 50[$, $[50, 75[$, $[75, 100[$.

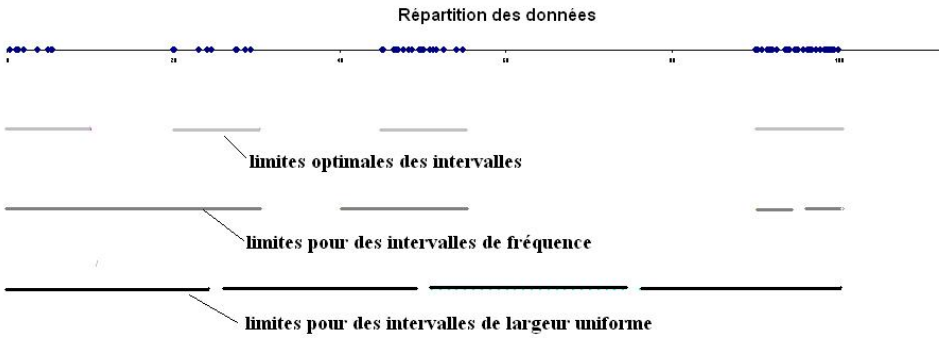


FIG. 2.2 – Exemple de répartition de données à discrétisées

2.3.5 Phase de pré-traitement

Il apparaît que la discrétisation doit être réalisée par analyse des données et tout particulièrement qu'elle peut être obtenue par clustering.

Ainsi, la phase de pré-traitement des données consiste en un traitement par clustering de chacun des attributs indépendamment.
L'indépendance entre les traitements des attributs permet de réaliser ces traitements de manière parallèle sur une distribution verticale de la base B.
Pour rappel, une distribution verticale ou distribution hétérogène d'une base consiste à distribuer les attributs (continus) de manière à ce que toutes les informations relatives à un attribut soient accessibles sur un même fragment de données donc localement.
Les clusters résultants de cette discrétisation de la base constituent un premier savoir extrait des données par technique de fouille.

2.3.6 Sécurité

- Confidentialité des données :
  Les communications sont utilisées dans les méthodes de résolution distribuée (toutefois limitées et ne menant pas à des synchronisations) et il convient de s'intéresser à la confidentilaité des données utilisées.
  Le but étant d'exploiter des données pouvant revêtir un caractère confidentiel (données médicales, bancaires, ...), il est nécessaire de s'assurer que de telles données ne transiteront pas en clair sur le réseau.
  Du fait du type de problème envisagé (traitements de Data Mining), un formatage des données est nécessaire (nettoyage, discrétisation des attributs en items...). Le

format interne à l'application résultant de la phase de pré-traitement permettra de faire transiter des données non "compréhensibles" (codées et partielles), assurant ainsi leur confidentialité.

- Tolérance aux pannes :
  Un des problèmes des infrastructures réseau de type grille réside dans le problème de tolérance aux pannes. Ce problème n'a pas été étudié dans ces travaux car il est plutôt du ressort d'un middleware visant à optimiser l'exécution sur la plateforme.

2.3.7 Limitation de la granularité du parallélisme

La distribution intelligente est effectuée par clustering et doit permettre l'identification de profils d'instances.
De la granularité de cette distribution dépend la granularité du traitement parallèle distribué des fragments qui suivra.

La limitation à apporter sur le nombre de groupes résultants du clustering de distribution est liée au nombre de machines disponibles pour le pré-traitement, mais doit également tenir compte du fait que l'on ne doit pas avoir une granularité trop fine dans le contexte de la recherche de règles.

Le nombre de fragments (**nbFrag**) à générer lors de la phase de distribution doit répondre à plusieurs critères :

- ce nombre doit être suffisant pour permettre la distribution des données (instances) de manière à ce que les données soient à la fois stockables et traitables localement (sur les noeuds de traitement)
  Ainsi on doit avoir : $nbFrag > seuilStockage$.

- ce nombre doit être limité pour avoir une cohérence de la distribution intelligente. Si les profils sont trop spécialisés (trop nombreux et de trop petite taille), on aura certe un gain de complexité de traitement local (nombre d'items à considérer et donc d'itemsets générables), mais on aura également besoin de compenser ce gain par des communications (la probabilité qu'un itemset fréquent soit présent sur plus d'un noeud augmentant)
  Ainsi on doit avoir : $nbFrag < seuilCoherence$.

La granularité en $nbFrag$ fragments doit donc être :

$$seuilStockage < nbFrag < seuilCoherence >$$

- Dans le cas où le nombre de fragments obtenus est plus grand que le nombre de noeuds ou ressources disponibles ($nbFrag > nbRessDispo$) on devra alors stocker et traiter plusieurs fragments sur un même noeud.

2.4 Schéma général de recherche de règles d'association

Le schéma général proposé pour la résolution du problème des règles d'association en environnement distribué peut être décomposé en plusieurs phases correspondant aux deux étapes majeures : préparation, traitement (voir Figure 2.3) :

- la **phase de préparation** des données (sur base de clustering)
  - Pré-traitement : nettoyage, filtrage, discrétisation, formatage des données pour le problème
  - Distribution : distribution intelligente en fragments par recherche de profils d'instances
- la **phase de traitement** distribué des fragments obtenus
  - Extraction des itemsets fréquents : exécution séquentielle sur chaque fragment avec éventuellement des collaborations
  - Validation des résultats et génération de règles.

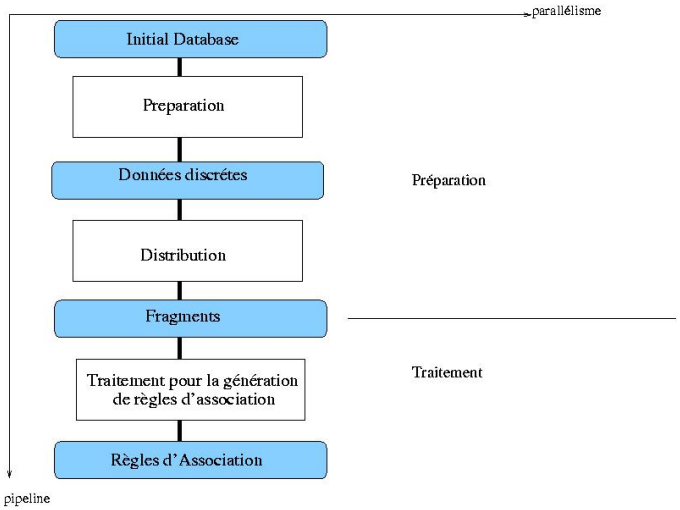


FIG. 2.3 – Schéma Général DisDaMin pour la recherche de règles d'association

Chacune des étapes du schéma doit être considérée dans un environnement parallèle distribué et le schéma complet doit donc utiliser les méthodes de parallélisations et de pipeline plutôt que des traitements centralisés.

Ainsi, partant d'une base initiale B, il convient tout d'abord de choisir une distribution $d1$ de cette base (verticale ou horizontale) et d'appliquer la phase de pré-traitement/discrétisation de manière distribuée. La manière dont est réalisée la discrétisation dépend de la distribution $d1$ effectuée (voir Figure 2.4).
On obtient dès lors une base de données discrétisée distribuée sur la plateforme en fonction de la distribution $d1$ précédemment choisie.
La base discrétisée D obtenue peut une nouvelle fois être distribuée selon une distribution $d2$ (toujours verticale ou horizontale), mais on peut également conserver la distribution existante à la fin de l'étape de discrétisation ($d2 = d1$).
La phase de fragmentation intelligente est exécutée sur la base de la distribution $d2$ (voir Figure 2.4).

Un modèle de fragmentation intelligente de la base est obtenu, et permet d'effectuer une redistribution intelligente $d3$ de la base ($d3$ horizontale ou verticale).
Après cette redistribution, la base de donnée discrétisée D est donc répartie en fragments intelligents sur lesquels on effectue le traitement effectif de génération des règles d'association (voir Figure 2.4).
Les résultats (règles d'association) sont répartis en fonction de la fragmentation $d3$ réalisée avant le traitement.

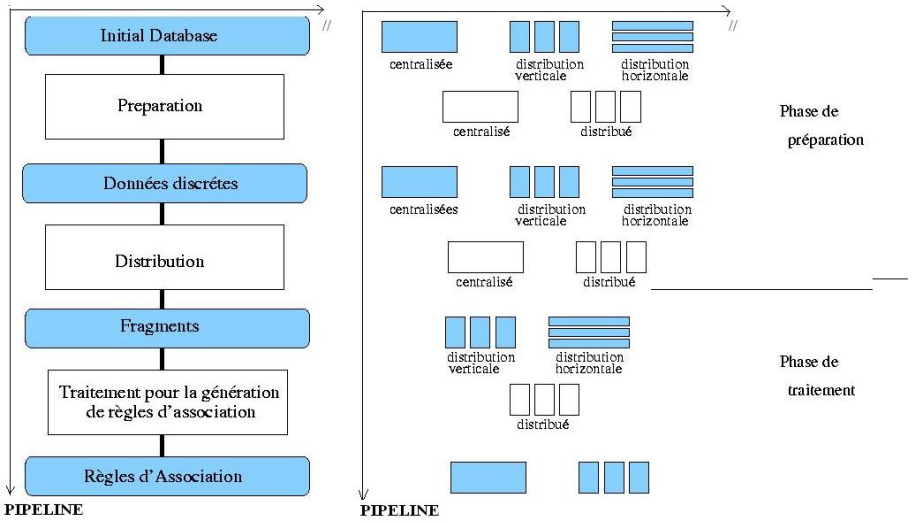


FIG. 2.4 – Schéma Général DisDaMin pour la recherche de règles d'association : possibilités de distribution

Chacunes des étapes est détaillée dans les Sections suivantes.

2.4.1 Phase de préparation

Partant d'une base B composée de m attributs à valeurs continues.
L'étape de discrétisation des données consiste à transformer la base B continue en une base D discrète (voir Figure 2.5), en transformant indépendamment chacun des attributs continus de B en un ou plusieurs items (attributs discrets) (voir Figure 2.6).

Soit un attribut a et le vecteur de valeurs associées à cette attribut : $\vec{v_a}$, (à chaque instance i de B est associée une valeur pour l'attribut a dans le vecteur).

Pour le problème de recherche de règles d'association, on ne peut pas considérer chaque attribut comme un item, le nombre de combinaisons possibles serait très important avec une probabilité de fréquence (la fréquence est la mesure de base pour la résolution du problème) de chacune des valeurs (et donc de l'item associé) trés faible.
On effectue donc une discrétisation des données de manière à identifier des classes de valeurs, afin d'obtenir ces attributs de type nominal.
Cette identification de classes correspond à un traitement par clustering (classification non

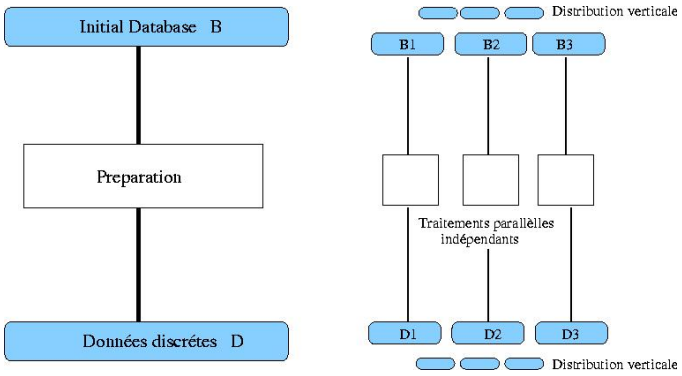


FIG. 2.5 – Phase de discrétisation : distribution

supervisée voir Partie II et Annexe B Section B.3).

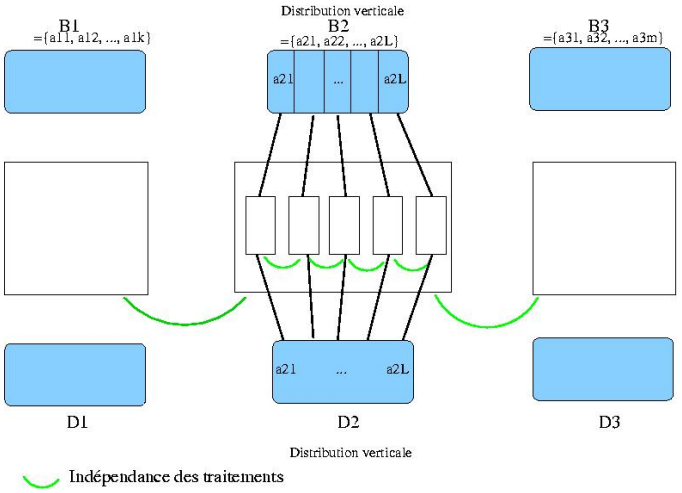


FIG. 2.6 – Phase de discrétisation : exécution

La discrétisation fournit un ensemble de classes $\{it_{a_i}\}$ associé à l'attribut continu a. Chacune des classes devient un attribut discret dans la base de données discrétisée D. Ces attributs discrets représentent les items pour le problème des règles d'association.

Après la phase de discrétisation, le terme attribut désignera non plus un attribut à valeurs continues issu de la base de données brute à traiter B, mais un attribut discret ou item à considérer dans la base de données discrétisée D.

Exemple :

$\vec{v_a} = \{0,\ 3,\ 7,\ 8,\ 4,\ 1\}$.

Si l'on identifie 3 items it_{a_1}, it_{a_2} et it_{a_3}, respectivement associés aux ensembles de valeurs continues {0, 1}, {3, 4} et {7, 8}.

On peut associer à $\vec{v_a}$ le vecteur discret $\vec{it_a}$ ou encore la matrice binaire b_a.

$\vec{it_a} = \{it_{a_1},\ it_{a_2},\ it_{a_3},\ it_{a_3},\ it_{a_2},\ it_{a_1}\}$

$$b_a = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

La discrétisation pour une base $B = \{\vec{v_{a_1}}, \vec{v_{a_2}}, \ ..., \vec{v_{a_m}}\}$ fournit une base

$$D = \{\vec{it_{a_1}}, \vec{it_{a_2}}, \ ..., \vec{it_{a_m}}\}$$

à laquelle on peut associer une représentation matricielle

$$D_M = \{b_{a_1}, b_{a_2}, \ ..., b_{a_m}\}.$$

Remarque : Dans la matrice binaire associée à la discrétisation de chaque attribut, un seul attribut discret issu d'un attribut a peut être contenu pour chaque ligne. Donc pour chaque ligne de la matrice binaire b_a, une seule valeur peut prendre pour valeur 1 (dans l'exemple, on aura soit la valeur pour la colonne associée à it_{a_1}, soit celle associée à it_{a_2}, soit celle associée à it_{a_3}).
Pour chaque ligne, on a une exclusion mutuelle des colonnes de la matrice b_a.
On a donc au maximum m bits de valeur 1 sur chaque ligne de la base D_M (représentation matricielle).

2.4.2 Phase de fragmentation

Partant d'une distribution verticale de la base D (distribution par items issue de la phase de discrétisation) en plusieurs sous-bases D_i, (voir D_1, D_2 et D_3 sur la Figure 2.7), nous souhaitons obtenir une distribution horizontale de la base D (distribution par instances) en fragments F_j, (voir F_1, F_2 et F_3 sur la Figure 2.7).

La base entière discrétisée D peut être représentée comme une matrice n, m (n lignes correspondant aux n instances et m colonnes correspondant aux m items). Cette matrice peut être obtenue par concaténation verticale des matrices associées aux sous-bases D_j.

La phase de fragmentation permet (sur la base de traitements collaboratifs) d'obtenir une partition P ($P = \{P_j\}$) des instances et donc les fragments F_j relatifs à cette partition (F_j étant associé à P_j).

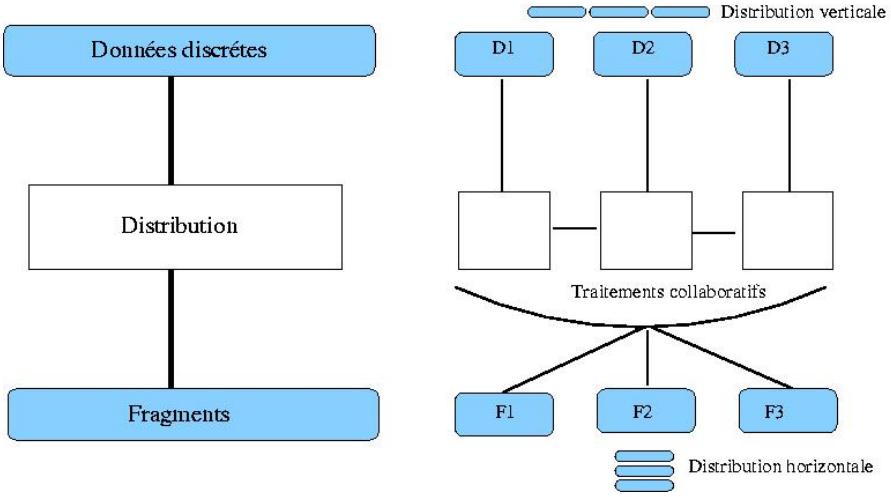


FIG. 2.7 – Phase de fragmentation : distribution

Exemple : Soit D_1 associée à une matrice n,k (n instances, items 1 à k de D), D_2 associée à une matrice n,l-k (n instances, items k+1 à l de D), et D_3 associée à une matrice n,m-l (n instances, items l+1 à m de D).

La partition P fournit une répartition des instances en 3 fragments F_1, F_2 et F_3 (à une permutation près des instances) de tailles respectives u, v et w, tels que F_1 est associé à une matrice u,m (u instances, m items), F_2 à une matrice v,m (v instances, m items) et F_3 à une matrice w,m (w instances, m items).

Soit une instance j (représentée en vert sur la Figure 2.8). Initialement les données relatives à l'instance j sont distribuées dans les sous-bases D_i (instances partielles I_{j_1}, I_{j_2} et I_{j_3} sur la Figure 2.8).
L'instance I_j est affectée au fragment F_3 par la partition P, et est obtenue par concaténation des instances partielles I_{j_1}, I_{j_2} et I_{j_3}.
Ainsi $I_j = I_{j_{11}} \dots I_{j_{1k}} I_{j_{2k+1}} \dots I_{j_{2l}} I_{j_{3l+1}} \dots I_{j_{3m}}$.

2.4.3 Phase de traitement

Sur base des fragments F_j obtenus lors de la phase de fragmentation, la phase de traitement consiste à identifier les itemsets fréquents sur chacun des fragments F_j par des traitement locaux (voir Figure 2.9).

Le but de la méthode étant de générer les itemsets fréquents globaux (et donc les règles d'association globales sur toute la base), il convient d'utiliser ces résultats locaux pour obtenir les itemsets fréquents globaux. L'utilisation des résultats locaux peut être effectuée durant le traitement par collaboration, mais pourrait également être réalisée a posteriori (nécessitant alors une phase de validation) (voir Figure 2.9 et Section 4.2).
Ces résultats (itemssets fréquents) locaux obtenus sont intéressants en tant que tels (voir

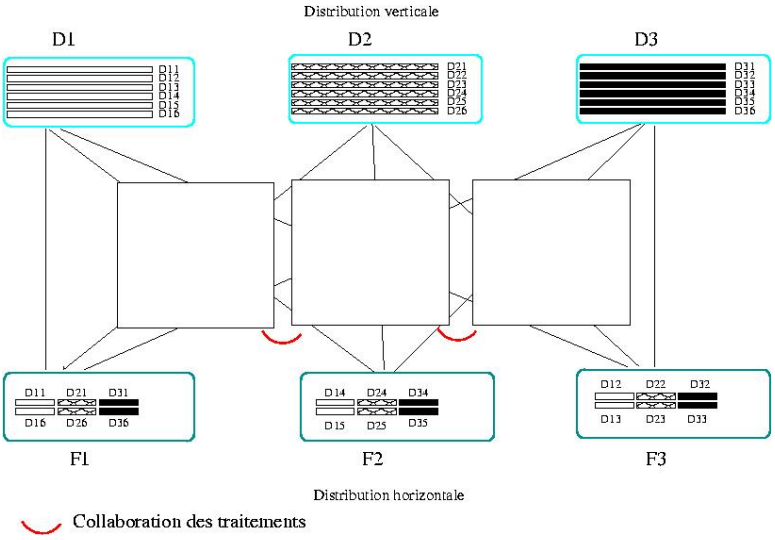


FIG. 2.8 – Phase de fragmentation : exécution

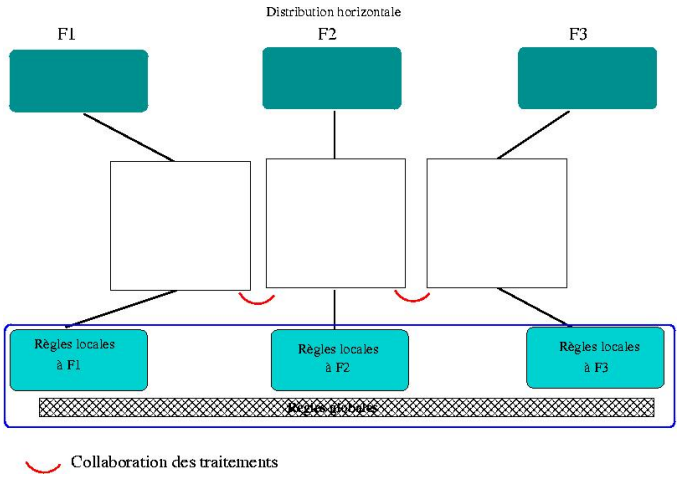


FIG. 2.9 – Phase de traitement

Section 5.2).

2.5 Conclusions

Dans ce chapitre, nous avons proposé le schéma général du projet DisDaMin pour la résolution du problème de recherche de règles d'association.
Ce schéma se base sur un principe de distribution intelligente pour diminuer la taille de l'espace de recherche visité pour la résolution du problème. et en permettre un traitement distribué. La validité de ce principe ayant été justifiée par les résultats de tests préalables, nous allons maintenant nous intéresser aux différentes étapes de ce schéma.
Celui-ci utilisant principalement deux types de traitement de data mining (le clustering pour la fragmentation intelligente, puis la recherche de règles d'association en elle-même), nous allons maintenant détailler ces deux problèmes, et plus particulièrement nous présenterons les solutions du projet DisDaMin pour les résoudre.

Ainsi, nous détaillerons tout d'abord le **clustering** (voir Partie II), utilisé pour les phases de discrétisation et de fragmentation, et proposons l'algorithme CDP (Clustering Distribué Progressif, voir Section 3.2).

Puis nous détaillerons le problème de **génération d'itemsets fréquents** et la génération de règles d'association (voir Partie III), qui correspond au traitement effectif, et présenterons une adaptation distribuée de l'algorithme DIC, l'algorithme DICCoop (DIC Coopératif, voir Section 4.2).

Nous effectuerons ensuite un bilan de la méthode globale proposée (voir Chapitre IV).

Deuxième partie

Clustering Distribué

Clustering Distribué dans DisDaMin

Le principe de distribution proposé dans le schéma général de recherche de règles d'association (Section 2.1.2) est basé sur des critères de clustering.
On attend tout d'abord de cette distribution "intelligente" des données la possibilité de distribuer le traitement (la recherche de règles d'association) de manière efficace, c'est-à-dire permettant une exécution optimale sur grille et une diminution de la taille de l'espace de recherche visité (complexité du traitement - voir Complexité - Section 2.2).

Les gains attendus inhérents à cette distribution ont été effectivement obtenus lors des premiers tests de validation du schéma (voir Sction 2.3.1 et Annexe A). Un deuxième avantage induit par cette distribution est la fragmentation elle-même, puisqu'elle constitue une information précieuse de description des données. Les fragments identifiés par clustering correspondent à des profils d'instances (profil de clients avec des comportements spécifiques, profil de patients avec pathologies communes, ..., selon le domaine d'application) pour lesquels des caractéristiques communes (autres que les règles d'association recherchées) pourraient être exploitées, voir déduites directement de nos traitements.
Ces groupes d'instances similaires constituent une connaissance à part entière et les traitements effectués sur ces différents groupes d'instances (Pré-traitements des données voir Sections 2.3.4 et 2.4.1, et traitements locaux lors de la recherche de règles d'association voir Section 4.2) peuvent amener à posséder au final beaucoup plus d'informations que les règles recherchées (voir Section5.2).

Ceci étant, l'étape d'identification des profils ne doit pas être une barrière dans le schéma général et ne peut donc pas être réalisée de manière centralisée.
De même, le schéma général devant s'exécuter en contexte distribué de type grille de calculs, la phase de fragmentation doit prendre en compte les spécificités de la plateforme de déploiement visée (voir Section 1.3.3).

Nous présentons ici le principe de clustering (ou classification non supervisée) ainsi que les différentes catégories d'algorithme de résolution de ce problème.
Deux de ces catégories (KMeans et le clustering agglomératif) sont détaillées ainsi que leurs inadéquations au déploiement sur grille, avant de présenter notre algorithme distribué de clustering : **CDP** (Clustering Distribué Progressif - voir Section 3.2).
Nous détaillerons ensuite les possibilités de déploiements et d'asynchronisme sur grille de calcul (voir Section 3.5), avant de présenter les résultats de l'évaluation de l'algorithme (voir Section 3.6).

3.1 Le Clustering

Le Clustering consiste en l'élaboration d'un partitionnement des données (en groupes ou clusters) par un apprentissage non supervisé, c'est-à-dire sans disposer de connaissances a priori sur les groupes existants (y compris leur nombre).

3.1.1 Le principe du clustering

Le clustering est considéré comme une tâche non supervisée, puisque l'on ne connaît pas les classes au préalable. Ces classes (associées aux clusters) doivent être identifiées et définies par analyse des données.
Le principe consiste, à partir d'un ensemble de données D, à trouver une fragmentation en clusters C, de manière à optimiser deux critères :

- diminuer la distance entre les données d'un même cluster ;
- augmenter la distance entre données de clusters différents.

Le principe nécessite donc une mesure de distance, fonction du type de données traitées. Dans certains cas on pourra envisager d'utiliser une mesure de similarité (respectant moins de propriétés qu'une distance et donc apportant plus de souplesse).

3.1.2 Les algorithmes existants

Il existe différentes méthodes de clustering. Une première séparation qui peut être effectuée consiste à considérer deux types de clustering, les méthodes hiérarchiques et les méthodes de partitionnement :

- les méthodes hiérarchiques :
  - agglomératives : qui considèrent initialement une partition constituée par des clusters regroupant au départ une seule instance de données et qui regroupent ensuite les clusters "voisins" jusqu'à un critère d'arrêt (**Single-Link**, **Complete-Link**, **Average-Link** 05 [16], **Agnes** 90 [42], **Chameleon** 99 [41], **Birch** 96 [66], **Cure** 98 [34]).
  - divisantes : qui considèrent initialement une partition constitué d'un seul cluster contenant l'ensemble des instances de données et découpent ensuite les clusters de manière itérative jusqu'à un critère d'arrêt (**DIANA** [42]).
- les méthodes de partitionnement :
  - basées sur la distance (**KMeans** 65-67 [45], [26], KMedoids, **KModes** 98 [36], **DBSCAN** 95 [19] et [20], **CLARANS** 94-02 [46] et [47]).
  - basées sur la densité (**Expectation Maximisation** EM 95 [11]) ou les probabilités (**CobWeb** 87 [30]).

Dans la réalité d'autres critères permettent de distinguer les différentes méthodes (voir [8], [40]) :

- les méthodes jouant sur le nombre d'attributs considérés en même temps : la plupart des algorithmes basent leurs décisions sur la totalité des attributs dans les données.

- les méthodes basées sur le degré d'appartenance des instances aux clusters :
  - dures : une instance de données est affectée à un seul cluster (telles que les méthodes citées précédemment).
  - floues : une instance de données possède un degré d'appartenance à plusieurs clusters (Fuzzy Clustering voir [9] et [10]).
- les méthodes incrémentales ou non :
  - non incrémentales : l'ensemble des instances de données est disponible et traité en une seule fois (méthodes précédemment citées).
  - incrémentales : les instances de données sont traitées au fur et à mesure, venant enrichir le résultat (voir [21]).
- les méthodes basées sur une recherche de voisinage : k-nearest neighbours KNN [54] et [55]

Nous présentons dans la section suivante deux méthodes de clustering : l'algorithme des K-Moyennes (KMeans - voir MacQueen [45], Forgy [26]) qui est une méthode de partitionnement par degré d'appartenance, et un algorithme hiérarchique agglomératif (voir Sokal [58]).

3.1.3 Comparaison de deux types de clustering

Les deux algorithmes qui vont être présentés diffèrent par la qualité de leurs résultat mais également par la complexité des traitements permettant d'obtenir ces résultats.
L'algorithme des K-Moyennes est une heuristique qui fournit une solution approximative mais possède une complexité temporelle acceptable.
Les algorithmes agglomératifs fournissent quant à eux une bonne qualité de résultats, mais leur utilisation est limitée du fait de leur complexité temporelle.

Principe de l'algorithme des K-Moyennes

Entrée : les données, le nombre de clusters à produire (k)
Sortie : clusters de données
(1) initialiser k objets comme centres initiaux
(2) répéter
(3) (re)assigner chaque objet au cluster le plus proche
(4) mettre à jour les valeurs moyennes de clusters
(5) jusqu'au critère d'arrêt
(6) **retourner les k clusters identifiés**

TAB. 3.1 – Principe de l'algorithme des K-Moyennes

L'algorithme des K-Moyennes est un algorithme dit de partitionnement consistant en une méthode itérative dite des "centres mobiles", qui construit une partition initiale des données de cardinalité k. Une technique de ré-affectation permet d'améliorer le partitionnement en déplaçant les objets d'un groupe à un autre jusqu'à un critère terminal de stabilité

(voir Table 3.1). L'algorithme des K-Moyennes converge vers un optimum local, la qualité de cet optimum dépend du choix de k (nombre de groupes à identifier) et du choix des centres initiaux.
Certains groupes pouvent être vides à la fin des itérations, le nombre k est le nombre maximal de clusters.
De plus, la valeur optimale de k n'étant a priori pas connue au départ, des méthodes itératives permettent de trouver une valeur adaptée pour k. Ces méthodes itérent sur différentes valeurs de k jusqu'à trouver la valeur pour laquelle la qualité des groupes est maximale.

Pour ce qui est de l'initialisation des centres initiaux, les centres choisis au départ doivent fournir une bonne couverture de l'espace de données. Il est nécessaire de disposer d'une mesure de similarité pour affecter les données aux groupes, ainsi que d'une fonction de calcul de moyenne.

L'algorithme s'arrête si les clusters n'évoluent plus (ne sont pas modifiés d'une itération à la suivante), ou après un nombre maximal d'itérations. L'algorithme converge vers un optimum local (voir [12]).

Principe des algorithmes de clustering agglomératif

Entrée : les données, critère d'arrêt
Sortie : clusters de données
(1) considérer chaque donnée en tant que cluster
(2) répéter
(3) fusionner les deux clusters les plus proches
(4) jusqu'au critère d'arrêt
(5) **retourner les clusters identifiés**

TAB. 3.2 – Principe de l'algorithme de Clustering Agglomératif

Le clustering agglomératif est une méthode de classification hiérarchique qui consiste en une approche "bottom-up" du problème. Au départ, toutes les données sont considérées séparément comme des clusters, et l'on regroupe les clusters les plus proches deux à deux à chaque itération, jusqu'à ce qu'une condition d'arrêt soit atteinte (voir Table 3.2).

La condition d'arrêt peut être soit qu'un certain nombre de clusters ait été atteint, soit que la distance séparant les deux clusters les plus proches est supérieure à un seuil... La méthode utilise une matrice de mesures de distance ou de similarité entre clusters qui rend la méthode inutilisable pour de grands jeux de données.

Les différents algorithmes agglomératifs diffèrent par la façon dont sont calculées les distances entre clusters.

Les clusters sont représentés par un unique élément, généralement leurs centres (pondérés), ainsi que l'ensemble des instances associées.

Les algorithmes hiérarchiques (tel que le clustering agglomératif) sont plus coûteux en temps et en espace que les algorithmes de partitionnement, tel que KMeans (voir détails de complexité Section 3.4).

3.1.4 Les inadéquations des algorithmes existants

Les deux méthodes présentées précédemment (les K-Moyennes et le clustering agglomératif) nécessitent, pour atteindre une solution correcte, l'accès à la base de données entière entre chaque itération. En version parallèle, l'impossibilité d'accèder à la base de données entière est compensée par des communications entre chaque itération pour synchroniser les résultats locaux.

Des méthodes parallèles existent pour les K-Moyennes (voir Forman et Zhang [31]) et pour le clustering agglomératif (voir Johnson [39], Samatova and al.[56]) sur les deux types de distribution de données existantes (verticale et horizontale i. e. par attributs ou par instances de données voir Section 1.5.2). De même, des parallélisations existent pour les autres méthodes citées en Section 3.1.2 (voir [22] et [23]).

Pour qu'un clustering parallèle puisse fournir un résultat de qualité comparable à celui d'une version séquentielle, un grand nombre de communications sont nécessaires, et ce pour réunir les informations de chaque site afin de prendre des décisions sur des critères globaux communs à tout le jeu de données. De plus, entre chaque itération, une phase de synchronisation est nécessaire pour partager les résultats de l'itération précédente et permettre une exécution optimale de l'itération suivante.
Ces communications et ces synchronisations répétées ne sont pas admissibles pour une exécution distribuée sur grille (trop coûteuses et donc trop pénalisantes).

Ces méthodes sont essentiellement adaptées à des super-calculateurs utilisant une mémoire partagée ou commune et un réseau d'interconnection interne rapide(Parallel Data Miner pour IBM-SP3 par exemple). Les communications apparaissent comme des opérations de lecture/écriture dans une architecture à mémoire partagée. En l'absence de mémoire commune, ces communications deviennent de véritables communications réseau, d'où des problèmes de performance dans un contexte de grille. Les méthodes classiques nécessitent alors d'être re-visitées pour prendre en compte les contraintes d'une architecture de type grille (pas de mémoire commune, communications lentes). La méthode de **Clustering Distribué Progressif** présentée dans la section suivante est basée sur l'observation de ces contraintes.

3.2 Le Clustering Distribué Progressif

3.2.1 Introduction

La méthode du Clustering Distribué Progressif est inspirée d'un algorithme séquentiel de clustering : l'algorithme CLIQUE (voir Agrawal [5], [6]). L'algorithme CLIQUE consiste en un clustering des données par projections de celles-ci dans chaque dimension (pour chaque attribut ou colonne), et par l'identification de clusters denses dans les projections de données (régions denses de l'espace dans les dimensions). La méthode suppose que la base de données complète puisse être accédée pour les projections. Dans le contexte de traitement sur grille pour le Clustering Distribué Progressif, nous supposons que la base de données est distribuée de manière verticale (multibase).

3.2.2 Inspiration

Dans le schéma général de recherche de règles d'association, le travail initial ayant fait apparaître la nécessité d'une méthode distribuée de clustering (voir Section 2.1), un pré-traitement des données est nécessaire (voir Section 2.3.4). Ce pré-traitement consiste à discrétiser les données par un clustering effectué sur une partition de la base en fragments verticaux. Chaque fragment correspond à l'un des attributs de la base et doit être discrétisé indépendamment des autres (clustering unidimensionnel).
Nous cherchons ici une méthode de clustering basée sur tous les attributs (clustering multidimensionnel).
L'idée de l'algorithme proposé est de construire les clusters multidimensionnels à partir des clusters disponibles sur des sous-espaces (clusters unidimensionnels issus de la discrétisation). Les clusters sur les sous-espaces étant générés de manière parallèle indépendante, si la construction peut être réalisée en respectant les critères d'un traitement distribué et conduire à des résultats de qualité, la méthode résultante consistera en un algorithme distribué.
C'est ce que nous appellerons le **Clustering Distribué Progressif : CDP**.

Cette vision du problème consiste donc à considérer le problème inverse de l'algorithme CLIQUE, à savoir que nous ne disposons pas ici de la base entière mais de fragments verticaux. Reste à gérer, dans la reconstruction des clusters multidimensionnels, le problème existant dans les travaux d'Agrawal, à savoir le risque d'obtenir un cluster unique à la fin du traitement.

Dans la suite, les données à traiter étant continues, on considérera l'utilisation de la distance euclidienne : $d_e(x,y) = \sqrt{\sum_{k=1}^{n}(x_k - y_k)^2} = || \ x - y \ ||$, n étant le nombre de dimensions.
On utilisera la moyenne pondérée : $m(x_k, y_k) = \frac{p_x * x_k + p_y * y_k}{p_x + p_y}$, p_x et p_y étant respectivement le poids de la donnée x et le poids de la donnée y.

3.2.3 Principe du Clustering Distribué Progressif

Le Clustering Distribué Progressif (voir [29]) fonctionne de manière inverse par rapport à l'algorithme CLIQUE, dans une approche "bottom up", en considérant les attributs ou colonnes de la base de données. La méthode consiste à générer dans un premier temps les clusters sur des fragments verticaux contenant quelques attributs (voir Section 3.3.3) et de combiner ensuite ces clusters (sur un sous-espace) pour obtenir des clusters dans un espace de plus grande dimension (voir Sections 3.3.3 and 3.3.3). Les deux étapes (i. e. le clustering de fragments verticaux et la combinaison des clusters obtenus) s'exécutent de manière distribuée. L'étape de combinaison des résultats tire bénéfice de l'exécution distribuée (voir Section 3.5.1).

Le formalisme des différentes étapes de la méthode va être donné dans la Section suivante, un exemple est donné un peu plus loin pour aider à la compréhension.

3.3 Définitions

3.3.1 Définitions des notations utilisées

Base de données et réprésentation matricielle

Une base de données de **n** lignes (instances) et **m** attributs est représentée par $B = (A, K, V)$ où :

- A={A_1,A_2, ... A_m} est un ensemble fini d'attributs ;
- K={K_1,K_2, ... K_n} est l'ensemble des clés (identifiants d'instances) de la base de données.
- V={V_{ij} / $1 \leq i \leq n$ et $1 \leq j \leq m$},
  v_{ij} représente la valeur de l'attribut j (j_{eme} coordonnée) pour l'instance i (clé K_i).

On peut limiter la représentation de cette base de données B à la matrice V, c'est ce que nous appelerons dans la suite représentation matricielle (voir Tables 3.3 et 3.4).

attributs / clés	A_1	A_j	A_m	
K_1	v_{11}	$\ldots v_{1j} \ldots$	v_{1m}	
		$\ldots$		
		$\ldots$		
K_i	v_{i1}	$\ldots v_{ij} \ldots$	v_{im}	i_{eme} instance de B
		$\ldots$		
		$\ldots$		
K_n	v_{n1}	$\ldots v_{nj} \ldots$	v_{nm}	

TAB. 3.3 – Base de données B

Dans la suite, on utilisera plûtot la représentation matricielle pour représenter la base B.

	1	j	m
1	v_{11}	$\ldots v_{1j} \ldots$	v_{1m}
		$\ldots$	
		$\ldots$	
	v_{i1}	$\ldots v_{ij} \ldots$	v_{im}
		$\ldots$	
		$\ldots$	
n	v_{n1}	$\ldots v_{nj} \ldots$	v_{nm}

i_{eme} ligne de la matrice associée à B

TAB. 3.4 – Définition de la matrice V associée à la base de données B

Partition d'instances et partition d'attributs

On note $\mathcal{U}$ une partition des clés de la base B, telle que $\mathcal{U} = \{U_1, \ldots, U_p\}$, avec $U_i = \{K_l \in K\}$, $\bigcup_i U_i = K$ et $U_i \cap U_j = \emptyset$.

On note $\mathcal{A}$ une partition des attributs de B, telle que $\mathcal{A} = \{X_1, \ldots, X_q\}$, avec $X_j = \{A_k \in A\}$, $\bigcup_j X_j = A$ et $X_j \cap X_k = \emptyset$.

Matrice projetée associée à une partition d'attributs $\mathcal{A}$

On note P_X une projection de la base B sur un sous-ensemble d'attributs X ($X \in \mathcal{A}$).

Soit $X = \{A_k \ldots A_r\}$, la matrice associée à P_X possède n lignes et q colonnes (une ligne pour chaque instance de B et une colonne pour chaque attribut A_j de X).
La j_{eme} colonne de P_X correspond à la j_{eme} colonne de B (voir Table 3.5).

	1	j	m
1	v_{11}	$v_{1k} \ldots v_{1j} \ldots v_{1r}$	v_{1m}
		$\ldots$	
		$\ldots$	
	v_{i1}	$v_{ik} \ldots v_{ij} \ldots v_{ir}$	v_{im}
		$\ldots$	
		$\ldots$	
n	v_{n1}	$v_{nk} \ldots v_{nj} \ldots v_{nr}$	v_{nm}

i_{eme} ligne de B

base de données B

	k		r
1	v_{1k}	$\ldots$	v_{1r}
		$\ldots$	
		$\ldots$	
	v_{ik}	$\ldots$	v_{ir}
		$\ldots$	
		$\ldots$	
n	v_{nk}	$\ldots$	v_{nr}

$A_k \ldots A_r \in X$

i_{eme} ligne de P_X

projection P_X

TAB. 3.5 – Représentation de B et de P_X

Matrice contractée associée à une partition d'instances $\mathcal{U}$ et à la base B

Soit une partition d'instances $\mathcal{U}$ de la base B, on peut associer à la partition $\mathcal{U}$ et à la base B une matrice contractée R.
La matrice R possède p lignes (une ligne pour chaque U_i de $\mathcal{U}$) et m colonnes (une colonne pour chaque colonne de B) (voir Table 3.6).

	R		
	1	...	m
U_1	$R_{U_1 1}$	...	$R_{U_1 m}$
		...	
		...	
U_j	$R_{U_j 1}$	...	$R_{U_j m}$
		...	
		...	
U_p	R_{p1}	...	$R_{U_p m}$

TAB. 3.6 – Définition de matrice contractée R associée à une partition $\mathcal{U}$ et à la base B

Toute ligne de R est associée à un sous-ensemble d'instances U_i de B.

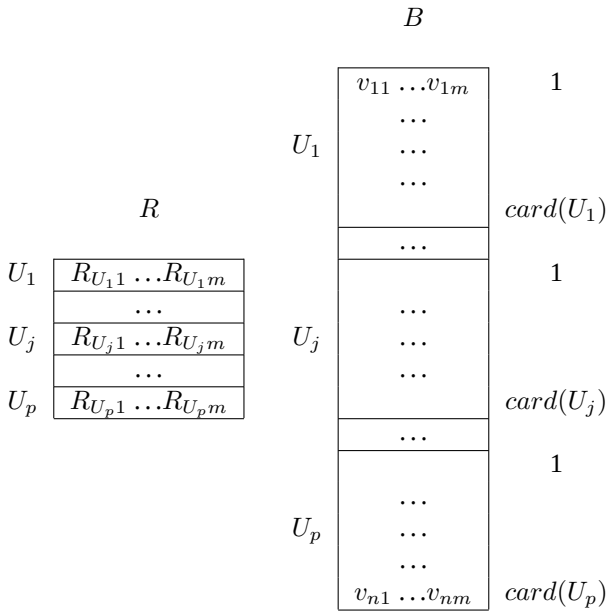
A partir de R (matrice p*m), on peut revenir à une matrice R' n*m, en dupliquant, pour chaque U_i de R, la ligne de R associée à U_i selon la cardinalité de U_i (voir Table 3.7).

	R		R'	
		U_1	$R_{U_1 1} \ldots R_{U_1 m}$	1
			...	
			...	
			...	
			$R_{U_1 1} \ldots R_{U_1 m}$	$card(U_1)$
			...	
U_1	$R_{U_1 1} \ldots R_{U_1 m}$	U_j	$R_{U_j 1} \ldots R_{U_j m}$	1
	...		...	
U_j	$R_{U_j 1} \ldots R_{U_j m}$		...	
	...		...	
U_p	$R_{p1} \ldots R_{U_p m}$		$R_{U_j 1} \ldots R_{U_j m}$	$card(U_j)$
			...	
			$R_{p1} \ldots R_{U_p m}$	1
			...	
		U_p	...	
			...	
			$R_{p1} \ldots R_{U_p m}$	$card(U_p)$

TAB. 3.7 – Schéma explicatif de l'association de R et R'

On peut également revenir à la matrice B, en remplaçant, pour chaque U_i de R, la ligne

de R associée à U_i par les lignes associées à U_i dans B (voir Table 3.8).



TAB. 3.8 – Schéma explicatif de l'association de R et B (**à une permutation d'instances près**)

Le remplacement des n lignes de B par p lignes dans R permet de diminuer la taille des données à traiter.

Chaque ligne de R (associée à un U_i) représente la **moyenne** des lignes associées à U_i dans B (voir plus loin l'opération de fragmentation 3.3.2).

On notera R_X la matrice contractée associée à une projection P_X de B.

3.3.2 Définitions des opérations liées à l'algorithme CDP

Projection $\mathcal{M}$

Soit X un sous-ensemble d'attributs de la base B issu d'une partition d'attributs $\mathcal{A}$.

On note $\mathcal{M}$ l'opération de projection définie par :

$$\mathcal{M} : B, X \rightarrow P_X$$

qui associe la projection P_X de B au sous-ensemble d'attributs X.

P_X est obtenue par application d'un masque M_X sur la matrice B.

Le masque M_X est défini sous forme d'une matrice de n lignes m colonnes telle que $M_{X_{ij}} = 1$, $\forall$ i, $\forall$ j tel que $A_j \in X$ et $M_{X_{ij}} = 0$, $\forall$ i, $\forall$ j tel que $A_j \notin X$ (voir Table 3.9).

B

	1	j	m
1	v_{11}	$\ldots v_{1j} \ldots$	v_{1m}
		$\ldots$	
		$\ldots$	
	v_{i1}	$\ldots v_{ij} \ldots$	v_{im}
		$\ldots$	
		$\ldots$	
n	v_{n1}	$\ldots v_{nj} \ldots$	v_{nm}

i_{eme} ligne de la matrice associée à B

M_X

1	0 …0	1 …1	0 …0
		…	
		…	
		1 …1	
		…	
		…	
n	0 …0	1 …1	0 …0
		$A_k \ldots A_r \in X$	

TAB. 3.9 – Représentation d'un masque M_X associé à une base B

Pour effectuer une projection de B dans un sous-espace lié au sous-ensemble d'attributs X, il suffit d'appliquer le masque M_X sur B pour obtenir P_X. (voir Table 3.10).

P_X

1	0 …0	$v_{1k} \ldots v_{1r}$	0 …0
		$v_{jk} \ldots v_{jr}$	
n	0 …0	$v_{nk} \ldots v_{nr}$	0 …0
		$A_k \ldots A_r \in X$	

TAB. 3.10 – Représentation de la projection P_X issue de B et M_X

L'opération de projection $\mathcal{M}$ est donc définie par :

$\mathcal{M}$(B, X) = ${M_X}^t.X = P_X$.

P_X est égale au produit de la représentation matricielle de B par la transposée de M_X.

Fragmentation $\mathcal{F}$

La fragmentation d'une base matricielle correspond à une partition des lignes de la matrice avec calcul de valeurs moyennes de lignes pour les fragments.
Cette opération est effectuée par utilisation d'un algorithme de clustering classique en tant

qu'étape de l'algorithme de Clustering Distribué Progressif.

Soient R une matrice contractée de n lignes, m colonnes, et $\mathcal{U}$ est une partition des instances de R telle que $\mathcal{U} = \{U_1, \ldots, U_p\}$.

On note $\mathcal{F}$ l'opération de fragmentation définie par :

$$\mathcal{F} : \{\mathcal{U}, R\} \rightarrow \{\mathcal{V}, S\}.$$

Une fragmentation $\mathcal{F}$ associe à R une matrice contractée S (voir Table 3.11).

$\mathcal{V}$ est une partition des instances de R telle que $\mathcal{V} = \{V_1, \ldots, V_q\}$
S est une matrice contractée de q lignes (correspondant aux $V_1, \ldots, V_q$ définis par $\mathcal{V}$), et m colonnes (correspondent aux attributs $A_1, \ldots, A_m$ de R) .

L'opération de fragmentation $\mathcal{F}$ est définie par :

$\forall j, \mathcal{S}_{Vj} = \frac{\sum_{k \in V_j} V_{kj} * card(k \cap V_j)}{card(V_j)}$, avec ici $card(k \cap V_j) = card(k)$.

R

	$R_{U_1 1} \ldots R_{U_1 m}$	1
V_1	…	
	…	
	…	
		$card(V_1)$
	…	
		1
	…	
V_j	…	
	…	
		$card(V_j)$
	…	
		1
	…	
V_q	…	
	…	
	$R_{U_p 1} \ldots R_{U_p m}$	$card(V_q)$

S

V_1	$S_{V_1 1} \ldots S_{V_1 m}$
	…
V_j	$S_{V_j 1} \ldots S_{V_j m}$
	…
V_q	$S_{V_q 1} \ldots S_{V_q m}$

TAB. 3.11 – Schéma explicatif de lien entre R et S (**à une permutation d'instances près**)

Croisement $\otimes$

Soient $\mathcal{U}$ et $\mathcal{V}$ deux partitions des instances de B et S_X et S_Y deux matrices contractées relatives aux sous-ensembles d'attributs X et Y de $\mathcal{A}$ (et respectivement associées à $\mathcal{U}$ et $\mathcal{V}$).

S_X est une matrice p*l (p lignes associées aux U_i, et l colonnes associées aux l attributs de X.
S_Y est une matrice q*l' (q lignes associées aux V_i, et l' colonnes associées aux l' attributs de Y.

On note $\otimes$ l'operation de croisement définie par :

$$\otimes : \{\mathcal{U}, S_X\}, \{\mathcal{V}, S_Y\} \rightarrow \{\mathcal{W}, \mathcal{T}_Z\}.$$

$\mathcal{W}$ est une partition des instances de T_Z telle que $\mathcal{W} = \{W_1, \ldots, W_r\}$

T_Z est une matrice contractée de r lignes (correspondant aux $W_1, \ldots, W_r$ définis par $\mathcal{W}$) et l+l' colonnes associé aux attributs du sous-ensemble Z de $\mathcal{A}$.

L'opération de croisement $\otimes$ est définie par :

$\mathcal{W} = \{W_i | W_i = U_j \cap V_k$ non vides, $U_j \in \mathcal{U}, V_k \in \mathcal{V}.\}$

$Z = \{A_j\} = X \cup Y.$

$\mathcal{Z}_{W_i j} = \frac{S_{X U_k j} * card(U_k \cap W_j) + S_{Y V_l j} * card(V_l \cap W_j)}{card(W_j)}$, avec $card(U_k \cap V_l) = card(W_j)$.

La matrice T_Z est définie par : $T_{Z_{ij}} = S_{X_{U_k j}}$ pour $U_k \subset Z_i$, si $A_j \in X$ et
$T_{Z_{ij}} = S_{Y_{V_l j}}$ pour $V_l \subset Z_i$ si $A_j \in Y$ (voir Table 3.13).

Exemple :
Soient les partitions $\mathcal{U}$ et $\mathcal{V}$, telles que :

$\mathcal{U} = \{U_1 = \{5, 7, 20\}, U_2 = \{2, 3, 6, 8, 10, 11, 13, 17\}, U_3 = \{4, 9, 12, 15\}, U_4 = \{1, 16\}, U_5 = \{14, 18, 19\}\}$
et $\mathcal{V} = \{V_1 = \{5, 7, 20\}, V_2 = \{2, 3, 6, 11, 17\}, V_3 = \{8, 10, 13\}, V_4 = \{1, 4, 9, 12, 14, 15, 16, 18, 19\}\}$.

La partition $\mathcal{W}$ issue du croisement est telle que
$\mathcal{W}1 = U_1 \cap V_1 = \{5, 7, 20\}, W_2 = U_2 \cap V_2 = \{2, 3, 6, 11, 17\}, W_3 = U_2 \cap V_3 = \{8, 10, 13\}, W_4 = U_3 \cap V_4 = \{4, 9, 12, 15\}, W_5 = U_4 \cap V_4 = \{1, 16\}, W_6 = U_5 \cap V_4 = \{14, 18, 19\}\}$
(voir Figure 3.12).

S_X		S_Y	
X=$\{A_1, A_2, A_5\}$		Y=$\{A_6, A_8, A_9\}$	
clés	ligne dans la matrice contractée	ligne dans la matrice contractée	clés clés
U_1=$\{5, 7, 20\}$	$S_{X_{U_1}} = \begin{pmatrix} S_{X_{U_1 1}} \\ S_{X_{U_1 2}} \\ S_{X_{U_1 5}} \end{pmatrix}$	$S_{Y_{V_1}} = \begin{pmatrix} S_{Y_{V_2 6}} \\ S_{Y_{V_2 8}} \\ S_{Y_{V_2 9}} \end{pmatrix}$	$V_1 = \{5, 7, 20\}$
U_2= {2,3,6,8,10,11,13,17}	$S_{X_{U_2}} = \begin{pmatrix} S_{X_{U_1 1}} \\ S_{X_{U_1 2}} \\ S_{X_{U_1 5}} \end{pmatrix}$	$S_{Y_{V_2}} = \begin{pmatrix} S_{Y_{V_2 6}} \\ S_{Y_{V_2 8}} \\ S_{Y_{V_2 9}} \end{pmatrix}$	V_2= {2,3,6,11,17}
U_3= {4,9,12,15}	$S_{X_{U_3}} = \begin{pmatrix} S_{X_{U_3 1}} \\ S_{X_{U_3 2}} \\ S_{X_{U_3 5}} \end{pmatrix}$	$S_{Y_{V_3}} = \begin{pmatrix} S_{Y_{V_3 6}} \\ S_{Y_{V_3 8}} \\ S_{Y_{V_3 9}} \end{pmatrix}$	V_3= {8,10,13}
U_4= {1,16}	$S_{X_{U_4}} = \begin{pmatrix} S_{X_{U_4 1}} \\ S_{X_{U_4 2}} \\ S_{X_{U_4 5}} \end{pmatrix}$	$S_{Y_{V_4}} = \begin{pmatrix} S_{Y_{V_4 6}} \\ S_{Y_{V_4 8}} \\ S_{Y_{V_4 9}} \end{pmatrix}$	V_4= {1,4,9,12,14,15, 16,18,19}
U_5= $\{14, 18, 19\}$	$S_{X_{U_5}} = \begin{pmatrix} S_{X_{U_5 1}} \\ S_{X_{U_5 2}} \\ S_{X_{U_5 5}} \end{pmatrix}$		

TAB. 3.12 – Définition de S_X et S_Y

	Z= X ∪ Y = $\{A_1, A_2, A_5, A_6, A_8, A_9\}$	
	clés	ligne dans la matrice contractée
W	$W = U_1 \cap V_1$ ={5,7,20}	$S_{X_{W_1}} = \begin{pmatrix} S_{X_{U_1 1}} \\ S_{X_{U_1 2}} \\ S_{X_{U_1 5}} \\ S_{Y_{V_1 6}} \\ S_{Y_{V_1 8}} \\ S_{Y_{V_1 9}} \end{pmatrix}$
	$W_2 = U_2 \cap V_2$ ={2,3,6,11,17}	$S_{X_{W_2}} = \begin{pmatrix} S_{X_{U_2 1}} \\ S_{X_{U_2 2}} \\ S_{X_{U_2 5}} \\ S_{Y_{V_2 6}} \\ S_{Y_{V_2 8}} \\ S_{Y_{V_2 9}} \end{pmatrix}$
	$W_3 = U_2 \cap V_3$ ={8,10,13}	$S_{X_{W_3}} = \begin{pmatrix} S_{X_{U_2 1}} \\ S_{X_{U_2 2}} \\ S_{X_{U_2 5}} \\ S_{Y_{V_3 6}} \\ S_{Y_{V_3 8}} \\ S_{Y_{V_2 9}} \end{pmatrix}$
	$W_4 = U_3 \cap V_4$ ={4,9,12,15}	$S_{X_{W_4}} = \begin{pmatrix} S_{X_{U_3 1}} \\ S_{X_{U_3 2}} \\ S_{X_{U_3 5}} \\ S_{Y_{V_4 6}} \\ S_{Y_{V_4 8}} \\ S_{Y_{V_4 9}} \end{pmatrix}$
	$W_5 = U_4 \cap V_4$ ={1,16}	$S_{X_{W_5}} = \begin{pmatrix} S_{X_{U_4 1}} \\ S_{X_{U_4 2}} \\ S_{X_{U_4 5}} \\ S_{Y_{V_4 6}} \\ S_{Y_{V_4 8}} \\ S_{Y_{V_4 9}} \end{pmatrix}$
	$W_6 = U_5 \cap V_4$ ={14,18,19}	$S_{X_{W_6}} = \begin{pmatrix} S_{X_{U_5 1}} \\ S_{X_{U_5 2}} \\ S_{X_{U_5 5}} \\ S_{Y_{V_4 6}} \\ S_{Y_{V_4 8}} \\ S_{Y_{V_4 9}} \end{pmatrix}$

TAB. 3.13 – Définition de T(Z)

3.3.3 Clustering Distribué Progressif CDP : Algorithme Général

L'algorithme général pour le Clustering Distribué Progressif (**CDP**) est un algorithme distribué itératif qui conduit à une fragmentation des instances de la base de données B (relative à tous les attributs de A) par association de deux fragmentations de projections de B à chaque itération.

- Fixer une partition $\mathcal{A}$ = $\{X, Y, Z\}$ des attributs de B.
- Poser $\mathcal{R}$ = $\{R_X, R_Y, R_Z\}$.
- **phase initiale** : appliquer $\mathcal{F}$ sur $\mathcal{R}$.

  $\mathcal{F}(\mathcal{R})$ = $\{\mathcal{F}(R_X), \mathcal{F}(R_Y), \mathcal{F}(R_Z)\}$

  i.e.$\mathcal{F}(\mathcal{R})$ = $\{(\mathcal{U}, R_X), (\mathcal{V}, R_Y), (\mathcal{W}, R_Z)\}$

  $\mathcal{R}$ = $\{S_X, S_Y, S_Z\}$
- **itérer** :

  $\forall S_X \subset \mathcal{R}$, $S_Y \subset \mathcal{R}$,

  $\mathcal{R}$ = $\mathcal{R} - S_X - S_Y \cup \mathcal{F}(\otimes((\mathcal{U}, S_X), (\mathcal{V}, S_Y)))$

  (avec $\otimes(S_X, S_Y) \rightarrow T_Z, \mathcal{F}(T_Z) \rightarrow S_Z$)
- **jusqu'à** avoir $card(\mathcal{R}) = 1$.

La phase initiale et les phases itératives sont exécutées de manière parallèle distribuée.

Phase Initiale de l'algorithme CDP

On choisit une partition $\mathcal{A}$ des attributs de la base B ($\mathcal{A}$=$\{X_j, /j = 1$ a l, $X_j = \{A_k \in A\}$, $\bigcup_j X_j = A$ et $X_j \cap X_k = \emptyset\}$).

On effectue ensuite les projections de B sur chaque X_j de $\mathcal{A}$ de manière à obtenir un ensemble $\mathcal{B} = \{P_{X_j}, j = 1l\}$.

Puis on associe à chaque fragment P_X (p lignes, q colonnes) de $\mathcal{B}$, la matrice contractée R_X et la partition d'instances $\mathcal{U}$ associée à R_X, telles que

R_X possède p lignes q colonnes, $R_X = P_X$ et $R_{X_{ij}} = P_{X_{ij}}$

$\mathcal{U} = \{U_1, \ldots U_n, avec U_i = \{K_i\}\}$.

On dénote par $\mathcal{R}$ l'ensemble des matrices contractées R_X associées aux projections P_X : $\mathcal{R}$ = $\{R_X, R_Y, R_Z\}$.

La phase initiale consiste à appliquer l'opération de fragmentation $\mathcal{F}$ (voir Section 3.3.2) sur $\mathcal{R}$.

Cela revient à appliquer $\mathcal{F}$ à chaque élément de $\mathcal{R}$:

$\mathcal{F}(\mathcal{R}) = \{\mathcal{F}(R_X), \mathcal{F}(R_Y), \mathcal{F}(R_Z)\}$

$\mathcal{F}(\mathcal{R}) = \{(\mathcal{U}, R_X), (\mathcal{V}, R_Y), (\mathcal{W}, R_Z)\}$

On obtient finalement $\mathcal{R} = \{S_X, S_Y, S_Z)\}$, une matrice contractée, représentation du résultat de la fragmentation (c'est-à-dire résultat de clustering) sur chacune des projections de la base B dans un des sous-espaces d'attributs.

On a donc un résultat de fragmentation par clustering pour des parties de la base B.

Remarque : Pour la génération des règles d'association (but final de la méthode), il peut être intéressant d'appliquer la phase initiale à chaque attribut indépendamment, i. e. une partition des attributs en m groupes de cardinalité 1, de manière à obtenir une information globale concernant chaque attribut.

Exemple d'exécution

Soient une base $B = \{A, K, V\}$ avec $K = \{1, \ldots, 20\}$ et $A = \{A_1, A_2\}$.
On suppose une répartition des données dans l'espace comme indiqué sur la Figure 3.1.

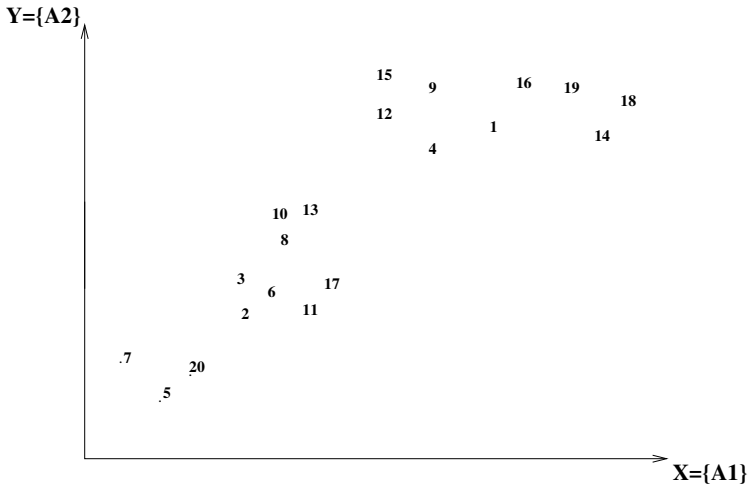


FIG. 3.1 – Répartition des données dans les dimensions de A.

La phase initiale de fragmentation sur P_X (sur la matrice contractée R_X associée à P_X), a fourni une partition $\mathcal{U}$ de K à 5 éléments (clusters) U_1, U_2, U_3, U_4 et U_5 (voir Figure 3.3),

qui sont représentés sur l'axe des abscisses sur la Figure 3.2.

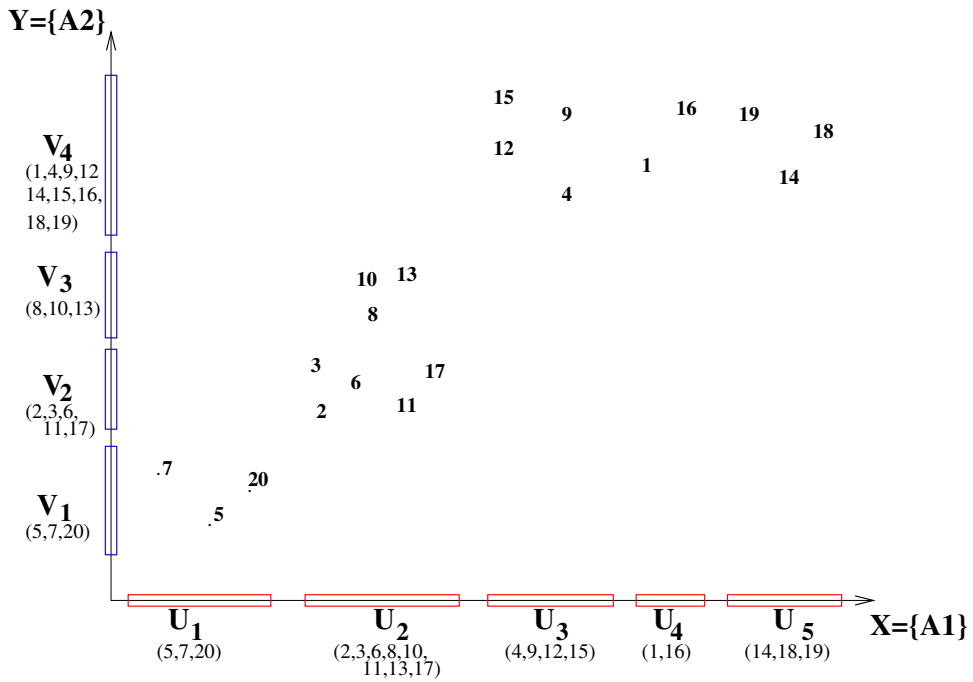


FIG. 3.2 – Fragmentation de P_X et fragmentation de P_Y.

La phase initiale de fragmentation sur P_Y (sur la matrice contractée R_Y associée à P_Y), a fourni une partition $\mathcal{V}$ de K à 5 éléments (clusters) V_1, V_2, V_3 et V_4 (voir Figure 3.3), qui sont représentés sur l'axe des ordonnées sur la Figure 3.2.

S X	S Y
U1	V1
U2	V2
	V3
U3	V4
U4	
U5	

FIG. 3.3 – Fragmentation de P_X et fragmentation de P_Y (à une permutation d'instances près).

Phase de croisement

Considérons deux résultats de fragmentations de projections de la base de données issues de la phase initiale (voir Section 3.3.3) : S_X et S_Y, tels que $S_X \subset \mathcal{R}$, $S_Y \subset \mathcal{R}$, avec $X \cap Y = \emptyset$ (par définition d'une partition des attributs).

Considérons également les partitions d'instances $\mathcal{U}$ et $\mathcal{V}$, associées respectivement à S_X et S_Y.

On soustrait S_X et S_Y de l'ensemble $\mathcal{R}$ obtenu en Section 3.3.3.

Soit $Z = X \cup Y$, $\mathbf{Z} \subset \mathcal{A}$.

La phase de croisement permet de définir une fragmentation des instances sur la projection P_Z, sous la forme d'une matrice contractée T_Z et d'une partition d'instances associée $\mathcal{W}$.

$$\otimes((\mathcal{U}, S_X), (\mathcal{V}, S_Y)) = (\mathcal{W}, T_Z).$$

Exemple d'exécution

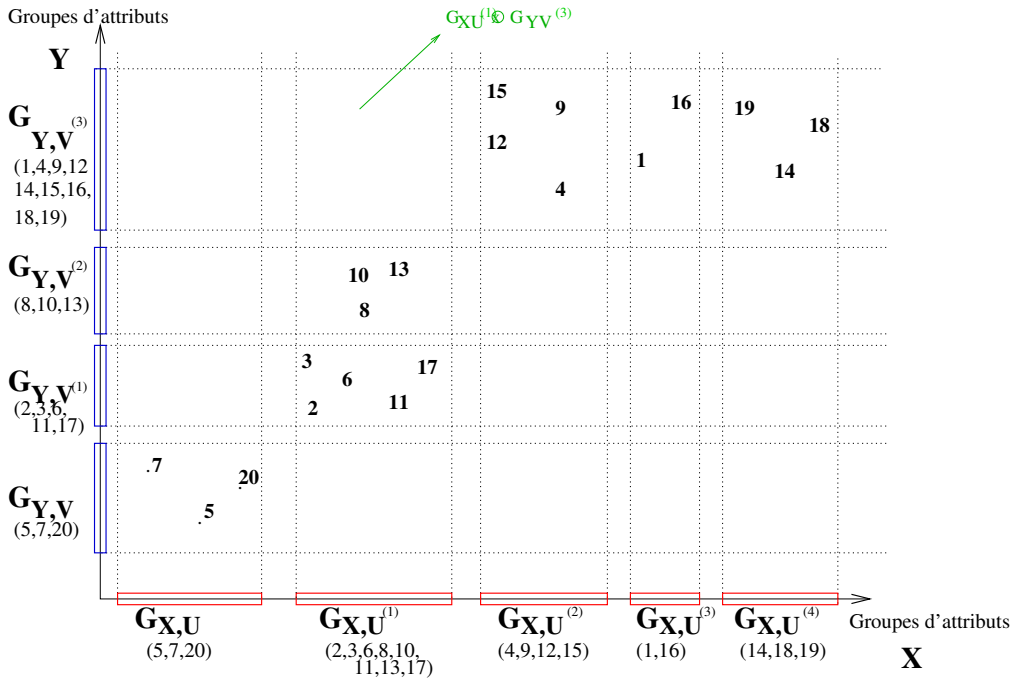


FIG. 3.4 – Croisement de S_X et S_Y (répartition dans l'espace).

La phase de croisement appliquée à S_X et S_Y fournit une partition $\mathcal{W}$ à 6 éléments (clusters non vides) : W_1, W_2, W_3, W_4, W_5 et W_6 (voir Figure 3.4) représentés sur la Figure 3.5.

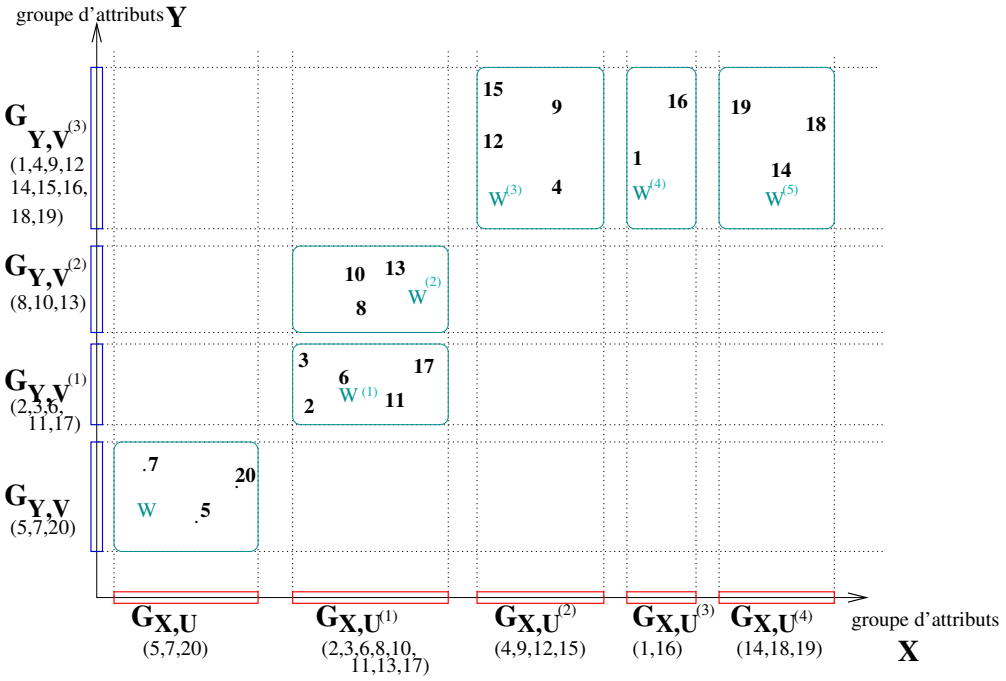


FIG. 3.5 – Croisement de S_X et S_Y (répartition dans l'espace, après suppression des groupes vides).

Pour obtenir les éléments de $\mathcal{W}$, on effectue d'abord le croisement des groupes à proprement parlé (voir Figure 3.6), puis on élimine les groupes vides (voir Figure 3.7).

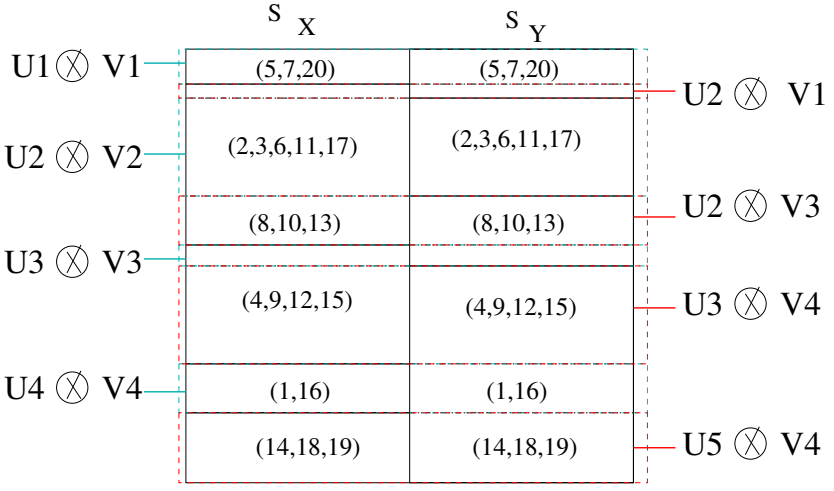


FIG. 3.6 – Croisement de S_X et S_Y (définition des groupes).

Phase de regroupement optimisé

La dernière phase du clustering progressif consiste à optimiser la fragmentation de la base de données T_Z obtenu lors de la phase de croisement de S_X et S_Y (voir Section 3.3.3).

	S X	S Y
W1	(5,7,20)	
W2	(2,3,6,11,17)	
W3	(8,10,13)	
W4	(4,9,12,15)	
W5	(1,16)	
W6	(14,18,19)	

FIG. 3.7 – Croisement de S_X et S_Y (définition des groupes, après suppression des groupes vides).

La phase de regroupement optimisé consiste à effectuer une nouvelle fragmentation par clustering, et donc à appliquer l'opération de fragmentation $\mathcal{F}$ à T_Z .

Ainsi, on obtient $\mathcal{F}(W, T_Z) \rightarrow (V, S_Z)$
avec V et S_Z définis par la fonction $\mathcal{F}$ (voir Section 3.3.2).

Exemple d'exécution

La phase de regroupement optimisé sur la matrice contractée T_Z et la partition $\mathcal{W}$ associée permet d'obtenir une matrice contractée S_Z et la partition associée $\mathcal{V}$ à 4 éléments (4 clusters) : W_1, W_2, W_3 et W_4 au lieu de la partition $\mathcal{W}$ à 6 éléments représentés sur la Figure 3.9.

Dans l'exemple, les deux premiers fragments sont regroupés ainsi que le 3_{eme} et le 4_{eme} (voir Figure 3.8).

Finalement la méthode de clustering distribué progressif résulte donc en une partition à 4 éléments (4 clusters) : W_1, W_2, W_3 et W_4 représentés dans l'espace sur la Figures 3.10.

Critère itératif

Les phases de croisement et de regroupement optimisé sont itérées par associations successives de couples (S_X, S_Y) jusqu'à obtenir une unique partition de la base B. L'ordre utilisé pour les associations durant la phase de croisement offre différentes versions (voir Section 3.5.1), reposant sur la disponibilité des résultats S_{X_i}.
Puisque le nombre de clusters résultants peut être limité à chaque ajout de nouveaux attributs en utilisant la phase de regroupement optimisé , les méthodes agglomératives, bien

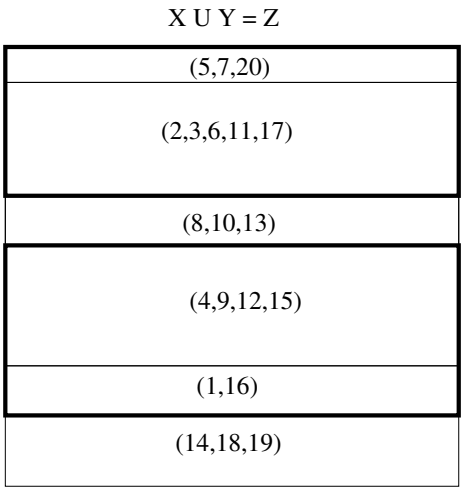


FIG. 3.8 – Fragmentation par clustering - Clustering de Regroupement (définition des groupes).

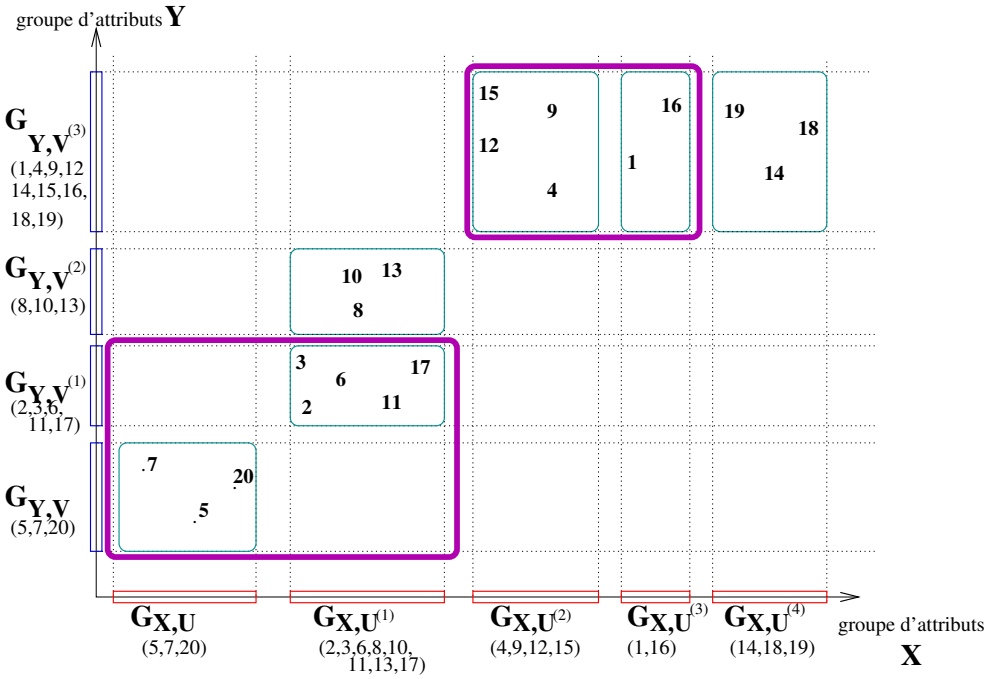


FIG. 3.9 – Fragmentation par clustering - Clustering de Regroupement (répartition dans l'espace).

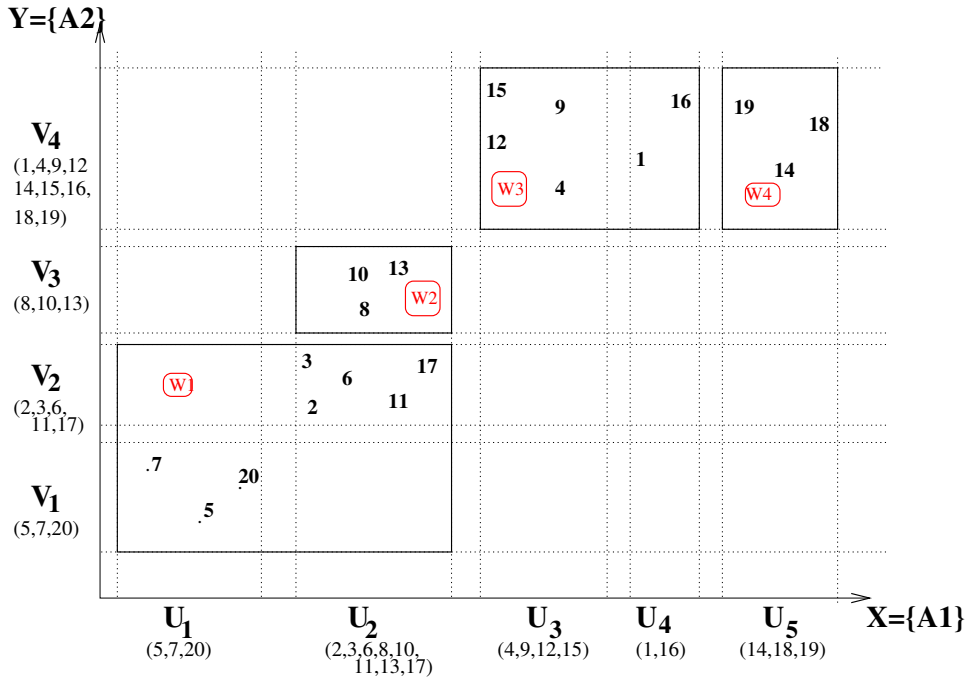


FIG. 3.10 – Résultat du clustering distribué progressif sur $A = \{A_1, A_2\}$ (répartition dans l'espace).

que coûteuses peuvent être utilisées, aussi bien que les méthodes de type K-Moyennes (voir Section 3.6.2 pour les comparaisons de performances entre les deux types de méthodes).

3.4 Complexité

3.4.1 Complexité des algorithmes classiques de clustering

Complexité Soient :

- n : le nombre d'instances à traiter,
- m : le nombre d'attributs des instances,
- k : le nombre de clusters à identifier,
- q : le nombre d'itérations nécessaires jusqu'à la convergence.

Pour des données à une seule dimension, les algorithmes de partitionnement ont une complexité de l'ordre de $O(knq)$ en temps et $O(k+n)$ en espace.
Toujours, pour des données de dimension 1, les algorithmes hiérarchiques ont au mieux une complexité de l'ordre de $O(n^2 log(n))$ en temps et $O(n^2)$ en espace.

Pour des données en plusieurs dimensions, les algorithmes de partitionnement ont une complexité de l'ordre de $O(knmq)$ en temps et $O(km+nm)$ en espace.
Toujours, pour des données en plusieurs dimensions, les algorithmes hiérarchiques ont une complexité de l'ordre de $O((nm)^2 log(nm))$ en temps et $O((nm)^2)$ en espace.

Critères d'arrêt utilisés Le nombre de clusters doit être fixé pour K-Means, même si un système itératif peut être utilisé pour trouver le nombre k le plus approprié.
Dans la méthode agglomérative, le niveau le plus approprié de la hiérarchie, et donc le nombre optimal de clusters, doit être identifié lors de la construction de cette hiérarchie.
Les clusters sont représentés par un unique élément, généralement leurs centres (pondérés).

Les algorithmes hiérarchiques sont donc plus coûteux en temps et en espace que les algorithmes de partitionnement.

3.4.2 Complexité du clustering progressif : CDP

On s'intéresse ici aux **complexités de traitement**(temps d'exécution).

Pour effectuer un **clustering KMeans multidimensionnel** sur n instances, m attributs afin d'identifer k groupes en q itérations de la méthode :

On doit effectuer le calcul d'une distance pour des instances de m attributs. La complexité du calcul d'une distance est de l'ordre de $O(m)$ soustractions et $O(m-1)$ additions, soit $O(2m-1)$.

La complexité du clustering réside essentiellement dans les calculs de distance effectués. A chaque itération, on calcule la distance de chaque instance à chacun des k groupes, soit kn calculs de distances,

soit $(2m-1) \times k \times n$ calculs de distance à chaque itération.

D'où une complexité de l'ordre de $O(qkn(2m-1))$, soit $O(qknm)$ en temps de traitement.

Le facteur k (nombre de clusters) peut être négligé (il est relativement faible par rapport au nombre d'instances). Le facteur q (nombre d'itération pour atteindre la convergence de la méthode) peut être grand, il ne doit pas être négligé. Le facteur m (nombre d'attributs) ne doit pas être négligé puisque le but est le traitement de base de données de grande taille en terme de nombre d'instances **et** nombre d'attributs.
On se ramène donc à un fonctionnement en $O(qnm)$ pour le clustering de référence (algorithme Kmeans multidimensionnel).

Pour effectuer un clustering via la méthode **CDP**.

Dans la version de base, on identifie 3 phases :

- phase initiale : clustering unidimensionnel ;
- phase de croisement ;
- phase de clustering optimisé sur les groupes obtenus par croisement.

Pour la phase **initiale de clustering unidimensionnel** sur n instances, m attributs, visant à identifier k groupes en q itérations.
Chaque attribut est traité indépendamment des autres, soit m clustering unidimensionnels, on a donc m traitements en $O(qkn)$, soit un traitement complet en $O(qknm)$.

Pour chaque étape de la phase de croisement de deux fois k groupes comportant en moyenne $\frac{n}{k}$ instances.
On doit réaliser k^2 jointures de $\frac{n}{k}$ instances, soit une complexité de l'ordre de $O(kn)$.

Pour chaque étape de la phase de clustering optimisé, dans le pire des cas (on n'a obtenu aucun groupe vide lors de la phase de croisement), on doit effectuer un clustering sur k^2 instances (les groupes issus du croisement), p attributs (p allant de 1 à m selon l'itération), pour identifier l groupes, en r itérations.
On obtient une complexité de l'ordre de $O(rlk^2p)$.

On effectue $m-1$ croisements (chacun suivi d'un clustering optimisé), soit une complexité de : $m-1$ traitements en $O(kn)+O(rlk^2p)$.

D'où une complexité de l'ordre de $O(nmqk^2)$
($O(qkmn)+(m-1)(O(kn)+O(rlk^2p))$).

Une fois encore le facteur k (nombre de clusters) peut être négligé (de même que le facteur l). Les facteur q (nombre d'itération pour atteindre la convergence de la méthode) et m (nombre d'attributs) ne doivent pas être négligés.
On se ramène donc à un fonctionnement en $O(qnm)$ (même complexité que l'algorithme de référence KMeans multidimensionnel).

Dans la version **macro-itérative**, la phase initiale consiste en un clustering mutlidimensionnel sur m instances, $\frac{m}{g}$ attributs (g étant la granularité du traitement macro-itératif, i.e. le nombre d'attributs considérés ensemble pour le clustering initial), visant à identifier k groupes en q itérations.
Chaque groupe de g attributs est traité indépendamment des autres, soit $\frac{m}{g}$ clustering multidimensionnels sur g attributs, on a donc $\frac{m}{g}$ traitements en $O(qkng)$, soit un traitement complet en $O(qknm)$.

Pour chaque étape de la phase de croisement, la complexité reste identique à celle observée dans la version basique de CDP, soit une complexité de l'ordre de $O(kn)$.

Pour chaque étape de la phase de clustering optimisé, dans le pire des cas (on n'a obtenu aucun groupe vide lors de la phase de croisement), on doit effectuer un clustering sur k^2 instances (les groupes issus du croisement), p attributs (p allant de 1 à m selon l'itération),

pour identifier l groupes, en r itérations.
On obtient une complexité de l'ordre de $O(rlk^2p)$.

On effectue $\frac{m}{g} - 1$ croisements puis clustering optimisé, soit une complexité de : $\frac{m}{g} - 1$ traitements en $O(kn) + O(rlk^2p)$.

D'où une complexité également de l'ordre de $O(nmqk^2)$
($O(qknm) + (\frac{m}{g} - 1)(O(kn) + O(rlk^2p))$).

Pour cette version macro-itérative, on se ramène donc également à un fonctionnement en $O(qnm)$.

3.5 Exploitation de la distribution

3.5.1 Parallélisme

Distribution

Pour la phase de clustering initial, les dimensions devant être traitées indépendamment, un découpage vertical des données (contexte multibase comme pour le schéma général de recherche de règles) est utilisé. Une approche macro itérative (voir Section 3.5.1) peut dès lors être utilisée sur l'ensemble des attributs présents sur un site.
Pour la phase de croisement, un modèle arborescent peut être utilisé (voir Section 3.5.1). On peut également considérer les résultats intermédiaires un par un en fonction de leur disponibilités (voir Section 3.5.1), plutôt que d'imposer un ordre de croisement.

La répartition de charge doit être assurée autant que possible dans la grille en fonction du nombre de machines, de leurs puissances respectives...

Spécificités d'un déploiement sur grille dans les traitements

La méthode de clustering distribué progressif est itérée, en croisant des résultats de fragmentation par clustering (unidimensionnel ou multidimensionnel, voir Section 3.5.1).
Les étapes de croisement et de regroupement optimisé peuvent être utilisés sur des R_{X_i} distribués (en respectant le partition initiale P_X, P_Y...) de la base de données (voir Section 3.5.1).

Scénarii d'incrémentation progressive du croisement

Plusieurs versions ont été comparées pour le scénario d'incrémentation progressif du croisement (visant à insérer des résultats de clustering), afin de tester l'impact sur les résultats de l'ordre des associations (des attributs).

Ainsi, deux types de scénarii d'incrémentation apparaissent :

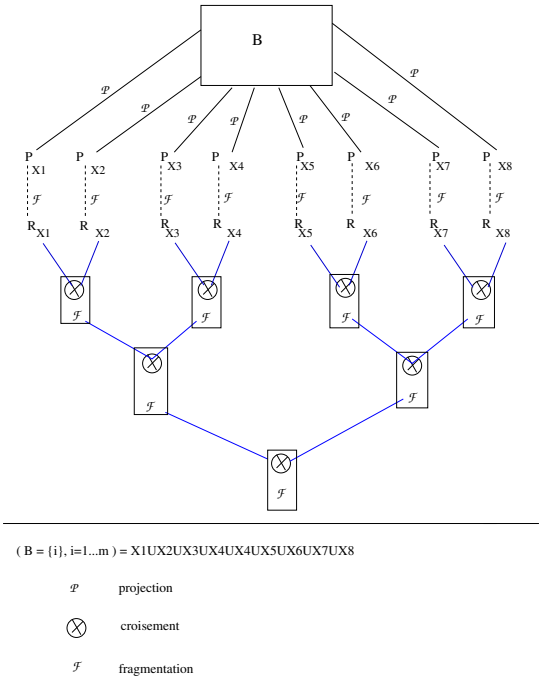


FIG. 3.11 – Scénario arborescent statique d'incrémentation progressive du croisement

- un scénario arborescent statique (voir Figure 3.11),
- un scénario arborescent dynamique par disponibilités (voir Figure 3.12).

Le modèle arborescent statique (voir Figure 3.11) consiste à croiser les R_{X_i} issus de la phase initale deux par deux selon un ordre imposé par la distribution.
Le résultat de ce croisement subit alors un regroupement optimisé par fragmentation.
Puis le résultat du regroupement optimisé est considéré pour le croisement avec un autre résultat du regroupement optimisé R_{X_i} imposé , et ainsi de suite.

Le modèle arborescent dynamique par disponibilité (voir Figure 3.12) consiste à croiser les R_{X_i} issus de la phase initale ou de regroupement optimisé deux par deux, non plus en fonction d'un ordre imposé comme dans la version arborescente statique, mais en fonction de la disponibilité des résultats R_{X_i}.
Le résultat de ce croisement subit alors un regroupement optimisé par fragmentation.
Puis le résultat du regroupement optimisé est considéré pour le croisement avec un des R_{X_i} disponible, et ainsi de suite.
L'ordre des croisements effectués dans ce second scénario (le croisement par disponibilité) dépend de l'exécution asynchrone sur la grille.

Le fait de tester ces deux scénarios permet de s'assurer que l'ordre utilisé pour associer (voir étape de croisement des résultats, Section 3.3.3) les résultats distribués de l'étape initiale (dépendant de l'exécution sur la grille) avant l'étape de regroupement optimisatoire ne modifie pas les résultats obtenus.

Sur les Figures 3.11 et 3.12, les P_{X_i} représentent les projections de la base au départ ; les R_{X_i} représentent les résultats de fragmentation par clustering (unidimensionnel ou multi-

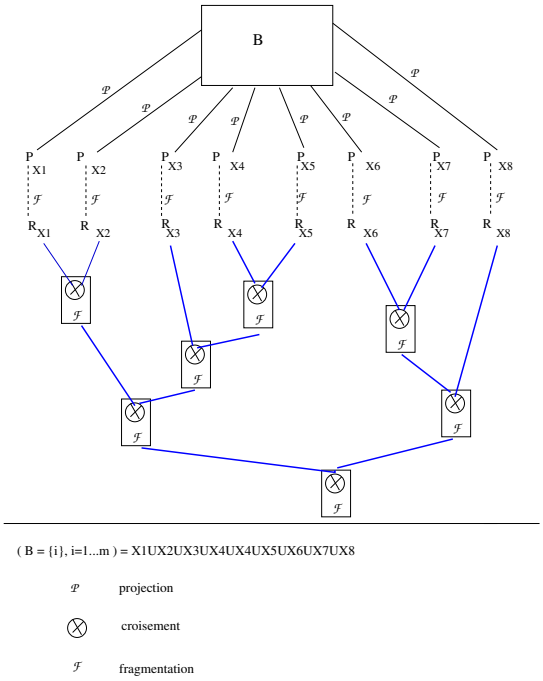


FIG. 3.12 – Scénario arborescent dynamique d'incrémentation progressive du croisement par disponibilité

dimensionnel voir Section 3.5.1) sur les R_{X_i}.

Vision Macro-Itérative

De manière à tester l'impact sur les résultats d'une exécution distribuée, nous avons décider de comparer deux cas :

1. Chacune des partitions verticales contient un seul et unique attribut : le clustering initial est unidimensionnel, c'est la version de base (voir Figures 3.13 et 3.14)
2. Chacune des partitions verticales contient un certain nombre (> 1) d'attributs : le clustering initial est multidimensionnel, c'est la version **macro-itérative** (voir Figures 3.15 et 3.16).

Dans le cas de clustering initial unidimensionnel, des itérations de l'algorithme CDP (croisement puis fragmentation) doivent être exécutées sur chaque site de pré-traitement pour arriver à un résultat de clustering pour toutes données du site. Puis des croisements entre résultats des différents sites sont exécutées, selon le scénario choisi par arborescence ou par disponibilité (voir Figures 3.13 et 3.14).

Dans le cas de clustering initial multidimensionnel (pour tous les attributs présents sur le site), dès les premières itérations de l'algorithme CDP (croisement puis fragmentation), on travaille entre différents sites.
Le clustering multidimensionnel peut être effectué sur chaque noeud (exploitant tous les attributs présents sur ce noeud) au lieu d'une exécution de clustering unidimensionnel sur chacun des attributs du noeud de la grille.

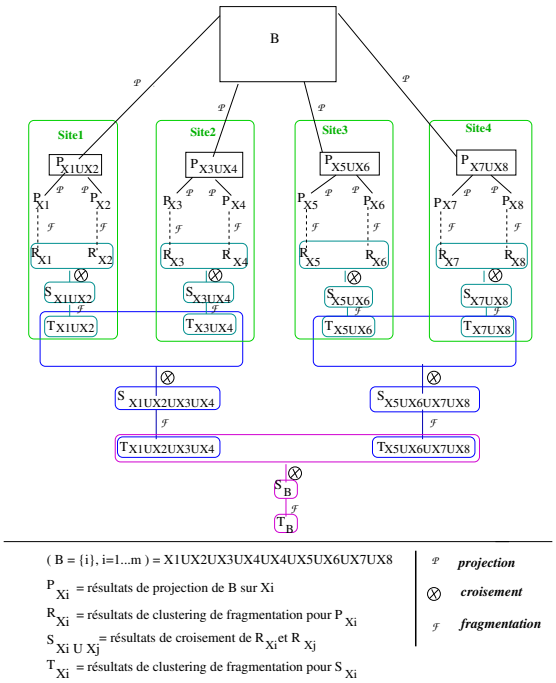


FIG. 3.13 – Exécution du scénario arborescent statique d'incrémentation progressive du croisement (version de base)

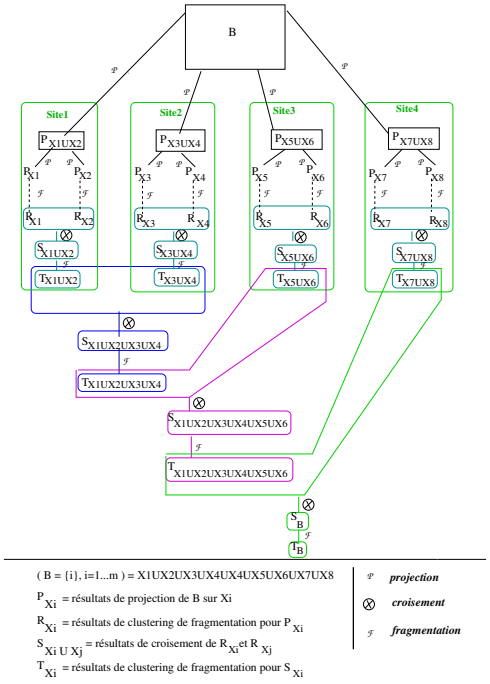


FIG. 3.14 – Exécution du scénario arborescent dynamique d'incrémentation progressive du croisement par disponibilité (version de base)

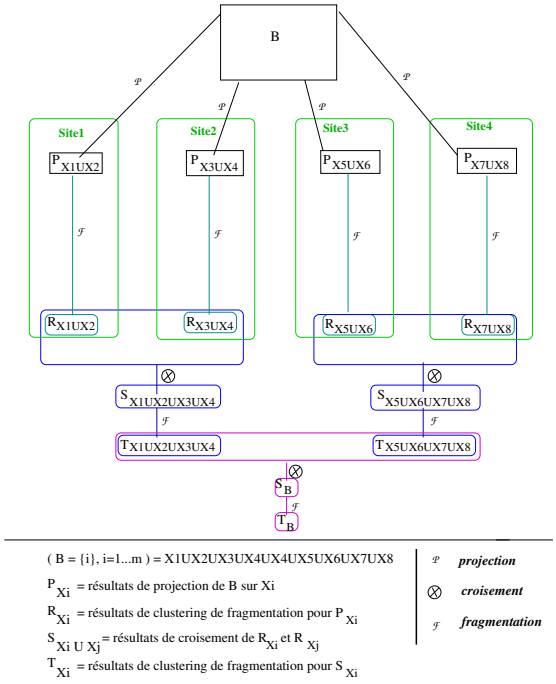


FIG. 3.15 – Exécution du scénario arborescent statique d'incrémentation progressive du croisement avec utilisation de l'amélioration macro-itérative

Ainsi les premières phases de l'algorithme CDP (croisement puis fragmentation) se font entre des résultats $R_{\bigcup_i U_i}$ plutôt que sur des résultats R_{X_i} dans la version classique.
On peut effectuer les croisements selon un scénario arborescent (voir Figure 3.15) ou selon un scénario par disponibilité (voir Figure 3.16).

Sur les Figures 3.15 et 3.16, $R_{\bigcup_i U_i}$ représentent les résultats de fragmentation (clustering multidimensionnel) cette version macro-itérative, dont la phase initiale est réalisée en une seule étape, sur tous les attributs présents sur un site. On obtient ainsi un résultat macro-itératif pour chaque site, ces résultats servent de base aux étapes de croisement et de regroupement optimisé. Cette considération multidimensionnelle pour l'étape initiale sera appelée vision macro-itérative dans les résultats présentés dans la section suivante.
Le degré maximal de distribution pour la phase initiale consiste à fragmenter la base de données de m attributs sur m sites de traitement. Ainsi, chaque site doit effectuer le traitement initial de fragmentation pour une projection $R_X i$ relative à un seul attribut.

Le degré minimal de distribution pour la phase initiale consiste à traiter toute la base de données de m attributs sur un unique site de traitement. Ainsi, le site effectue un clustering de fragmentation multidimensionnel sur les m attributs de la base. Cela correspond à une version macro-itérative centralisée.

Dans la réalité, il faut fragmenter les projections de la base de données de manière à obtenir un traitement efficace en exploitant les noeuds disponibles, sans pour autant obtenir une granularité trop fine (un attribut par noeud, degré maximal de paralèlisme), les phases de croisement-fragmentation entre résultats de différents sites engendrant des communications.

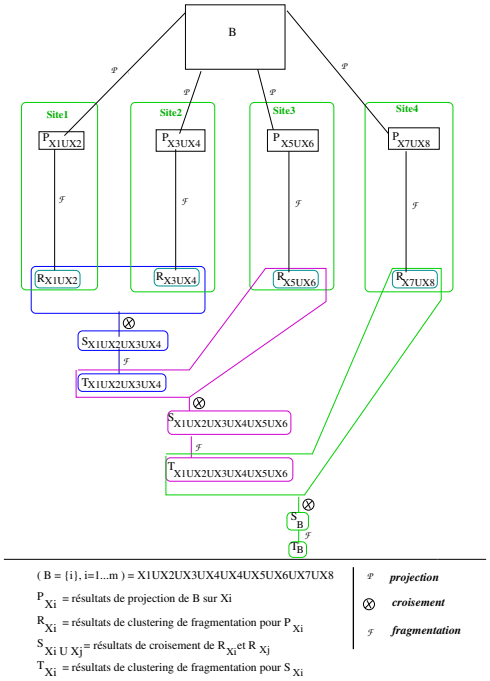


FIG. 3.16 – Exécution du scénario arborescent dynamique d'incrémentation progressive du croisement par disponibilité avec utilisation de l'amélioration macro-itérative

Parallélisme et communications

Des étapes de fragmentation par clusterings parallèles (phase initiale) sont effectuées sur les attributs de la base de données en fonction de la distribution initiale de la multibase.
Les résultats des fragmentations sur chaque R_{X_i} (avec $X_i = A_i$ clusterings unidimensionnels) sont stockés pour être utilisés plus tard dans le problème de règles d'association. Ces résultats de clustering (de la phase initiale de clustering) apparaissent comme une information globale (sur chacun des attributs, indépendamment des autres) qui permettra d'optimiser les traitements locaux parallèles des algorithmes de génération de règles d'association, et qui représente une importante source de connaissances.
Durant l'étape de fragmentation par le clustering distribué progressif, les données ne sont pas communiquées en tant que telles entre les noeuds de la grille puisque la méthode travaille uniquement sur les résultats des clusterings précédents. Ainsi les communications sont composées des T_{X_i} (voir Section 3.2) qui doivent être traités par un scénario de croisement du clustering progressif (voir Figures 3.11 et 3.12). Les communications sont limitées en fonction du scénario de croisement utilisé.

On considère le degré maximal de distribution où chacun des R_{X_i} (issu de la projection initiale) stocké sur un noeud particulier de la grille correspond à un seul attribut ($X_i = A_i$), pour i allant de 1 à m.
On considère également que le traitement par croisement d'un couple (T_{X_i}, T_{X_j}) (ou (R_{X_i}, R_{X_j})) s'effectue toujours sur le site stockant T_{X_i}. Le nombre de communications est alors égal au nombre de croisements.

Le nombre de communications pour le scénario de croisement arborescent statique cor-

respond donc aux nombre de croisements effectués dans ce scénario.
Lors du premier jeu de croisements (premier niveau de l'arbre voir Figure 3.11), $\frac{m}{2}$ sites envoient leur résultats intermédiaires aux $\frac{m}{2}$ autres sites qui effectuent le traitement de croisement. Lors du deuxième jeu de croisement (niveau de l'arbre), $\frac{m}{4}$ sites envoient leur résultats intermédiaires aux $\frac{m}{4}$ autres sites qui effectuent le traitement de croisement, et ainsi de suite. (m étant le nombre d'attributs de la base), on a donc : $\frac{m}{2} + \frac{m}{4} + \frac{m}{8} + \ldots = m$ croisements. On effectue donc environ **m** communications dans ce pire des cas (ceci correspond à la moitié du nombre d'arcs d'un arbre binaire).

Par exemple, si $T_{X_1 \cup X_2}$ est généré sur le noeud stockant R_{X_1}, alors R_{X_2} doit être envoyé à ce noeud pour qu'il puisse effectué son traitement de croisement. Si $T_{X_3 \cup X_4}$ est généré sur le noeud stockant R_{X_3}, alors R_{X_4} doit être envoyé à ce noeud. Si $T_{X_1 \cup X_2 \cup X_3 \cup X_4}$ est généré sur le noeud stockant $T_{X_1 \cup X_2}$, alors $T_{X_3 \cup X_4}$ doit être envoyé à ce noeud ...

Le nombre de communications pour le scénario de croisement arborescent dynamique par disponibilité correspond également au nombre de croisements effectué dans ce scénario.
Considérant que c'est toujours le premier site qui effectue le traitement de croisement, les $m - 1$ autres sites devront lui envoyer leur résultats intermédiaires (voir Figure 3.12). m étant le nombre d'attributs de la base, on a donc : $m - 1$ croisements. On effectue donc également de l'ordre de **m** communications dans ce pire des cas.

Par exemple, si $T_{X_1 \cup X_2}$ est généré sur le noeud stockant R_{X_1}, alors R_{X_2} doit être envoyé à ce noeud. Si $T_{X_1 \cup X_2 \cup X_3}$ est généré sur le noeud stockant $T_{X_1 \cup X_2}$, alors R_{X_3} doit être envoyé à ce noeud ...

Le nombre de communications est donc identique pour les scénarii de croisement arborescent statique et dynamique par disponibilité, puisqu'il s'agit dans les deux cas d'arbres binaires (l'un complet, l'autre incomplet) ayant les mêmes feuilles.

Dans le schéma général DisDaMin pour la génération de règles d'association, cette étape de fragmentation par l'algorithme CDP (clustering distribué progressif), permet d'obtenir des fragments de données discrétisées à traiter. Les données réelles ne sont plus utilisées par la suite, les règles d'association sont générées sur les données discrétisées (résultats de la fragmentation sur chaque attribut lors de la phase initiale en version de base de l'algorithme).
Les données discrétisées sont communiquées aux noeuds de traitement en fonction du résultat final du clustering distribué progressif.

3.6 Evaluation du Clustering Distribué Progressif

Les expérimentations concernant la qualité du Clustering Distribué Progressif ont été réalisées sur des données synthétiques comportant de 2 à 25 dimensions (2 à 25 attributs continus au départ). Les données ont été synthétisées de manière à assurer l'existence de groupes cohérents au sein des données. Ceci grâce à la génération de valeurs aléatoires

pour les centres des groupes avec l'utilisation de variances pour générer des données autour de ces points centraux.

Du fait de la complexité des méthodes de clustering agglomératif (et afin de pouvoir comparer l'utilisation d'un tel clustering à l'utilisation d'un clustering de type K-Moyennes), le nombre de lignes dans les données a été limité à 1000. Les différents scénarios décrits précédemment ont été implémentés puis testés (scénario de croisement arborescent ou par disponibilité, clustering agglomératif ou de type K-Moyennes). Les résultats obtenus pour ces différents scénarios ont été comparés aux clusters présents dans les données initiales (les données ont été générées de manière à assurer l'existence de tels clusters, les informations concernant ces clusters "initiaux" sont donc connues), et également comparés avec les méthodes globales séquentielles multidimensionnels de clusterings par K-Moyenne et par clustering agglomératif. Les résultats présentés sont constitués de moyennes réalisées sur 50 tests sur des données de 25 dimensions. (les résultats concernant les données comportant moins de 25 dimensions sont comparables à ceux présentés ici en terme de qualité des résultats comme de complexité d'exécution).

3.6.1 Principes des Combinaisons utilisées

Le choix des méthodes de clustering apparait à deux niveaux de la méthode de clustering progressif :

- en tant que clustering initial sur les attributs pris individuellement (voir phase initiale - Section 3.3.3),
- et en tant que clustering de regroupement optimisé (voir étape de regroupement optimisé - Section 3.3.3).

A ces deux niveaux, les méthodes de type K-Moyenne tout comme les méthodes de clustering agglomératif peuvent être considérées. Ainsi les tests permettront de comparer les combinaisons des deux types de clustering pour :

- la phase de clustering initial : K-Moyennes unidimensionnel ou clustering agglomératif unidimensionnel,
- la phase de regroupement optimisé : K-Moyennes ou clustering agglomératif.

MA et MK représentent respectivement la clustering agglomératif multidimensionnel et le clustering par K-Moyennes multidimensionnel.
Nous utilisons la notation suivante, **WXYZ** représente les versions de clustering progressif avec :

- W : la granularité du clustering initial, 1 pour unidimensionnel ou $n \in]1, m]$ pour la version macro-itérative (voir Section 3.5.1) ;
- X : l'algorithme utilisé pour le clustering initial, A = agglomératif, K = K-Moyennes (voir phase initiale - Section 3.3.3) ;
- Y : l'algorithme utilisé pour le clustering de regroupement optimisé, A ou K (voir étape de croisement - Section 3.3.3) ;
- Z : le scénario pour la méthode d'incrémentation progressive du croisement, B : arborescent statique (voir Figure 3.11) ou D : arborescent dynamique par disponibilité (voir Figure 3.12).

Par exemple, nAKD représente une version macro-itérative utilisant le clustering agglomératif pour l'étape de clustering initial, le clustering K-Means pour le clustering de regroupement optimisé et un scénario arborescent dynamique d'incrémentation progressive de croisement par disponibilité.

3.6.2 Qualité des clusters générés

La qualité des résultats est appréciée par comparaison des clusters générés par la méthode de clustering progressif et des clusters initialement présents dans les données (groupes initiaux de génération des données). Ces derniers servent de référentiel.

Les groupes de référence sont identifiés par la méthode de clustering agglomératif multidimensionnel (MA) ainsi que la méthode de clustering multidimensionnel par K-Moyennes (MK). La comparaison des résultats consiste à vérifier que la répartition des instances est la même entre les deux ensembles de clusters (référentiel et solution évaluée).

- Les scénarios de clustering progressif utilisant le clustering agglomératif pour l'étape de clustering initial (version de base ou version macro-itérative - voir versions 1AY*, nAY* sur les Figures 3.17 et 3.19), produisent des clusters qui représentent un mélange des groupes initiaux (référentiel). En particulier, les scénarios utilisant le clustering agglomératif pour les deux étapes (versions 1AA* sur les Figures 3.17 et 3.19) produisent les pires résultats concernant ce mélange.

- Les scénarios de clustering progressif utilisant le clustering agglomératif pour l'étape de regroupement optimisé (1KA*, nKA*) convergent en un cluster principal (incluant la plupart des instances) et des clusters "unitaires" avec peu d'instances.

- Les scénarios de clustering progressif utilisant la méthode des K-Moyennes pour l'étape de clustering initial (versions 1KY* and nKY* sur les Figures 3.17 et 3.19) produisent des clusters similaires à ceux du référentiel avec quelques agglomérations de groupes voisins (+/- 10% des groupes existant n'apparaissent pas en tant que groupes indépendants, mais sont inclus dans un groupe voisin). Malgré ces petites imperfections, les clusters générés peuvent être considérés comme similaires aux clusters du référentiel, et donc ces scénarios de clustering progressif peuvent être considérés comme de bonnes méthodes heuristiques si l'on prend en compte les possibilités de distribution offertes par la méthode de clustering progressif.

3.6.3 Temps d'exécution

Rappels des légendes :

Les temps d'exécution peuvent être décomposés en 2 composantes :

- un **temps initial : T-Init** correspondant à la phase de clustering initial (voir Section 3.3.3) ;

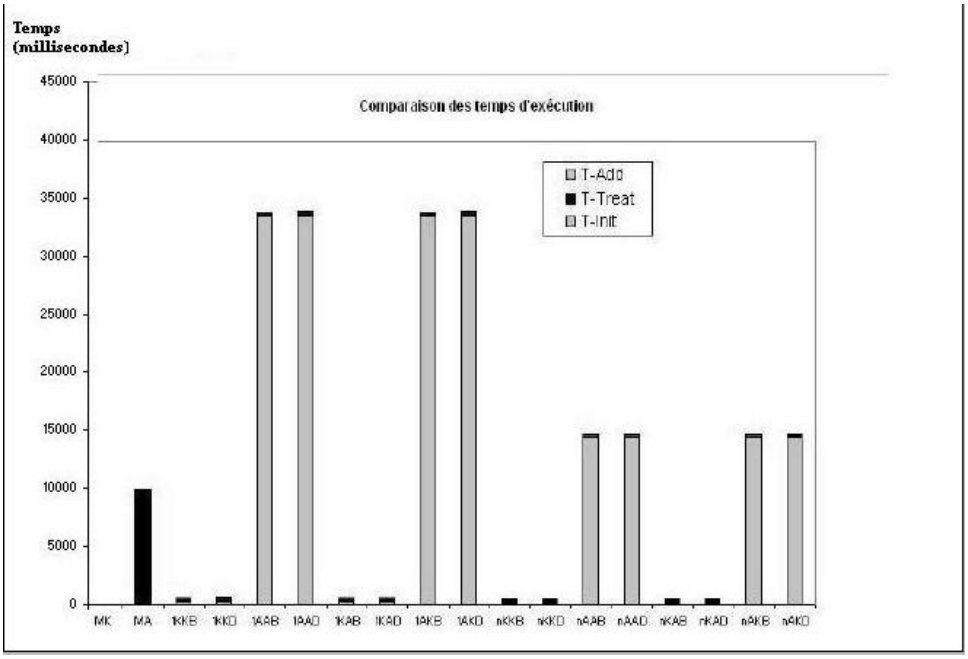


FIG. 3.17 – Comparaison des temps d'exécution selon les mode de clustering utilisé

- un **temps de traitement : T-Treat** correspondant aux phases de croisement et de clustering de regroupement optimisatoire (voir Section 3.3.3).

Pour les versions macro-itératives, on ajoutera éventuellement :

- un **temps additionnel : T-Add** correspondant à un clustering unidimensionnel de chacun des attributs, nécessaires dans le schéma de recherche de règles d'association et apportant de nombreuses informations dans les autres cas, mais qui n'est pas indispensable au clustering proprement dit.

Observations :

- Les tests ont permis de vérifier que les scénarios incluant l'utilisation du clustering agglomératif (pour le clustering initial aussi bien que pour le clustering de regroupement optimisatoire) sont les plus coûteuses en terme de temps d'exécution (voir Figure 3.17), et ne peuvent donc pas être utilisables sur de grandes quantités de données (spécialement pour la phase de clustering initial unidimensionnel).

  Puisque ces versions ne produisent pas de clusters justes, les temps d'exécution pour ces méthodes ne sont pas détaillés. La Figure 3.18 met en évidence les temps d'exécution de ces méthodes. On peut simplement noter que la version multidimensionnelle agglomérative (version MA sur la Figure 3.17) nécessite un temps d'exécution 3340 fois plus grand que la version multidimensionnel par algorithme KMeans (version MK sur la Figure 3.17) ; et que les versions entièrement agglomératives (versions *AA* sur la Figure 3.17)) nécessitent des temps d'exécution de 6700 à 13500 fois plus grands que la version MK).

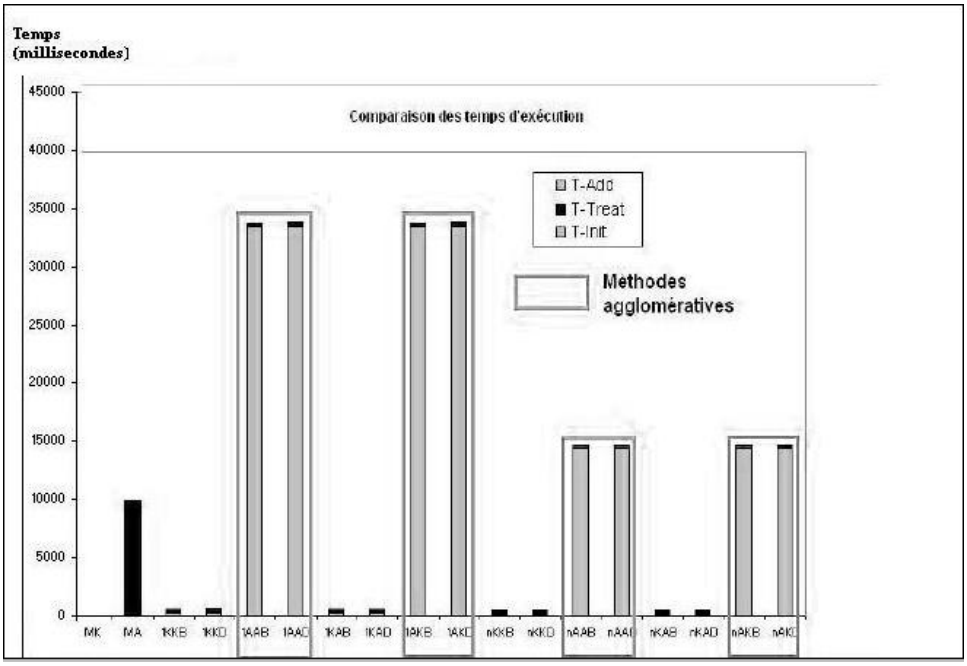


FIG. 3.18 – Mise en évidence des temps d'exécution pour les scénarios incluant le clustering agglomératif

La Figure 3.19 présente les temps d'exécution pour les versions du clustering progressif basé sur l'algorithme des K-Moyennes (MK, 1KY* et nKY*).

- Les versions 1KY* nécessitent des temps d'exécution 15 fois plus grands que la version MK, mais offre de grandes possibilités de parallélisation.

- Les versions nKY* nécessitent des temps d'exécution 6 fois plus grand que la version MK, mais offre également de grandes possibilités de parallélisation.

Les scénarios basés sur les versions macro-itératives apparaissent comme offrant de bons résultats :

- au vu des clusters produits, ce qui est le plus important ;

- mais également au vu des temps d'exécution, même en incluant le surcoût d'un clustering unidimensionnel (nécessaire pour obtenir de précieuses informations - voir Section 2.3.4), représenté par les zones rouges sur la Figure 3.20), avec de fortes possibilités de parallélisations à plusieurs niveaux : phase de clustering initiale macro-itérative aussi bien que unidimensionnelle si nécessaire, phase de croisement...). Ces versions nKK* (utilisant uniquement la méthode des K-Moyennes) sont celles qui sont retenues pour les utilisations ultérieures.

3.6.4 Tableaux récapitulatifs

La Table 3.14 présente un bilan des méthodes de clustering (Agglomérative ou K-Moyennes) aux deux étapes du clustering progressif (clustering initial et clustering de regroupement

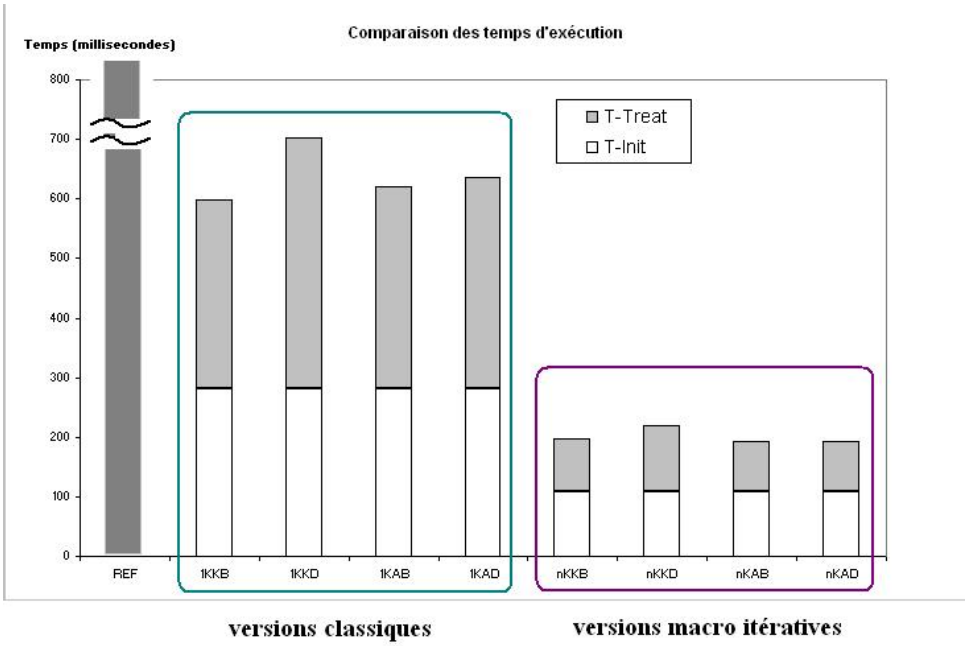


FIG. 3.19 – Comparaison des temps d'exécution pour les scénarios incluant le clustering par méthode des K-Moyennes

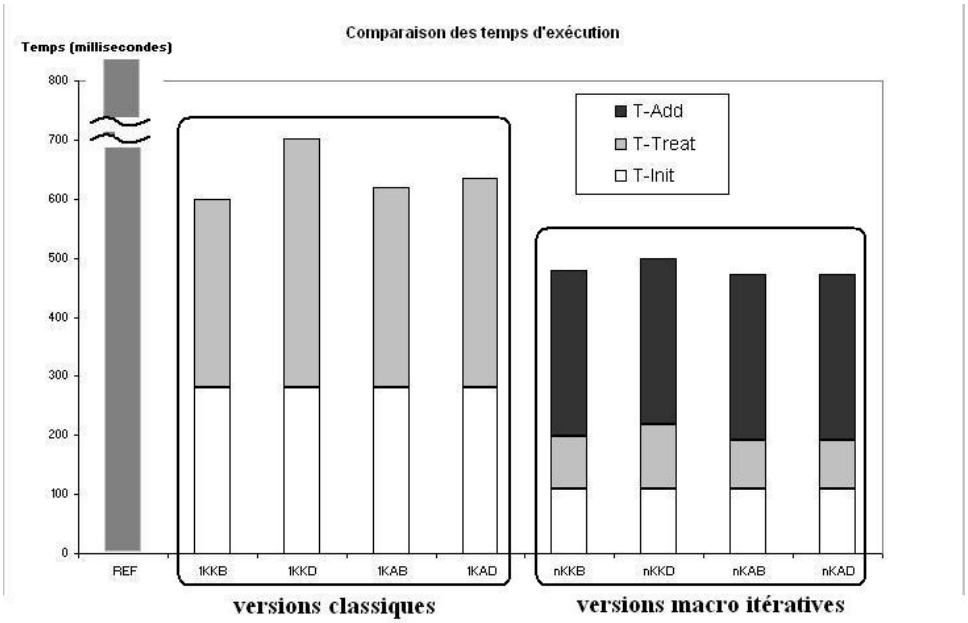


FIG. 3.20 – Comparaison des temps d'exécution pour les scénarios incluant le clustering par méthode des K-Moyennes avec ajout des temps additionnels de phase initiale unidimensionnelle

- voir Section 3.3.3).

Les versions utilisant le clustering agglomératif fournissent des résultats faux (voir Table 3.14), malgré le fait que les méthodes agglomératives soient moins volatiles a priori que les méthodes de type K-Moyennes de par la non-existence d'aléatoire dans leur fonctionnement, mais elles nécessitent des temps d'exécution inappropriés. En particulier, l'utilisation du clustering agglomératif aux deux niveaux de la méthode de clustering progressif (phase initiale et de regroupement) fournit les pires résultats par rapport à la complexité temporelle et à la qualité (un groupe unique est obtenu, ce qui peut être relié aux problèmes mis en évidence dans les travaux d'Agrawal (voir [5]).
Ceci peut être expliqué par le fait qu'une méthode stricte n'est pas appropriée au processus itératif du clustering progressif et confirme que la combinaison de méthodes heuristiques peut fournir de meilleurs résultats que la combinaison de méthodes "exactes", permettant une réelle diminution des temps d'exécution ainsi que l'obtention de résultats de bonne qualité.

	Clustering de regroupement			
	Agglomératif		K-Moyennes	
Clustering initial	Temps d'exécution	Résultats	Temps d'exécution	Résultats
Agglomératif	- - - - -	faux (groupe unique)	- - -	faux
K-Moyennes	+	faux	+ + +	**bon**

TAB. 3.14 – Bilan de l'utilisation de clustering agglomératif ou K-Moyennes

La Table 3.15 présente le bilan de l'utilisation de la méthode de clustering progressif ou du clustering classique.

La méthode de clustering progressif présentée dans ce chapitre apporte un gain de parallélisation (sans aucune synchronisation) et permet également de conserver la qualité des résultats. Les versions de clustering progressif avec utilisation de la méthode des K-Moyennes offrent de nombreuses possibilités de parallélisation qui permettent d'exploiter les ressources de traitement disponibles.

Version	Temps d'exécution séquentielle	Qualité des clusters	Possibilités de parallélisation
MA	- - - -	++	0
MK	+++	++	0
1KKZ	+	++	+++
nKKZ	++	++	+++

TAB. 3.15 – Bilan de l'utilisation du clustering progressif ou du clustering classique

Remarque : Des expérimentations complémentaires ont été réalisées sur Grid'5000 (voir Annexe D) pour évaluer l'évolution le facteur d'accélération de l'algorithme CDP en fonc-

tion du taux de parallélisme utilisé. Ces résultats sont présentés en Annexe B.

3.6.5 Pourquoi une version progressive basée entièrement sur le clustering agglomératif fournit les moins bons résultats en terme de qualité de groupes ?

La méthode de clustering agglomératif est une méthode rigoureuse qui prend à chaque étape la meilleure décision de regroupement. Une fois une décision prise, elle ne pourra plus être modifiée, on ne pourra pas revenir en arrière. Ainsi, lors des décisions prises sur chaque dimension (phase initiale de clustering), la méthode prend des décisions de regroupement des données sur cette dimension. Les décisions définitives sont transmises lors du croisement et influeront sur la phase de clustering de regroupement optimisé.

Pour la version basée sur la méthode des k-moyennes, les centres sont "ajustés" jusqu'à convergence pour chaque dimension, mais les décisions effectuées pourront être ajustées lors d'un clustering de regroupement en ajustant les centres de groupes par rapport aux données issues de plusieurs dimensions. La méthode permet ainsi de corriger des erreurs de décisions qui auraient été prises dans les dimensions précédentes, mais qui ne sont plus valables avec un plus grand nombre de dimensions. On a ici, la vérification que le regroupement de solutions optimales n'est pas nécessairement optimal et peut même conduire à des résultats inintéressants. La combinaison de plusieurs méthodes "heuristiques" (telle les k-moyennes) amène souvent à de meilleurs résultats que la combinaison de méthodes rigoureuses (qui ne laissent pas une part d'aléatoire dans les choix effectués).

3.7 Conclusions concernant le clustering parallèle progressif

Le clustering progressif fournit une bonne solution pour distribuer la base de données sur une grille de manière "intelligente". La méthode offre une bonne qualité des clusters obtenus, ainsi que des possibilités d'exécution parallèle et distribuée.
Elle peut ainsi être utilisée en tant que méthode **heuristique de clustering parallèle** qui produit de bons résultats au vu des gains de parallélisations offerts. Le clustering progressif apparaît donc comme une solution distribuée raisonnable pour résoudre le problème de clustering dans un environnement distribué puisque des algorithmes réellement distribués pour ce problème n'existent pas encore.

La méthode d'incrémentation (sur modèle arborescent statique ou arborescent dynamique par disponibilité voir Section 3.5.1) utilisée pour le clustering apparaît ne pas avoir une grande influence sur les résultats, puisque les résultats sont similaires pour chacune des deux méthodes (en utilisant un même algorithme pour le clustering initial et le clustering de croisement). Les tests ayant été réalisés sur une plateforme non perturbée (par des travaux externes) et en utilisant une répartition de charge équilibrée, le choix de la méthode

d'incrémentation n'influence pas les temps d'exécution dans les résultats présentés.
Cependant la méthode arborescente dynamique par disponibilité est plus adaptée aux spécificités d'une l'exécution distribuée. En effet elle permet de diminuer les temps d'attente inactive en permettant d'utiliser les résultats dès que possible plutôt qu'en suivant un ordre imposé.

Puisque le problème du clustering progressif est apparu dans la recherche d'une fragmentation intelligente de la base de données dans le problème de règles d'association, l'étape suivante de ces travaux consiste à incorporer cette méthode dans le schéma général DisDaMin (voir Section 2.1).
Le clustering progressif sera donc utilisé dans ce schéma général. Le Clustering Progressif peut alors prendre sa place en tant que phase de distribution dans le problème de génération des règles d'association, au lieu d'un clustering centralisé utilisé dans les premiers travaux (voir Section 2.3.1 et [27]) qui avaient permis de valider la théorie de diminution de la taille de l'espace de recherche visité à partir d'une distribution intelligente. Il permettra d'obtenir une diminution de la complexité et également un gain d'exécution de par l'utilisation du parallélisme. Les considérations distribuées de l'incrémentation seront dès lors liées à la répartition initiale de la multibase. La phase de clustering progressif devra s'inclure dans un mode de "pipeline" complet sur le schéma général en fonction du mode d'incrémentation utilisé.

Troisième partie

Recherche Distribuée de Règles d'Association

Recherche Distribuée de Règles d'Association dans DisDaMin

Une fois qu'une fragmentation "intelligente" est obtenue par clustering (voir Section 2.1 et Partie II), le schéma général présenté pour la recherche de règles d'association (voir Section 2.4) nécessite le traitement parallèle des fragments obtenus de manière la plus indépendante possible ou tout du moins asynchrone pour respecter les contraintes de la plateforme d'exécution.
Nous présenterons tout d'abord les algorithmes de recherche de règles d'association existants (en particulier l'algorithme de référence dans ce domaine : APriori, ainsi que l'algorithme DIC qui a inspiré nos travaux), puis les versions parallèles de ces algorithmes et leurs inadéquations à l'exploitation sur grille. Nous introduirons ensuite notre méthode basée sur une distribution intelligente par clustering, l'algorithme DICCoop et son adaptation à grande échelle, DICCoop Méta Version, avant de présenter les résultats d'expérimentations de l'algorithme.

4.1 Rappel du problème de la Recherche de Règles d'Association

Nous cherchons à identifier des règles de type : {lait, bière} $=>$ {fromage} (voir Section 1.4).

Le problème de recherche de règles d'association se décompose en deux étapes. Tout d'abord la recherche des itemsets fréquents (voir Section 1.4.2), puis à partir de ceux-ci, la génération de règles à proprement parler (voir Section 1.4.3).

4.1.1 Rappels des définitions et notations

On appelle **itemset** un ensemble d'items ; le nombre d'items d'un itemset constitue sa longueur (un itemset contenant k items est appelé un k-itemset).
Chaque instance d'une base D est un ensemble d'items, c'est donc un itemset.

La recherche de règles d'association consiste à produire des règles de dépendance, des relations entre les items d'une base D, et ceci afin de prédire l'occurrence d'autres items.

Soit **I={**i_1**,** i_2**, ...,** i_m**}** un ensemble de ***m*** items et soit **D** un ensemble d'instances (constituées des items i_i, i_j, ..., i_k de **I**).

Une règle d'association est une implication de la forme **X =>** **Y**, où X et Y sont inclus dans I et X ∩ Y= Ø[1].

- X est la condition ou l'antécédent,
- Y est la conclusion ou la conséquence.

Nous utilisons deux mesures liées aux règles d'association :

- Le **support** d'un itemset : à chaque itemset est associé un support défini comme étant le pourcentage d'instances de D qui contiennent cet itemset.

- La **confiance** d'une règle d'association : à chaque règle d'association est associée une mesure confiance définie par :
  $confiance(X \rightarrow Y) = \frac{support(X \cup Y)}{support(X)}$.

La recherche de règles d'association consiste à trouver les règles de support et de confiance (ou autre mesure) supérieurs à certains seuils fixés au départ.

Notations :

On désignera par **k-itemset** u un itemset de longueur k, auquel on associera un support $support(u)$.

4.1.2 Les algorithmes existants

Le plus connu des algorithmes de recherche de règles d'association est l'algorithme **Apriori** (Agrawal et al. [3], [60]).
La plupart des algorithmes existants se basant sur Apriori, nous avons décidé de présenter celui-ci en détails.
L'algorithme DIC sera également détaillé car, ayant inspiré la méthode DICCoop, il sera utilisé dans la suite.
Les grandes lignes des autres algorithmes sont présentées ainsi que les versions parallèles existantes.

Notations

[1] Formulation du problème proposée par Agrawal et al. [1] et [3]

On considérera les notations suivantes :

C_k désigne l'ensemble des k-itemsets **candidats** (i. e. potentiellement fréquents).

F_k désigne l'ensemble des k-itemsets fréquents.

L'algorithme Apriori (Agrawal et Srikant [3], [60])

L'algorithme A Priori procède de **manière itérative** pour identifier les $k-$itemsets fréquents, c'est à dire que l'ensemble des itemsets fréquents est parcouru par niveau (parcours en largeur du haut vers le bas du treillis d'itemsets). Lors de la k^{eme} itération, on recherche les k-itemsets fréquents, c'est-à dire les itemsets de longueur k qui sont fréquents. A l'itération suivante, la $(k+1)^{eme}$, on recherchera les (k+1)-itemsets fréquents et ainsi de suite.

Pour chaque itération k, on génére les k-itemsets candidats (à être fréquents) C_k à partir de l'ensemble des itemsets fréquents identifiés à l'itération Précédente F_{k-1}.
On scanne les données (balayage) afin de calculer le support de chaque itemset candidat c ($c \in C_k$).
Puis les éléments c de C_k ayant un support suffisant (les éléments c de C_k fréquents) sont ajoutés à l'ensemble des itemsets fréquents F_k (voir Table 4.1).

Durant la première itération de l'algorithme (Table 4.1, ligne 1), tous les itemsets de taille 1 sont considérés, et un balayage est réalisé pour déterminer l'ensemble des 1-itemsets fréquents F_1.
Chaque itération k suivante (Table 4.1, lignes 2 à 8) se subdivise en deux phases. Durant la première phase (Table 4.1, ligne 3), l'ensemble C_k des k-itemsets candidats est construit par association des $(k-1)-$itemsets fréquents de F_{k-1}. Cette phase est réalisée par la fonction Apriori-Gen (voir Table 4.2).
Durant la deuxième phase (Table 4.1, lignes 4 à 8), un balayage des données est réalisé afin de déterminer le support de chacun des $k-$itemsets candidats de C_k. Les $k-$itemsets fréquents sont insérés dans F_k.

Remarque : L'opération consistant à ne pas introduire les k-itemsets infréquents dans F_k, et donc à ne pas les prendre en compte pour la suite du traitement, est appelée **pruning** ou **élagage** .

L'algorithme APrioriGen(F_{k-1})

La procédure AprioriGen reçoit un ensemble F_{k-1} de $(k-1)-$itemsets fréquents comme paramètre. Elle retourne un ensemble C_k de $k-$itemsets candidats.
Les éléments de C_k sont générés par combinaison des éléments de F_{k-1}.

---

l'algorithme Apriori :
Entrée : la base, le seuil de support minimal
Sortie : l'ensemble F_k des k-itemsets fréquents
(1) $F_1 =$ 1-itemsets fréquents
(2) **pour**($k = 2; F_{k-1} \neq 0; k++$)**faire**
(3) C_k = AprioriGen(F_{k-1})
(4) **pour chaque** instance lue $inst$ **faire**
(5) **pour** chaque élément c de C_k tel que $c \in inst$ **faire**
(6) $c.support++$
(7) **fin pour**
(8) **fin pour**
(9) $F_k = \{c \in C_k \| c.support \geq minsupport\}$
(10) **fin pour**
(11) **retourner** $\cup F_k$

TAB. 4.1 – Principe de l'algorihme Apriori

Propriété 1 : Tous les sous-ensembles d'un itemset fréquent sont fréquents.
(voir Section 1.5.1)

La procédure se décompose en deux étapes, la première visant à générer des candidats, la deuxième visant à en éliminer une partie.
Durant une première partie de la procédure, les $(k-1)-$itemsets fréquents de F_{k-1} sont combinés, et les résultats de ces combinaisons sont insérés dans C_k.
Durant la deuxième partie, les éléments de C_k dont l'un des sous-ensembles est infréquent sont supprimés de C_k (**élagage** de C_k).

Au départ (voir Table 4.2, lignes 1 à 4), on effectue des combinaisons de deux $(k-1)-$itemsets fréquents p et q, ayant $k-2$ items communs. Il en découle un $k-$itemset c dont les de $k-2_{eme}$ premiers items sont les items communs à p et q, et donc les $k-1_{eme}-$ et $k_{eme}-$ items sont les items distincts de p et q. c est donc un candidat à être fréquent, il est ajouté à l'ensemble itemsets candidats que l'on est en train de construire, C_k.
Une fois l'ensemble C_k des $k-$itemsets candidats générés par combinaison des $(k-1)-$itemsets fréquents (F_{k-1}), on supprime une partie de ces candidats (Table 4.2 lignes 5 à 9) en se basant sur la **Propriété 1 : Tout sous-ensemble d'un ensemble fréquent doit être fréquent**, (donc si un ensemble à un de ses sous-ensembles infréquent, il est infréquent, **Propriété 2**).
Ainsi, tous les éléments c de C_k dont l'un des sous-ensembles s n'est pas fréquent ($s \notin F_{k-1}$) sont supprimés de C_k, soit un élagage de C_k (voir Table 4.2).

Propriété 2 : Tous les sur-ensembles d'un itemset infréquent sont infréquents.

(voir Section 1.5.1)

---

l'algorithme AprioriGen :
Entrée : l'ensemble F_{k-1} des (k-1)-itemsets fréquents
Sortie : l'ensemble C_k des k-itemsets candidats
(1) insérer dans C_k
(2) *select p[1], p[2], ..., p[k-1], q[k-1]*
(3) *from* F_{k-1} *p ,* F_{k-1} *q*
(4) *where p[1] = q[1],..., p[k-2]= q[k-2], p[k-1] < q[k-1]*
(5) **pour** chaque itemset candidat c de C_k **faire**
(6) **pour** chaque sous-ensemble s de c tel que $\|s\| = k-1$ **faire**
(7) **si** $s \notin F_{k-1}$ alors supprimer c de C_k
(8) **fin pour**
(9) **fin pour**
(10) **retourne** C_k

TAB. 4.2 – La procédure AprioriGen

Exemple d'exécution de l'algorithme APriori sur une base D, pour un support minium de $2/6$:

Soit le base D telle que représentée Table 4.3.

Base D

Identifiant de l'instance	Items contenus
1	A C D
2	B C E
3	A B C E
4	B E
5	A B C E
6	B C E

TAB. 4.3 – Exemple d'exécution APriori : la base D

On identifie tout d'abord l'ensemble F_1 des $1-$itemsets fréquents.
L'ensemble C_1 des $1-$itemsets candidats est construit à partir de chaque item contenu dans D (il y a autant de $1-$itemsets candidats que d'items apparaissant dans D), soit $C_1 = \{A, B, C, D, E\}$. Les supports de chacun de ces $1-$itemsets sont calculés (voir Table 4.4) et les $1-$itemsets fréquents (dont le support est supérieur à $2/6$) sont insérés dans F_1 (voir Table 4.4). .
On obtient donc F_1 ={{A}, {B}, {C}, {E}} ({D} n'est pas fréquent voir Table 4.4).

C_1 (et supports associés)

1-itemsets	Support
⟨ A ⟩	3/6
⟨ B ⟩	5/6
⟨ C ⟩	5/6
⟨ D ⟩	1/6
⟨ E ⟩	5/6

→

F_1

1-itemsets
⟨ A ⟩
⟨ B ⟩
⟨ C ⟩
⟨ E ⟩

TAB. 4.4 – Exemple d'exécution APriori : les ensembles C_1 et F_1

C_2 (et supports associés)

2-itemsets	Support
⟨ A B ⟩	2/6
⟨ A C ⟩	3/6
⟨ A E ⟩	2/6
⟨ B C ⟩	4/6
⟨ B E ⟩	5/6
⟨ C E ⟩	4/6

→

F_2

2-itemsets
⟨ A B ⟩
⟨ A C ⟩
⟨ A E ⟩
⟨ B C ⟩
⟨ B E ⟩
⟨ C E ⟩

TAB. 4.5 – Exemple d'exécution APriori : les ensembles C_2 et F_2

A partir de F_1, on génére les $2-$itemsets candidats de C_2 (voir Table 4.5).
On calcule leurs supports, puis on met dans F_2 les $2-$itemsets fréquents de C_2.
Dans l'exemple (voir Table 4.5), tous les candidats contenus dans C_2 sont fréquents, on obtient donc F_2 ={{A B}, {A C}, {A E}, {B C}, {B E}, {C E}}($= C_2$).

C_3 (et supports associés)

3-itemsets	Support
⟨ A B C ⟩	2/6
⟨ A B E ⟩	3/6
⟨ A C E ⟩	2/6
⟨ B C E ⟩	4/6

→

F_3

3-itemsets
⟨ A B C ⟩
⟨ A B E ⟩
⟨ A C E ⟩
⟨ B C E ⟩

TAB. 4.6 – Exemple d'exécution APriori : les ensembles C_3 et F_3

A partir de F_2, on génére les $3-$itemsets candidats de C_3 (voir Table 4.6).
On calcule leurs supports, puis on met dans F_3 les $3-$itemsets fréquents de C_3.
Dans l'exemple(voir Table 4.6), tous les candidats contenus dans C_3 sont fréquents, on obtient donc F_3 ={{A B C}, {A B E}, {A C E}, {B C E}}($= C_3$).

A partir de F_3, on génére les $4-$itemsets candidats de C_4 (voir Table 4.7).
On calcule leurs supports, puis on met dans F_4 les $4-$itemsets fréquents de C_4.
Dans l'exemple (voir Table 4.7).
, C_4 ne contient qu'un élément et il est fréquent, on a donc F_4 ={{A B C E}}($= C_4$).

C_4 (et supports associés)			F_4
4-itemsets	Support	→	4-itemsets
⟨ A B C E ⟩	2/6		⟨ A B C E ⟩

TAB. 4.7 – Exemple d'exécution APriori : les ensembles C_4 et F_4

A partir de F_4, on génére les $5-$itemsets candidats de C_5.
Il se trouve que C_5 est vide, l'algorithme s'arrête.

Améliorations proposées à Apriori :

- **DHP** (Direct Hashing and Pruning) proposé par Park et al. [52].
  DHP est une légère modification de Apriori qui utilise des tables de hachage pour diminuer le nombre d'itemsets candidats générés.
  Cet algorithme propose également une diminution de la taille de la base de données, au fur et à mesure des itérations.
  Lors des deux premières itérations (gestion des $1-$itemsets et des $2-$itemsets), on utilise une table de hachage sur les numéros des itemsets [2]. A chacune des cases de la table de hachage, on associe un compteur. A chaque fois qu'un instance contient l'un des itemsets associés (par la fonction de hachage) à une case de la table de hachage, on incrémente le compteur de cette case.
  On garde les itemsets associés aux cases dont le compteur est suffisant (pour une case associée à 3 itemsets, on garde les itemsets si le compteur associé à la case possède une valeur supérieure à trois fois le seuil de support fixé).
  A partir de la deuxième itération, on effectue un pruning (voir page 104) des instances de la base. Pour chaque instance, si le nombre de bits de cette instance est supérieur à k, l'instance peut être utile à la k^{eme} itération, sinon, on la supprime.

- **AprioriTid** proposé par Agrawal et Srikant [3].
  Cet algorithme utilise des listes d'itemsets associées aux identifiants des objets en relation avec les itemsets, afin de diminuer progressivement la taille du jeu de données. Il permet de diminuer progressivement la taille de la base, dans le but de la stocker en mémoire et de ne plus réaliser d'opérations d'entrée-sortie, après le premier balayage.
  A partir de la deuxième itération, la procédure AprioriGen est utilisée pour générer les itemsets candidats de l'itération suivante dans un ensemble $\overline{C_k}$. Chaque élément de l'ensemble $\overline{C_k}$ est un couple (ID, c_k) dans lequel c_k est la liste des k-itemsets candidats contenus dans l'instance d'identifiant ID. Le support d'un itemset candidat c_k correspond au nombre d'apparitions de c_k dans $\overline{C_k}$.

[2] Cela suppose d'avoir au préalable associé un numéro à chacun des itemsets possibles pour l'itération concernée

- **AprioriHybrid** proposé par Agrawal et Srikant [3].
  AprioriHybrid constitue un mélange de Apriori et AprioriTid en fonction de l'itération à laquelle on se trouve.

- **Partition** proposé par Savasere et al. [59].
  Cet algorithme ne nécessite que deux balayages du jeu de données.
  Durant le premier balayage, la base est divisée en n partitions disjointes qui seront considérées une à une successivement. Pour chaque partition, l'ensemble des itemsets fréquents dans la partition (les itemsets fréquents locaux) sont extraits. Les ensembles d'itemsets fréquents de chaque partition sont ensuite fusionnés pour obtenir un ensemble d'itemsets candidats à être fréquents sur la base entière. On a ainsi un sur-ensemble de l'ensemble des itemsets fréquents.
  Durant le second balayage les supports pour ces candidats sont calculés sur toute la base, les itemsets fréquents sont identifiés.
  La taille de la partition est choisie de telle façon que chaque partition tienne en mémoire.

- **MaxEclat / MaxClique** proposé par Zaki et al. [64] et [65] .
  A chaque itemset on associe une tid-liste (liste des identifiants des instances qui contiennent cet itemset). Ces algorithmes effectuent ensuite des jointures sur les tids-listes des instances associés à chaque itemset. Le nombre d'instances contenus dans une tid-liste est le support de l'itemset associé à cette tid-liste.

- **Sampling** proposé par Toivonen [62].
  Cet algorithme effectue la génération des itemsets sur un échantillon de la base. Leurs supports sont ensuite calculés sur la base entière. Au fur et à mesure du parcours de la base, on affine les résultats (la valeur des supports). On supprime des itemsets qui se révèlent non fréquents.
  Cet algorithme suppose de disposer d'un échantillon représentatif de la base !

- **DIC** (Dynamic Itemsets Counting) proposé par Brin et al. [14].
  Cet algorithme propose une diminution du nombre de balayages nécessaires du jeu de données. L'algorithme DIC est basé sur une méthode similaire a Apriori. Le jeu de données est décomposé en sous-ensembles de taille donnée, et durant chaque itération, un sous-contexte (i. e. un sous-ensemble d'instances) est lu. Ainsi, plusieurs ensembles d'itemsets candidats de tailles différentes sont traités simultanément durant chaque itération, ce qui permet de diminuer le nombre total de balayages du contexte.
  L'algorithme utilise un principe de fenêtrage de la base (on forme des blocs de M instances). On affecte un type aux itemsets : fréquent confirmé, fréquent potentiel, infréquent confirmé, infréquent potentiel.
  Le parcours du premier bloc permet d'estimer les fréquences des 1-itemsets. Lorsqu'on a parcouru tout ce premier bloc, on génére les 2-itemsets candidats à partir des estimations de fréquences réalisées pour les 1-itemsets. Le parcours du deuxième bloc permet d'améliorer les estimations des fréquences des $1-$itemsets et de débuter l'estimation des fréquences des $2-$itemsets...

Caractéristiques de l'Algorithme DIC (Brin et al. [14])

L'algorithme DIC (Dynamic Itemsets Couting) permet d'améliorer la phase de recherche des itemsets fréquents en permettant une diminution du nombre de parcours nécessaires du jeu de données. Il utilise deux principes :

- un fenêtrage de la base (utilisation de blocs de M instances de manière itérative plutôt que d'utiliser tous les instances) permettant de diminuer le nombre de passage sur les données.
- un marquage des itemsets permettant de connaître la catégorie à laquelle ils appartiennent : fréquent confirmé, fréquent potentiel, infréquent confirmé, infréquent potentiel (voir plus loin).

Le principe de fenêtrage On définit une taille M de fenêtre. A chaque itération, on effectue le parcours des données d'un bloc (une fenêtre). Lorsque toutes les instances du jeu de données ont été parcourues (tous les blocs ont été pris en compte, on a effectué un balayage complet du jeu de données), on recommence les lectures de blocs à partir du début du jeu de données. Après chaque lecture de bloc, on effectue des conclusions partielles avec modification des marqueurs (voir plus bas) et génération de nouveaux itemsets candidats à traiter lors de l'itération suivante.

(**Remarque :** Apriori correspond à DIC pour une fenêtre de taille n : le nombre d'instances de la base).

Le principe de marquage A chaque itemset on va associer un marqueur en fonction des informations connues pour l'itemset.
Ces marqueurs sont constitués de deux critères :

- la fréquence ou l'infréquence des itemsets, liée aux supports et au seuil fixé pour la fréquence ;
- la validité totale ou partielle du support, selon que l'ensemble des blocs ait été parcouru ou seulement une partie.

Pour la fréquence on identifie ainsi les deux catégories principales d'itemsets (également utilisé dans les autres algorithmes) :

- les itemsets fréquents : ils seront alors représentés par des **boites** (**B** pour Box) Voir Figures 4.1, 4.2, 4.3, 4.4 et 4.5 ;
- les itemsets non fréquents : ils seront représentés par des **cercles** (**C** pour Circle) Voir Figures 4.1, 4.2, 4.3, 4.4 et 4.5.

En fonction du parcours des blocs, les résultats de fréquence/infréquence sont partielles ou totales. On identifie ainsi :

- les itemsets d'informations partielles : ils seront alors représentés par des traits en **pointillés** (**D** pour Dashed) Voir Figures 4.1, 4.2, 4.3, 4.4 et 4.5 ;

- les itemsets d'informations totales : ils seront représentés par des traits **pleins** (**S** pour Solid) Voir Figures 4.1, 4.2, 4.3, 4.4 et 4.5.

En associant ces deux types d'information, on a ainsi quatre marqueurs pour les itemsets dans l'algorithme DIC :

- **SB** - Solid Box : pour les itemsets **avérés fréquents** sur la totalité des instances (après prise en compte de tous les blocs) ;

- **SC** - Solid Circle : pour les itemsets **avérés infréquents** sur la totalité des instances (après prise en compte de tous les blocs) ;

- **DB** - Dashed Box : pour les itemsets **fréquents "incomplets"** sur une partie des instances (après prise en compte d'un sous-ensemble de blocs) ;

- **DC** - Dashed Circle : pour les itemsets **infréquents "incomplets"** sur une partie des instances (après prise en compte d'un sous-ensemble de blocs).

Extraction des itemsets fréquents avec l'algorithme DIC

1. L'ensemble d'itemsets vide est marqué **SB**. Tous les 1-itemsets sont marqués **DC**. Les autres itemsets ne sont pas encore générés et peuvent donc être considérés comme non-marqués (voir Figure 4.2).

2. M instances sont lues et les supports des itemsets marqués **Dashed** (DC ou DB) sont incrémentés.

3. Si un itemset marqué **DC** a un support supérieur au seuil fixé, modifié son marqueur en **DB**.
   Si l'un des sur-ensemble direct de celui-ci a tous ces sous-ensembles marqués **SB** ou **DB** le marquer **DC** et lui ajouter un compteur (voir Figures 4.3 et 4.4).

4. Si un itemset marqué **Dashed** (DC ou DB) a été traité sur l'ensemble des instances, le marqueur **SC** et stopper son comptage.

5. Si la fin de la base de données a été atteinte "rembobiner" jusqu'à la première instance 4.5.

6. Si il reste des itemsets marqués **Dashed** (DC ou DB) retourner à l'étape 2.

La Table 4.8 présente le pseudo-code de la méthode.

Exemple d'exécution de l'algorithme DIC :

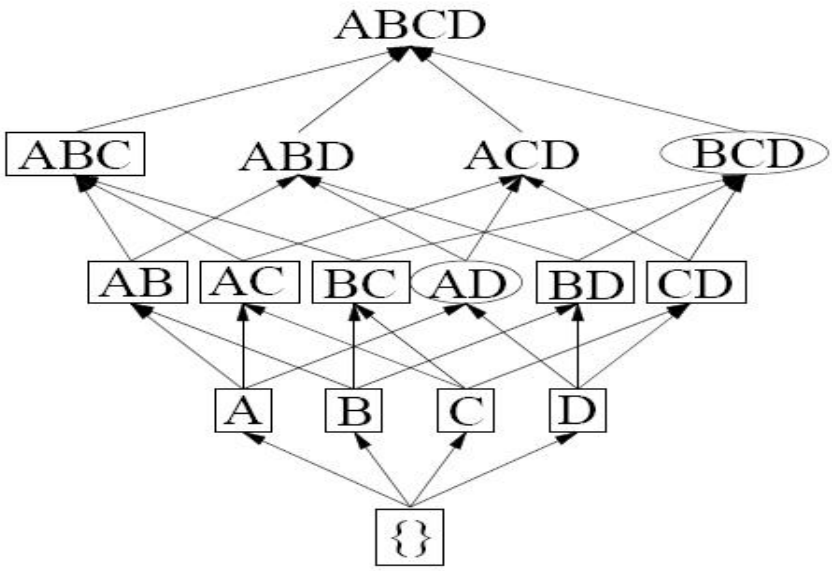


FIG. 4.1 – Exemple de parcours chaotique par l'algorithme DIC.

En utilisant un jeu de données menant à l'identification des itemsets fréquents : A, B, C, D, AB, BC, BD, CD, ABC (voir Figure 4.1).

L'étape 1 consiste à marquer tous les $1-$itemsets **DC** (voir Figure 4.2) .
A la première itération (lecture d'un bloc d'instances), l'étape 3 amène au marquage des 1-itemsets **A, B, C, D** avec **DB** (leur support sur le bloc lu est suffisant), et donc à la génération des 2-itemsets candidats **AB, AC, BC, AD, BD, CD** marqués **DC** (voir Figure 4.3).

A la deuxième itération, l'étape 3 amène au marquage des $2-$itemsets **AB, AC, BC, BD, CD** avec **DB** (leur support sur le bloc lu est suffisant), le 2-itemset **AD** reste marqué **DC**. On génére donc les $3-$itemsets candidats **ABC, BCD** marqué **DC** (voir Figure 4.4).

A la fin du parcours des instances le comptage des $1-$itemsets est terminé (il a commencé au début du parcours des instances et elles ont toutes été parcourues, le support est donc définitif et complet), ils sont marqués **SB** (voir Figure 4.5).
Les autres itemsets restants sont toujours marqués **DX** (l'étape 6 est valide) , et donc un

l'algorithme DIC :
Entrée : la base, le seuil de support minimal, la taille de fenêtre
Sortie : l'ensemble F_k des k-itemsets fréquents

(1) $C_1 \leftarrow$ tous les 1-itemsets marqués DC
Lecture d'un bloc de M instances
(2) **pour chaque** instance lue $inst$ **faire**
(3) **pour** chaque élément c de C_k
tel que $c \in inst$ et $c.marqueur = DCouDB$ **faire**
(4) $c.support++$
(5) **fin pour**
(6) **fin pour**
Création de candidats
(7) **pour** chaque élément c de C_k ayant été **modifié**,
tel que $c.marqueur = DC$ et $c.support \geq minsupport$ **faire**
(8) $c.marqueur \leftarrow DB$
(9) **pour** chaque sur-ensemble x de c
tel que $|x| = |c| + 1$ et $x \notin C_{|x|}$ **faire**
(10) **si** $\forall s$ sous-ensemble de x de taille $|x| - 1$, $s.marqueur = DBouSB$ **alors**
(11) **insérer** x dans $C_{|x|}$ avec $x.marqueur = DC$
(12) **fin pour**
(13) **fin pour**
Mise à jour des marqueurs
(14)**pour** chaque élément c de C tel que $c.marqueur = DCouDB$ **faire**
(15) c.nombrelu ← c.nombrelu + M
(16) **si** ($c.nombrelu = |D|$) **alors**
(17) **si** ($c.support \geq minsupport$) **alors**
(18) $c.marqueur \leftarrow SB$
(19) **sinon**
(19) $c.marqueur \leftarrow SC$
(21) **finsi**
(22)**fin pour**
Test d'arrêt
(23)**si** (∃ c ∈ C c.marqueur = SB ou SC) **alors** aller ligne 2
(24)$F_k \leftarrow \{c \in C_k c.marqueur = SB\}$
(25)**retourner** $\cup F_k$

TAB. 4.8 – Principe de l'algorihme DIC

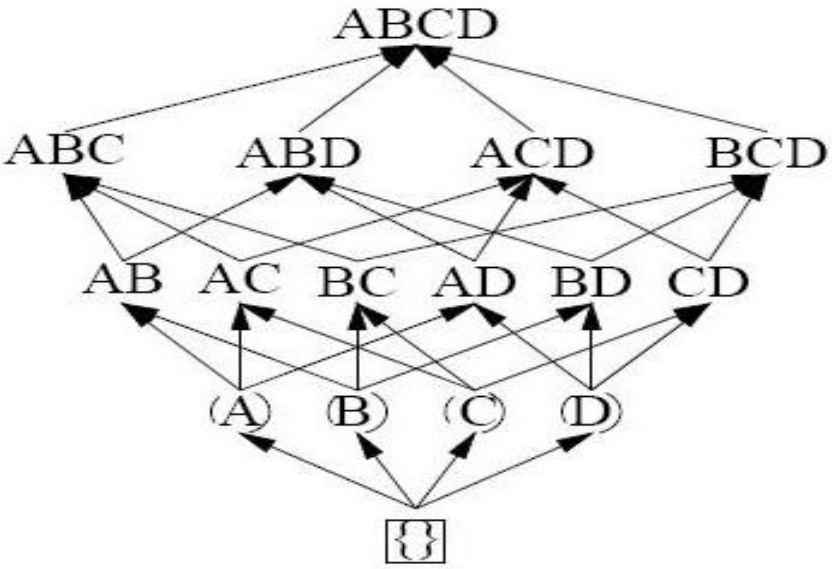


FIG. 4.2 – Première étape de l'algorithme DIC.

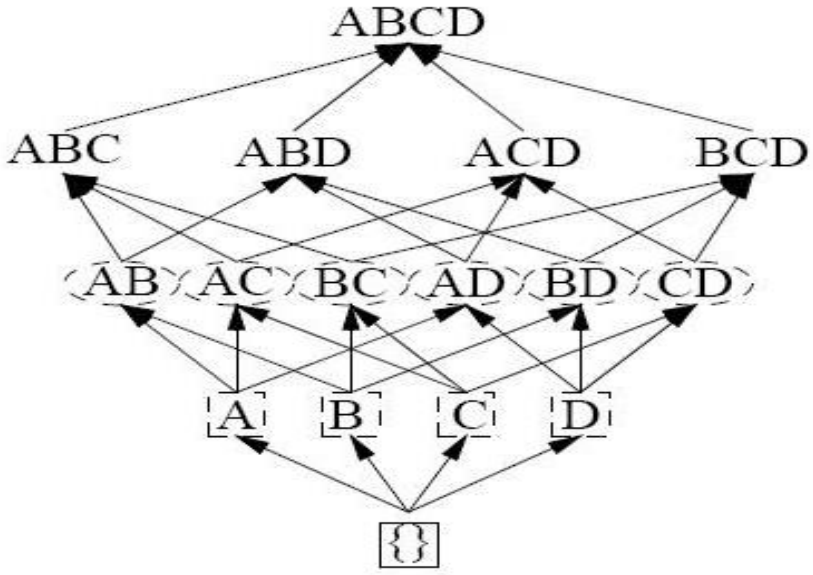


FIG. 4.3 – Algorithme DIC, après la lecture de M instances

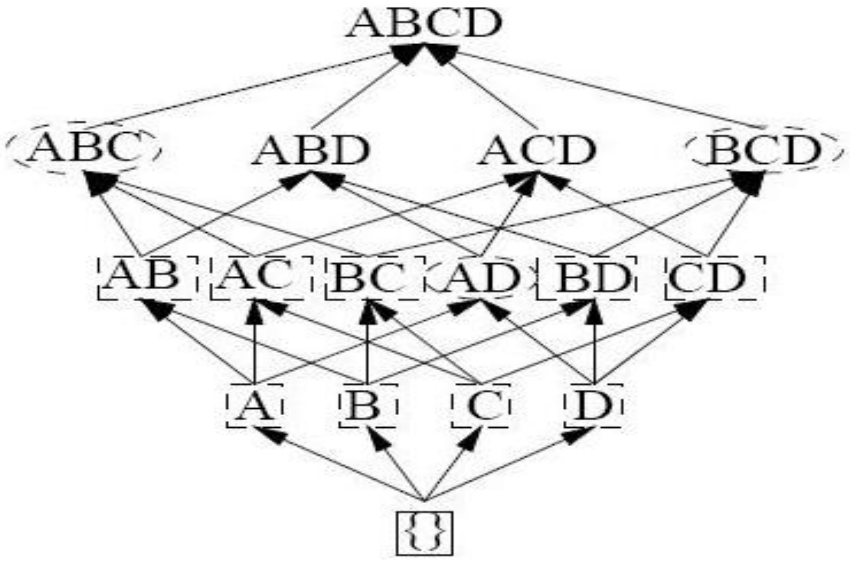


FIG. 4.4 – Algorithme DIC, après la lecture de 2M instances

"rembobinage" de la base est nécessaire avec de nouvelles itérations.

Dans l'algorithme Apriori, la première itération nécessite le passage sur toutes les instances pour permettre le traitement des 1-itemsets, la deuxième itération (passage sur toutes les instances) permet le traitement des $2-$itemsets et ainsi de suite (voir Figure 4.6).

Dans l'algorithme DIC, la première itération nécessite le passage sur seulement M instances pour permettre dès la deuxième itération le traitement d'une partie des 2-itemsets. Dès la troisième itération, après le passage sur seulement 2M instances, le traitement de 3 itemsets peut démarrer et ainsi de suite (voir Figure 4.7).

La méthode permet ainsi une anticipation par rapport à l'algorithme Apriori et donc une diminution du nombre de passage nécessaire sur les instances.
Le traitement complet des 3-itemsets qui nécessitent 3 passages sur les données dans l'algorithme Apriori, peut être effectué dans le meilleur des cas en seulement un passage et demi (si le traitement des 3-itemsets a pu être démarré dès le parcours du troisième bloc - 3ème itération).

Exemple 2 d'exécution de l'algorithme DIC :

Soit la base de données D représentée Table 4.9, et en considérant un seuil de support de

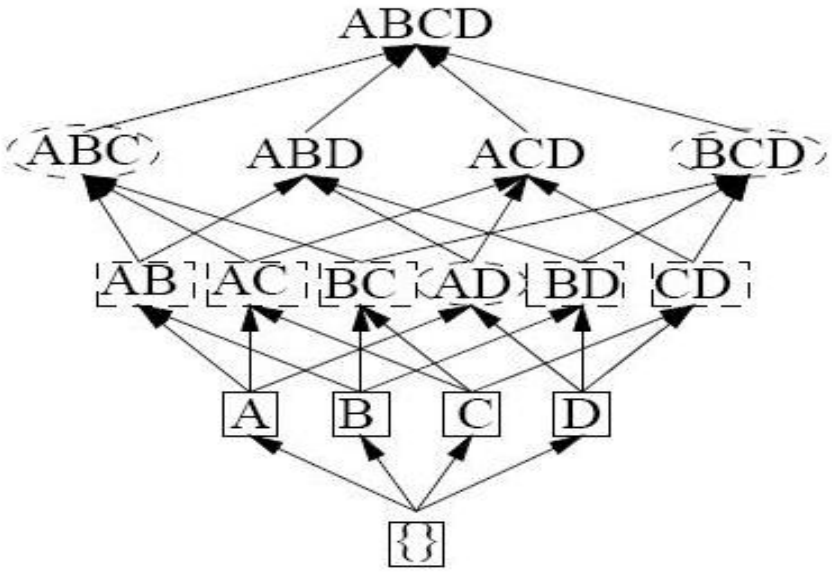


FIG. 4.5 – Algorithme DIC, après la lecture de toutes les instances

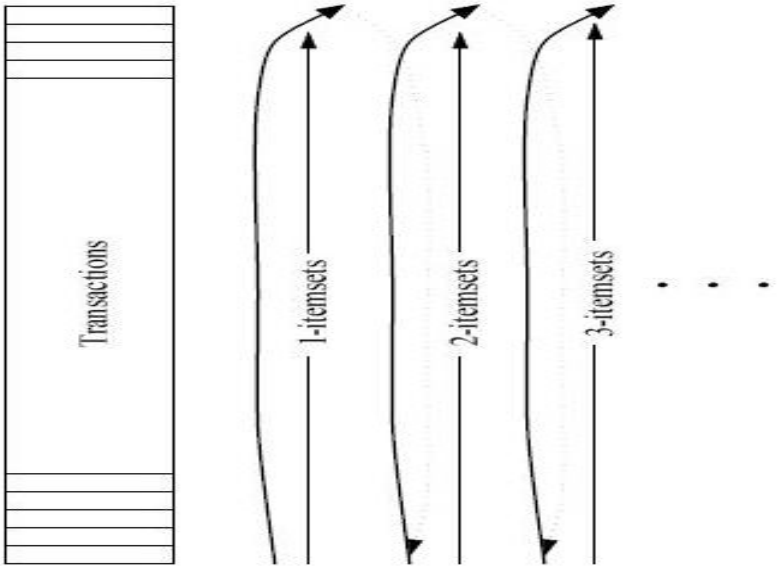


FIG. 4.6 – Parcours des données dans l'algorithme Apriori.

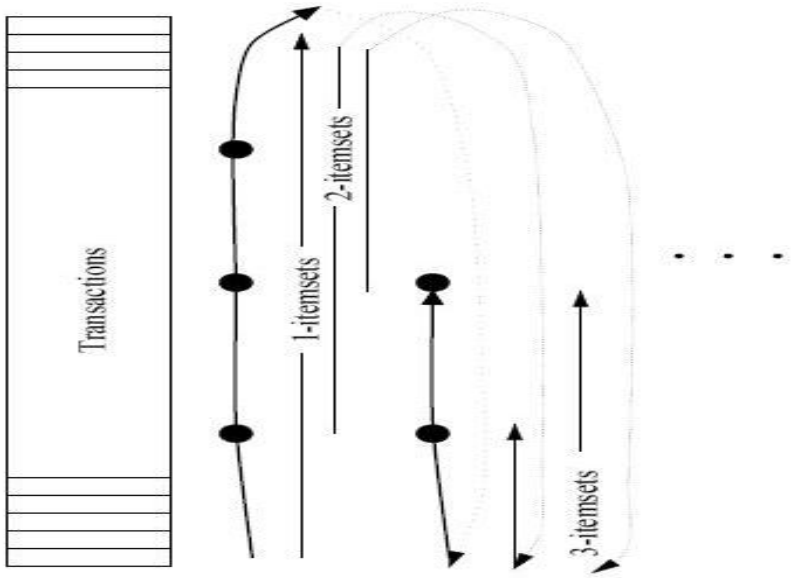


FIG. 4.7 – Parcours des données dans l'algorithme DIC.

$2/6$ et une fenêtre de taille3.

Base D

Tid [3]	Items contenus
1	A C D
2	B C E
3	A B C E
4	B E
5	A B C E
6	B C E

TAB. 4.9 – Exemple d'exécution DIC : la base D

Les Tables 4.10 et 4.11 présentent les étapes d'exécution de l'algorithme DIC sur D.

Initialement, tous les $1-$itemsets sont marqués DC.

Après la première itération de l'algorithme (passage sur les instances 1, 2, 3), 4 $1-$itemsets ({A}, {B}, {C}, {E}) ont un support suffisant pour être fréquents et changent donc de marqueurs pour passer de DC à DB.
Le dernier $1-$itemset ({D}) reste marqué DC.
6 $2-$itemsets ({AB}, {AC}, {AE}, {BC}, {BE}, {CE}) sont générés à partir des $1-$itemsets nouvellement marqués DB, les $2-$itemsets nouvellement générés sont marqués DC.

Après la deuxième itération de l'algorithme (passage sur les instances 4, 5, 6), tous les $1-$itemsets ont été entièrement comptés (on ne les considerera plus dans les itérations suivantes). 4 d'entre eux ({A}, {B}, {C}, {E}) passent du marqueur DB à SB, et le 5_{eme} ({D}) passe du marqueur DC à SC (son support est toujours insuffisant pour qu'il soit fréquent).
3 $2-$itemsets ({BC}, {BE}, {CE}) ont un support suffisant pour être fréquents et changent donc de marqueurs pour passer de DC à DB.
Les autres $2-$itemsets ({AB}, {AC}, {AE}) restent marqués DC.
1 $3-$itemset ({BCE}) est généré à partir des $2-$itemsets nouvellement marqués DB, ce $3-$itemsets est marqué DC.

Après la troisième itération de l'algorithme (de nouveau passage sur les instances 1, 2, 3), tous les $2-$itemsets ont été entièrement comptés (on ne les considerera plus dans les itérations suivantes). 3 d'entre eux ({BC}, {BE}, {CE}) passent du marqueur DB à SB.
Les autres $2-$itemsets ({AB}, {AC}, {AE}) passent du marqueur DC à SB (leurs supports sont suffisants pour qu'ils soient fréquents).
L'unique $3-$itemsets {BCE} a un support suffisant pour être fréquent et change donc de marqueur pour passer de DC à DB.
3 $3-$itemset ({ABC}, {ABE}, {ACE}) sont générés à partir des $2-$itemsets marqués SB, ces $3-$itemsets sont marqués DC.

Après la quatrième itération de l'algorithme (passage sur les instances 4, 5, 6), 1 $3-$itemset ({BCE}) a été entièrement compté (on ne le considerera plus dans les itérations suivantes). Il passe du marqueur DB à SB.
Les 3 autres $3-$itemsets ({ABC}, {ABE}, {ACE}) restent marqués DC (leur supports sont toujours insuffisants pour qu'ils soient fréquents).
Aucun itemset n'est généré.

Après la cinquième itération de l'algorithme (de nouveau passage sur les instances 1, 2, 3), tous les $3-$itemsets restants ({ABC}, {ABE}, {ACE}) ont été entièrement comptés (on ne les considerera plus dans les itérations suivantes). Ils passent du marqueur DC à SB (leurs supports sont suffisants pour qu'ils soient fréquents).
1 $4-$itemset ({ABCE}) est généré à partir des $3-$itemsets marqués SB, il est marqué DC.

Après la sixième itération de l'algorithme (passage sur les instances 4, 5, 6), l'unique $4-$itemset ({ABCE}) reste marqué DC (son support est toujours insuffisant pour qu'ils soit fréquent).
Aucun itemset n'est généré.

Après la septième itération de l'algorithme (de nouveau passage sur les instances 1, 2, 3), l'unique $4-$itemsets restants ({ABCE}) a été entièrement compté (on ne le considerera plus dans les itérations suivantes). Il passe du marqueur DC à SB (son support est suffisant pour qu'il soit fréquent).
Aucun itemset n'est généré.

Il ne reste plus d'itemset marqué DC ou DB, l'algorithme s'arrête.

Initialement

1-itemsets	Support	Marqueur
⟨ A ⟩	0/6	DC
⟨ B ⟩	0/6	DC
⟨ C ⟩	0/6	DC
⟨ D ⟩	0/6	DC
⟨ E ⟩	0/6	DC

Après la 1ère itération
(instances 1 à 3 lues)

1-itemsets	Support	Marqueurs
⟨ A ⟩	2/6	**DB**
⟨ B ⟩	2/6	**DB**
⟨ C ⟩	3/6	**DB**
⟨ D ⟩	1/6	DC
⟨ E ⟩	2/6	**DB**
2-itemsets	Support	Marqueurs
⟨ **A B** ⟩	0/6	DC
⟨ **A C** ⟩	0/6	DC
⟨ **A E** ⟩	0/6	DC
⟨ **B C** ⟩	0/6	DC
⟨ **B E** ⟩	0/6	DC
⟨ **C E** ⟩	0/6	DC

Après la 2ème itération
(instances 4 à 6 lues)

1-itemsets	Support	Marqueurs
⟨ A ⟩	3/6	**SB**
⟨ B ⟩	5/6	**SB**
⟨ C ⟩	5/6	**SB**
⟨ D ⟩	1/6	**SC**
⟨ E ⟩	5/6	**SB**
2-itemsets	Support	Marqueurs
⟨ A B ⟩	1/6	DC
⟨ A C ⟩	1/6	DC
⟨ A E ⟩	1/6	DC
⟨ B C ⟩	2/6	**DB**
⟨ B E ⟩	3/6	**DB**
⟨ C E ⟩	2/6	**DB**
3-itemsets	Support	Marqueurs
⟨ **B C E** ⟩	0/6	DC

Après la 3ème itération
(instances 1 à 3 lues)

2-itemsets	Support	Marqueurs
⟨ A B ⟩	2/6	**SB**
⟨ A C ⟩	3/6	**SB**
⟨ A E ⟩	2/6	**SB**
⟨ B C ⟩	4/6	**SB**
⟨ B E ⟩	5/6	**SB**
⟨ C E ⟩	4/6	**SB**
3-itemsets	Support	Marqueurs
⟨ B C E ⟩	2/6	**DB**
⟨ **A B C** ⟩	0/6	DC
⟨ **A B E** ⟩	0/6	DC
⟨ **A C E** ⟩	0/6	DC

TAB. 4.10 – Exemple d'exécution DIC sur D : premières itérations.

Après la 4ème itération
(instances 4 à 6 lues)

3-itemsets	Support	Marqueurs
⟨ A B C ⟩	1/6	DC
⟨ A B E ⟩	1/6	DC
⟨ A C E ⟩	1/6	DC
⟨ B C E ⟩	4/6	**SB**

Après la 5ème itération
(instances 1 à 3 lues)

3-itemsets	Support	Marqueurs
⟨ A B C ⟩	2/6	**SB**
⟨ A B E ⟩	2/6	**SB**
⟨ A C E ⟩	2/6	**SB**
⟨ B C E ⟩	4/6	SB
4-itemsets	Support	Marqueurs
⟨ **A B C E** ⟩	0/6	DC

Après la 6ème itération
(instances 4 à 6 lues)

4-itemsets	Support	Marqueurs
⟨ A B C E ⟩	1/6	DC

Après la 7ème itération
(instances 1 à 3 lues)

4-itemsets	Support	Marqueurs
⟨ A B C E ⟩	2/6	**SB**

TAB. 4.11 – Exemple d'exécution DIC sur D : 4ème, 5ème et 6ème itérations

---

Les Algorithmes Parallèles

Devant l'immense quantité de données à traiter, une nécessité d'algorithmes parallèles apparaît clairement. Pour rappel, il existe trois grandes familles :

- Duplication des itemsets candidats (CD 1999 [65], Parallel Partition 1995 [59], PDM 1995 [53])
- Partitionnement des itemsets (DD 1999 [65], IDD 1999 [65], HPA 1996 [57])
- Approche Hybride : Réplication partielle des itemsets candidats (HD 1999 [65], HPA-ELD 1996 [57])

Rappel Propriété 3 : **Pour qu'un itemset soit globalement fréquent il faut qu'il soit localement fréquent sur au moins un site.**

Notation :

Dans la suite on notera **Di**, un fragment de la base présent sur le site i.

Les parallèlisations du calcul présentées par Zaki [4], [65]

- **CD (Count Distribution)**
  Cet algorithme se base sur une répartition "aléatoire" des données, on effectue un découpage horizontal des données (répartition des instances sur les processeurs). Chaque processeur compte les supports des mêmes itemsets candidats.
  A chaque itération, les supports locaux de chaque itemset sont calculés, puis une synchronisation permet de sommer ces supports locaux, de manière à prendre les décisions sur des supports globaux. Cette synchronisation de supports a lieu entre chaque itération.

- Une variante de cet algorithme existe, NPA [57]. Cette variante propose d'introduire des fonctions de hachage sur les tids pour partitionner les instances
  .
- **DD (Data Distribution)**
  Tout comme CD, cet algorithme se base sur une répartition "aléatoire" des données, mais ici, chaque processeur génére des candidats différents.
  Les supports des itemsets sont toujours calculés de façon locale, à chaque itération. Puis, entre deux itérations, on effectue une diffusion des partitions locales de données (c'est-à-dire des instances). Cette diffusion des Di locaux s'effectue par une communication de type all-to-all.

- Comme pour CD, une variante de cet algorithme introduisant des fonctions de hachage sur les tids existe : SPA [57].

- **CaD (Candidate Distribution)**
  L'algorithme CaD se base toujours sur un découpage horizontal des instances, mais également sur une distribution des candidats. De façon à assurer une cohérence entre les itemsets que l'on calcule et les instances qu'on utilise pour effectuer ces calculs, il est nécessaire d'effectuer une redistribution des itemsets et des Di entre chaque itération.

D'autres types de parallèlisme

- **DMA (Distributed Mining of Association rules)**
  DMA est une parallélisation de l'algorithme Apriori (plus précisément de son amélioration DHP) proposée par Cheung et al. en 1996 [15]. Les supports sont calculés de façon locale, puis, entre chaque itération, on effectue un échange des supports de façon à identifier les itemsets globalement fréquents. Ensuite, on effectue une diffusion, à tous les processeurs, des itemsets fréquents. DMA différe de Data Distribution car ici ce sont les supports qui sont échangés lors des communications synchronisées entre chaque itération plutôt que les données.

- **IDD (Intelligent Data Distribution)**
  L'algorithme IDD est une modification de DD présentée par Han, Karypis, Kumar en 1999 [38]. Dans IDD, la diffusion des Di s'effectue sur une structure en anneau

plutôt que par communication de type all-to-all, et ce pour améliorer les performances en diminuant les communications.

- Deux variantes introduisant des fonctions de hachage pour le partitionnement existent (comme pour CD et DD) : HPA (hachage pour assigner les itemsets aux noeuds), HPA-ELD (hachage pour assigner les itemsets aux noeuds avec réplication partielle des itemsets candidats) [57].

- **HD (Hybrid Distribution)**
  Solution hybride de IDD et CD proposée par Han, Karypis, Kumar en 1999 [38]. Les processeurs sont divisés en groupes. On effectue l'algorithme IDD au sein des groupes (communications des Di sur une structure en anneau), et l'algorithme CD entre les groupes (communications pour la réduction des supports).

- **Parallel PARTITION**
  L'algorithme séquentiel PARTITION proposé par Savasere et al. [59] (présenté plus haut) s'inscrit tout à fait dans une vision parallèle. On affecte une partition à chacun des processeurs. Des communications sont nécessaires entre chaque itération pour échanger les itemsets qui sont fréquents sur au moins un processeur (respect de la propriété 3 voir Section 4.1.2).

- **PDM**
  PDM est une parallélisation de DHP (selon le modèle de CD) proposée par Park et al. en 1995 [53]. Entre chaque itération, on effectue des communications similaires à celles de l'algorithme CD (réduction des supports).

4.1.3 Inadéquation des méthodes parallèles existantes

Pour rappel, il existe deux types de parallélisme : le parallélisme de type SMP/CCNUMA (voir Section 1.3.1), et le parallélisme sur réseau de stations de travail ou grille de calcul (voir Section 1.3.2).
Dans le premier type de parallélisme (SMP/CCNUMA), on dispose d'une mémoire commune ou partagée et d'un réseau d'interconnexion rapide.
Dans le deuxième type de parallélisme (sur réseau de stations de travail ou grille de calcul), on ne dispose pas de mémoire commune et le réseau est relativement lent par rapport à la puissance des machines.
Dans le projet DisDaMin, nous visons une architecture de réseau de stations ou grille de calcul pour l'exécution de la recherche de règles d'association.

Les algorithmes qui proposent une répartition des itemsets

Dans ce type d'algorithmes, on fait "tourner" les instances sur chaque site (les Di), d'où énormément de communications. Si la base est très grande (comporte beaucoup d'ins-

tances) comme c'est généralement le cas, il faudra que chaque instance passe par chaque site !

Ce type d'algorithme est adapté au parallélisme à mémoire partagée et à réseau d'interconnexion rapide (SMP/CCNUMA), à cause des échanges nécessaires d'instances entre les noeuds. Il est non satisfaisant sur une architecture parallèle de type réseau de stations de travail ou grille de calcul, les communications y étant trop pénalisantes.

Les algorithmes qui proposent une répartition des instances

Dans ce type d'algorithme, on doit échanger les informations calculées sur les différents sites, d'où énormément de communications que l'on peut considérer assez courtes ; dans le meilleur des cas, on ne communique que des entiers (les supports).

Cependant, ces communications ont lieu entre chaque itération et sont bloquantes (pour passer aux k+1-itemsets, il faut avoir reçu les supports des k-itemsets de tous les sites) !
Ici encore, ce type d'algorithme est adapté au parallélisme à mémoire partagée et à réseau d'interconnexion rapide (SMP/CCNUMA), à cause des échanges d'informations (supports) nécessaires. La solution ne peut pas être utilisée de manière satisfaisante sur une architecture parallèle de type réseau de stations de travail ou grille de calcul.

Les problèmes et manques des différents algorithmes présentés

Le nombre de communications dans ces algorithmes s'explique par le fait que la recherche de connaissances doit s'effectuer à partir de critères globaux. Cependant, à cause de ces communications, ils sont plutôt adaptés à une architecture parallèle à mémoire partagée et réseau d'interconnxion rapide qu'à une architecture de type réseau de stations de travail ou grille de calcul.
C'est d'ailleurs toujours sur des machines parralèles de type SMP que sont testés ces algorithmes. Des problèmes de délais de communication ralentiraient clairement tous les traitements effectués parallèlement.

Se pose également le problème du stockage. Aux nombreuses communications, il convient d'ajouter les coûts de lectures-écritures sur disque, inévitables étant donné la grande quantité de données. Une approche distribuée poserait des problèmes au niveau des communications mais pourrait par contre apporter une solution au problème de stockage. Les algorithmes présentés s'inscrivent dans une logique de mémoire partagée, une vision par mémoire distribuée pourrait permettre d'atténuer le problème de stockage des données.

Les algorithmes présentés plus haut se basent toujours sur le problème type de l'analyse du panier de la ménagère. Sur ce type de problèmes (données de consommation), généralement, la taille des itemsets fréquents reste limitée, (le plus grand nombre k pour lequel on génére les $k-$itemsets reste faible), on n'a donc peu d'itérations à faire. Sur des données médicales (comme celles que l'on souhaite traiter), ou d'autres types de données, il serait intéressant de ne pas se limiter pour les jeux de données ; la longueur, la largeur de

la base ainsi que le nombre moyen d'items par instance ne doivent pas être soumis à des bornes supérieues trop faibles à cause de la complexité de traitement !

4.1.4 Exploitation d'une distribution par clustering

Intérêt de la méthode

La distribution des données par utilisation du clustering amène à obtenir des fragments de données respectant les critères de similarité au sein d'un fragment et de dissimilarité entre les fragments.
Cette distribution intelligente avec similarité au sein d'un fragment de données permet d'améliorer les temps d'exécution pour la résolution du problème de deux manières :

- le nombre d'**items infréquents localement** augmente (absence totale ou forte sur ce cluster d'instances). Ces items seront identifiés à la première itération et ne devront dès lors plus être considérés dans les combinaisons à partir de la deuxième itération diminuant ainsi le nombre d'itemsets à générer ;
- le nombre d'**items fortement fréquents localement** augmente (présence totale ou forte sur ce cluster d'instances, items quasiment triviaux sur ce cluster), voir page 45 et Annexe A Section A.2
  Ces items peuvent être écartés du traitement si le taux de présence est proche de 100% (ou d'un autre seuil à fixer en fonction du type de données), puis réintroduits à la fin. D'où une fois de plus une diminution du nombre d'items à considérer et donc une diminution de la taille de l'espace de recherche à visiter.

La distribution par clusters d'instances similaires consiste donc en une spécialisation des traitements locaux sur un sous-ensemble d'items d'où une diminution des temps d'exécution. Pour rappel, la complexité du problème de génération des itemsets fréquents est liée aux nombres d'items à considérer dans le traitement (exponentielle par rapport au nombre d'items) : potentiellement $n*2^m$ itemsets possibles, n étant le nombre d'instances, m étant le nombre d'items.
Le but étant de ne plus être limité dans les traitements quant au nombre d'items traitables aussi bien que quant au nombre d'instances.

Validité de la méthode

L'utilisation de la distribution par clustering associée à des traitements indépendants des fragments par l'algorithme APriori a permis de constater une conservation de 95 à 100% des itemsets fréquents globaux (voir Section 2.3.1).

Cependant cette version par traitements indépendants des fragments utilisant l'algorithme APriori ne permet pas d'obtenir un support correct pour ces itemsets fréquents (le support obtenu par regroupement des supports locaux obtenus était incomplet, et donc inférieur au support sur la base de données complète) (voir Section 2.3.1).
L'incomplétude des supports et la perte de certains résultats sont accompagnées d'un problème de taille : la présence de nombreux parasites.
De nombreux itemsets sont identifiés comme fréquents sans que l'on puisse assurer qu'ils

le soient globalement et sans que l'on puisse valider leur fréquence sans un nouveau comptage de supports nécessitant l'accès aux données, problématique dans le contexte distribué (vérification nécessaire puisque les supports obtenus étaient erronés).

On considère donc deux approches de validation des "parasites" (voir Section 2.3.3) :
- obtenir des résultats imprécis (par des traitements locaux indépendants) et effectuer une phase de validation a posteriori ;
- enrichir les traitements locaux (par des communications) pour obtenir des résultats valides.

L'utilisation d'une phase de validation, pose toujours le problème de l'accès aux données. De plus , les itemsets "parasites" handicapent fortement le traitement, au vu de leur nombre par rapport au nombre d'itemsets réellement fréquents globalement.

Nécessité de collaboration

Afin d'identifier au préalable ces itemsets "parasites", il apparaît nécessaire d'autoriser des communications de collaboration entre les sites de traitement, afin d'empêcher, lors du calcul des supports locaux, la génération des "parasites".

L'algorithme APriori utilisé étant itératif sur la taille des itemsets considérés à chaque itération, la prise en compte d'informations extérieures (communications collaboratives) ne pourraient se faire qu'entre les itérations et nécessiterait des retours en arrière pour éventuellement traiter des itemsets de taille inférieure à la taille considérée à l'itération actuelle.
Afin de pouvoir librement utiliser les informations issues des communications de collaboration, nous proposons d'utiliser le parcours chaotique de l'ensemble des itemsets proposé par l'algorithme DIC.

4.2 Proposition d'une version collaborative pour la génération des itemsets fréquents : DIC-Coop

4.2.1 Utilité des communications de collaboration

L'utilisation des communications entre les traitements des différents fragments permet :
- la gestion des itemsets "parasites" observés dans la version de traitements indépendants. Cette gestion pourra consister à ne pas les considérer, ou au moins à les identifier pour limiter les communications (voir plus loin).
- de forcer le calcul de support des itemsets fréquents globaux pour obtenir un support exact en fin de traitement.

A partir des informations locales, nous allons assurer la construction d'informations globales au fur et à mesure de l'avancée des traitements. Les informations globales ainsi générées seront utilisées pour enrichir les traitements locaux.

Afin de simplifier les explications, la construction des informations globales sera décrite comme étant centralisée, mais cette construction peut être effectuée de manière distribuée, en utilisant une technique de hachage par exemple.

4.2.2 Version initiale de l'algorithme DIC-Coop - DIC coopératif

L'adaptation DICCoop repose sur une collaboration entre un fédérateur (centralisé ou distribué) et des sites de traitements exécutant une version modifiée de l'algorithme DIC. Il y a accumulation des informations sur le fédérateur, puis prises de décisions globales par ce fédérateur, ce qui correspond à une phase de validation anticipée, qui permet d'enrichir les traitements restant à effectuer par collaboration.

Définitions

On appelle **information locale**, une connaissance concernant la fréquence ou l'infréquence d'un itemset sur un site donné (par rapport aux instances présentes sur ce site).

On appelle **information globale**, une connaissance concernant la fréquence ou l'infréquence d'un itemset pour l'ensemble des sites (par rapport à l'ensemble des instances).

Notations

On identifiera les sites de traitements T_i (voir Figure 4.8) : Le site T_k assure le traitement du fragment de données D_k issu de la distribution par clustering. Pour une distribution en n fragments, on aura donc n sites T_i (i allant de 1 à n). Selon la taille des fragments plusieurs d'entre eux pourraient être traités sur un même noeud de la grille auquel cas on aurait plusieurs T_i physiquement présents sur un même noeud. Le traitement du fragment D_k consiste en l'utilisation de l'algorithme DIC-Coop local sur le site T_k (voir plus loin).

L'information globale est construite sur un site **F** appelé **Fédérateur** (voir Figure 4.8) : Le rôle du fédérateur est de "collecter" les informations locales sur les sites T_i (voir plus loin : envoi et réception sur le fédérateur), de les agglomérer pour en déduire l'information globale (voir plus loin agglomération des informations sur le fédérateur) et de transmettre les informations globales ainsi construites aux sites de traitement T_i.

Utilisation des informations globales sur les sites T_i (enrichissement du traitement)

Chaque site T_i peut enrichir son traitement en utilisant les informations globales de fréquence et d'infréquence de la manière suivante :

- les informations de fréquence globale servent à forcer le traitement d'itemsets localement infréquents mais globalement fréquents (donc intéressants).
  Ces itemsets ne seraient pas considérés localement sans informations extérieures, on force leur prise en compte dans le traitement local.
- les informations d'infréquence globale servent :
  - soit à limiter le traitement des itemsets infréquents globalement, ils ne sont pas intéressants pour le résultat global final, on peut donc interrompre leur traitement

ainsi que celui de leur "descendance" ou au moins limiter les incidences de ces traitements (voir intérêt des résultats locaux ci-dessous),
- soit à limiter les communications vers le fédérateur. Ces itemsets étant globalement infréquents, même s'ils sont localement fréquents, le fédérateur n'a pas besoin d'informations les concernant eux ainsi que leurs "descendants".

Remarque concernant l'arrêt de traitement et l'intérêt de résultats locaux :

La fragmentation correspondant à des groupes d'instances similaires, ceci constitue un découpage en sous-populations à caractéristiques communes (voir Section 2.1.2).
Il peut être intéressant de générer les règles d'association spécifiques à chacune de ses sous-populations, d'autant plus que les résultats locaux de fréquence et les supports sont disponibles, et peuvent permettre d'offrir des informations importantes sur les groupes d'instances.
Les infréquents globaux qui sont fréquents locaux ne doivent donc pas être éliminés des traitements locaux au sein desquels ils sont pertinents (fréquents), afin de bien avoir un résultat complet sur chaque fragment : pour chaque fragment, un ensemble de règles d'association pertinent.

Ainsi, on fera une différence entre :
- les règles d'association obtenues sur la base complète ;
- les règles d'association supplémentaires obtenues sur un sous-ensemble "intelligent" d'instances, un cluster.

Les itemsets fréquents locaux non pertinents globalement vont influer sur la poursuite du traitement. En effet, il conviendra de poursuivre le traitement tant que l'on n'a pas obtenu tous les itemsets fréquents globaux générables, mais également tant que l'on n'a pas obtenu tous les itemsets fréquents locaux générables.

De plus, lorsqu'un site a terminé la recherche locale des itemsets fréquents, il est possible qu'il reste des itemsets globaux à traiter. Dans ce cas, le site doit rester disponible pour répondre à d'éventuelles besoin d'informations du fédérateur.

4.2.3 Principe de l'algorithme DIC-Coop

Le principe de l'algorithme DIC-Coop est donc d'utiliser un algorithme inspiré de l'algorithme DIC sur chaque fragment (traitements locaux sur les sites T_i), et d'effectuer des communications de collaboration entre ces sites T_i et le site fédérateur F qui enrichit le traitement des sites locaux (voir Figure 4.8).

1. Exécution de DIC-Coop local
2. Envoi d'informations locales au fédérateur
3. Déduction d'informations globales sur le fédérateur F.
4. Envoi d'informations globales aux sites T_i

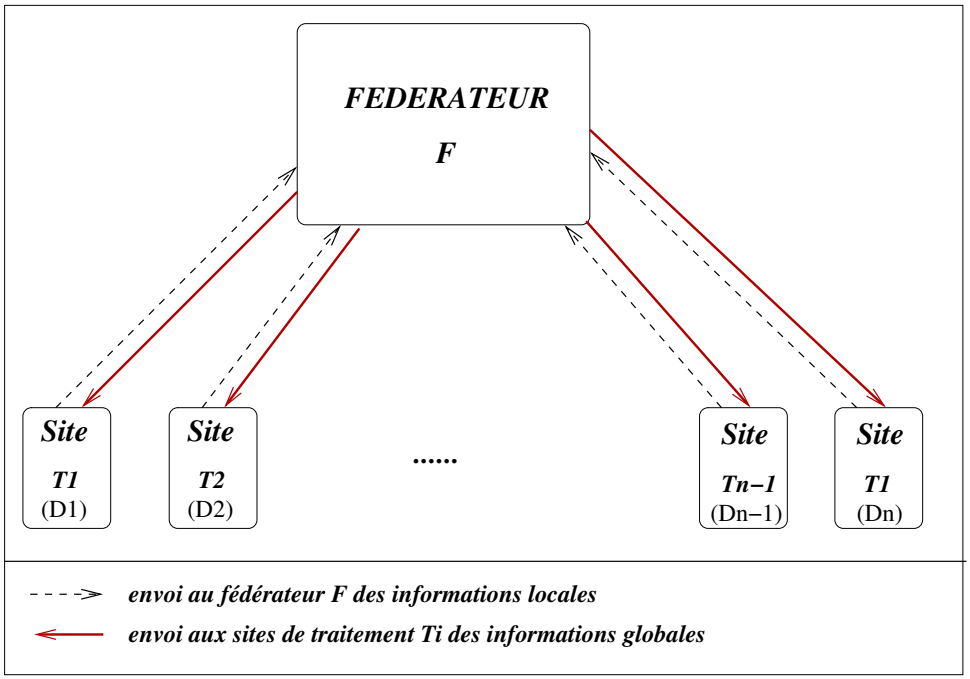


FIG. 4.8 – Schéma de déploiement et communications de l'algorithme DIC-Coop

Le pseudo-code de la méthode DIC-Coop local est donné Tableau 4.13, celui permettant la déduction d'informations sur le fédérateur est donné Tableau 4.12.

Les échanges d'information sont basés sur l'utilisation de marqueurs (tels que ceux existants dans l'algorithme DIC voir Section 4.1.2). On adapte les marqueurs aux différents types d'information (locale, globale).

Rappel du fonctionnement de l'algorithme DIC classique pour la génération de nouveaux itemsets

Pour chaque itemset k que l'on traite, on a un couple d'informations $R_k = (s_k, m_k)$:

- un support : s_k
- un marqueur : m_k

On incrémente les compteurs de support s_k par rapport au scan d'une tranche de données. Les marqueurs sont ensuite mis à jour (ils seront utilisés pour déterminer les nouveaux itemsets à générer).

Puis, pour tout itemset k qui change d'état (passage de Dashed à Solid ou passage de Circle à Box), on tente de générer les sur-ensembles k.
Pour la génération d'un sur-ensemble K (de taille $Card(K) = Card(k) + 1$), on effectue une vérification de tous les sous-ensembles l tel que $l \in K$, avec $Card(l) = Card(K) - 1$:

- si $m_k =$ DB ou $m_k =$ SB, K est générable si $\forall l \in K$, avec $Card(k) = Card(l)$, $m_l =$ DB ou $m_l =$ SB,

Les sur-ensembles K seront placés dans l'ensemble des itemsets dont il faut calculer le support.

L'algorithme itère jusqu'à ce qu'il n'y est plus d'itemsets marqués "dashed".

Marqueurs utilisés dans l'algorithme DICCoop

Il est nécessaire de procéder à un marquage des itemsets localement et globalement, en utilisant le marquage de l'algorithme DIC complété avec de nouveaux marqueurs globaux :

1. marqueurs locaux (ceux de l'algorithme DIC classique)
   - DB : fréquent en cours de traitement (dashed box)
   - SB : fréquent traitement fini (solid box)
   - DC : infréquent supposé (dashed circle) valeur par défaut en début de traitement
   - SC : infréquent avéré (solid circle)
2. marqueurs globaux
   - GF : globalement fréquent
   - GI : globalement infréquent

Echanges d'informations dans l'algorithme DICCoop

Il existe deux cas d'échanges dans l'algorithme :

1. les envois d'informations des sites T_j vers le fédérateur F ;
2. les envois d'informations du fédérateur F vers les sites T_j.

Le premier type d'échange correspond aux informations locales envoyées des sites T_j vers le fédérateur F : I_{T_j} (voir flèches en pointillés sur la Figure 4.8).
On a $I_{T_j} = \{I_{T_{j\,k}}\}$, pour chaque itemset k pour lequel on communique des informations .
Avec $I_{T_{j\,k}} = (k, s_{kj}, m_{kj})$

- une description de l'itemset : k
- un support local : s_{kj}
- un marqueur local : m_{kj}

Le deuxième type d'échange correspond aux informations globales envoyées du fédérateur F vers les sites T_j : I_F (voir flèches pleines sur la Figure 4.8).
On a $I_F = \{I_{F_k}\}$, pour chaque itemset k pour lequel on communique des informations.
Avec $I_{F_k} = (k, mg_k)$

- une description de l'itemset : k
- un marqueur glocal : mg_k

4.2.4 Rôle des composants de DICCoop

Rôle des composants de DICCoop : site fédérateur F

Le site Fédérateur F effectue le regroupement d'informations (voir pseudo-code Table 4.12)
Il est nécessaire de connaître le **seuil de support** global souhaité et les tailles des partitions sur chaque site ($Card(D_j)$). Le traitement du Fédérateur se décompose en trois composantes :

- La réception spontanée et progressive d'informations locales de fréquence I_{T_j} de la part des sites de traitement (résultats locaux)
- L'accumulation par le fédérateur et prise de décisions globales : R_F
- Les envois d'informations globales I_F utiles aux sites de traitement ou représentant un questionnement de ces sites

Sur le fédérateur, pour chaque itemset k, on a un triplet d'informations $R_{F_k} = (M, S, mg_k)$, avec :

- une liste des marqueurs locaux des sites T_j : $M = \{m_{kj}\}$
- une liste des supports locaux des sites T_j : $S = \{s_{kj}\}$
- un marqueur global : mg_k (initialement $mg_k = GU$)

Accumulation et prise de décision sur le fédérateur Les décisions sur le fédérateur sont prises selon le scénario décrit Table 4.12.

Lors de la réception d'information reçue, on vérifie si la quantité d'informations reçues pour les itemsets concernés est complète (tous les sites ont communiqué des informations pour un itemset donné), auquel cas on peut tirer une conclusion globale de fréquence ou infréquence.
Si la quantité d'information n'est pas complète, on vérifie si elle est suffisante pour tirer une conclusion globale de fréquence, ou si le nombre de sites non informateurs (et la quantité de données qu'ils contiennent) est insuffisant pour arriver à une situation de fréquence (auquel cas on se trouve dans une situation d'infréquence).
Si des informations globales sont identifées, elles sont préparées pour être communiquées aux sites de traitement T_j.

Rôle des composants de DICCoop : site de traitement T_j

Un site de traitements T_j effectue un traitement local (sur les instances présentes sur ce site) sur un site de calcul par l'algorithme DIC-Coop local (voir pseudo-code Table 4.13).
Le traitement effectué sur un site de traitement T_j se décompose en quatre composants :

- Le traitement sur données locales D_j (par génération classique itérative avec marqueurs locaux)
- L'obtention de résultats locaux : R_{T_j}
- L'envoi **progressif** d'informations de fréquence locales à un fédérateur : I_{T_j} (marqueurs locaux)
- La réception d'informations de fréquence globales I_F de la part du fédérateur (marqueurs globaux)

Sur un site de traitement, pour chaque itemset k, on a un quadruplet d'informations $R_{T_k} = (s_k, m_k, mg_k, d_k)$:

- un support : s_k
- un marqueur local : m_k
- un marqueur global : mg_k

```
( 1)Pour chaque information I_{T_{l_k}} recue
( 2)   on insere cette information
( 3)     i.e. mise a jour de m_{kl} et s_{kl}
( 4)   puis on vérifie :
( 5)   si (on a reçu I_{T_{j_k}} ∀j) alors
( 6)      calcul du seuil global S_k = Σ_j s_{kj}
( 7)     //prise de decision globale definitive de frequence
( 8)       (mg_k passe de GU à GI ou à GF)
( 9)      si (S_k < seuil) alors
(10)          mg_k= GI
(11)      sinon //i.e. S_k ≥ seuil alors
(12)          mg_k = GF
(13)      envoi de I_{F_k} = (k, mg_k) aux sites T_j
(14)   sinon
(15)      calcul du seuil global S_k = Σ_{jrecu} s_{kj}.
(16)      //tentative de prise de decision globale partielle de frequence :
(17)      si (S_k > seuil) alors
(18)          mg_k = SB
(19)          envoi de I_{F_k} = (k, GF) aux sites T_j
(20)      sinon
(21)          calcul de S'_k = Σ_{jnonrecu} Card(D_j)
(22)          si (S_k + S'_k < seuil) alors
(23)             mg_k = SC
(24)             envoi de I_{F_k} = (k, GI) aux sites T_j
(25)          finsi
(26)      finsi
(27)   finsi
(28)Fin Pour
```

TAB. 4.12 – Principe de déduction d'informations sur le fédérateur F dans l'algorithme DIC-Coop

- une indication des instances déjà prises en compte dans le calcul de support : d_k

Génération d'informations sur les sites de traitement La génération d'informations sur les sites de traitement fonctionne de la manière itérative classique de l'algorithme DIC (par tranches de données) voir Table 4.13.

Les informations globales reçues du fédérateur permettent d'enrichir le traitement, en forçant le traitement d'itemsets localement non intéressants mais globalement utiles. Elles permettent également de limiter les communications, en ne communiquant pas au fédérateur des informations sur des itemsets de branche globalement infréquentes (branches du treillis selon le mode de génération des sur-ensembles).

Détail du traitement sur les sites de traitement Le traitement sur chaque site T_j est réalisé selon le scénario décrit Table 4.14.

(1)**Pour** chaque iteration
(2) insertion informations du fédérateur
(3) **envoi** de réponses aux questions du fédérateur
(4) traitement local
(5) **envoi** d'informations au fédérateur
(6)**Fin pour**

TAB. 4.13 – Principe de fonctionnement de l'algorithme DIC-Coop local sur les sites T_i

On incrémente les compteurs de support s_k par rapport au scan d'une tranche de données présente sur le site (principe de fenêtrage de l'algorithme DIC).

Puis, pour tout itemset k qui change d'état durant l'itération (passage de Dashed à Solid ou passage de Circle à Box), on tente de générer les sur-ensembles K tels que $Card(k) = Card(K) - 1$:

- soit sur base de fréquence locale ($m_k = SB$ ou $m_k = DB$)
- soit sur base de fréquence globale ($mg_k = GF$), avec $m_k = SC$ ou $m_k = DC$

Les deux critères cités sont utilisés sans qu'il existe une priorité pour l'un ou pour l'autre.

(1)incrementation de support s_k //par rapport au scan d'une tranche de données
(2)//mise à jour des marqueurs locaux
(3)**Si** ($m_k = DC$ ou $m_k = DB$ et toutes les données ont été prises en compte) alors
(4) **Si** ($m_k = DB$) alors
(5) $m_k = SB$
(6) **Sinon**
(7) **Si** ($s_k \geq seuil$) alors
(8) m_k = SB
(9) **Sinon**
(10) m_k = SC
(11) **Finsi**
(12) **Finsi**
(13)**Sinon**
(14) **Si** ($m_k = DCets_k \geq seuil$) alors
(15) $m_k = DB$
(16) **Finsi**
(17)**Finsi**
(18)**Pour** chaque itemset ayant change d'etat de XC a XB
(19) //generation du sur-ensemble
(20) generation des itemsets K tels que
(21) $\forall l \in K$ avec $Card(l) = Card(K) - 1$
(22) m_l = DB ou m_l = SB ou mg_l = GF
(23)**Fin pour**

TAB. 4.14 – Génération de sur-ensembles sur les sites T_i dans l'algorithme DICCoop

Pour la génération d'un sur-ensemble K, on effectue une vérification de tous ces sous-

ensembles l tel que $l \in K$ avec $Card(l) = Card(K) - 1$:

- si m_k = DB ou m_k = SB, K est générable si $\forall l \in K$ avec $Card(l) = Card(K) - 1$ m_l = DB ou m_l = SB,
- si mg_k = GF, K est générable si $\forall l \in K$ avec $Card(l) = Card(K) - 1$ g_l = GF (localement, on a m_l = DC ou m_l = SC).

Un itemset k ne sera pris en compte (pour la génération de sur-ensembles) que lors de son changement d'état (passage de $m_l = DC$ à $m_l = SC$, $m_l = DB$ ou $m_l = SB$), principe de l'algorithme DIC.
Le sur-ensemble K généré sera placé dans l'ensemble des itemsets dont il faut calculer le support avec un marqueur m_K = DC.

Réponses au fédérateur Si on reçoit une information marquée GU (un itemset marqué GU), le fédérateur a besoin de cette information pour prendre une décision particulière. On lui envoie donc immédiatement la réponse (le support local de l'itemset) si elle est disponible. Si l'information n'est pas disponible, on inclut l'itemset concerné dans l'ensemble des itemsets à traiter (dont il faut calculer le support) avec un marqueur m_k = DC et mg_k=GU. Dès que l'information sera disponible elle sera envoyée.

Limitation des envois d'informations : Les envois d'informations ne sont effectués que si on n'a pas déjà un marqueur GI dans la hiérarchie d'un itemset. Si l'un de ses parents est globalement infréquent (marqué GI), toute la branche est éliminée des communications, mais le traitement local peut continuer si nécessaire localement (marqueurs locaux XB).

4.2.5 Résumé des expérimentations sur DIC-Coop

Différentes versions de méthodes de communication entre le fédérateur F et les sites T_i ont été comparées (Voir Annexe C.1 pour une synthèse des résultats) :

- **Aucune communication** :
  On n'effectue aucune communications des sites de traitement T_j vers le fédérateur F ou du fédérateur F vers les sites T_j. Cela revient à effectuer des traitements indépendants sur chaque site T_j.

- **Tout communiquer** :
  On effectue des communications aussi souvent que nécessaires.
  A la fin de chacune des itérations de son exécution, un site T_j effectue une communication vers le fédérateur F, quelle que soit la quantité d'informations qu'il ait à lui transmettre.
  De même le fédérateur F effectue un envoi d'informations aux sites T_j, dès qu'il est en possession d'une information globale, ou dès qu'il a besoin d'un résultat local non communiqué par un site.

- **Les communications sont limitées et paramétrées** :
  Les communications ayant lieu de T_j vers le fédérateur F (en fin de chaque itération sur les sites de traitement) ne sont effectuées que si les envois d'informations correspondent à l'un des deux critères suivants :
  – Il faut que la quantité d'informations à envoyer soit suffisante :
    on fixe un paramètre seuil correspondant au nombre minimal d'informations nécessaire pour effectuer un envoi.
    Ceci permet de ne pas effectuer de communications de trop petite taille, puisque, à quantité égale, un grand nombre de petites communications est plus pénalisant qu'un petit nombre de grandes communications.
  – il faut qu'un certain délai soit dépassé depuis le dernier envoi :
    un paramètre précise le nombre maximum d'itérations possibles sur un site sans effectuer de communications.
    Le paramètre délai de communications permet d'éviter de transmettre des informations trop anciennes au fédérateur.
  – Le fédérateur n'utilise quant à lui pas ces paramètres, et effectue les communications aux sites $T_j$dans tous les cas.

La version sans **aucune communication** consiste en fait à effectuer des traitements indépendants par l'algorithme DIC classique, et amène donc aux problèmes observés dans la pré-version utilisant des traitements indépendants sur base de l'algorithme Apriori (voir Section 2.3.1), à savoir les "parasites" locaux et le problème des cas limites (incomplétude des supports).

La version avec **communications paramètrées** est bien entendue celle qui est la plus adaptée au déploiement distribué sur grille et à la génération rapide et exacte des résultats.

Cependant la version **tout communiquer** s'avère intéressante si on considère qu'elle ne demande pas de synchronisation handicapante. *En particulier la méthode permet de traiter des données distribuées de manière aléatoire* (donc non fragmentées de manière "intelligente").

4.3 Méta version de l'algorithme DIC-Coop

Sur la base des résultats observés sur la version **Tout communiquer** (voir Annexe C.1, nous proposons d'utiliser cette version pour augmenter le parallélisme de traitement des fragments de trop grandes tailles.
En effet, le nombre de fragments devant être limité (voir Section 2.3.7) pour conserver les gains de la distribution "intelligente", on pourrait être amené à devoir traiter des fragments de trop grande taille en termes de nombre d'instances.

Ainsi, dans le cas où la fragmentation fournirait des fragments d'instances trop grands pour être traités localement sur un seul site (complexité liée au traitement de ce fragment ou problème de capacité de stockage des instances), nous proposons de distribuer leur

traitement non plus sur un site (ou noeud de la grille), mais sur une sous-grappe (voir Section 1.3.2), en utilisant l'algorithme DIC-Coop en version **Tout communiquer**.
C'est ce que nous appellerons la **Méta-version de DIC-Coop**.

4.3.1 Principe de fragmentation dans la méta version de DIC-Coop

La fragmentation initiale de la base de données est basée sur l'algorithme de clustering distribué progressif CDP (voir Chapitre II Section 3.2), on obtient donc une fragmentation intelligente. Si certains fragments issus de celle-ci s'avèrent être de trop grandes tailles pour être traités sur un seul site, ces fragments sont redistribués de manière aléatoire au sein d'une sous-grappe.
Ainsi, pour une base de données fragmentée en n fragments par le clustering distribué progressif et en admettant que les fragments obtenus soient de trop grandes tailles pour subir un traitement local (problème de complexité et/ou de stockage), on considère le traitement de chacun de ces fragments non plus sur un site T_i, mais sur une sous-grappe SG_i (voir Figure 4.9).

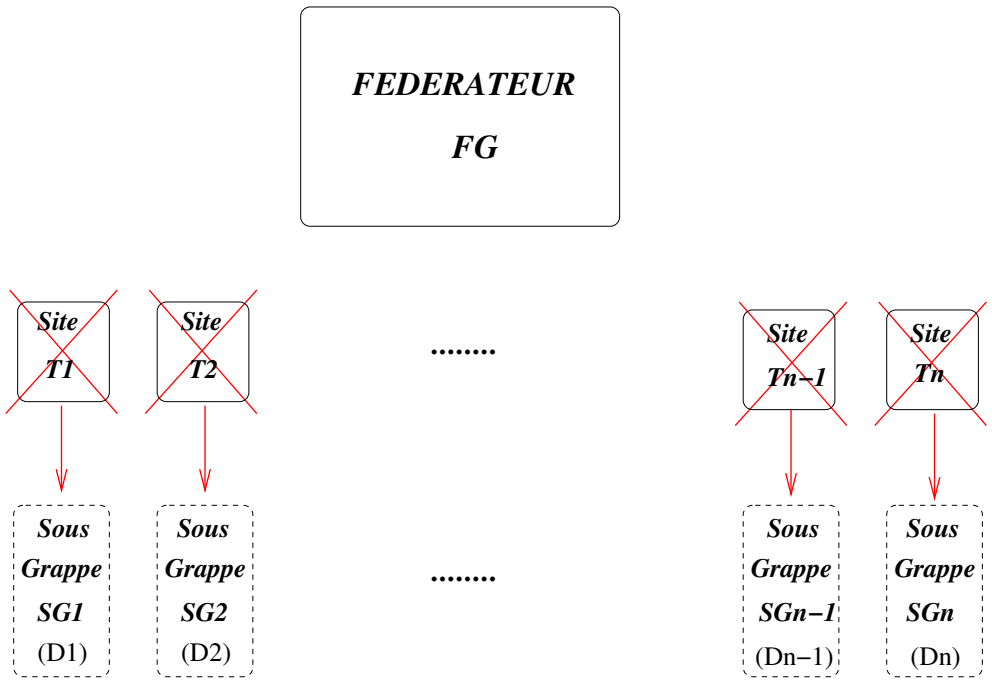


FIG. 4.9 – Passage à la Méta version de l'algorithme DIC-Coop

Remarque : On peut également effectuer le traitement de chaque fragment sur une sous-grappe plutôt que sur un seul site, sans que la nécessité apparaisse à cause des tailles des fragments.
Cependant la version **Tout communiquer** engendrant plus de communications que la version avec **communications paramétrées**, et les communications étant coûteuses dans ce type d'architecture parallèle, il convient de ne pas abuser du procédé de redistribution sur des sous-grappes.

Ainsi le traitement du fragment D_i de base de données sera assuré par une collaboration sur une sous-grappe SG_i.

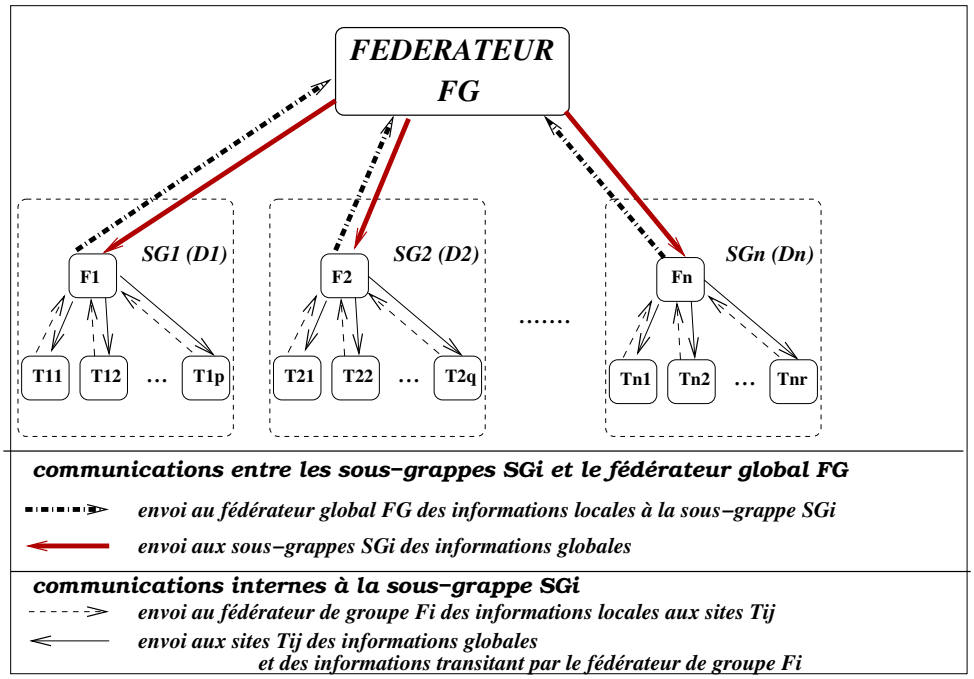


FIG. 4.10 – Schéma de déploiement et communications de la Méta version de l'algorithme DIC-Coop

Dans la version précédente, on considère un seul fédérateur et des sites de traitements qui traitent les groupes d'instances issus de la distribution intelligente.

En considèrant que ces groupes d'instances peuvent être de taille quelconque, et de ce fait de taille ne permettant pas un traitement optimal sur un noeud (volume des données ne permettant pas un stockage des instances en mémoire ou complexité du traitement menant à un dépassement des capacités en mémoire du noeud), on éclate les groupes de manière aléatoire, puis on les redistribue sur une sous-grappe.
On considère alors un fédérateur principal, et pour chaque groupe de données une sous-architecture incluant un fédérateur et des sites de traitement tels que ceux décrits plus haut (voir Figure 4.10).

Le traitement d'un fragment D_i dans la sous-grappe SG_i sera assuré sur le modèle de l'algorithme DIC-Coop en version **Tout communiquer** (voir Figure 4.11), à l'aide d'un fédérateur de groupe F_i et de sites de traitements T_{ij}.

Les communications

Les communications de collaboration se font de manière hiérarchique (voir Figure 4.10).

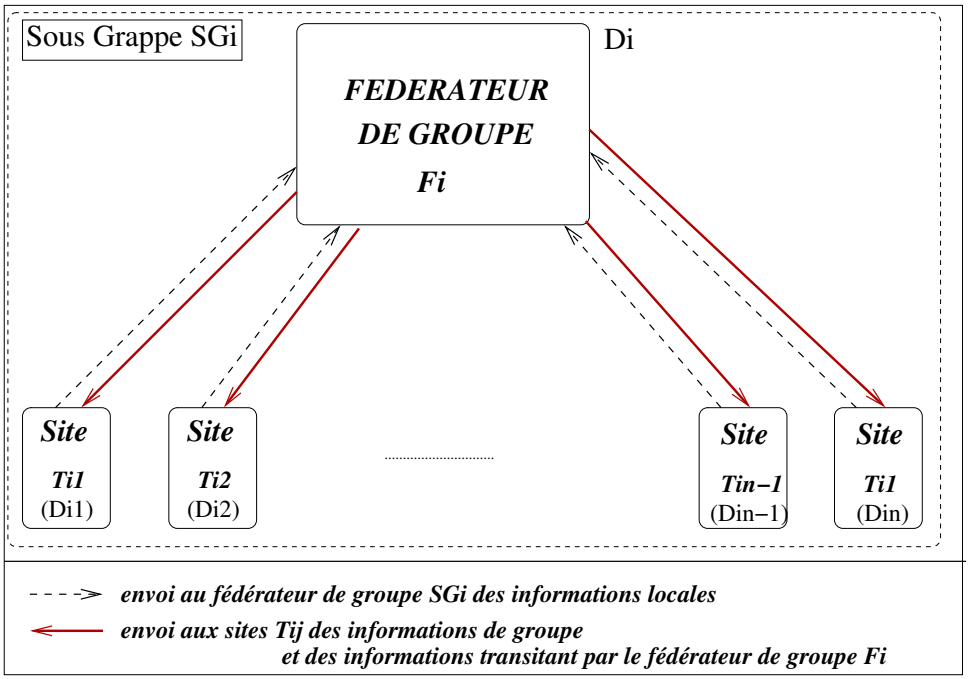


FIG. 4.11 – Redistribution d'un fragment sur une sous-grappe dans la Méta version de l'algorithme DIC-Coop

On a donc les entités de traitements suivantes :

- Un fédérateur global : FG ;
- Des fédérateurs locaux : un F_i pour chaque groupe de données D_i ;
- Des sites de traitement locaux : un ensemble $T_i = \{T_{ij}\}$ pour chaque D_i.

Les communications sont possibles :

- du fédérateur global FG vers les fédérateurs locaux F_i : $I_{FG} = \{I_{FG_k}\} = \{(k, smg_k)\}$
  (voir flèches pleines et épaisses Figure 4.10) ;
- des fédérateurs locaux F_i vers le fédérateur global FG : $I_{F_i} = \{I_{F_{ik}}\} = \{(k, g_{ik})\}$
  (voir flèches en pointillés épais Figure 4.10) ;
- d'un fédérateur local F_i vers les les sites de traitement T_{ij}, de même i : $I_{F_i} = \{I_{F_{ik}}\} = \{(k, g_{ik}, smg_k)\}$
  (voir flèches pleines et fines Figure 4.10) ;
- des sites de traitements T_{ij} vers le fédérateuir local F_i de même i : $I_{T_{ij}} = \{I_{T_{ijk}}\} = \{(k, m_{ijk})\}$
  (voir flèches en pointillés fins Figure 4.10).

Il n'y a pas de communications directes entre les sites de traitements T_{ij} y compris à l'intérieur d'une sous-grappe (i commun).

Nouveaux marqueurs et changements de comportement

Les marqueurs associés aux traitements sont modifiés afin de distinguer cette fois-ci :

- les informations locales aux sites T_{ij},

- les informations locales aux sous-grappes SG_i (ou au fragment D_i),
- les informations globales.

On considère donc une série de marqueurs super-global smg_k sur le fédérateur globale F qui prendront les valeurs GI, GF, GU (voir Section 4.2.3).

4.3.2 Fonctionnement des fédérateurs FG et F_i

Accumulation et prise de décision sur le fédérateur FG

Le fédérateur FG reçoit des informations issues des fédérateurs de groupes F_i (supports et marqueurs).

Sur le fédérateur global (comme sur le fédérateur dans la version de base), pour chaque itemset k, on a un triplet d'informations $R_{Fmg_k} = (M, S, smg_k)$, avec :

- une liste des marqueurs $M = \{g_{ik}\}$ correspondant aux informations sur les fédérateurs de groupe F_i ;
- une liste des supports $S = \{s_{ik}\}$ sur fédérateurs de groupe F_i ;
- un marqueur super global : smg_k (initialement $smg_k = GU$)

Le scénario de prise de décisions sur le fédérateur global (voir pseudo-code Table 4.15) ressemble fortement à celui du fédérateur F dans la version de base de DICCoop de base (voir Section 4.2.2).
Si des informations super-globales sont identifées, elles sont préparées pour être communiquées aux fédérateurs de groupe F_i.

Accumulation et prise de décision sur les fédérateurs de groupe F_i

Sur le fédérateur de groupe F_i, pour chaque itemset k, on a un quadruplet d'informations RF_{ik} :

- une liste des marqueurs relatifs aux sites T_{ij} : $M = \{g_{ijk}\}$
- une liste des supports des sites T_{ij} : $S = \{s_{ijk}\}$
- un marqueur de groupe : mg_k (initialement $mg_k = GU$)
- un marqueur super global : smg_k (initialement $smg_k = GU$)

Réception d'informations issues des T_{ij} Le scénario de prise de décisions sur les fédérateurs locaux F_i (voir pseudo-code Table 4.16) est à peu près similaire à celui du fédérateur F dans la version de base de DICCoop de base (voir Section 4.2.2). Les envois d'informations pour le groupe sont par contre effectuées vers les sites de traitements T_{ij} **et** vers le fédérateur global FG.

(1)**Pour** chaque information $I_{F_{l_k}}$ reçue
(2) insertion de $I_{F_{l_k}}$
(3) **Si** ($I_{F_{i_k}} recu \forall i$) alors
(4) calcul du seuil global $S_k = \sum_i s_{ik}$
(5) //prise de décision globale définitive de fréquence
(6) (smg_k passe de GU à GI ou à GF)
(7) **Si** ($S_k < seuil$) alors
(8) $smg_k =$ GI
(9) **Sinon** //i.e. $S_k \geq seuil$
(10) $smg_k =$ GF.
(11) **FinSi**
(12) **envoi** de $I_{FG_k} = (k, smg_k)$ aux fédérateurs F_i.
(13) **Sinon**
(14) calcul du seuil global $S_k = \sum_{irecu} s_{ik}$
(15) //tentative de prise de décision partielle de fréquence
(16) **Si** ($S_k > seuil$) alors
(17) $smg_k =$ SB
(18) **envoi** de $I_{FG_k} = (k, GF)$ aux fédérateurs F_i.
(19) **Sinon**
(20) calcul de $S'_k = \sum_{inonrecu} Card(D_i)$
(21) **Si** ($S_k + S'_k < seuil$) alors
(22) $smg_k =$ SC.
(23) **envoi** de $I_{FG_k} = (k, GI)$ aux fédérateurs F_i.
(24) **FinSi**
(25) **FinSi**
(26) **FinSi**
(27)**FinPour**

TAB. 4.15 – Génération de sur-ensembles sur le fédérateur global FG dans l'algorithme Méta DICCoop

Réception d'informations issues de FG avec marqueur msg_F. Pour chaque information I_{FG_k} reçue, on insère cette information en mettant à jour le marqueur smg_k associé On envoie $I_{Fi_k} = (k, mg_k, smg_k)$ aux sites de traitements T_{ij} dès que possible.

4.3.3 Fonctionnement des sites de traitement : T_{ij}

Le comportement des sites de traitements T_{ij} de la Méta version de l'algorithme DICCoop est plus ou moins similaire à celui des sites T_j dans la version de base (voir Section 4.2.2). Les communications sont effectuées vers le fédérateur de groupe F_i. Seule la génération des sur-ensembles d'itemsets est légèrement différente (voir pseuo-code Table 4.17).

Sur un site de traitement T_{ij}, pour chaque itemset k, on a un $R_{T_k} = (s_k, m_k, mg_k, smg_k)$:

- un support : s_k
- un marqueur local : m_k
- un marqueur de groupe : mg_k
- un marqueur super global : smg_k

On tente ici de générer les sur-ensembles K à partir des itemsets k (tels que $Card(k) =$

(1)**Pour** chaque information $I_{T_{ij_k}}$ reçue
(2) insertion de $I_{T_{ij_k}}$
(3) **Si** ($I_{T_{ij_k}} recu \forall j$) alors
(4) calcul du seuil global $S_k = \sum_j s_{ijk}$
(5) //prise de décision globale définitive de fréquence
(6) (mg_k passe de GU à GI ou à GF)
(7) **Si** ($S_k < seuil$) alors
(8) $mg_k =$ GI
(9) **Sinon** //i.e. $S_k \geq seuil$
(10) $mg_k =$ GF.
(11) **FinSi**
(12) **envoi** de $I_{Fi_k} = (k, mg_k)$ aux sites de traitements T_{ij} et a FG.
(13) **Sinon**
(14) calcul du seuil global $S_k = \sum_{jrecu} s_{ijk}$
(15) //tentative de prise de décision partielle de fréquence
(16) **Si** ($S_k > seuil$) alors
(17) $mg_k =$ SB
(18) **envoi** de $I_{Fi_k} = (k, GF)$ aux sites de traitements T_{ij} et a FG.
(19) **Sinon**
(20) calcul de $S'_k = \sum_{jnonrecu} Card(D_j)$
(21) **Si** ($S_k + S'_k < seuil$) alors
(22) $mg_k =$ SC.
(23) **envoi** de $I_{Fi_k} = (k, GI)$ aux sites de traitements T_{ij} et a FG.
(24) **FinSi**
(25) **FinSi**
(26) **FinSi**
(27)**FinPour**

TAB. 4.16 – Génération de sur-ensembles sur les fédérateurs de groupe F_i dans l'algorithme Méta DICCoop

$Card(K) - 1$ et $k \in K$) :

- soit sur base de fréquence locale ($m_k = SB$ ou $m_k = DB$)
- soit sur base de fréquence de groupe ($mg_k = GF$), avec $m_k = SC$ ou $m_k = DC$
- soit sur base de fréquence super globale ($smg_k = GF$), avec $m_k = SC$ ou $m_k = DC$

Réponses des sites T_{ij} au fédérateur de groupe F_i et Limitation des envois d'informations au sein d'une sous-grappe Les réponses au fédérateur de groupe, tout comme la limitation des communications en fonction des informations globales fonctionnent sur le même principe que dans la version de base de l'algorithme.

4.3.4 Digression concernant la génération des règles d'association

Une fois les itemsets fréquents générés (phase préalable et coûteuse à la génération de règles d'association), il reste à générer les règles à proprement parler, à partir des itemsets fréquents obtenus (voir Section 1.4.3).

(1)**Pour** chaque itemset ayant change d'etat de XC a XB
(2) //generation du superset
(3) generation des itemsets K telsque
(4) $\forall l \in K$ avec $Card(l) = Card(K) - 1$
(5) m_l = DB ou m_l = SB ou mg_l = GF ou smg_l = GF
(6)**Fin pour**

TAB. 4.17 – Génération de supersets sur les sites T_{ij} dans Méta DICCoop

Cette génération s'effectue grâce à une procédure GenRules (voir pseudo-code Table 4.18).

$\forall l_k \in F_k$, tel que $k \geq 2$ do
genrules(l_k, l_k)

l'algorithme GenRules :
Entrée : l_k : un $k-$itemset fréquent et
a_m : un $m-$itemset fréquent.
Sortie : l'ensemble $\mathcal{R}$ des règles d'association

(1) A = { a_{m-1}} tel que $a_{m-1} \subset a_m$
(2) **pour chaque** $a_{m-1} \in A$ **faire**
(3) $conf = support(l_k)$/support(a_{m-1})
(4) si ($conf \geq minconf$) alors
(5) $\mathcal{R} = \mathcal{R} \cup R(a_{m-1} \rightarrow (l_k - a_{m-1}))$
(6) de confiance $conf$ et de support $support(l_k)$
(7) si ($m - 1 > 1$) alors
(8) $\mathcal{R}$ = $\mathcal{R} \cup$ genrules(l_k, a_{m-1})
(9) //(generation avec les sous-ensembles de a_{m-1} comme antecedent de regle)
(10) **retourne** $\mathcal{R}$

TAB. 4.18 – La procédure GenRules

Chaque itemset fréquent obtenu ($l_k \in F_k$) est découpé en combinaisons de deux itemsets (l'un pour l'antécédent de la règles, l'autre pour la conclusion de la règle).

Les combinaisons amenant à une mesure statistique (confiance ou autre, voir 1.4.1), de qualité suffisante (supériorité à un seuil fixé) forment une règle qui est ajoutée à l'ensemble $\mathcal{R}$ des règles générées (voir lignes 4-5-6 Table 4.18).

Il existe d'autres algorithmes de génération des règles d'association à partir des itemsets fréquents, en particulier pour éviter la génération de règles dites redondantes.
De plus, de nombreuses mesures statistiques de qualité des règles existent (voir [7]), la tendance actuelle consistant à ne plus se baser sur une seule de ces mesures, mais sur une

combinaison de mesures pour sélectionner les règles pertinentes.

A la fin de l'algorithme DICCoop, on dispose de deux types d'itemsets fréquents :

- les itemsets globalement fréquents sur la totalité des instances ;
- les itemsets localement fréquents sur chacun des fragments d'instances (résultats locaux intéressants voir Section 5.2) ;

Pour générer les règles d'association par l'algorithme DICCoop, on a besoin d'accèder à la totalité des itemsets, ainsi qu'aux supports qui leur sont associés (pour effectuer le calcul des mesures de qualité, confiance ou autre).

On peut effectuer la génération de règles de manière indépendante sur les deux types d'informations obtenues, c'est-à-dire :

- les itemsets fréquents localement pour un fragment de données D_i, et qui sont stockés sur le noeud T_i en charge du traitement de ce fragment. Les supports associés à ces itemsets dont également stockés sur T_i.
- les itemsets fréquents globalement sur la totalité des instances, et qui sont stockés sur le fédérateur, ainsi que les supports globaux qui leur sont associés.

Les itemsets fréquents locaux correspondant à des profils d'instances similaires identifiés par clustering (voir Section 5.2). Il est intéressant de générer les règles internes à chaque fragment de données.
Cette génération est effectuée par l'algorithme genRules classique exécuté indépendamment sur chaque site de traitement T_i.
Il en découle un ensemble de règles $\mathcal{R}_i$: règles d'association associées au fragment de donnée D_i (et donc à la sous-population représentée dans D_i).

Le but initial des travaux étant de générer les règles d'association sur l'ensemble des instances de la base, les itemsets fréquents globalement sont utilisés pour effectuer la génération de ces règles globales, par l'algorithme genRules classique exécuté sur le site fédérateur F.
Il en découle un ensemble de règles $\mathcal{R}$: règles d'association associées à la base de données traitée.

4.4 Evaluation de la génération collaborative des itemsets fréquents

Nous présentons dans cette section des résultats d'expérimentations sur l'algorithme DICCoop.

Afin de ne pas intéragir, dans les conclusions, avec l'algorithme CDP utilisé pour la fragmentation intelligente des données (voir Partie II), les données utilisées ici ont été générées de manière à assurer les critères d'intelligence dans les fragments.

4.4.1 Conditions d'expérimentations

Les implémentations java utilisées lors des expérimentations effectuent seulement en partie le recouvrement de communications, de manière à permettre une observation des comportements engendrés par les apports d'informations extérieures. Les communications sont toutefois non synchronisées donc non bloquantes.
De plus, les temps complets d'exécution présentés sur les figures (*exécution complète*) incluent des traitements liés au traçage du programme.
Ces deux facteurs expliquent la différence significative que l'on peut obtenir entre le temps de traitement effectif (comptage + accès à la structure de données) et le temps complet d'exécution sur chaque site.
Des mécanismes d'invocation de méthodes asynchrones offerts par le projet ADAJ (Adaptive Distributed Applications in Java, voir Section 5.1.2) permettront, lors de futures expérimentations sur Grid'5000 (voir Annexe D), de maximiser le recouvrement de communications, et donc d'optimiser les temps complets d'exécution.
Les temps de communications des sites T_i (ou T_{ij}) vers le fédérateur incluent le temps de traitement de ces informations sur le fédérateur (c'est la réception d'informations spontanées qui engendrent des phases de traitement sur les fédérateurs), même si ces temps de traitements sont négligeables, (ceci du fait de la non synchronisation des horloges des différents noeuds de l'architecture utilisée).
Les temps de comptage et d'accès à la structure de données présentés ne sont quant à eux pas perturbés par des phases de traçage, ni temps de communications.

Les expériences ont été réalisées sur une grappe homogène de stations (Pentium 4 2.66Ghz, 256Mo, système d'exploitation Windows 2000, type salle de travaux pratiques universitaires).
Ce type d'architecture est non dédiée. Lors des expérimentations, les processeurs des machines étaient réservés au traitement, mais pas le réseau. Le fait d'utiliser ce type d'architecture pour les expérimentations permet de ne pas baser les observations uniquement sur l'utilisation d'une architecture dédiée telle que Grid'5000. De même, la puissance des processeurs utilisées (2.66Ghz) et surtout leur capacité mémoire limitée (256Mo) sont largement inférieures aux capacités d'une grille de calcul (actuellement en moyenne 2Ghz et surtout de 2 à 4GB de mémoire sur chaque noeud de Grid'5000). Le portage de l'application sur Grid'5000 promet donc des améliorations non négligeables des performances de l'algorithme DICCoop.

Les mesures effectuées concernent deux aspects :

1. la qualité de la méthode par rapport aux résultats générés et aux taux d'informations (itemsets fréquents et infréquents) générées globalement et localement ;
2. les performances de la méthode en terme de temps d'exécution et de temps de communication.

4.4.2 Remarque préliminaire

Une première remarque qui peut être effectuée concerne la répartition des temps de traitement dans l'algorithme DIC. Que ce soit par une exécution séquentielle de l'algorithme DIC classique (voir Section 4.1.2) ou par des exécutions parallèles de celui-ci sur plusieurs fragments (exécutions parallèles indépendantes de l'algorithme DIC classique ou exécutions parallèles collaboratives de l'algorithme DICCoop, voir Section 4.2), le temps de traitement peut se décomposer en deux étapes principales à chaque itération :

- le comptage des supports d'itemsets (qui nécessite l'accès aux données elles-même, c'est-à-dire aux instances, et l'accès à la structure de données utilisée par la méthode, le treillis) : **accés données** sur la figure 4.12 ;
- la génération des sur-ensembles de niveaux supérieurs et conclusions intermédiaires (qui nécessite uniquement l'accès à la structure de données) : **accés structure** sur la figure 4.12.

La première étape peut être considérée comme très handicapante si l'on considère un grand nombre d'instances (avec un grand nombre d'attributs pour chacune d'entre elles).

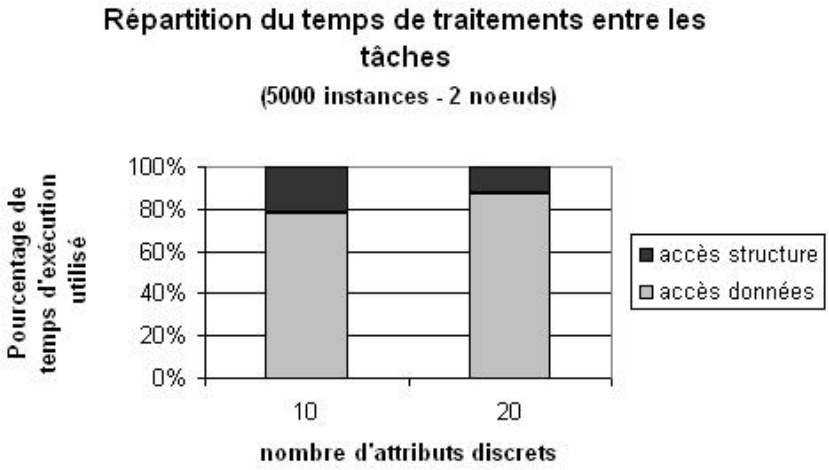


FIG. 4.12 – Observation des temps consacrés aux différentes étapes dans l'algorithme DIC (accès aux instances, accès aux structures) (5000 instances - exécution sur 2 noeuds)

La première observation concerne donc la répartition de temps d'exécution entre ces deux étapes, (accès données et accès structure) cette répartition en fonction du nombre d'attributs considérés est représentée la Figure 4.12. On peut constater que l'étape nécessitant l'accès aux instances de données représente de 80 % (pour 5000 instances de 10 attributs) à près de 90% (pour 5000 instances de 20 attributs) du temps total de traitement. Il est à noter que cette étape n'est pas réalisée en une seule fois, mais par petites parties à chaque itération de l'algorithme, et que lors de chaque itération toutes les instances ne sont pas accèdées, mais seulement une tranche (principe de l'algorithme DIC).

Les expérimentations présentées dans la suite préciseront la répartition des temps de traitement de ces deux tâches en fonction de l'évolution du taux de parallélisme et du nombre d'instances utilisées.

4.4.3 Génération de données

Les données ont été générées de manière à assurer les critères d'intelligence retenus (voir Section 2.1), c'est-à-dire un maximum d'items présents communs aux instances d'un groupe et un maximum d'items absents au sein d'un groupe d'instances. Des détails concernant le mode de fonctionnement du générateur sont disponibles en annexe E.

Le générateur fonctionne sur un mode de probabilités sur chaque attribut pour chaque fragment d'instances.

4.4.4 Comparaisons effectuées

4.4.5 Résultats générés

Les résultats générés par l'algorithme DICCoop en version classique ou en méta version (itemsets globaux fréquents sur l'ensemble des instances des fragments), sont identiques à ceux obtenus par un traitement séquentiel des données. On obtient donc bien le résultat recherché.

Les itemsets fréquents globaux obtenus sont accompagnés de leurs supports (mesure statistique nécessaire à la génération de règles d'association, voir Section 4.3.4). Il n'est pas nécessaire d'effectuer une phase de validation (comme dans les premiers travaux d'étude de traitements distribués par fragmentation intelligente, voir Section 2.3.1), celle-ci est réalisée durant le traitement grâce au principe de collaboration.

De plus, les résultats locaux (itemsets locaux fréquents sur un fragment intelligent d'instances) sont disponibles sur les sites T_i dans la version classique, et sur les fédérateurs de groupe F_i dans la méta version, apportant une connaissance supplémentaire.

4.4.6 Temps d'exécution

Comparaison des temps d'exécution par rapport à une version séquentielle classique

La figure 4.13 présente la comparaison entre :

- une exécution par algorithme DIC classique : **séquentiel**,
- et une exécution par traitements collaboratifs (algorithme DICCoop) en mode tout communiquer : **cumul de 8 sites**

Les mesures présentées sur la figure 4.13 sont : le temps total de traitement sur chaque site (**exécution complète**), le détail du temps de **comptage** des supports (accès aux instances et à la structure de données) et du temps de génération des sur-ensembles et des conclusions intermédiaires par **parcours** de la structure de données uniquement.
Les temps de communications engendrés par chaque site sont également représentés :

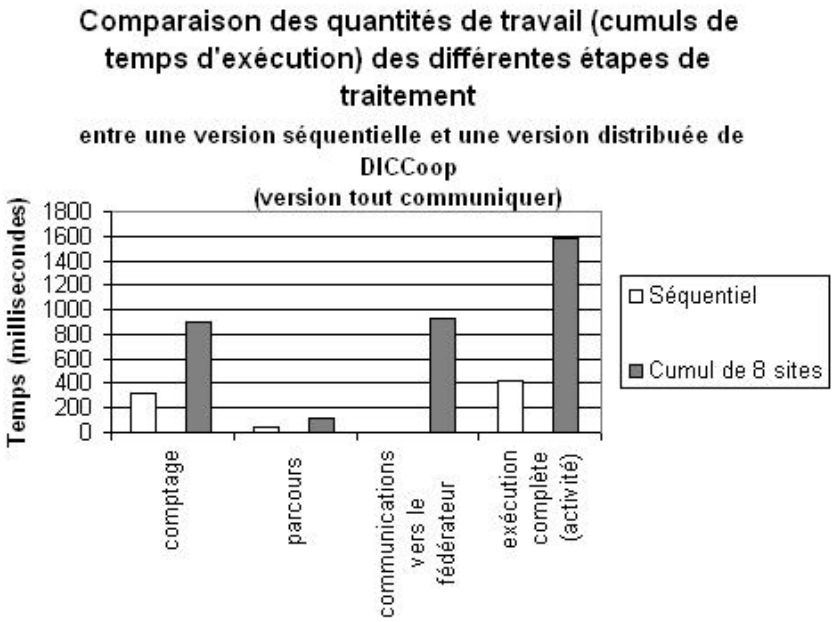


FIG. 4.13 – Comparaison des quantités de travail (temps d'exécution cumulé) de l'algorithme DIC entre une version séquentielle et une version collaborative DICCoop (20 attributs - 5000 instances - 8 noeuds de traitement - version DICCoop tout communiquer)

communications vers le fédérateur.

La version collaborative DICCoop nécessite une quantité de travail (temps d'exécution complète) plus importante que la version DIC classique (près de quatre fois plus de temps d'exécution que la version séquentielle, environ 1600 ms contre près de 400). Les temps des différentes étapes sont comparés également sur la figure 4.13. Ainsi, on peut constater que la phase de comptage nécessite trois fois plus de travail en version collaborative qu'en version séquentielle, et que la phase de parcours de la structure nécessite deux fois plus de temps en version collaborative qu'en version séquentielle. Le temps de communications n'existe pas dans la version classique de DIC, mais on peut remarquer (toujours sur la figure 4.13), que le temps de communications est équivalent au temps de comptage.
La quantité de travail plus importante dans la version collaborative distribuée DICCoop peut paraître surprenante du fait de la parallélisation, mais est explicable de deux manière : le type de traitement effectué (et les résultats obtenus dans chaque version) et les coûts fixes dans les deux méthodes pour certaines étapes.

Il faut noter que cette exécution a été réalisée sur un nombre d'instances restreint (5000) et sur un nombre d'attributs également restreint (20).
Des expériences effectuées sur une plus grande quantité de données (un plus grand nombre d'instances et un plus grand nombre d'attributs) amènent à des dépassements de capacités mémoire sur la machine effectuant le traitement séquentiel, ne permettant pas d'obtenir un référentiel.
De plus, les deux versions (DIC classique et version collaborative DICCoop) n'effectuent

pas le même traitement et fournissent des quantités d'informations différentes. La version séquentielle classique fournit bien les itemsets fréquents globaux (principaux résultats recherchés).
La version par traitements collaboratifs DICCoop (tout communiquer) fournit les itemsets fréquents **globaux et locaux**, fournissant ainsi deux types de connaissances : le savoir principal recherché c'est-à-dire les itemsets globalement fréquents, mais également des résultats propres à chaque fragment qui correspondent à une description de l'intelligence contenue dans les fragments (voir Section 5.2).
La complexité de parcours de la structure de données dépend du nombre d'éléments de cette structure. La version collaborative distribuée fournissant plus de résultats (générant plus d'itemsets fréquents, les globaux et les locaux), la quantité de travail nécessaire à cette phase est plus importante en version distribuée.

Les conclusions ont été effectuées sur des temps d'exécution cumulés. La figure 4.14 présente les temps moyens et maximum de traitement sur les différents sites dans la version collaborative distribuée par rapport au temps d'exécution de ces mêmes étapes dans la version séquentielle classique.
La figure 4.14 présente les mêmes mesures que la figure 4.13, à savoir : le temps d'**exécution complète** (le temps total de traitement sur chaque site), le temps de **comptage** des supports (accès aux instances et à la structure de données), le temps de **parcours** (génération des sur-ensembles et de conclusions intermédiaires avec accès à la structure de données uniquement), et les **communications vers le fédérateur** .

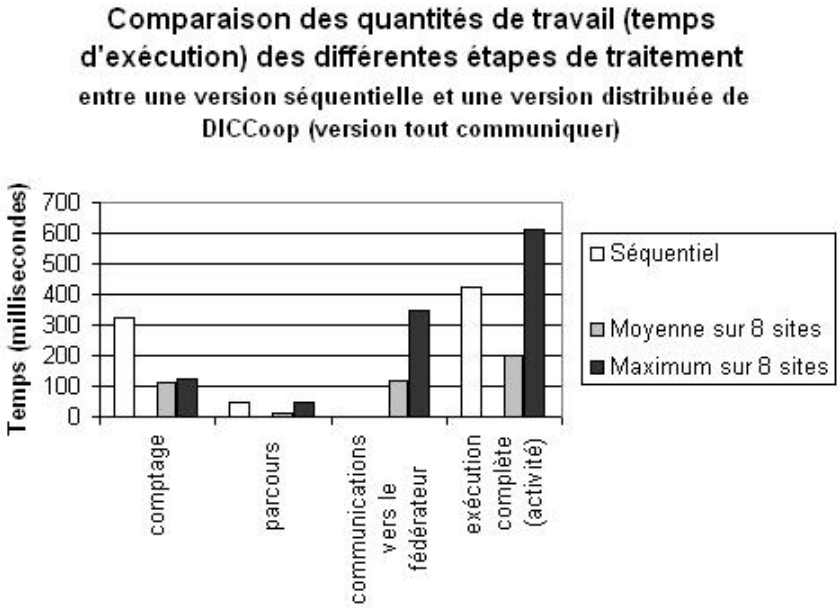


FIG. 4.14 – Comparaison des quantités de travail (temps d'exécution minimum et maximum) de l'algorithme DIC entre une version séquentielle et une version collaborative DICCoop (20 attributs - 5000 instances - 8 noeuds de traitement - version DICCoop tout communiquer)

On remarque que le temps d'exécution moyen sur chaque site est deux fois moins élevé dans la version distribuée que dans la version séquentielle.
L'observation des temps d'exécution sur la figure 4.14 permet de confirmer que tous les sites n'effectuent pas la même quantité de travail ou effectue une quantité de travail comparable mais à des vitesses différentes (voir temps maximum d'exécution des étapes et temps maximum d'exécution complète). Certains sites sont plus lents et viennent ralentir le traitement complet. Ce phénomène sera plus détaillé dans la partie "Comparaisons liées au taux de parallélisme" (voir Section 4.4.6).

La version par traitements collaboratifs apparaît intéressante par rapport aux résultats apportés (résultats globaux ET locaux), et par le fait qu'elle permet de reculer les limites de capacités de traitement (voir plus haut, dépassement de capacité mémoire obtenu en version séquentielle).
Le surcoût en temps de traitement obtenu sur de faible quantité de données peut être limité par l'augmentation des quantités de données traitées (voir plus loin).

Comparaison des temps d'exécution selon le mode de communication utilisé

Nous comparons ici, les temps d'exécution et temps de communications engendrées dans différentes versions de l'algorithme DICCoop.

On peut distinguer trois modes de communications :

- une exécution **sans communication** (avec les problèmes sur les résultats obtenus décrits ci-dessus).
- une exécution avec communication aussi souvent que nécessaire : **tout communiquer**. A la fin de chaque itération locale, si des résultats locaux sont disponibes sur les sites de traitement T_i (ou sur le fédérateur F) (quel que soit le nombre de ces résultats), ils sont communiqués au fédérateur F (ou aux sites de traitements T_i). Attention toutefois, ce mode de communications peut surcharger le réseau par de petites communications coûteuses.
- une exécution avec **communications paramétrées**, plus adaptée à une exécution distribuée sur grappe de stations ou grille de calcul. Deux paramètres sont utilisés pour limiter les communications, la taille de la communication à effectuer et le délai depuis la dernière communication.
  Le premier paramètre utilisé concerne la quantité de résultats (informations) à transmettre. Un **seuil minimal de taille de communications** est fixé. A la fin de chaque itération locale de l'algorithme DICCoop, si le nombre d'informations disponibles est supérieur ou égal au seuil, la communication est réalisée. Sinon, les résultats sont mis en attente, et seront envoyés plus tard, dès que suffisamment de résultats seront disponibles pour respecter la taille minimale requise pour un envoi.
  Ce paramètre permet de ne pas engendrer de trop nombreuses communications de petites tailles. En effet, il vaut mieux réaliser quelques communications de grande taille que beaucoup de petites communications sur le type d'insfrastructure parallèle visée (grappe de stations ou grille de calcul). La valeur de ce paramètre a été fixé à 10 (envoi d'un minimum de 10 informations concernant les itemsets) dans les

expérimentations présentées.

Le second paramètre utilisé concerne le délai de temps écoulé depuis la dernière communication. Ce délai est lié au nombre d'itérations de la méthode. Un **seuil maximal de délai de communications** est fixé, c'est-à-dire un nombre maximal d'itérations autorisées sans effectuer de communication. A la fin de chaque itération locale de l'algorithme DICCoop, si le nombre d'informations disponibles est insuffisant pour effectuer une communication selon le critère de taille, on regarde depuis combien d'itérations il n'y a pas eu d'envoi d'informations. Si le nombre d'itérations effectuées depuis la dernière communication est supérieur ou égal au seuil de délai, la communication est effectuée, même si la quantité d'informations est insuffisante.
Ce paramètre permet de ne pas envoyer les informations trop tard. Les informations sont utilisées pour construire les résultats globaux (à partir de résultats locaux) et pour enrichir, anticiper ou limiter les traitements locaux (résultats globaux). Les informations ne doivent pas être envoyées trop tardivement, auquel cas, elles pourraient s'avérer moins pertinentes.
La valeur de ce paramètre a été fixé à 5 (5 itérations sans envoi) dans les expérimentations présentées.

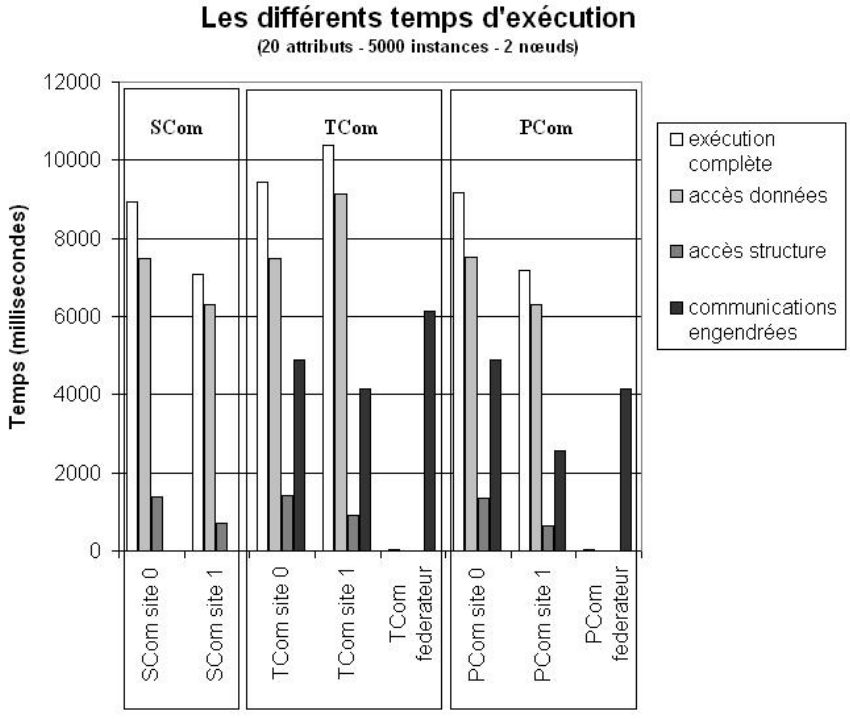


FIG. 4.15 – Comparaison des temps d'exécution entre les différentes versions de communications de l'algorithme DIC (20 attributs - 5000 instances - 2 noeuds de traitement)

La Figure 4.15 présente la comparaison des temps d'exécution entre les trois modes de

communication cités précedemment, avec à chaque fois une exécution de l'algorithme DICCoop sur deux sites de traitement T_0 et T_1 :

- une exécution par traitements parallèles indépendants sur chaque fragment (également par l'algorithme DIC classique) : **sans communication (SCom)**
  (cette exécution nécessite une phase de validation a posteriori dont le temps de calcul n'est pas pas inclu dans les mesures présentées),
- et une exécution par traitements collaboratifs (algorithme DICCoop), avec des communications aussi souvent que nécessaire : **tout communiquer (TCom)**
- et une exécution par traitements collaboratifs (algorithme DICCoop), avec des communications limitées par des critères de taille minimale ou de délai minimal : **communications paramétrées (PCom)**

Les mesures présentées sur la figure 4.15 sont : le temps total de traitement sur chaque site (**exécution complète**), le détail du temps de comptage des supports : **accès données** et du temps de génération des sur-ensembles et des conclusions intermédiaires : **accès structure**, ainsi que les temps de **communications engendrées** par chaque site.

Dans la version sans collaboration (SCom), le fédérateur n'est pas représenté puisqu'il n'est pas utilisé.

La version sans communication (SCom) nécessite un peu moins de temps de traitement sur chaque site que la version collaborative DICCoop tout communiquer (TCom) (5% de temps de traitement en moins, donc négligeable), et des temps de traitement équivalent sur chaque site par rapport à la version collaborative DICCoop avec communications paramétrées (PCom).

Les temps d'exécution ne doivent pas être évalués seuls, en effet il convient de les **croiser avec les résultats obtenus**.

La version par traitements parallèles indépendants (SCom), ne permet pas de générer les itemsets fréquents globaux (ou du moins pas directement). On obtient uniquement les itemsets fréquents locaux sur chaque fragment d'instances. Une phase de validation doit être réalisée a posteriori, le problème étant que cette validation nécessite l'accès aux instances (voir Résultats d'expérimentation préalables, Section 2.3.1).
Sans la validation, le seul résultat définitif obtenu est donc la connaissance des itemsets fréquents locaux sur chaque fragment. Cependant, ces résultats sont intéressants et représentent un savoir relatif à la distribution intelligente (voir Section 5.2).

Les versions par traitements collaboratifs DICCoop (TCom et PCom) fournissent à la fois les itemsets fréquents **globaux et locaux** (voir Section 5.2).

Pour ce qui est des temps de communications, ils sont bien sûr plus élevés dans la version tout communiquer (TCom), y compris pour les communications engendrées par le fédérateur F. Les temps de communications engendrés par les sites T_i sont supérieurs de 10 à 15% aux temps de communications engendrés par ces mêmes sites (traitant les même fragments d'instances) en version avec communications paramétrées (PCom). Pour ce qui est du fédérateur F, les temps de communications qu'il engendre dans la version

tout communiquer (TCom) sont en moyenne 30% plus elevés que ceux engendrés par le fédérateur F en version avec communications paramétrées (PCom).

La version collaborative DICCoop avec communications paramétrées (PCom) apparait comme la solution la plus adaptée à une exécution sur une plateforme de type grappe de stations ou grille de calcul. Cependant, dans les expérimentations, les valeurs des deux paramètres (seuil minimal de taille de communication et seuil maximal de délai sans communication) ont été fixées pour l'exécution. Il serait intéressant de faire évoluer ces valeurs en fonction du déroulement de l'exécution (valeurs adaptatives), ce qui permettrait de prendre en compte automatiquement non seulement les spécificités du support d'exécution mais également les spécificités des données utilisées.
De plus, les temps de communications peuvent, en partie, être recouverts par du traitement (voir Conditions d'expérimentations Section 4.4.1).

Evolution de temps d'exécution en fonction du taux de parallélisme

Cette section décrit l'évolution des temps de traitements sur chaque site et des quantités de travail (temps cumulés d'exécution) en fonction du nombre de noeuds utilisé. Les expérimentations présentées sont effectuées sur 16 groupes homogènes de 2000 instances chacuns, soit 32000 instances au total.

Dans cette section, deux types de figures sont présentées, celles liées à l'exécution des différentes tâches de traitement sur les noeuds, celles liées au traitement effectué sur le fédérateur. Dans les figures représentant l'exécution de traitement sur les différents sites, trois types de mesures sont présentées : **envois au fédérateur** d'informations locales, **parcours** de la structure de données locales pour générer les sur-ensembles d'itemsets, et **comptage** des supports avec accès aux instances.
Dans les figures représentant le traitement effectué sur le fédérateur, deux types de mesures de temps sont présentées : le temps **réception** d'informations envoyées par les noeuds, et le temps de **traitement** de ces informations sur le fédérateur.

Remarque :
Le traitement de 16 groupes homogènes par l'algorithme DICCoop nécessite normalement l'utilisation de 16 noeuds (principe de distribution intelligente proposé par le projet DisDaMin), pour générer à la fois les itemsets fréquents globaux (ceux initialement recherchés), et les itemsets fréquents locaux à chaque fragment homogène (résultat supplémentaire fourni par la méthode).
Il faut noter ici, que les versions utilisant moins de 16 noeuds de traitement fournieront, pour résultats locaux, les itemsets fréquents sur un regroupement de fragments (regroupement de 2 fragments intelligents par noeud pour l'exécution sur 8 noeuds, et regroupement de 4 fragments intelligents par noeud pour l'exécution sur 4 noeuds).
Les connaissances locales obtenues sont donc moins riches, elles ne sont que partiellement liées aux découpages en fragments intelligents. De plus, le regroupement de fragments amènent à n'obtenir que partiellement la spécialisation des traitements attendue par

l'utilisation d'une fragmentation intelligente (et donc la diminuton complète de leur complexité de traitement).
Nous avons dû utiliser les mêmes données dans les différentes versions (sur 4, 8 et 16 noeuds), afin de pouvoir comparer les temps d'exécution obtenus. Or ces données sont générées de manière à obtenir des framents intelligents, dont le nombre est fixé lors de la génération (voir générateur Annexe E), 16 fragments pour les données utilisées ici puisque l'on utilise jusqu'à 16 noeuds.
Le taux de parallélisme à utiliser est lié, dans la version classique de l'algorithme DICCoop, au nombre de fragments intelligents obtenus par la fragmentation intelligente. Cependant, le taux de parallélisme peut être augmenté par l'utilisation de la méta version de l'algorithme DICCoop (voir expérimentations de la méta version, Section 4.4.6).
Les expérimentations présentées ici permettent cependant d'observer le comportement de la méthode DICCoop, en fonction de son déploiement sur 4, 8 ou 16 noeuds pour 16 fragments.

Le traitement de X fragments sur Y noeuds (avec X>Y) nécessite normalement l'utilisation de X composants effectuant l'algorithme DICCoop, répartis sur Y noeuds. Nous n'avons pas évalué cette distribution de X composants sur Y noeuds, car les composants présents sur un même noeud provoquent dans ce cas des changements de contexte (passage d'un thread à l'autre dans la JVM) fortement handicapants et qui faussent les mesures effectuées.

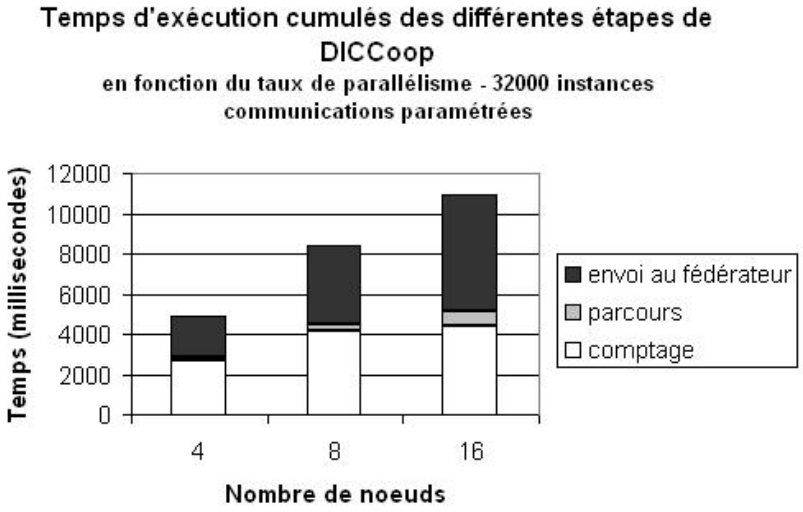


FIG. 4.16 – Comparaison des quantités de travail (temps d'exécution cumulés) en fonction du taux de parallélisme utilisé pour l'algorithme DICCoop (30 attributs - 32000 instances - communications paramétrées)

La figure 4.16 présente l'évolution des quantités de travail (cumul des temps d'exécution) des différentes tâches pour des exécutions sur 4, 8 et 16 noeuds (comptage avec accès aux instances, parcours de la structure de données pour la génération des sur-ensembles

d'itemsets et communications des sites vers le fédérateur).
La figure 4.17 présente les temps moyens d'exécution des tâches sur les noeuds.

On notera que la quantité totale de travail (temps cumulés d'exécution, voir Figure 4.16) augmente avec le nombre de noeuds. Ainsi, cette quantité de travail sur 8 noeuds est égale à 1,7 fois celle sur 4 noeuds (8450 ms contre 4900 ms). Celle sur 16 noeuds est égale à 1,3 fois celle sur 8 noeuds (11000 ms contre 8450 ms) et 2,2 fois celle sur 4 noeuds (11000 ms contre 4900 ms).
L'augmentation des quantités totales de travail provient essentiellement d'une augmentation des temps de communications (**envoi** des informations locales) : 2100 ms pour 4 noeuds, contre 3900 ms pour 8 noeuds et 5800 ms pour 16 noeuds.
Les temps moyens de communications vers le fédérateur sur chaque noeud (voir Figure 4.17) diminuent sur chaque noeud lorsque le nombre de noeuds augmente, mais pas de manière proportionnelle à l'augmentation du nombre de noeuds.
On passe d'un temps moyen de communications engendrées par noeud de 520 ms pour une exécution sur 4 noeuds, à 490 ms pour 8 noeuds et 360 ms pour 16 noeuds.
Tous les sites informent le fédérateur avec des informations locales en quantité équivalente, amenant à une saturation des communications vers celui-ci, d'où un temps de communication plus élevé.

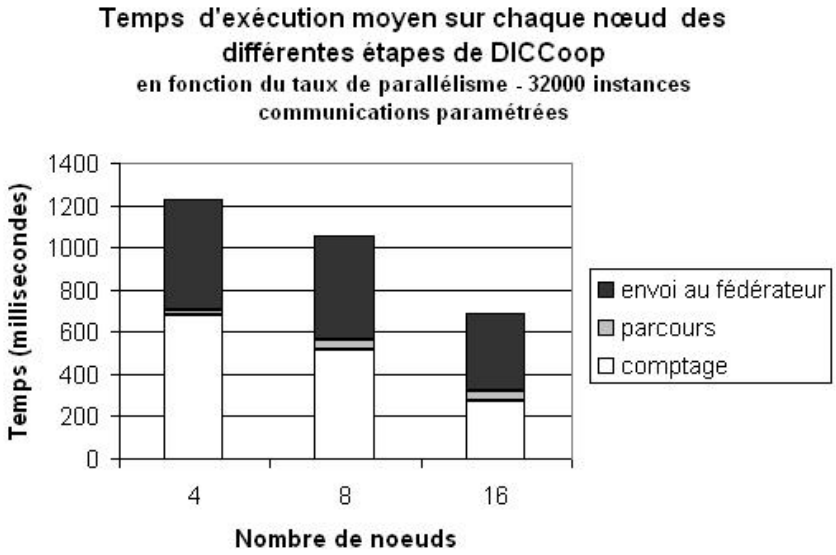


FIG. 4.17 – Comparaison des temps moyens d'exécution sur chaque site en fonction du taux de parallélisme utilisé pour l'algorithme DICCoop (30 attributs - 32000 instances - communications paramétrées)

Les temps cumulés d'accès aux instances (**comptage** des supports) augmentent avec le nombre de noeuds. Ils passent de 2700 ms pour 4 noeuds, à 4150 ms pour 8 noeuds et évolue peu entre une exécution sur 8 ou 16 noeuds (4150 ms pour 8 noeuds et 4400 ms pour 16 noeuds).
Pour rappel, le temps de la tâche de comptage nécessite l'accès aux instances **et** à la struc-

ture de données.
L'observation des temps moyens d'exécution de cette tâche de comptage sur chaque site (voir Figure 4.17) confirme cependant bien une diminution du temps nécessaire sur chaque noeud (du fait de la diminution du nombre d'instances), avec un passage de 680 ms sur 4 noeuds, à 515 ms sur 8 noeuds et 275 ms sur 16 noeuds.

Par contre, les temps cumulés de parcours de la structure de données augmentent avec le nombre de noeuds. Dans l'exécution sur 8 noeuds, ces temps cumulés de parcours de la structure sont plus de trois fois plus élevés que dans l'exécution sur 4 noeuds (390 ms sur 8 noeuds contre 110 ms sur 4 noeuds), et deux fois plus élevés dans l'exécution sur 16 noeuds que dans l'exécution sur 8 noeuds (740 ms sur 16 noeuds contre 390 ms sur 8 noeuds).
Ceci est dû au fait que tous les noeuds effectuent le parcours d'une structure de données dont la taille dépend des itemsets examinés localement. La taille de la structure à parcourir n'est pas proportionnelle au taux de distribution.

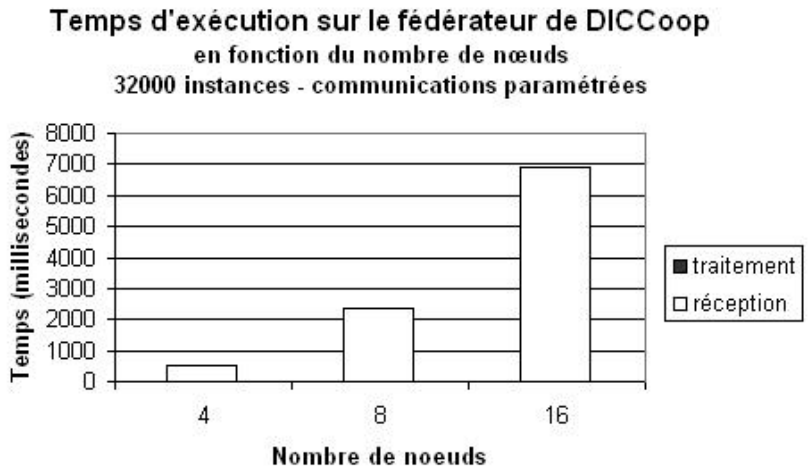


FIG. 4.18 – Comparaison des temps d'exécution sur le fédérateur en fonction du taux de parallélisme utilisé pour l'algorithme DICCoop (30 attributs - 32000 instances - communications paramétrées)

Les observations sont réalisées sur les temps cumulés d'exécution et les temps moyens sur chaque noeud.
Les temps minimum et maximum d'exécution sur chaque noeud sont présentés en Annexe C. On peut simplement préciser ici que le retard dans l'exécution (temps maximum) de certains sites augmente avec le taux de parallélisme, alors que, dans le même temps, l'avance des noeuds plus rapides est plus importante (grâce, en particulier, à la diminution du temps de traitement de la tâche de comptage de support). Le déséquilibre de charge augmente avec le taux de paralélisme, les noeuds les plus rapides sont rapidemment inactifs et pourraient être utilisés à d'autres tâches, ou permettre un déplacement de traitement.

Les observations effectuées sur les temps d'exécution des différents noeuds sont confir-

més par les temps d'exécution sur le fédérateur (voir Figure 4.18).
Ainsi, les envois d'informations locales par les noeuds vers le fédérateur augmentent considérablement avec le nombre de noeuds. Le fédérateur consacre ainsi 515 ms à la réception d'informations pour 4 noeuds, contre 2350 ms pour 8 noeuds et 6900 ms pour 16 noeuds.
On a clairement une saturation du fédérateur qui peut être limité par un passage à la méta version de l'algorithme DICCoop (voir Section 4.4.6).
Le temps de traitement des informations locales par le fédérateur est quant à lui négligeable par rapport au temps de communications. Les quantités d'informations envoyées et reçues par le fédérateur sont données en annexe C. Ces quantités augmentent avec le taux de parallélisme.

Bilan des communications engendrées : les communications peuvent être recouvertes en partie par des instructions CPU. Elles le sont partiellement dans les expérimentations réalisées, mais ceci peut encore être amélioré (voir Conditions d'expérimentations Section 4.4.1).
Le principe collaboratif de l'algorithme DICCoop réside sur ces communications, elles sont nécessaires à l'obtention des résultats locaux et globaux (itemsets fréquents).

Evolution de temps d'exécution en fonction du nombre d'instances

Cette section décrit l'évolution des temps de traitements sur chaque site et des quantités de travail (temps cumulés d'exécution) en fonction du nombre d'instances utilisé. Les expérimentations présentées sont effectuées sur 16 noeuds avec 16 groupes homogènes de 2000 instances chacuns (soit 32000 instances au total) et 4000 instances chacun (soit 64000 instances au total).
Les remarques effectuées en début de section 4.4.6, concernant le traitement sur un même noeud d'un regroupement de fragments homogènes, ne sont donc pas valables ici.

Deux types de figures sont présentées, celles liées à l'exécution des différentes tâches de traitement sur les noeuds, celles liées au traitement effectué sur le fédérateur. Dans les figures représentant l'exécution de traitement sur les différents sites, trois types de mesures sont présentées : **envois au fédérateur** d'informations locales, **parcours** de la structure de données locales pour générer les sur-ensembles d'itemsets, et **comptage** des supports avec accès aux instances.
Dans les figures représentant le traitement effectué sur le fédérateur, deux types de mesures de temps sont présentées : le temps **réception** d'informations envoyées par les noeuds, et le temps de **traitement** de ces informations sur le fédérateur.

La figure 4.19 présente l'évolution des quantités de travail (cumul des temps d'exécution) des différentes tâches pour des exécutions sur 32000 et 64000 instances (comptage avec accès aux instances, parcours de la structure de données pour la génération des sur-ensembles d'itemsets et communications des sites vers le fédérateur).
La figure 4.20 présente les temps moyens d'exécution de ces tâches sur les noeuds.

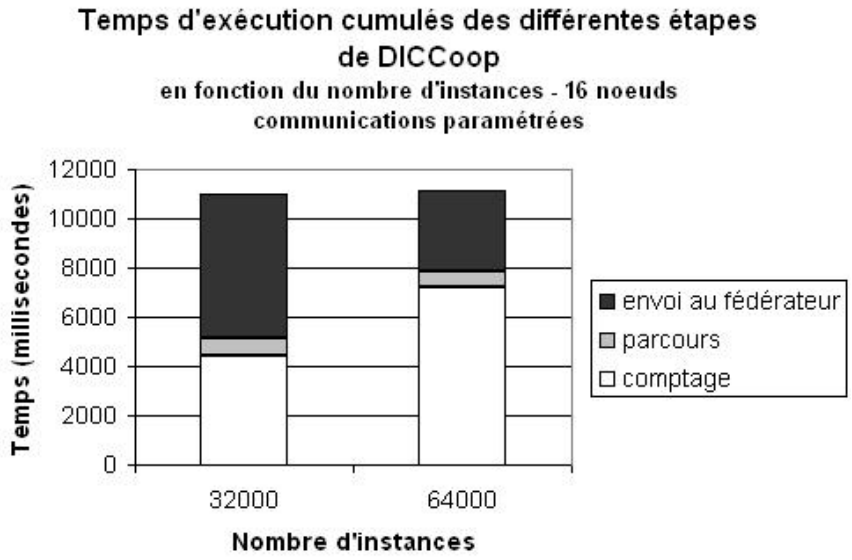


FIG. 4.19 – Comparaison des quantités de travail (temps d'exécution cumulés) en fonction du nombre d'instances pour l'algorithme DICCoop (30 attributs - 16 noeuds - communications paramétrées)

On notera que la quantité totale de travail sur 32000 ou 64000 instances est presque identique (11000 ms contre 11200 ms).

Cependant on note des différences significatives dans les évolutions des quantités de travail (temps d'exécution cumulé) des différentes tâches. Ainsi, la quantité de travail associée au **comptage** des supports augmente (ce qui est cohérent avec l'augmentation du nombre d'instances sur chaque noeud qui passe de 4400 ms pour 32000 instances à 7200 ms pour 64000 instances). Tandis que le temps réservé au comptage augmente, la quantité de travail liée aux **communications** diminue (5800 ms pour 32000 instances contre 3300 ms pour 64000 instances).

La quantité de travail associée au **parcours** de la structure de données reste stable (740 ms pour 32000 instances contre 640 ms pour 64000 instances).

Les observations des temps moyens d' exécution sur chaque noeud sont similaires aux observations des quantités de travail (stabilité des temps cumulés de traitement avec même phénomène d'augmentation des temps de comptage et diminution des temps de communications).
Les temps minimum et maximum sur chaque noeud sont présentés en Annexe C. On peut simplement préciser ici que les noeuds retardataires dans l'exécution (temps maximum) augmentent leur retard avec l'augmentation du nombre d'instances (mais en proportion moindre que dans les expérimentations basées sur l'augmentation du taux de parallélisme), alors qu'une fois encore, l'avance des noeuds plus rapides augmente (toujours en proportion moindre que dans les expérimentations basées sur l'augmentation du taux de parrallélisme).

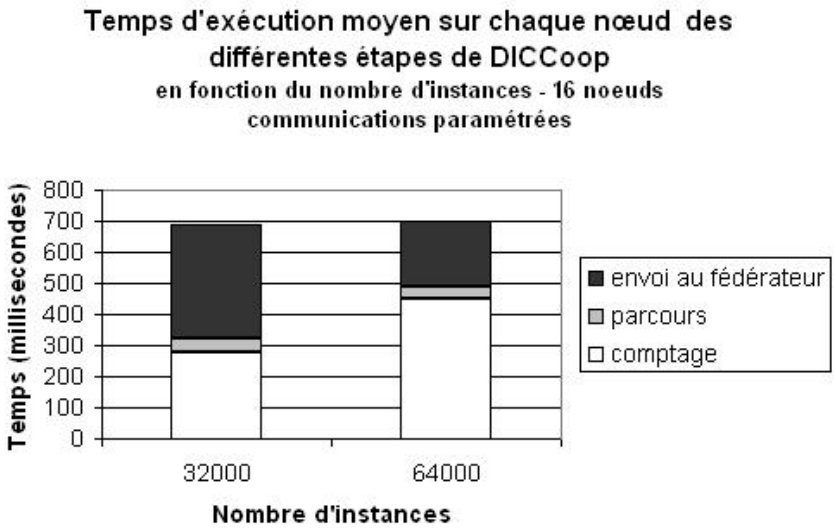


FIG. 4.20 – Comparaison des temps moyens d'exécution sur chaque site en fonction du nombre d'instances pour l'algorithme DICCoop (30 attributs - 16 noeuds - communications paramétrées)

Les observations effectuées sur les temps d'exécution des noeuds sont confirmés par les temps d'exécution sur le fédérateur (voir Figure 4.21).
Ainsi, les envois d'informations locales par les noeuds vers le fédérateur diminuent légèrement lorsque le nombre d'instances augmente. Le fédérateur consacre ainsi 6900 ms à la réception d'informations pour 32000 instances, contre 6600 ms pour 64000 instances.
La quantité de travail sur le fédérateur est faiblement allégée.
Le temps de traitement des informations locales par le fédérateur est toujours négligeable par rapport au temps de communications. Les quantités d'informations envoyées et reçues par le fédérateur sont données en annexe C. Ces quantités diminuent avec l'augmentation du nombre d'instances.

La diminution des temps de communications obtenue lors de l'augmentation du nombre d'instances est relativement faible. La quantité d'informations à échanger reste stable lors de l'augmentation du nombre d'instances, puisque ces informations sont liées au taux d'intelligence dans les fragments (items communs aux instances d'un fragment ou absents dans un fragment). On se situe ici dans des expérimentations réalisées sur 16 noeuds pour 16 fragments intelligents et donc le taux d'intelligence exploitable sur chaque site est stable.

Comparaison des temps d'exécution pour la méta Version de DICCoop

La Figure 4.22 présente une comparaison entre le traitement d'un même fragment d'instances par une version clasique de l'algorithme DICCoop et par une méta version de cet algorithme. Pour rappel, dans la méta version, un fragment d'instances (potentiellement

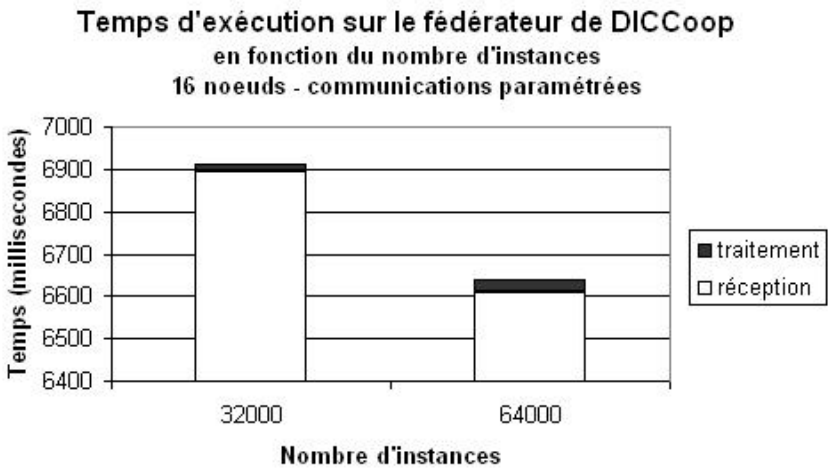


FIG. 4.21 – Comparaison des temps d'exécution sur le fédérateur en fonction du nombre d'instances pour l'algorithme DICCoop (30 attributs - 16 noeuds - communications paramétrées)

de trop grande taille pour être traité sur un seul site T_i) est redistribué sur une sous-grappe. La Figure 4.22 présente les temps d'exécution de :

- la version classique par communication paramétrées :
  sur un site T_0 et sur le fédérateur
- la méta version sur une sous-grappe composée des sites T_{00} et T_{01} et du fédérateur local de groupe F_0, le fédérateur global de cette méta version est également représenté

Concernant les temps de traitement sur les sites T_{00} et T_{01}, la somme des temps de traitements est légèrement inférieure à l'exécution réalisée sur le site T_0 dans la version classique de l'algorithme.
En particulier, on peut remarquer que ce n'est pas le temps d'accès à la structure de données qui diminue dans la distribution du traitement. Sur le site T_{00}, par exemple, le temps de génération des sur-ensembles et de conclusions intermédiaires est équivalent à celui nécessaire sur le site T_0 dans la version classique (soit un peu plus de 1s).
Par contre, le temps de comptage des supports (et donc le temps de traitement nécessitant l'accès aux instances de données) est inférieur dans la méta version. Ainsi, le temps de comptage est lègèrement supérieur à 7s sur le site T_0 dans la version classique, il n'est que de 4s environ sur le site T_{00} de la méta version, et de moins de 2s sur le site T_{01}. Soit une somme des temps de traitements nécessitant l'accès aux données de moins de 6s en méta version contre un peu plus de 7s en version classique.

De même, que pour les temps d'accès aux structures de données (étape de génération des sur-ensembles et de conclusions intermédiaires), les temps de communications engendrées par les sites T_{00} et T_{01} dans la méta version sont inférieurs ou égals à ceux engendré par le site T_0 dans la version classique.

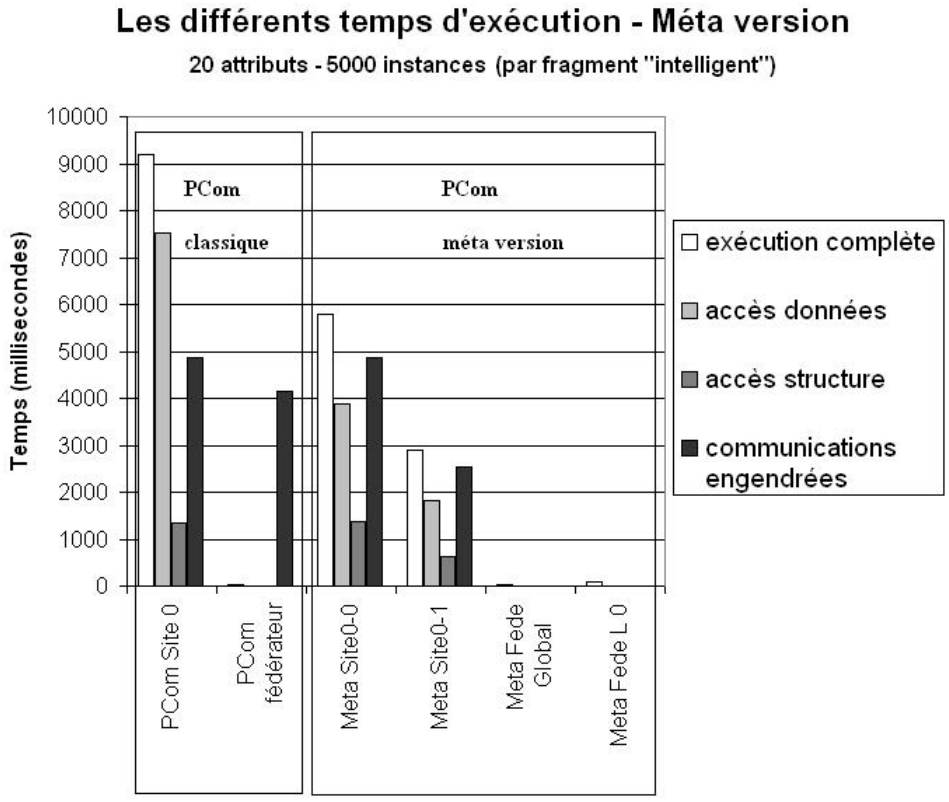


FIG. 4.22 – Comparaison des temps d'exécution de l'algorithme DICCoop dans la version classique et la méta version (20 attributs discrets - 5000 instances par fragment "intelligent" - PCom classique sur 2 noeuds, PCom en méta version sur 2 sous-grappes de 2 noeuds chacunes)

Le travail des fédérateurs F_0 et FG est quant à lui quasiment nul, de même que les communications qu'ils engendrent.
La diminution plus que significative des communications engendrées par le fédérateur de groupe F_0 vers les sites T_{00} et T_{01} est dûe au fait que les sites T_{0j} travaillant sur les mêmes attributs (du fait de la distribution aléatoire, donc a priori uniforme, des instances similaires issues d'une même fragment "intelligent" redistribué sur la sous-grappe), ils informent le fédérateur de groupe de manière équivalente. Celui-ci n'a dès lors pas besoin de questionner les sites, et les communications qu'il engendre sont limitées à les informer de décisions de fréquence et d'infréquence globales des itemsets. Il n'y a pas ici d'enrichissement des traitements locaux.
La quasi absence de communications engendrées par le fédérateur global FG provient du fait que celui-ci n'est informé que par les fédérateurs de groupe F_i. Ceux-ci effectuent en quelques sortes un tri des informations intéressantes, limitant le rôle du fédérateur global. De plus, celui-ci étant averti plus tard des informations relatives aux différents sites (retard dû à la présence du fédérateur de groupe comme intermédiaire), il est amené à moins questionner les sites et à moins enrichir leurs traitements locaux. Son rôle est moins important sur le déroulement des traitements en méta version.

4.4.7 Quantité de travail à effectuer et progression des résultats

La Figure 4.23 présente l'évolution de la quantité de travail à effectuer sur un site de traitement T_i. La quantité de travail est liée au nombre d'itemsets dont le comptage de support n'est pas terminé (itemsets marqués "dashed" voir Section 4.2.3). A chaque itération, un certain nombre d'itemsets sont générés et marqués "dashed" (nouveau généré sur la Figure 4.23), ils viennent alors se rajouter aux itemsets précédement marqués dashed et donc ajouter du traitement pour les itérations suivantes. Certains itemsets marqués "dashed", pour lesquels on a fini le comptage, sont soustraits de l'ensemble et donc viennent diminuer la quantité de traitement des prochaines itérations.

On constate sur la Figure 4.23 que la quantité de travail augmente de manière exponentielle dès les premières itérations, puis décroit de manière quasi linéaire à partir de la 15ème itération.
Le nombre d'itemsets nouvellement générés (et qui viennent donc s'ajouter à la quantité de travail à effectuer) est relativement important lors des premières itérations (d'où la croissance exponentielle du nombre d'itemsets marqués "dashed" lors de ces itérations). Puis le nombre de générations d'itemsets diminue, avec simplement des pics de génération anecdotique à partir de la 40ème itération, permettant une diminution de la quantité de travail à réaliser et donc un arrêt de la méthode.
La génération d'itemsets par pics est liée au parcours des données par tranche pour le comptage des supports.
La Figure 4.24 présente l'évolution des informations sur un site de traitement T_i.
Lors des 20 premières itérations, le nombre d'itemsets en cours de comptage (marqués dashed, box ou circle) est important.
A noter que la majorité des itemsets en cours de comptage sont infréquents supposés (dashed circle), puisque c'est le marqueur qui leur est attribué lors de leur génération.

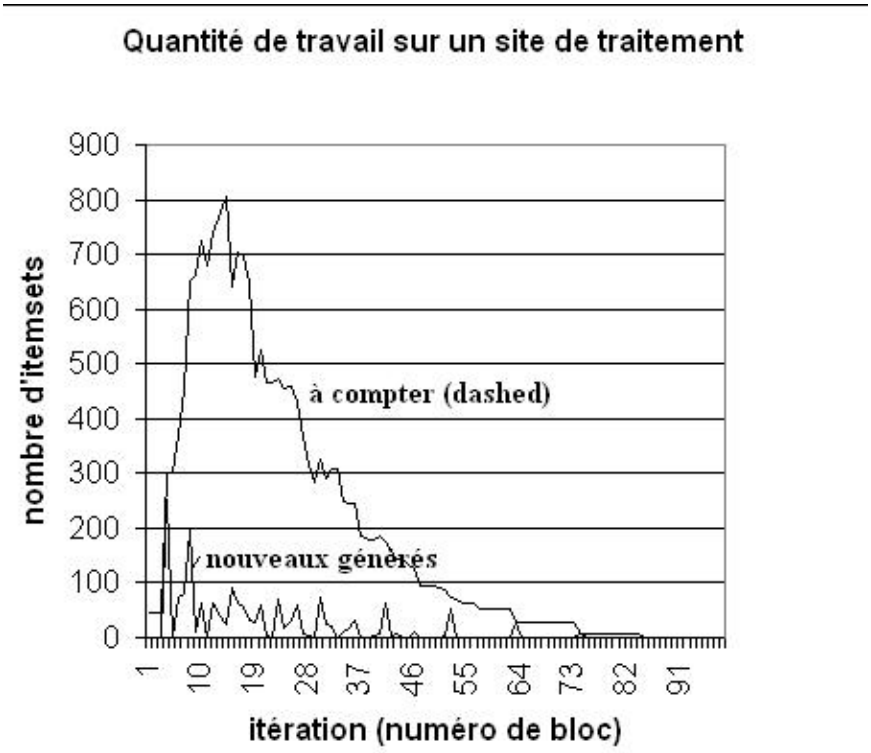


FIG. 4.23 – Evolution de la quantité de travail sur les sites de traitements T_i

Le taux d'itemsets dont le comptage de support est terminé, et donc marqués "solid" (circle = infréquents ou box=fréquents), augmente au fur et à mesure des itérations, tandis que celui des itemsets marqués "dashed" décroit. Le nombre d'informations, et donc de résultats connus, devient plus important que le nombre d'itemsets dont il faut poursuivre le traitement dès la 20 ème itération. Ceci correspond à la progression de connaissances des résultats dans l'algorithme DICCoop.

4.5 Conclusions concernant l'algorithme DICCoop

L'algorithme DICCoop est une adaptation de l'algorithme DIC compatible avec une exécution sur infrastructure parallèle de type grappe de stations ou grille de calcul. Les contraintes liées à ce type d'infrastructure sont respectées en particulier concernant les délais de communications.
L'exploitation d'une distribution intelligente amène à obtenir les itemsets fréquents sur l'ensemble des instances à partir de résultats locaux sur des fragments "intelligents", et ceci à moindre coût.

Les performances de l'algorithme DIC séquentiel sont limitées au regard de la nécessité de ressources de stockage et de traitement. L'algorithme DICCoop repousse ces limites permettant d'exploiter les ressources de plusieurs noeuds en terme de stockage et de calcul.

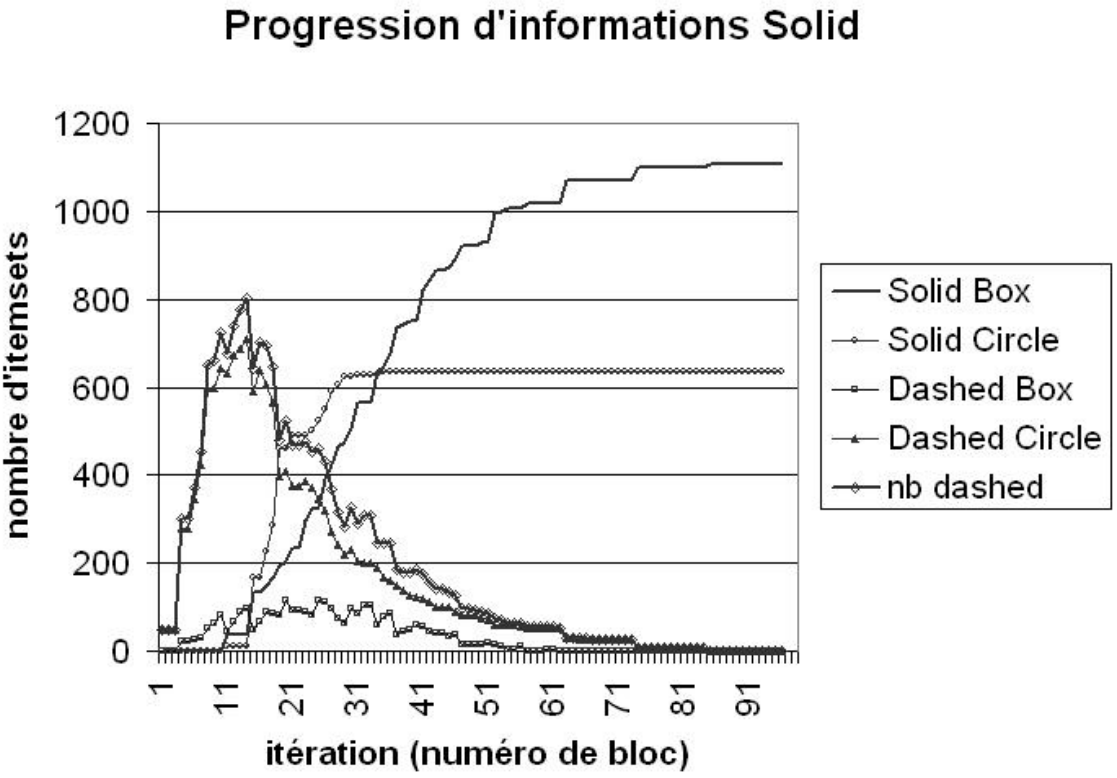


FIG. 4.24 – Progression de l'information sur les sites de traitements T_i

La proposition de la méta version propose de repousser encore plus loin ces limites en permettant une "redistribution" sur sous-grappes de fragments d'instances de trop grande taille.

De nombreuses améliorations sont encore possibles sur la méthode, en particulier pour améliorer les communications, et recouvrir les temps de ces communications par des traitements autant que possible.

L'algorithme DICCoop s'inscrivant comme étape de traitement postérieure à une fragmentation intelligente par clustering (voir Section 2.1), des améliorations par introduction de pipeline entre la phase de fragmentation et la phase de traitement effectif sont possibles et seront discutées dans le chapitre suivant (voir Section 5.1.1).

Quatrième partie

Bilan, Conclusions et Perspectives

Bilan, Conclusions et Perspectives

Ce chapitre présente un certain nombre d'aspects liés à un déploiement efficace des algorithmes précédemment présentés (clustering CDP voir Section 3.2 et identification des itemsets fréquents DICCoop voir Section 4.2) et au schéma général proposé dans DisDaMin pour la génération de règles d'association distribuée sur une infrastructure de type grille de calculs (voir Section 2.1).

5.1 Bilan de l'exécution distribuée

5.1.1 Bilan des aspects communications et asynchronisme dans le schéma général

Du fait des critères énoncés pour une exécution optimale sur un support de type grille de calcul (voir Section 1.3.3), les méthodes de recouvrement de communications et d'asynchronisme doivent être utilisées aussi souvent que possible dans le schéma général.
Ainsi ces optimisations sont utilisées à différents niveaux :

- Dans la phase de préparation des données, entre la phase de discrétisation et la phase de fragmentation (principe progressif de l'algorithme CDP) ;
- Dans la phase de traitement pour la génération de règles d'association à proprement parlé (principe collaboratif de l'algorithme DICCoop).

En particulier, l'utilisation des méthodes de **pipeline** dans la **phase de préparation** a amené à proposer la méthode de clustering CDP.
L'algorithme CDP a été conçu pour réutiliser les résultats de clustering unidimensionnel disponibles dans le traitement complet de la base B (discrétisation des attributs). Ainsi, une partie des calculs pour la phase de fragmentation est déjà réalisé dans un contexte d'exécution cohérent avec la suite du traitement.
La méthode de clustering **CDP** présentée (voir Partie II) est utilisée dans le projet DisDaMin en tant que phase de fragmentation des données, d'autant que ce sont les particularités du schéma des règles d'association qui ont inspiré les étapes de cette méthode de clustering.
La phase de discrétisation des données (phase préliminaire dans CDP) s'effectue sur les attributs suivants pendant que les résultats du pré-traitement sur les attributs précédents

sont déjà exploités dans CDP pour construire la fragmentation.
A noter, qu'après la phase de discrétisation, les données ne sont pas communiquées, les résultats sont utilisés dans le clustering progressif de distribution, sous format «binaire » propre à l'implémentation.

On exploite ici les méthodes de pipeline, les résultats de discrétisation étant utilisés dès que disponibles. Le traitement de fragmentation s'effectuant de manière progressive (par la prise en compte des résultats de discrétisation), on évite des barrières de synchronisation entre cette discrétisation et la fragmentation à proprement parler(voir Section 3.5.1).

Tout comme pour le clustering distribué progressif, l'algorithme DICCoop (voir Section 4.2) utilisé dans l'étape de traitement respecte les critères d'exécution de l'architecture visée pour le déploiement (grille de calcul). Le modèle collaboratif inspiré de l'algorithme DIC , permet de respecter les caractéristiques de l'asynchronisme, en évitant les barrières de synchronisation existantes dans les méthodes existantes de recherche des itemsets fréquents.

5.1.2 Améliorations d'exécution - Ordonnancement de tâches

Afin d'optimiser l'exécution des méthodes présentées sur une infrastructure de type grille de calcul, et afin de permettre un contrôle de l'application (monitoring et des déplacements éventuels des traitements liés aux disponibilités des noeuds, à des défaillances éventuelles...), le schéma général doit être déployé en utilisant une plateforme adaptée.
L'exploitation de l'asynchronisme tout particulièrement peut être optimisé grâce à l'utilisation des travaux réalisés dans le cadre du projet ADAJ (Adaptative Distributed Application in Java, voir [24], [25] et [13]). ADAJ vise à transformer une grappe de stations en une machine parallèle pour applications hétérogènes.
Un environnement de développement est fourni (permettant l'expression de la distribution et du parallélisme), ainsi qu'un environnement pour une exécution transparente et efficace. Le projet offre des outils de monitoring et d'équilibrage de charge, et permet la migration (ou redistribution) d'objets java.
Plus particulièrement, dans le cadre d'une architecture de type grille de calculs, les travaux réalisés dans le cadre du projet DG-ADAJ (Desktop-Grid Adaptative Application in Java, voir [49] et [48]), visant à fournir un environnement de contrôle adaptatif des applications distribuées sur grille, doivent permettre d'optimiser le déploiement du schéma général de recherche de règles d'association, en tenant compte des puissances et charge réelles des stations.

Le recouvrement des communications doit quant à lui être maximisé et il est encore possible d'améliorer l'exploitation du pipeline dans le schéma général, en se basant sur les dépendances entre les différentes tâches de traitement.

Les différentes tâches :
La méthode générale présentée dans le projet DisDaMin propose une fragmentation de données suivie d'une fragmentation des traitements associés. Deux phases principales

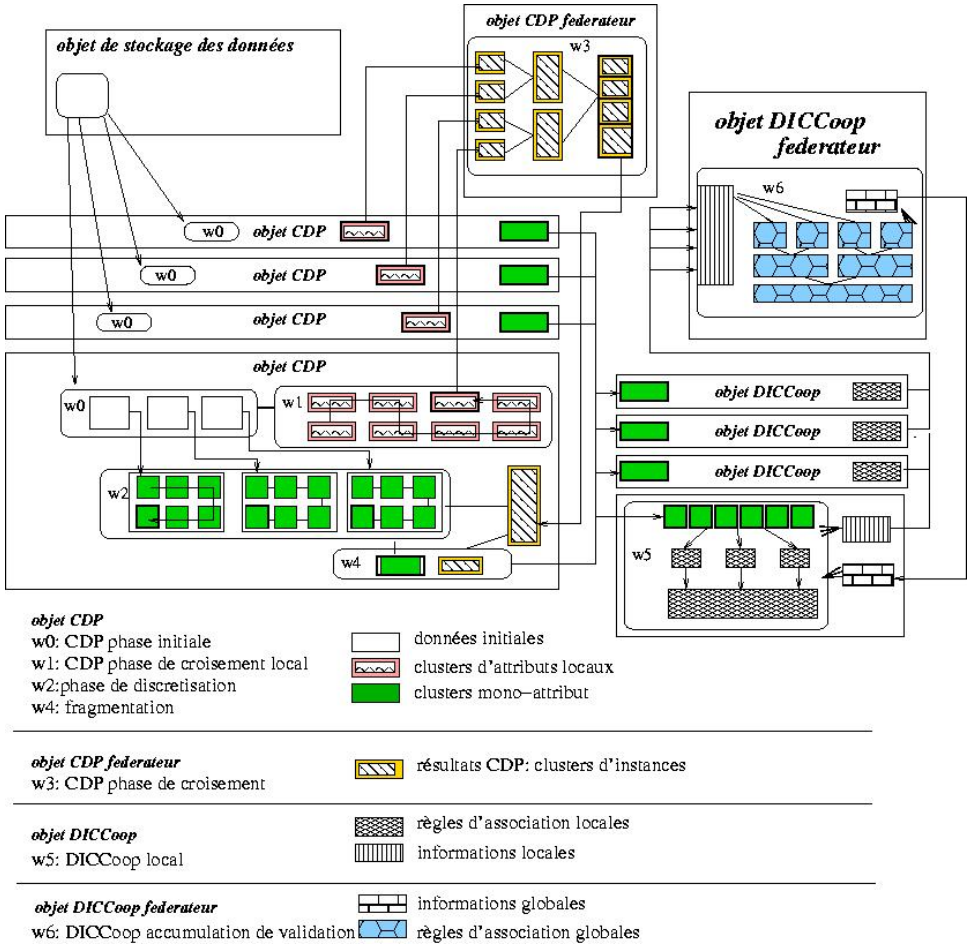


FIG. 5.1 – Schema général DisDaMin

peuvent être identifiées (fragmentation et génération de règles d'association). Chacune de ces phases peut être divisée en plusieurs tâches (voir la figure 5.1.2).

Nous définissons sept tâches : $W_0, \ldots W_6$.

Nous distinguons également différents types de noeuds de traitement :

- les noeuds de stockage avec accès direct à la base de données. Ces noeuds sont chargés de transmettre les données aux noeuds de traitement (voir ci-dessous)
- les noeuds de traitement qui effectuent les tâches des algorithmes CDP et DICCoop.
- le(s) noeud(s) fédérateur(s) qui permettent la collaboration entre les noeuds de traitement pour les algorithme CDP et DICCoop.

La première étape du schéma, la fragmentation intelligente par clustering (algorithme CDP voir Partie II), est composée des tâches W_0 à W_3.

La tâche W_0 consiste à déployer les données. Cette tâche consiste à transmettre les données aux noeuds de traitement. Cette tâche se déroule donc entre les noeuds de stockage et les noeuds de traitement. Chaque noeud de stockage possède un fragment de la base de données et a connaissance des noeuds de traitement voisins auxquels il envoie des fragments verticaux de données. Si un unique noeud de stockage est utilisé, alors la fragmen-

tation initiale est simple. Si plusieurs noeuds de stockage sont utilisés pour le déploiement des données, il est nécessaire d'avoir une collaboration entre ces noeuds (pour la fragmentation initiale des données) afin d'éviter la saturation d'un noeud et la sous-exploitation d'autres.
Cette tâche W_0 peut être divisée sur chaque noeud de traitement en $W_{0,0} \ldots W_{0,n}$.

La tâche W_1 consiste à analyser indépendamment chaque attribut pour collecter des informations pour chacun d'entre eux. Ces informations sont utilisées pour transformer les données dans un format adapté à la recherche de règles d'association (données discrétisées composées d'items). Cette tâche correspond à une discrétisation des données.

La tâche W_2 permet de fragmenter les instances de données (indépendamment sur chaque attribut continu) par une technique de clustering.
Les données initiales (continues) sont utilisées, et donc cette tâche ne peut se terminer que lorsque la tâche W_0 est terminée (pour le noeud de traitement considéré).
La tâche W_2 produit des clusters pour chaque attribut de la base (chaque dimension).

La tâche W_3 correspond à l'étape de croisement del'algorithme CDP.
En utilisant les clusters identifiés dans la tâche W_2, des clusters sur plus d'une dimension sont générés.
Le début d'exécution de la tâche W_3 nécessite qu'au moins deux noeuds de traitement aient terminé la tâche W_2. Le noeud fédérateur est utilisé pour choisir lequel des deux noeuds de traitement (dont on croise les résultats) va effectuer le croisement.

La seconde étape du schéma, la génération de règles d'association (génération des itemsets fréquents plus particulièrement) est composée des tâches W_4 à W_6.

La tâche W_4 consiste à effectuer le placement des fragments "intelligents" de données sur les noeuds de traitement. Ces fragments sont identifiés par la fragmentation intelligente qui se termine à la tâche W_3, et sont composés des résultats partiels de la tâche W_1. Donc la tâche W_3 doit être entièrement achevée por que la tâche W_4 puisse démarrer , et la tâche W_1 doit être entièrement achevée pour que la tâche W_4 puisse se terminer.

La tâche W_5 consiste à effectuer une recherche locale des itemsets fréquents (étape essentielle du problème de génération des règles d'association) à partir des données discrètes obtenues lors de la tâche W_1. Pour identifier des k-itemsets il est nécessaire qu'au moins k tâches W_1 indépendantes soient achevées.

La tâche W_6 consiste à rechercher des k-itemsets globalement fréquents.
Le noeud fédérateur reçoit spontanément des informations locales des noeuds de traitement, et effectue des combinaisons pour obtenir une décision globale de fréquence des itemsets.
La tâche W_4 doit être achevée, et les informations partielles issues de W_5 doivent être disponibles pour démarrer W_6. La tâche W_5 doit être terminée pour que la tâche W_6 puisse s'arrêter.

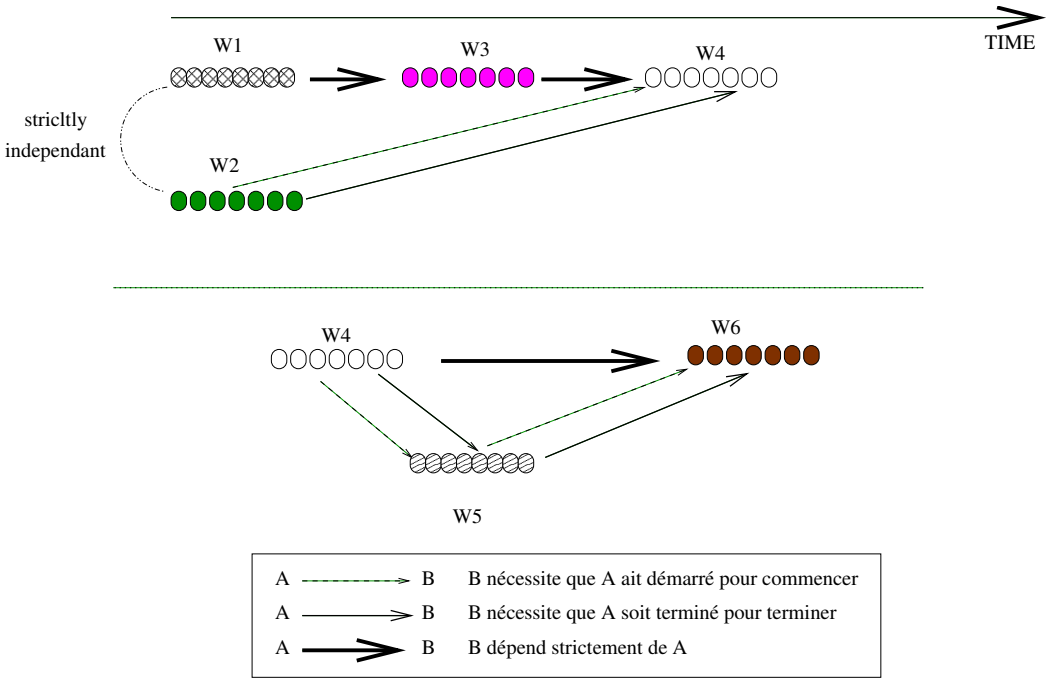


FIG. 5.2 – Dépendances entre tâches

Différents aspects nécessitent une attention particulière pour garantir une exécution optimale et pour préserver une bonne accélaration (speed-up) : la distribution des données, l'ordonnancement de tâches, l'équilibrage de charge et l'utilisation des périodes d'inactivité.

Ordonnancement de tâches :
Une attention particulière est donnée à l'ordonnancement des tâches. Par exemple, la tâche W_1 peut être réalisée à différents instants : au démarrage, dès que W_0 (distribution initiale des données) est commencée, ou lorsque W_0 est terminée, après W_2 ou après W_3. Les dépendances entre tâches, décrites ci-dessus sont synthétisées par la Figure 5.1.2.

Le choix du moment d'exécution pour les différentes tâches a non seulement un impact sur la durée des traitements (via les possibilités de recouvrement de communications, les disponibilités des processeurs...), mais également sur le volume de données devant être stocké. Pour la tâche W_1, il est nécessaire de travailler sur les données initiales, c'est-à-dire non discrétisées, pour la recherche de règles d'association, DisDaMin a uniquement besoin des données discrétisées. Travailler avec des données discrétisées (donc qui peuvent être stockées sous forme binaire), au lieu de travailler avec les données continues permet une réduction de la capacité de la mémoire nécessaire pour stocker ces données. Par exemple, si les données continues initiales sont de type flottant (4 octets) pour chaque attribut continu, et que l'étape de discrétisation (tâche W_1) identifie 4 items pour chaque attribut (un item pouvant être stocké sur 8 bits, c'est-à-dire 1 octet), et en considérant l'exclusion entre les items issus d'un même attribut continu, le gain est environ de 4. En traitant 100 millions d'instances, on réduit la capacité mémoire nécessaire de 3 Go à moins de 800 Mo, ce qui est loin d'être négligeable .

Equilibrage de charge :
Un autre aspect à prendre en compte est l'optimisation d'une distribution équilibrée. Une optimisation de l'équilibrage est simple à effectuer lors la tâche W_0 (distribution initiale des données).
Les choix à effectuer pour cet équilibrage sont liés à :

- des considérations physiques (espace de stockage limité) sur les noeuds. Il est nécessaire de distribuer les attributs continus (dimensions) de manière à minimiser les accès disque.
- la complexité estimée du temps de traitement des attributs initiaux sur les noeuds de traitement. Il semble possible d'établir une échelle de complexité de traitement de discrétisation de chaque attribut, en fonction d'informations sur les données. Par exemple, on peut considérer que des données binaires sont moins complexes à traiter (pour la discrétisation) que des données énumérées. De même, des données énumérées sont moins complexes à traiter que des données continues.

Une procédure d'équilibrage de charge doit donc tenir compte de ces deux critères ainsi que des informations locales sur les noeuds de traitement pour effectuer un choix optimal de distribution.

Un autre problème d'équilibrage de charge apparaît durant l'étape de croisement de l'algorithme de CDP (tâche W_3). À chaque itération de croisement, un noeud de traitement envoie ses informations à un autre. Le noeud récepteur doit alors mener à bien les travaux de croisement et devra participer à la prochaine itération de croisement. Le choix entre noeud émetteur et le noeud de croisement est possible. Il pourrait être ainsi intéressant de libérer de cette tâche des noeuds très chargés et pour la confier à un noeud sous-utilisé (on considère ici la charge totale des noeuds, pas seulement leur charge liée à l'application, contexte multi-utilisateurs sur architecture non dédiée).

Utilisation des périodes d'inactivité :
Une fois les tâches W_3 et W_1 effectuées, si le traitement global n'est pas encore terminé, un noeud de traitement peut se retrouver sans travail. Il est alors nécessaire de lui en trouver. Ce travail peut être de trois types :

- contribuer à accélérer les autres traitements, par exemple pour la tâche W_1, en délestant un noeud trop chargé de certains attributs continus.
- permettre d'obtenir des informations de meilleure qualité, en effectuant, par exemple de nouvelles étapes de clustering avec d'autres paramètres (modification du paramètre k dans le clustering par exemple) pour utiliser le meilleur résultat obtenu.
- anticiper les traitements à venir, c'est-à-dire la recherche de règles d'association. Ceci peut être réalisé par des échanges d'informations avec d'autres noeuds sous chargés.

5.1.3 Déploiement de l'algorithme CDP sur GRID'5000

Des expérimentations ont été réalisées sur l'infrastructure Grid5000 (voir Annexe D) pour l'algorithme CDP (étape de fragmentation intelligente) sur la base de données médicale *DiabCare* comportant environ 180 attributs et 30000 instances.

Les expérimentations amènent à un taux d'accélération (speedup) acceptable pour la méthode (voir la Figure 5.3), qui est proche d'une accélération linéaire en utilisant un nombre raisonnable de noeuds de la grille.
Au début de l'algorithme, les attributs de la base de données sont distribués puis indépendamment traités sur les noeuds. Donc l'augmentation du nombre de noeuds utilisés, permet d'améliorer le speedup (dans la limite du nombre d'attributs continus considérés). L'étape suivante l'algorithme consiste en une collaboration entre les noeuds pour croiser les résultats, Ainsi, si trop de noeuds sont employés par rapport au nombre d'attributs considérés, de nombreux composants deviennent rapidement inactifs pour la tâche de croisement (ceux qui ne participent pas à l'exécution du croisement) ce qui explique une limite dans le speedup obtenu. En outre, plus le nombre d'instances à traiter est grand, meilleur est le speedup, ce qui pourrait être expliqué par la plus grande part (dans le temps total d'exécution de la méthode) des temps de communications (coût fixe d'établissement des communications) et des périodes d'inactivité (attente de disponibilité pour le croisement) pour un petit nombre d'instances.

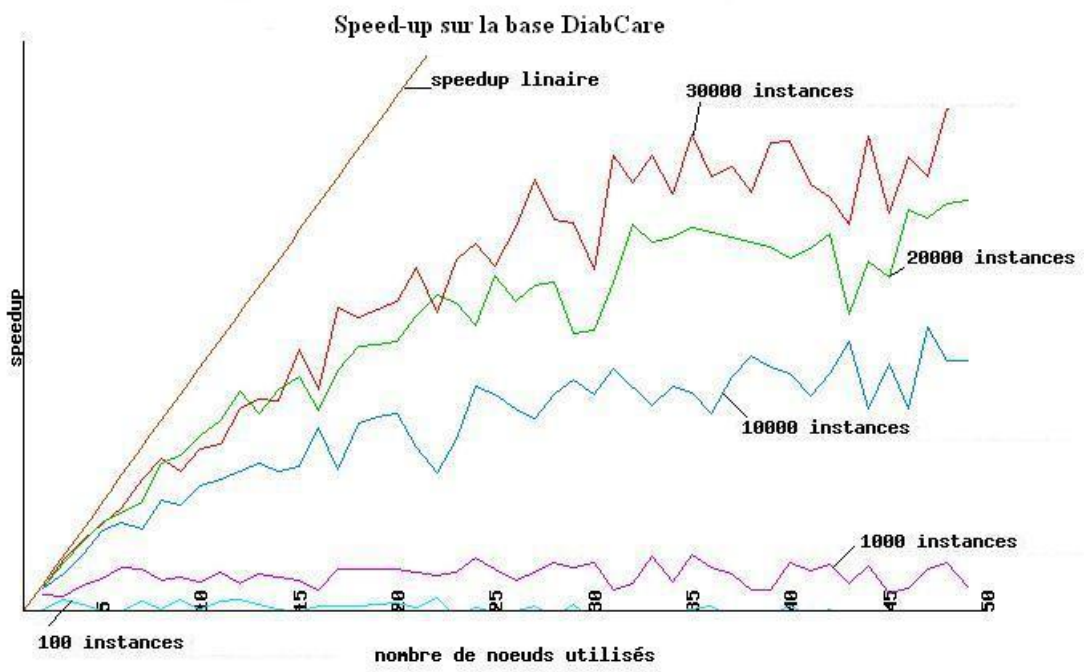


FIG. 5.3 – Speed up sur la base de données Diabcare.

Plus de détails ainsi que des résultats d'expérimentations complémentaires sont donnés en annexe B.

5.2 Informations extraites

Pour rappel, le principe de la distribution intelligente consiste à regrouper dans un même fragment les instances similaires (en terme d'items contenus), puis à effectuer des traitements parallèles sur les fragments obtenus.
On possède donc, en plus de l'information globale (itemsets globaux, fréquents sur toutes les instances de la base), des informations concernant l'ensemble des itemsets fréquents localement sur chaque groupes d'instances. On peut donc générer les règles d'association locales à chaque fragment d'instances.

Les fragments d'instances résultant d'une distribution par similarité (clustering), on a des groupes correspondant potentiellement à des profils d'instances (profils de clients d'un supermarché, profils de pathologie sur données médicales ...).
Les profils indentifiés constituent donc en eux-mêmes un savoir important : des sous-populations à caractéristiques communes.

De plus, la nécessaire discrétisation des données qui doit être réalisée préalablement à la recherche des règles d'association, répond à un manque d'informations sur les données. Il est possible, pour certains domaines que la classification d'un attribut soit connue. Dans le cas particulier de la base DiabCare, un découpage cohérent des valeurs liées à un attribut continu de départ n'a pas pu nous être fourni (parce qu'un tel découpage n'était pas connu). Dès lors la discrétisation réalisée, et plus particulièrement les groupes de valeurs identifés pour chaque attribut (classe de valeurs), constitue un savoir important.
Ainsi, pour chaque attribut continu on identifie un ensemble de classes (items) associé. Chacun de ces items est lié à un ensemble de valeurs pour lequel on peut fournir les bornes d'un intervalle (valeur minimale de l'intervalle lié à l'item, valeur maximale). Les intervalles sont d'ailleurs utilisés pour la présentation finale des règles d'association générées, le codage en item (propre au traitement) n'étant pas adapté à une analyse des résultats par un spécialiste des données. On fournira ainsi, non pas des règles d'association de type $A => B$ mais attribut1$[x_1, x_2]$ => attribut2$[y_1, y_2]$.

Remarque : Les items issus d'un même attribut continu sont présents en exclusion mutuelle dans les instances, cette information d'exclusion pourrait être exploitée dans le traitement.

Intérêt de résultats locaux
La fragmentation correspondant à des groupes d'instances similaires, ceci constitue un découpage en sous-populations à caractéristiques communes (voir Section 2.1.2).
Il peut être intéressant de générer les règles d'association spécifiques à chacune de ses sous-populations, d'autant plus que les résultats locaux de fréquence et les supports sont disponibles, et peuvent permettre d'offrir des informations importantes sur les groupes d'instances.
Les infréquents globaux qui sont fréquents locaux ne doivent donc pas être éliminés des traitements locaux au sein desquels ils sont pertinents (fréquents), afin de bien avoir un résultat complet sur chaque fragment : pour chaque fragment, un ensemble de règles d'association pertinent.

Ainsi, on fera une différence entre :

- les règles d'association obtenues sur la base complète ;
- les règles d'association obtenues sur un sous-ensemble "intelligent" d'instances, un cluster.

Les itemsets fréquents locaux non pertinents globalement vont influer sur la poursuite du traitement. En effet, il conviendra de poursuivre le traitement tant que l'on n'a pas obtenu tous les itemsets fréquents globaux générables, mais également tant que l'on n'a pas obtenu tous les itemsets fréquents locaux générables.

De plus, lorsqu'un site a terminé la recherche locale des itemsets fréquents, il est possible qu'il reste des itemsets globaux à traiter. Dans ce cas, le site doit rester disponible pour répondre à d'éventuelles besoin d'informations du fédérateur.

5.3 Conclusions et Perspectives

Pour résoudre le problème de la recherche d'association en contexte distribué de type grille de calcul, nous avons introduit, dans le cadre du DisDaMin, un schéma général proposant une fragmentaton intelligente des instances basée sur le clustering. Partant de cette fragmentation, le traitement distribué des fragments permet d'obtenir une diminution significative de la taille de l'espace de recherche visité, et donc de la complexité de traitement pour la génération des itemsets fréquents (étape essentielle et coûteuse de la recherche de règles d'association), tout en permettant de générer, outre les règles d'association globales sur les données, des connaissances annexes qui s'avèrent intéressantes, à savoir, l'identification de profils d'instances et les règles d'association locales à chacun de ces profils.

Le schéma général se base sur l'utilisation d'une phase de clustering pour la fragmentation, nous avons donc présenté notre algorithme CDP (Clustering Distribué Progressif - voir Partie II). Celui-ci permet de construire une solution de clustering multidimensionnel (sur plusieurs attributs) à partir de résultats de clustering unidimensionnel (sur un seul attribut).
L'algorithme a été conçu pour répondre aux spécificités des supports d'exécution de type réseau de stations de travail ou grille de calculs. Il a été inspiré du contexte particulier à notre schéma général de traitement mais apparaît comme une heuristique de clustering à par entière.

Partant d'une fragmentation intelligente issue de l'algorithme CDP, notre adaptation de l'algorithme DIC a ensuite été présenté, DICCoop (DIC Coopérative - voir Partie III). Cette adaptation respecte elle aussi les contraintes de la plateforme d'exécution visée et permet de générer les règles d'association sur la base de données entières ainsi que les règles d'association propres à chacun des fragments de données identifiés.

Les expérimentations effectuées ont permis de valider la conjecture initiale qui consistait à affirmer qu'une distribution intelligente des instances permettait de diminuer le facteur exponentiel de la taille de l'espace de recherche pour la génération de règles d'association.

Les deux méthodes présentées (CDP et DICCoop) peuvent bénéficier de nombreuses améliorations liées à un déploiement optimisé, exploitant au maximum l'asynchronisme, le pipeline et le recouvrement de communications. Ces optimisations sont propres au déploiement, mais peuvent être guidées par les dépendances observées entre les différentes étapes du traitement. De même, l'utilisation de la version macro-itérative de l'algorithme CDP lié à un ordonnancement optimal de tâches doit permettre de diminuer les périodes d'inactivité dans l'exécution de la méthode.
La phase de génération des règles d'association, postérieure à la phase d'identification des itemsets fréquents effectuée par DICCoop, peut également bénéficier d'optimisations liées à l'exécution de la méthode sur grille.

De nombreuses pistes ont été abordées sans pour autant être exploitées dans le schéma général, telle que, par exemple, la possibilité d'écarter du traitement des items très fortement fréquents pour les réincorporer par la suite.
Le principe de fragmentation intelligente pour la résolution du problème de règles d'association ayant été validé, de même que les algorithmes proposés, la prochaine étape consiste à déployer le schéma complet sur l'architecture GRID'5000 en utilisant les mécanismes distribués des projets ADAJ et DG-ADAJ, permettant ainsi une exécution optimale tenant compte des charges des différents noeuds, avec possibilité de migration des tâches, de tolérances aux pannes. . .

Les fédérateurs utilisés dans les algorithmes CDP et DICCoop sont pour l'instant des composants centralisés. De futurs travaux, qui seront effectués en collaboration avec des partenaires étrangers, doivent amener à décentraliser ces fédérateurs et permettre une implémentation sur modèle agent.

Le schéma général proposé dans le projet DisDaMin offre de nombreuses perspectives de recherche. Il inclut une phase de préparation des données (discrétisation des attributs). La fragmentation intelligente par l'algorithme CDP a été construite en utilisant des informations issues de cette préparation.
On peut dès lors envisager d'adapter les phases de préparation et de fragmentation pour permettre le traitement de bases de données spécifiques (données textes, images, . . .) selon ce principe de réutilisation d'informations.

De plus, d'autres informations issues des phases de préparation et de fragmentation (les items fortement fréquents identifiés lors de la discrétisation par exemple) pourraient être exploitées pour améliorer la phase de traitement (recherche des règles d'association dans les travaux présentés). Le principe de fragmentation intelligente pourrait également être adapté à d'autres problèmes afin de diminuer leur complexité de traitement (spécialisation des traitements) et de générer des résultats propres à des profils d'instances.

Table des figures

Liste des tableaux

Bibliographie

[1] R. Agrawal, T. Imielinski, and A. Swami. *Database mining : A performance perspective*. In IEEE Transactions on Knowledge and Data Engineering : Special issue on learning and discovery in knowledge-based databases, 5(6), pages 914-925, December 1993.

[2] R. Agrawal, . Imielinski, and A. Swami. *Mining association rules between sets of items in large databases*. In Proceeding of the 1993 ACM SIGMOD International Conference on Management Of Data (SIGMOD'93 : Special Interest Group on Managment of Data), pages 207-216, May 1993.

[3] R. Agrawal and R. Srikant. *Fast algorithms for mining associations rules in large databases*. In Proc. of the 20th Int. Conf. on Very Large Data Bases (VLDB'94), pages 478-499, September 1994.

[4] R. Agrawal and J.C. Shafer. *Parallel Mining of Association Rules*. IEEE Trans. on Knowledge and Data Eng., 8(6), 962-969, dec 1996.

[5] R. Agrawal, J. Gehrke, D. Gunopulos and P. Raghavan. *Automatic subspace clustering of high dimensional data for data mining application.*. In Proceeding of the 1998 ACM SIGMOD International Conference on Management Of Data, pages 94-105, 1998.

[6] R. Agrawal, J. Gehrke, D. Gunopulos and P. Raghavan. *Automatic subspace clustering of high dimensional data.*. In Data Mining and Knowledge Discovery Journal, 11(1), 2005.

[7] R.J. Bayardo and R. Agrawal. *Mining the most Interesting Rules*. In Proc. of the 5th ACM SIGMOD Int. Conf. (SIGMOD'99) , pages 145-154, 1999.

[8] P. Berkhin. *Survey of clustering data techniques*. Technical Report, Accrue Software, San Jose CA, 2002.

[9] James C. Bezdek, Joseph C. *Dunn : Optimal Fuzzy Partitions : A Heuristic for Estimating the Parameters in a Mixture of Normal Distributions.* In IEEE Trans. Computers 24(8) : 835-838, 1975.

[10] Richard J. Hathaway, James C. Bezdek. *Local convergence of the fuzzy c-Means algorithms.* In Pattern Recognition 19(6) : 477-480, 1986.

[11] C. Bishop *Neural Networks for Pattern Recognition*, Oxford University Press, November 1995.

[12] L. Bottou and Y. Bengio. *Convergence properties of the K-Means algorithms*. In Advances in Neural Information Processing Systems, Volume 7, pages 585-592, 1995.

[13] A. Bouchi, R. Olejnik and B. Toursel. *Java tools for measurement of the machine loads*. In Advanced Environments Tool and Applications for Cluster Computing, LNCS of Springer Verlag 2326, pages 271-278, September 2001.

[14] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. *Dynamic itemset counting and implication rules for market basket data*. In Proc. of the 1997 ACM SIGMOD Int. Conf. (SIGMOD'97), pages 255-264, May 1997.

[15] D.W. Cheung, J. Han, V. T. Ng, A. Fu and Y Fu. *A fast distributed algorithm for mining association rules*. In Proc. of the 4th Int. Conf. on Parallel and Distributed Information Systems, pages 31-42, 1996.

[16] S Dasgupta, P. M. Long. *Performance guarantees for hierarchical clustering*. Journal of Computer and System Sciences archive : Special issue on COLT 2002 , Volume 70 , Issue 4 :555-569, June 2005.

[17] www.datamininggrid.org.

[18] www.discovery-on-the.net

[19] M. Ester, H.-P. Kriegel and X. Xu. *A database interface for clustering in large spatial databases.* In Proc. of the 1st Int'l Conference on Knowledge Discovery in Databases and Data Mining (KDD'96). Montreal, Canada, August 1995.

[20] M. Ester, H.-P. Kriegel, J. Sander and X. Xu. *A density- based algorithm for discovering clusters in large spatial databases with noise.* In Proc. of the 2nd Int'l Conference on Knowledge Discovery in Databases and Data Mining, Portland, Oregon, August 1996.

[21] M. Ester, H.-P. Kriegel, J. Sander, M. Wimmerand and X. Xu. *Incremental Clustering for Mining in a Data Warehousing Environment.* In Proc. of the 24th Int. Conf. Very Large Data Bases (VLDB'98), pages 323-333, 1998.

[22] F. Farnstrom, J. Lewis and C. Elkan. *Scalability for Clustering Algorithms Revisisted*. In Proc. of ACM SIGKDD (SIGKDD Explorations : Special Interest Group on Knowledge Discovery in Data), 2 :(2) :1-7, juillet 2000.

[23] D. Fasulo. *An analysis on recent work on Clustering Algorithms*, 1999.

[24] V. Felea, B. Toursel, *Adaptive Distributed Execution of Java Applications*. In Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'04), pages 16-31, 2004.

[25] V. Felea, R. Olejnik and B. Toursel. *ADAJ : a Java Distributed Environment for Easy Programming Design and Efficient Execution*. Shedae Informaticae, UJ Press, Krakow Vol 13, pages 9-36, 2004.

[26] E. Forgy. *Cluster analysis of multivariate data : Efficiency vs. interpretability of classifications*. In Biometrics, Vol. 21 Number 3, page 768, 1965.

[27] V. Fiolet and B. Toursel. *Distributed Data Mining*. In Proc. of the the first Int. Symposium on Parallel and Distributed Computing (ISPDC'02), pages 349-365, July 2002.

[28] V. Fiolet and B. Toursel. *Distributed Data Mining*. In Scalable Computing : Practice and Experiences (SCPE), Special Issue : Internet-based Computing, Vol. 6, Number 1, pages 99-109, March 2005.

[29] V. Fiolet and B. Toursel, *Intelligent Database distribution on a Grid using Clustering*, In Lectures Notes In Computer Sciences. Proc. of AWIC 2005, page 466 Springer-Verlag, June 2005.

[30] D. H. Fischer. *Knowledge Acquisition Via Incremental Conceptual Clustering*. In Machine Learning. 2 pages 139-172, 1987

[31] G. Forman and B. Zhang. *Distributed data clustering can be efficient and exact*. In SIGKDD Explorations, Volume 2, 2000.

[32] www.wab.info.uvt.ro/ petcu/GRAI.html

[33] www.gridminer.org

[34] S. Guha, R. Rastogi, and K. Shim. *CURE : An efficient clustering algorithm for large databases.* In Proceedings of ACM SIGMOD International Conference on Management of Data, pages 73-84, New York, 1998.

[35] G. K. Gupta, A. Strehl, and J. Ghosh. *Distance based clustering of association rules*. In Intelligent Engineering Systems Through Artificial Neural Networks (Proceedings of ANNIE 1999), Volume 9, pages. 759-764, ASME Press, November 1999.

[36] Z. Huang. *Extensions to the k-means algorithm for clustering large data sets with categorical values*. In Data Mining and Knowledge discovery, Volume 2, pages 283-304, 1998.

[37] J. Han and Y. Fu. *Discovery of multiple-level association rules from large databases*. In Proc. of the 21st VLDB Int. Conf. (VLDB'95), pages 420-431, September 1995.

[38] E-H. Han and G. Karypis. *Scalable Parallel Data Mining for Associations Rules*. In IEEE Trans. on Knowledge and Data Eng. , 2(3) pages 337-352, May-June 2000.

[39] E. Johnson and H. Kargupta. *Collective, hierarchical clustering from distributed, heterogenous data*. In Lecture Notes in Computer Science, 1759, pages 221-244, Springer-Verlag, 1999.

[40] A. Jain, M. Murty and P. Flynn. *Data Clustering : a review*. In ACM Computing surveys, Vol.31, No 3, pp264-323 (September 1999).

[41] G. Karypis, E.H. Han, and V. Kumar. *Chameleon : A hierarchical clustering algorithm using dynamic modeling*. In Computers, Volume 32, Number 8 pages 68-75, 1999.

[42] J. Kaufmann, P.J. Rousseeuw. *Finding groups in data : an introduction to cluster analysis*. Wiley, 1990.

[43] www.grid.deis.unical.it/kgrid

[44] B. Lent, A. Swami and J. Widom. *Clustering Association Rules*. In Proc. of the 13th Int. Conf. on Data Eng. (ICDE'97), pages 220-231 , April 1997

[45] J. McQueen. *Some methods for classification and analysis of multivariate observations*. In Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, AD 669871, Univ. of California Press, Berkeley, Vol. 1, pages 281-297, 1967.

[46] R. T. Ng and J. Han. *Efficient and effective clustering methods for spatial data mining*. In Proc. of the VLDB Conference, Santiago, Chile, September 1994.

[47] R. T. Ng, J. Han. *CLARANS : A Method for Clustering Objects for Spatial Data Mining* In IEEE Transactions on Knowledge and Data Engineering archive. Volume 14 , Issue 5, pages 1003-1016, September 2002.

[48] R. Olejnik, A. Bouchi, B. Toursel. *Object observation for a java adaptative distributed application platform*. In Conference on Parallel Computing in Electrical Engineering (PARELEC 2002) pages 171-176., Warsaw Poland, September 2002.

[49] R. Olejnik, B. Toursel, M. Tudruj, E. Laskowski and I. Alshabani. *Application of DG-ADAJ environment in Desktop Grid*. A paraitre dans Future Generation Computer Systems, Springer Verlag.

[50] C. Lucchese, S. Orlando, R. Perego. *Fast and Memory Efficient Mining of Frequent Closed Itemsets* In IEEE Transactions on Knowledge and Data Engineering, vol. 18, no. 1, pages 21-36, January 2006.

[51] C. Silvestri and S. Orlando. *Distributed approximate mining of frequent patterns* In Proc. of SAC '05 (the 2005 ACM symposium on Applied computing), pages 529-536, 2005.

[52] J. S. Park, M.-S. Chen, and P. S. Yu. *An effective hash based algorithm for mining association rules*. In Proc. of the 1995 ACM SIGMOD Int. Conf. (SIGMOD'95), pages 175-186, May 1995.

[53] J. S. Park, M.-S. Chen, and P. S. Yu. *Efficient parallel data mining for association rules.* In Proc. of the 4th Int. Conf. on Information and Knowledge Management, pages 31-36, 1995.

[54] B. D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge, 1996.

[55] W. N. Venables, and B. D. Ripley *Modern Applied Statistics with S*, Springer, 2002.

[56] N. F. Samatova, G. Ostrouchov, A. Geist and A. Melenchko, *Rachet : An efficient covering based merging of clustering hierarchies from distributed datasets*. In International Journal of Distributed and Parallel Databases, 11(2), pages 157-180, 2002.

[57] T. Shintani and M. Kitsuregawa. *Hash-Based Parallel Algorithms fir Mining Association Rules*. In Proc. of the Int. Conf. on Parallel and Distributed Information Systems, 1996.

[58] R. Sokal and P. Sneath. *Numerical Taxonomy*. San Francisco : Freeman, 359. pages, (1963).

[59] A. Savasere, E. Omiecinski, and S. Navathe. *An efficient algorithm for mining association rules in large databses*. In Proc. of the 21st VLDB Int. Conf. (VLDB'95), pages 432-444, September 1995.

[60] R. Srikant. *Fast algorithms for mining association rules and sequential patterns*. PhD thesis, University of Wisconsin, 1996.

[61] H. Toivonen, M. Klemettinen, P.Ronkainen, K. Hatonen and H. Mannila. *Pruning and Grouping Discovered Association Rules*. In ECML-95 Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases. pages 47-52, Heraklion, Greece, (April 95).

[62] H. Toivonen. *Sampling large databases for association rules*. In Proc. of the 22nd VLDB Int. Conf. (VLDB'96), pages 134-145. September 1996.

[63] B. Toursel. *About design and Efficiency of Distributed Programming : Some Algorithmic Aspects*. In Proc. of International Worshop on Cluster Computing 2001. (IWCC'01), pages 36-46. September 2001.

[64] Zaki. *Association Mining.* 1997.

[65] M.J. Zaki. *Parallel and Distributed Association Mining : A survey.*. In IEEE Concurrency, special issue on Parallel Mechanisms for Data Mining, Volume 7, N˚4, pages 14-25 , December 1999.

[66] T. Zhang, R. Ramakrishnan, and M. Livny. *BIRCH : An efficient data clustering method for very large databases*. In Proc. of the ACM SIGMOD Conference on Management of Data, pages 103-114, Montreal, Canada, June 1996.

A Résultats d'expérimentations préalables

Nous présentons ici les résultats des premières expérimentations réalisées sur le schéma général proposé dans le projet DisDaMin pour la recherche de règles d'association (par fragmentation intelligente) .

A.1 Différentes propositions de fragmentation

A.1.1 Introduction

Une première idée de fragmentation par clustering consiste à utiliser les résultats de la phase de pré-traitement pour effectuer la fragmentation des instances. Un attribut continu de la base initiale (base brute) fournit X attributs discrétisés (items) dans la base de données à distribuer pour le traitement (de recherche d'itemsets fréquents et de règles d'association).
Un seul de ces attributs discrétisés peut être présent pour chaque instance (voir exclusion mutuelle des colonnes Figure A.1).

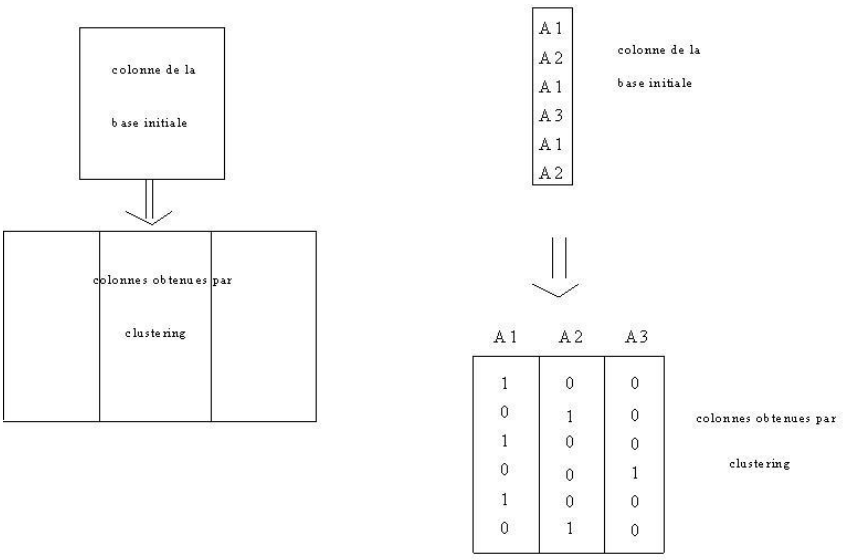


FIG. A.1 – Exclusion mutuelle des colonnes.

On peut dès lors imaginer distribuer les instances en utilisant la décomposition des attributs issue de la phase de discrétisation et répeter l'opération aussi souvent que nécessaire (en utilisant des attributs distincts) jusqu'à obtenir des fragments de taille raisonnable (suffisamment petits pour être traités localement) et une répartition acceptable (voir Figure A.2).

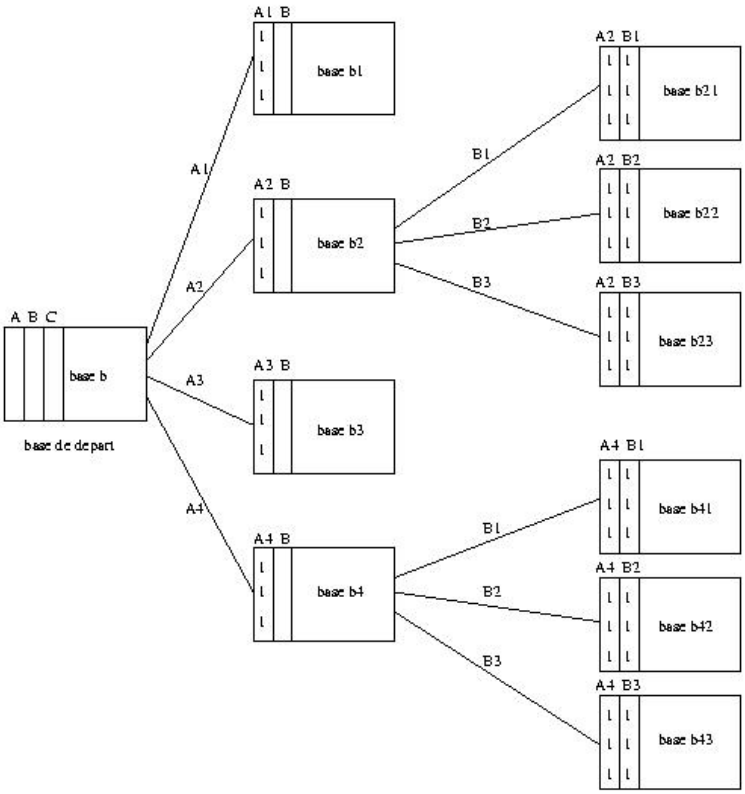


FIG. A.2 – Distribution selon certains attributs.

Ce procédé ne permet pas de diminuer la complexité de traitement (via la taille de l'espace de recherche visité). Seuls les attributs ayant joués un rôle dans la fragmentation deviennent triviaux (toutes les instances du fragment le contiennent, voir Table A.1).

Les autres attributs (items) doivent être considérés pour le traitement de manière classique, amenant au problème de complexité de traitement connu pour le problème. De plus, le choix des attributs servant à la fragmentation est arbitraire.

Une deuxième idée consiste à "plier" les instances (en utilisant des fonctions logiques : AND, XOR, OR...) sur un modèle table de hachage (voir effet de l'utilisation des fonctions logiques Tables A.2 et A.3), et de fragmenter les instances en fonction des valeurs de repliement obtenues (voir Figures A.3 A.4).

La principale difficulté de cette méthode est la perte de similarités locales des instances

items ayant servi pour distribuer	base correspondante dans le schema	items triviaux
A1	b1	A1
A2	b2	A2
A2 puis B1	b21	A2 et B1
A2 puis B2	b22	A2 et B2
A2 puis B3	b23	A2 et B3
A3	b3	A3
A4 puis B1	b41	A4 et B1
A4 puis B2	b42	A4 et B2
A4 puis B3	b43	A4 et B3

TAB. A.1 – Items devenus triviaux après distribution.

AND		
0	0	0
0	1	0
1	0	0
1	1	1

OR		
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
0	0	0
0	1	1
1	0	1
1	1	0

ANDNOT		
0	0	0
0	1	0
1	0	1
1	1	0

TAB. A.2 – Tables de vérité pour les fonctions logiques AND, OR, XOR et ANDNOT.

fonction logique	valeurs obtenues
AND	00...00 dans tous les cas
OR	11...11 dans la plupart des cas (pour les autres cas, majorité de bits à 1 dans le repliement)
ANDNOT	00...00 dans la plupart des cas (pour les autres cas, majorité de bits à 0 dans le repliement)
XOR	l'ensemble des valeurs de repliement est représenté

TAB. A.3 – Valeurs de repliement pour différentes fonctions logiques.

dans le "pliage". Ainsi, en utilisant la fonction XOR, deux instances similaires (en terme d'items contenus) peuvent se retrouver distribuées dans des fragments distincts.

FIG. A.3 – Découpage d'enregistrement.

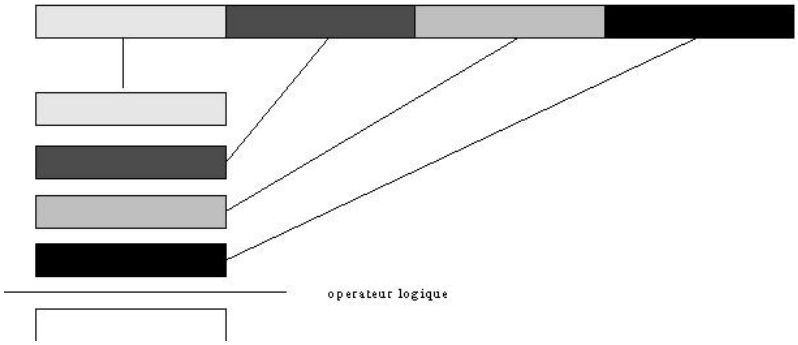


FIG. A.4 – Repliement d'enregistrement.

Une troisième idée (celle utilisée pour ces expérimentations préalables) consiste à utiliser un (ou plusieurs) modèle(s) d'instances, le même modèle sur tous les sites de traitement participant à la fragmentation et de fragmenter les instances en utilisant la distance à ce(s) modèle(s) d'instances.
Le(s) modèle(s) d'instances et la fonction de distance entre les instances et le(s) modèle(s) doivent être choisis de manière à permettre la conservation ds similarités entre instances.

Fragmentation par rapport à un modèle d'instances

Le premier paramètre pour l'étape de fragmentation proposée consiste à choisir le(s) **modèle(s) d'instances**.
Dans un premier temps, un modèle d'instances a été aléatoirement généré. Ce modèle ne correspondant pas aux spécificités de la base de données à traiter, cette solution n'était pas satifsaisante.
Le modèle d'instances est donc aléatoirement choisi parmi les instances de la base (une instance est élue comme modèle d'instances de référence). Les résultats obtenus correspondent dans ce cas aux spécificités de la base traitée.

Le choix aléatoire d'une instance en tant que modèle d'instances peut amener à sélectionner une instance particulière (dans sa structure, dans les informations contenues), ce qui

peut influencer la fragmentation résultante. Une variation possible consiste alors à utiliser non pas un mais plusieurs modèle (plusieurs instances choisies comme modèles d'instances) ou d ne sélectionner que des instances qui correspondent à certaines fonctions de qualités (afin d'assurer que ces instances sont bien représentatives de la base). Reste alors à fixer les critères de qualité pour la sélection.
Une autre possibilité consiste à construire un modèle cohérent avec la base utilisée, une sorte d'instance moyenne conservant la structure des instances, en utilisant les informations issues du pré-traitement. Par exemple, la phase de pré-traitement fournit la proportion de chaque item issu d'un même attribut continu.

Le second paramètre à prendre en compte dans la fragmentation est la **fonction de distance** utilisée pour effectuer la fragmentation par rapport à la distance à un modèle d'instances.
Travaillant sur des attributs discrets (items), ce sont des fonctions de distance binaire qui doivent être utilisées. On considère les instances comme des vecteurs de bits signalant la présence (1) ou l'absence (0) d'un item dans l'instance.
D'autres fonctions de distance et de dissimilarité sur données binaires sont dès lors étudiées, elles sont inspirées des mesures de similarités de Jaccard, Salton et Cosinus, ou d'utilisation des fonctions logiques (voir Section A.1.2).

Puisque le but de la fragmentation est de fournir des fragments d'instances tels que les instances au sein d'un fragment sont le plus similaire possible, en terme d'items contenus (bits communs à 1 dans la représentation binaire), une attention particulière doit être donnée à ces similarités d'items (et donc similarité de bits à 1).
Une fonction de dissimilarité, inspirée des travaux de bio-informatique pour l'alignement de séquences, a été construite. Cette fonction donne une pénalité particulière aux dissimilarités d'items, et un gain aux similarités d'items. Ainsi, en fonction des paramètres choisis, une dissimilarité d'items est admise pour x similarités d'items. Les valeurs de dissimilarité obtenues donnent une attention plus particulière aux similarités comme attendu (pour rappel, c'est la maximisation des similarités au sein d'un fragment d'instances qui permet la diminution de l'espace de recherche à visiter et donc du temps d'exécution nécessaire à la résolution du problème de génération d'itemsets fréquents, voir Section 2.1.2.

A.1.2 Fonctions de distance

On peut considérer deux types d'approches :
- le calcul du degré de similitude, de ressemblance.
- le calcul du degré de dissemblance, de distance à proprement parlé.

Pour le problème qui nous intéresse (la recherche d'itemsets fréquents puis de règles d'association qui doit suivre la fragmentation traitement), il convient de traiter de façon spécifique les cas de double zéros (bit à 0 dans le modèle et dans l'instance). En effet, la présence d'un double zéros ne représente pas une similarité de présence d'items, mais d'absence d'items. Il semblerait, au vu des premiers tests effectués avec la distance de Hamming, qu'il faille différencier le cas du double zéros du cas du double uns.

La Table A.4 présente plusieurs fonctions de distances ou similarité.

Certaines des distances définies par la suite s'appuie sur les notations et valeurs suivantes :
- c_1= #(1,1), (*nombre de doubles 1*) ;
- c_2= #(0,0), (*nombre de doubles 0*) ;
- c_3= #(0,1) + #(1,0), (*nombre de différence de bits*).
- *n* représente le nombre total de bits des enregistrements que l'on considère
- *nS* représente le nombre de bits significatifs (donc $nS = n - c_2$).

- Valeur du BitSet : considérer la valeur décimale des vecteurs de bits (instances), renvoyer la différence entre ces valeurs décimales.
- Permutation : considérer les vecteurs de bits comme des représentations de permutations (voir Exemple Table A.5).
  Problème : Les permutations n'ont pas le même nombre d'éléments (5 éléments pour l'une, 20 pour l'autre). De plus, on n'a pas les mêmes éléments dans les permutations. On a des permutations de longueurs différentes, et ne contenant pas les mêmes éléments. Les distances entre les permutations ne sont donc pas adaptées ici.
  Remarque : De plus, le calcul de distance entre permutations est un problème NP-dur.)

  1. Une solution pour comparer des permutations seraient de les considérer deux à deux. L'une des permutations (p1) serait prise comme origine. On calculerait alors le nombre de modifications à effectuer pour transformer p1 en p2.
     Attention dans ce cas à respecter l'inégalité triangulaire ! ! Attention également à la complexité dans ce cas.
  2. Une autre solution consisterait à calculer la distance par rapport à la permutation origine {1234...}.

  Exemple :Pour calculer la distance de {2143} à l'origine {1234}, on effectue un algorithme de tri sur {2143}, puis on compte le nombre de permutations nécessaires pour ce tri.
  Attention : Par le calcul de la distance à une origine, on ne conserve pas l'inégalité triangulaire.

- Bio-info (mesure de distance) : Ce qui nous intéresse réellement ce sont les égalités d'items.
  Ainsi :
  - une égalité de bits à 1 correspond à une égalité d'items, elle doit donc être récompensée.
  - une égalité de bits à 0 témoigne d'une ressemblance (pas celle qui nous intéresse), elle ne doit pas être pénalisée.
  - une différence de bits correspond à une différence d'items (différence handicapante pour notre problème), elle doit donc être fortement pénalisée.

La pénalité en cas de différence de bits (cas 3) doit-elle être supérieure/inférieure/égale à la "récompense" en cas d'égalité d'items (cas 1).

Fonction	valeur	intervalle de valeurs
Valeur de Bitset	différence des valeurs décimales	
Permutation	indices des permutations	
distance de Hamming	$\frac{c_3}{n}$	
distance par Présence d'items		$[0, 1]$
version de base	$\frac{n-c_1}{n}$	
amélioration	$\frac{n-c_1}{nS}$	
distance Bio-info		$[-1, 2]$
version de base	$\frac{2*c_3-1*c_1}{n}$	
amélioration	$\frac{2*c_3-1*c_1}{nS}$	
mesure de Salton	$\frac{c_1}{\#(1,0)}$	$[0, 1]$
cas particulier si #(1,0) = 0	1	
mesure du Cosinus	$\frac{c_1}{\sqrt{\#(0,1)*\#(1,0)}}$	$[0, c_1]$
cas particulier si #(1,0) = 0 ou si #(0,1) = 0	1	
mesure du Jaccard	$\frac{c_1}{nS}$	$[0, 1]$

TAB. A.4 – Les mesures de similarité

indices de permutation :	1	2	3	4	5	6	7	8
enregistrement :	1	0	1	0	0	0	1	1
Permutation	{1378}							

TAB. A.5 – Exemple de similarité basée sur les permutations

Dans la version basique, on considére :

- une pénalité de 2 en cas de différence de bits ((1, 0) ou (0, 1)) ;
- une récompense de 1 en cas d'égalité d'items ((1, 1)) ;
- pas de modification en cas d'absence d'items ((0, 0)).

L'intervalle de valeurs obtenu pour la distance Bio-Info (les cas extrêmes) :

- si on a uniquement des (0,1) et (1,0), on obtient : +2 $\frac{2*c_3}{c_3}$ ((#(0, 1) + #(1, 0) étant le nombre de bits significatifs nS dans ce cas).
- si on a uniquement des (1,1), on obtient -1 $\frac{-1*c_1}{c_1}$.
- si on a uniquement des (0,0), on obtient 0.

Au pire, on a : ((2*n)/n) = 2
Au mieux, on a : ((-1*n)/n) = -1

Remarques sur les distances

De légères transformations des implémentations ont dû être effectuées pour ne pas obtenir de valeurs infinies (éviter les divisions par 0).
Toutes les mesures ont pu être ramenées entre 0 et 1.

On peut facilement passer d'une mesure de similarité à une mesure de distance :

1 - similarité = distance
1 - distance = similarité

Propriétés des distances

Une fonction de distance, a proprement parlé, doit respecter les propriétés suivantes :

- égalité d(a,a)=0
- réflexivité d(a,b)=d(b,a)
- transitivité si d(a,b)=0 et d(b,c)=0 alors d(a,c)=0
- inégalité triangulaire : d(a,c) <= d(a,b) + d(b,c)

Pondération de distances

La première étape consiste à définir une fonction de distance **d** qui conviendrait pour le calcul de distance à un modèle.

On peut ensuite identifier deux types de pondérations de distances à plusieurs modèles (on supposera qu'on affecte le même poids à toutes les distances aux modèles) :

1. la sommation des distances

$$\frac{\sum_{i=1}^{n} d_i}{n}$$

2. la distance euclidienne

$$\frac{\sqrt{\sum_{i=1}^{n} d_i^2}}{n}$$

La pondération par sommation tolère une distance importante sur une partie des champs.

La pondération par calcul de distance euclidienne favorise des voisins dont tous les champs sont assez voisins.

Intérêt de la méthode :
- Limiter l'impact d'un unique modèle .
- Redistribuer les données si les partitions obtenues sont trop grandes.

A.1.3 Résultats de répartition

La nécessité d'utiliser un modèle d'instances qui respecte la strcture des instances de la base de données (une instance moyenne) étant apparue, il existe deux possibilités pour générer un tel modèle :
- un modèle binaire : chaque bit du modèle(associé à un item) prend la valeur majoritaire dans les instances (1 si l'item est présent pour plus de 50% des instances, 0 sinon).
- un modèle réel avec pour chaque item la proportion d'instances de la base qui le contiennent.

Toutes ces informations sont disponibles grâce à la phase de discrétisation.

Pour le premier modèle d'instances (le modèle binaire par bit majoritaire) les fonctions de distances binaires présentées plus haut sont utilisées. Pour le second modèle d'instances (le modèle réel), la distance par sommation sur chaque item peut être utilisée, tout comme la distance euclidienne .

La fragmentation est réalisée en fonction des distances au modèle d'instances. Des intervalles de valeurs (pour la distance) sont fixés, à chaque intervalle est associé un fragment d'instances. Chaque instance est affectée au fragment associé à l'intervalle dans lequel se trouve la valeur de sa distance au modèle.

La première fonction de distance à avoir été testée dans le schéma est la distance de Hamming.
Cette fonction permet de compter le nombre de bits équivalents (nombre d'items équivalent donc) entre le modèle et chaque instance à distribuer. La distance de Hamming semble être un choix adapté puisque l'on traite des instances de type binaire (voir Figure

Fonction de distance	Taux de distribution des instances (seuls les fragments non vides apparaissent dans la table)						
	modèle binaire = instance binaire **choisie** aléatoirement						
Hamming	1,7	16,5	33,5	37,7	9,8	0,8	
variation de Hamming	1,3	7,8	28,8	43,3	16,5	2	0,17
Bioinformatic	1,3	7,8	28,8	43,3	16,5	2	0,17
Salton	0,7	3,5	3,3	13,5	66,3	12,7	
Cosinus	0,7	1,8	2,17	22,7	22,8	38	11,8
Jaccard	0,17	99,8					
	modèle binaire **générée** aléatoirement						
Hamming	1,7	1,7	10,8	50,17	37,17		
variation de Hamming	6,7	68,17	25	0,17			
Bioinformatic	3,17	30,7	49,17	0,16	1		
Salton	2,3	97,7					
Cosinus	0,3	13,8	85,8				
Jaccard	99,7						
	modèle binaire= bit majoritaire						
Hamming	7,8	45,17	39,3	6,5	1,17		
variation de Hamming	7,8	45,17	38,3	6,8	1,7	0,17	
Bioinformatic	7,8	45,17	38,3	6,8	1,7	0,17	
Salton	47,3	28,5	8,8	15,3			
Cosinus	5,7	10,17	9,7	42,17	19,8	12,3	0,17
Jaccard	100						
	modèle réel : proportion de chaque item						
Sommation	8	85	3,3	1,7			
Distance Euclidienne	5,7	46	39,7	6,8	1,8		

TAB. A.6 – Taux de distribution des instances pour quelques fonctions de distance (fragmentation par intervalles)

A.1), mais s'avère trop simple pour le problème posé.
Les expérimentations réalisées (voir Table A.6) , en utilisant les distances citées ci-dessus et les intervalles de distance, n'ont pas amené les résultats attendus. Les valeurs de dissimilarités obtenues via les fonctions de salton, jaccard et cosinus utilisées ne se sont pas avérés adaptés au problème car elles ne permettent pas de distinguer réellement un match d'item (un item commun au modèle et à l'instance).

A.2 Items triviaux

A.2.1 Principe

L'idée est de conserver l'information des items liés par la phase de discrétisation (items issus d'un même attribut continue dans la base d'origine et représentant les différentes

Fonction de distance	nb de groupes	Taux de distribution des instances (seuls les fragments non vides apparaissent dans la table)						
		modèle binaire= bit majoritaire						
Hamming	5	14,8	17	10,7	10,3	3,5		
variation de Hamming	6	19,3	16,3	26,17	20,7	9,7	7,8	
Bioinformatic	7	19,3	16,3	13,8	12,3	20,7	9,7	7,8
Salton	3	42,67	35,67	19,67				
Cosinus	5	52,67	17	21,5	4,5	4,33		
Jaccard	2	80,67	19,33					
		modèle réel : proportion de chaque item						
Sommationmmation	3	19,33	38	42,67				
Distance Euclidienne	4	19,33	12,33	28,8	39,5			

TAB. A.7 – Taux de distribution des instances par clustering des distances

classes de valeurs possibles identifiées pour cet attribut).
On pourrait prendre en compte ce lien pour effectuer les calculs de distance

Le problème est que la phase de discrétisation peut être suivi d'une phase d'élagage. Cette phase d'élagage ne devrait laisser que l'item le plus fréquent (si le seuil de support utilisé est supérieure à 50%).

Une solution à ce problème est de se fixer un seuil de trivialité. Si le support d'un item est supérieur au seuil de trivialité, cet item est ajouté àune liste d'items triviaux (il ne sera plus pris en compte).
Ce seuil de trivialité est un paramètre àpréciser avec le seuil de support et le seuil de confiance en sortie de programme (ceci afin de ramener les résultats dans leur contexte).

Exemple : un item A apparait dans 95% des cas → il est considéré trivial.

Le fait d'éliminer de nombreux items par ce seuil de trivialité devrait permettre de s'intéresser aux cas rares.

Question : Que faire de ces items triviaux àla fin :

- les ajouter dans les règles : le problème se pose alors de savoir de quel côté les ajouter (cause ou conclusion) ; il n'y a aucune relation de cause àeffet entre les items triviaux et les règles.
- les écarter (puisqu'il n'y a pas de causalité) et en donner la liste séparemment des règles.

Fonction de distance	Taux d'items triviaux (seuls les fragments non vides apparaissent dans la table)						
	modèle binaire= bit majoritaire						
	Globally obvious : 26,09						
Hamming	43,48	56,52	26,09	34,8	4,3		
variation de Hamming	69,56	43,48	52,17	34,8	26,09	4,3	
Bioinformatic	69,56	43,48	60,87	52,17	34,8	26,09	4,3
Salton	26,09	52,17	8,7				
Cosinus	56,52	43,48	34,8	26,09	4,3		
Jaccard	26,09	30,4					
	modèle réel : proportion de chaque item						
Sommation	69,56	43,48	17,4				
Distance Euclidienne	69,56	69,56	43,48	17,4			

TAB. A.8 – Taux d'items triviaux (seuil de trivialité de : 95%)

A.2.2 Résultats de test

Si un seuil de trivialité est utilisé, seuil au dessus duquel un item peut être considéré comme apparaissant dans toutes les instances, de nombreux items peuvent être écartés du traitement, permettant ainsi d'obtenir une diminution significative de la taille de l'espace de recherche à visiter (exponentiel par rapport au nombre d'items à prendre en compte). Les items écartés sont alors réinjectés, en fin de traitement sur chaque fragment, ou donnés en tant que résultat indépendant. Dans les expériences réalisées (voir Table A.8), on identifie plus d'items triviaux dans chacun des fragments obtenus par clustering que dans la base entière. Ceci confirme l'intérêt de la distribution par intelligente. La complexité des traitements prend en compte le fait d'écarter ces items triviaux en début de traitement des fragments.

A.3 Recherche d'itemsets de fréquence faible

Le problème de la génération des règles d'association consiste à identifer des règles de support supérieur à un seuil fixé (règle dont le nombre d'occurences dans les instances est supérieur à un seuil).

L'espace de recherche visité pour la recherche des itemsets fréquents est grand (exponentiel par rapport au nombre d'items à considérer), mais peut être diminué en pratique grâce à l'utilisation du principe d'élagage possible grâce aux propriétés :

Propriété 1 : Tous les sous-ensembles d'un itemset fréquent sont fréquents.

Propriété 2 : Tous les sur-ensembles d'un itemset infréquent sont infréquents.

Cependant, ces propriétés ne permettent pas de résoudre le problème de recherche de règles et d'itemsets de support faible (cas rares plutôt que fréquent). **Idée** : au lieu d'un seuil maximal de support, on pourrait utiliser un seuil minimal de support.

- si sup(A) $<$ seuil, je conserve A,
- sinon élagage.

Le problème amené par l'uilisation d'un seuil de rareté plutôt que de fréquence provient du faut que pour que {A, B} soit infréquent, il n'est pas nécessaire que {A} et {B} soient infréquents.
Cela nécessite de partir à l'envers par rapport à la recherche des itemsets fréquents.

A.3.1 Recherche des k-itemsets rares

Principe :
- On parcours le treillis d'itemsets à l'envers par rapport aux algorithmes de recherche des itemsets fréquents.
- pour que {A} soit infréquent, il faut que{A, B} soit infréquent.

On peut démarrer la recherche par les $k-$itemsets (k = nombre moyen d'items par record) au lieu de $1-$itemsets.
Le problème est que le nombre de combinaisons possibles (de $k-$itemsets) est trés grand si k et m (nombre d'items) sont grands.

$$C_k = \frac{m!}{(m-k)!k!}$$

.

La solution que nous proposons consiste à croiser les deux bornes de support, une seuil de fréquence et un seuil de rareté (i.e. parcourir le treillis d'itemsets dans les deux sens).

Algorithme :
On commence par les (m-1) itemsets, m étant le nombre d'items ! ! ! !
Puis on traite les (m - 2) itemsets, puis les (m - 3) itemsets... (on peut espérer que trés vite des branches entières du treillis seront supprimées, comme c'est le cas dans le parcours de recherche des itemsets fréquents).
Eventuellement, on peut vérifier que le m-itemset est infréquent.

B Evaluation du clustering

B.1 Utilisation de l'algorithmes des K Moyennes pour la discrétisation

Exemple :
On se place dans un espace euclidien de dimension 2, on fixe k=2 (on cherche à former deux groupes).
On considère 8 points de l'espace : A, B, C, D, E, F, G, H dont les coordonnées dans l'espace deux dimensions sont données Table B.1.

La méthode des K- Moyennes consiste dans un premier temps à choisir aléatoirement deux points qui seront les centres initiaux des deux groupes recherchés.
Nous considérons que ce sont des points issus de l'ensemble de données qui sont choisis plutôt que des points générés aléatoirement.
B et D sont choisis, on répartit les points A, B, C, D, E, F, G, H dans les groupes associés aux centres B et D.
On calcule les nouveaux centres de groupe (par moyenne des valeurs affectées à chaque groupe). On obtient B et I. Ces deux nouveaux centres sont utilisés pour la deuxième itération, on réaffecte les points A, B, C, D, E, F, G, H aux groupes en fonction de leur proximité à D et I. Et ainsi de suite . . .
A la quatrième itération, on obtient la stabilité (les groupes ne sont pas modifiés entre la troisième et la quatrième itération). On obtient donc au final deux groupes : {A, B, C, D} et {E, F, G, H}.

Difficultés de l'algorithme des K Moyennes

- Il est nécessaire de disposer d'une bonne fonction de distance, ainsi que d'une méthode de calcul de la moyenne entre enregistrements.
- Les résultats sont trés dépendants de l'initialisation des centres.
- Le nombre de groupes que l'on va former est choisi arbitrairement si on fixe k. En réalité, on ne sait pas quelle valeur de k permettrait de mieux segmenter les données.

Pour résoudre le troisième problème soulevé, on utilise un algorithme itératif. On fait varier la valeur de k pour trouver celle qui est la plus adaptée aux données que l'on traite.

points	itération 1	itération 2	itération 3
	centres B(2,2) D(2,4)	centres D(2,4) I(27/7,17/7)	centres J(5/3,10/3) K(24/5,11/5)
A(1,3)	B	D	J
B(2,2)	B	I	J
C(2,3)	B	D	J
D(2,4)	D	D	J
E(4,2)	B	I	K
F(5,2)	B	I	K
G(6,2)	B	I	K
H(7,3)	B	I	K

TAB. B.1 – Affectation des points aux groupes.

La discrétisation par l'algorithme des K-Moyennes est simple à distribuer, chaque attribut devant être traité de manière indépendante. On se contente donc de distribuer la base par un découpage vertical. Chaque site traite une partie des attributs continus. Les caractèristiques de chaque attribut communes à toutes les instances de la base sont ainsi conservé (critères globaux), et on peut appliquer directement l'algorithme des K-moyennes.

B.2 Facteur k

Dans l'algorithme CDP, le paramètre k de l'algorithme des K- Moyennes est utilisé à plusieurs niveaux utilisant l'operation de fragmentation $\mathcal{F}$ (voir Section 3.3.2).
Pour la phase initiale correspondant à une fragmentation des attributs (dont peut d'ailleurs découler la discrétisation voir Section 3.3.3), et après la phase de croisement lors du regroupement optimisé, qui correspond à une fragmentation des clusters obtenus par croisement (voir Section 3.3.3).
Au vu des tests réalisés, il est apparu que pour obtenir une bonne qualité des résultats par l'algorithme CDP, il est nécessaire de fixer un nombre k deux à trois fois supérieur au moins au nombre de fragments souhaités pour les étapes intermédiaire de regroupement optimisé, ceci afin de laisser de la marge d'erreur à la méthode (voir Section 3.6.5). Lors de la dernière itération de regroupement optimisé, le nombre k lié aux nombre de fragments souhaités peut être ramené au nombre de noeuds disponibles pour le traitement qui doit suivre, c'est-à-dire la recherche de règles d'association.

B.3 Expérimentations de discrétisation des attributs continus

Les expériences ont été menées sur des données réelles issues de la base Diabcare (182 attributs continus avant phase de discrétisation et 30 000 enregistrements).

Les programmes d'expérimentation ont été réalisés sur base de Java RMI, en environnement linux hétérogène, avec implémentation des mécanismes de pipeline pour les différentes phases du schéma.

B.3.1 Résultats des expérimentations

Dans les test réalisés, le nombre de classes (ou items) à identifier est limité à 4 pour chaque attribut continu dans cette phase de discrétisation (au lieu de tenter de trouver le k le plus approprié). Cette borne est un paramètre et peut être augmentée, voir supprimée (au risque que certaines attributs continus de la base initiale fournissent trop d'items dans la base discrétisée).

La phase de clustering de discrétisation permet d'obtenir 480 items. En utilisant un maximum de $k = 4$ classes pour chaque attribut à discrétiser, on multiplie environ par 3 le nombre d'attributs de la base, la base discrétisée D est trois fois plus large que la base initiale B (en terme de nombre d'attributs), mais ce sont des attributs discrets cette fois.

Pour rappel (voir Chapitre IV, les données discrétisées peuvent être stockées sous forme binaire. En supposant les attributs continus initiaux de type flottant (4 octets), si l'étape de discrétisation identifie 4 items pour chaque attribut (un item pouvant être sur 8 bits, c'est-à-dire 1 octet et en considérant l'exclusion entre les items issus d'un même attribut continu), le gain est environ de 4. En traitant 100 millions d'instances, on réduit la taille mémoire nécessaire au stockage des données de 3 Go à moins de 800 Mo, par la phase de discrétisation, ce qui est loin d'être négligeable .

La largeur de la base a augmenté, mais cette discrétisation est nécessaire pour permettre le traitement de recherche de règles d'association. D'où l'intérêt de limiter le nombre d'items obtenables à partir d'un attribut initial (continu).

De façon à limiter cette augmentation du nombre d'items, nous avons songé à anticiper sur le traitement de recherche des itemsets fréquents, et à introduire une phase d'élagage (pruning).
De nombreux items identifiés lors de la discrétisation s'avèrent rares (peu fréquents) dès les premières itérations de la génération d'itemsets fréquents.

Cette phase d'élagage consiste à utiliser dès la phase de discrétisation un seuil de support minimal pour supprimer les items non fréquents (voir page 104). Les classes recherchées lors du clustering de discrétisation correspondent aux items utilisés dans la suite du traitement, on peut profiter d'avoir une vision globale (à toutes les instances) de

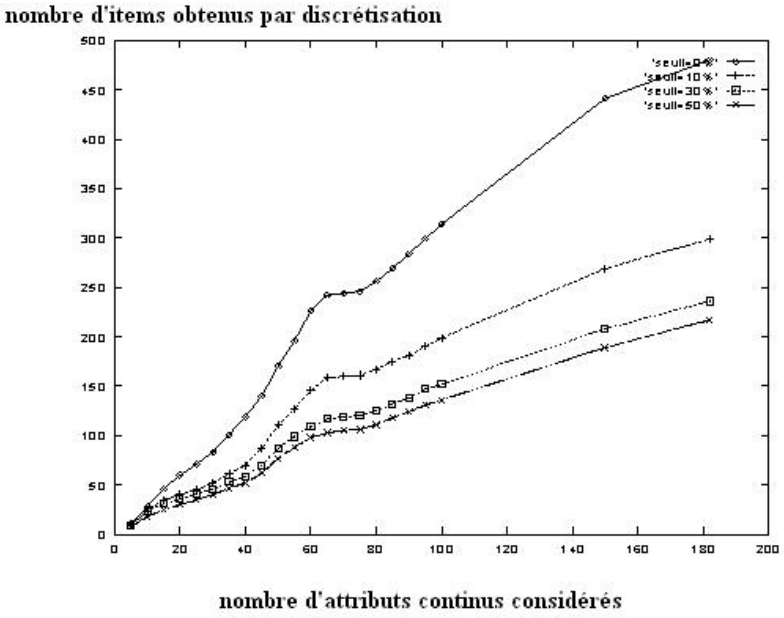


FIG. B.1 – Nombre d'items obtenus aprés la phase d'élagage.

chaque item pour supprimer ceux qui n'apparaissent pas assez souvent (prinicpe de fréquence/infréquence).

La figure B.1 montre l'évolution du nombre d'items obtenus par clustering de discrétiastion (en ordonnée), en fonction du nombre d'attributs de la base initiale B (en abscisse). Les courbes représentent cette évolution pour différentes valeurs de seuil utilisé pour l'élagage : 0%, 10%, 30%, 50% (voir Table B.2 pour les détails).

Cette phase d'élagage peut limiter les effets de l'utilisation d'un nombre k trop élévé dans l'algorithme des k-moyennes utilisé pour la discrétisation. Ainsi, cette discrétisation peut être réalisée normalement, sans limiter le nombre de classes (items) que l'on peut obtenir.

Remarque :
Le seuil de suport fixé à 0% correspond à un clustering de discrétisation sans élagage des items non fréquents.
Plus le seuil de support est élevé, moins le nombre d'items obtenus est élevé : on effectue un élagage global des 1-itemsets non fréquents.

B.4 Granularité du clustering Distribué Progressif

Pour rappel (voir Section 5.1.3), les expérimentations réalisées sur l'infrastructure Grid5000 pour l'algorithme de clustering distribué progressif CDP (étape de fragmentation intelligente) sur la base de données *DiabCare* avaient fourni un taux d'accélération (speedup) acceptable (voir la Figure B.2), proche d'une accélération linéaire.

	nombre d'items obtenus			
seuil de support	0%	10%	30%	50%
nombre d'attributs continus considérés				
5	11	9	9	8
10	28	24	24	18
15	46	35	31	25
20	60	40	36	30
25	71	45	41	35
30	83	52	46	40
35	101	61	53	47
40	119	70	58	52
45	140	87	70	62
50	170	111	87	77
55	196	127	99	88
60	226	146	109	98
65	242	158	117	103
70	244	160	119	105
75	246	161	120	106
80	256	167	125	111
85	269	175	132	118
90	284	181	138	124
95	299	190	147	131
100	314	199	152	136
150	441	269	208	189
182	480	299	236	217

TAB. B.2 – Evolution du nombre d'items obtenus aprés clustering de discrétisation et élagage (en fonction du seuil de support utilisé)

La base de données DiabCare est une base de données médicales présentant des informations réparties sur environ 180 attributs concernant des patients diabétiques. Cette base inclue des données manquantes (attributs non renseignés [1]), et différents types d'attributs : booléens, attributs date, valeurs continues ou énumérées.
Nous ne possédons pas d'information concernant le partionnement réel des patients, de même nous n'avons pas d'information permettant une discrétisation directe des attributs continus. La base possède environ 37 000 instances.

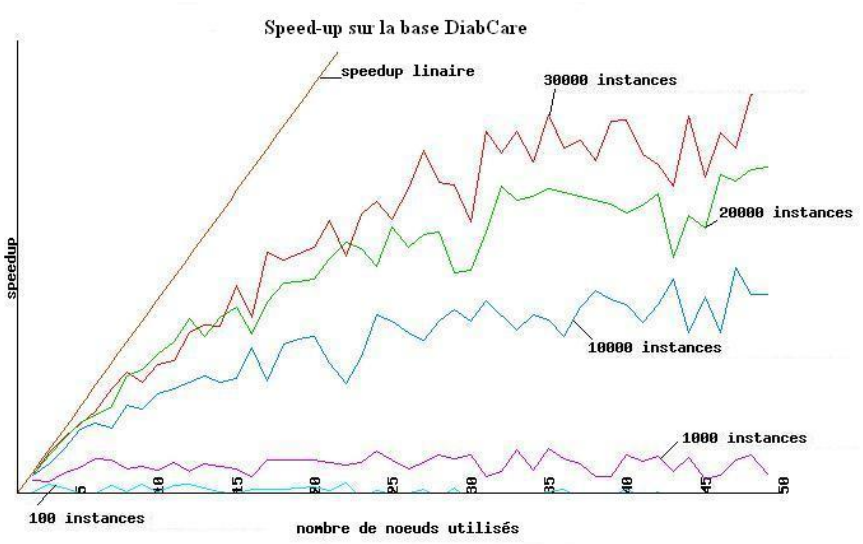


FIG. B.2 – Speed-up obtenu sur la base DiabCare en fonction du nombre de noeuds utilisés et du nombre d'instances considéré.

La base de données Diabcare ayant un nombre d'instances limité, une seconde base de données Forest Cover Type [2] a été utilisée.
Cette base, utilisée en classification, contient un nombre d'instances plus important que DiabCare (581012) et comporte 54 attributs. La base est utilisée en classification pour classer une instance décrivant une forêt parmi un des 7 types de forêts possible.

La Figure B.3 présente speed-up obtenu dans l'algorithme CDP sur cette base Foret en fonction du nombre d'instances considérées.
Le speed-up obtenu sur cette seconde base est comparable à celui obtenu sur la base DiabCare.

Les Figures B.4 et B.5 présente la répartition du temps de traitement entre les différentes étapes de l'algoritghme CDP (chargement des données sur les noeuds, fragmentation initiale et croisement) ainsi que les périodes d'inactivité, en fonction du nombre de noeuds utilisés.

[1] informations importantes à ne pas écarter surtout sur des données médicales. Cette valeur particulière sera considérée comme un item à part entière lors de la discrétisation de chaque attribut

[2] http ://kdd.ics.uci.edu/databases/covertype/covertype.html

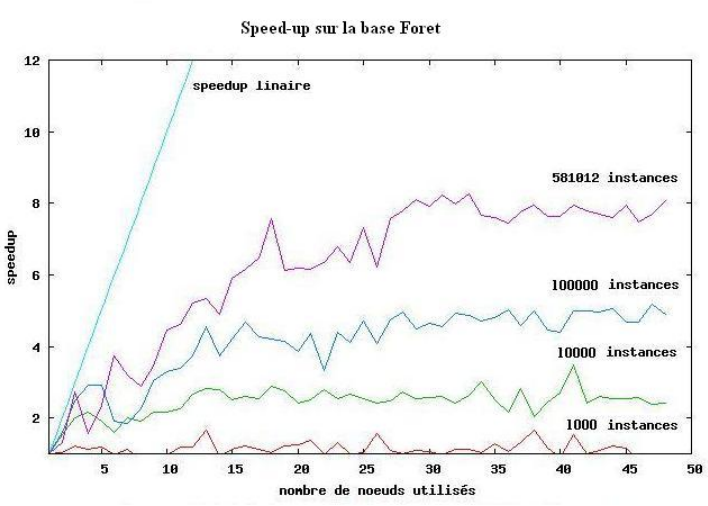


FIG. B.3 – Speed-up obtenu sur la base Foret en fonction du nombre de noeuds utilisés et du nombre d'instances considéré.

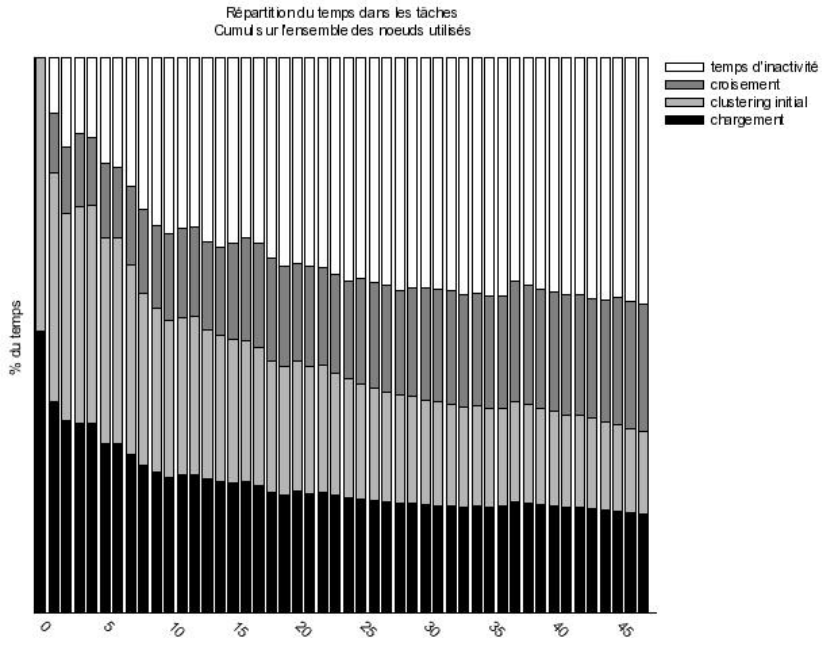


FIG. B.4 – Répartition du temps d'exécution entre les différentes phases de l'algorithme CDP (pour 1000 instances de la base Foret).

Ainsi, on peut noter la plus grand part des périodes d'inactivité dans le temps de traitement lorsque l'on augmente le nombre de noeuds. Ceci est dû à l'inutilité de certains noeuds au fur et à mesure des phase de croisement dans la méthode.
Le temps de chargement des données diminue quant à lui pour un nombre d'instances utilisé faible (voir Figure B.4, 1000 instances), mais s'avère indépendant du nombre de noeuds utilisé pour un nombre d'instances plus important (voir Figure B.5, 581012 instances).

Les parts occupées par la fragmentation initiale et la phase de croisement est comparable pour un nombre d'instances faible (voir Figure B.4, 1000 instances), alors que pour un nombre d'instances plus élevé (voir Figure B.5, 581012 instances), c'est la phase initiale de fragmentation qui prend une part majoritaire dans l'exécution. La phase de croisement travaillant sur des données beaucoup plus restreintes (résultats intermédiaires), sa part dans le temps total d'exécution devient négligeable, d'autant plus que le nombre de noeuds utilisé est important.

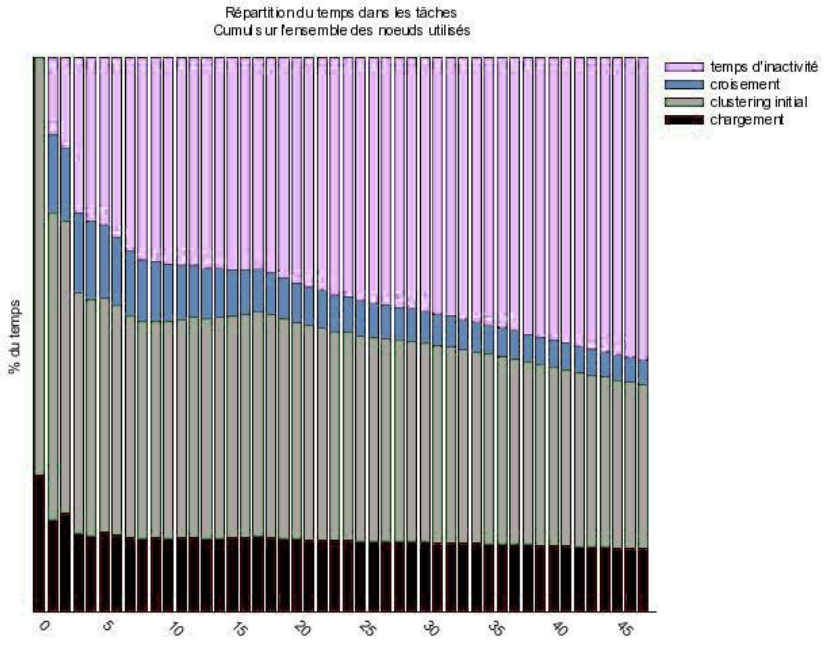


FIG. B.5 – Répartition du temps d'exécution entre les différentes phases de l'algorithme CDP (pour 581012 instances de la base Foret).

C Résultats d'expérimentations du traitement collaboratif par l'algorithme DICCoop

C.1 Synthèse des expérimentations préalables sur l'algorithme DICCoop

C.1.1 Conditions d'expérimentations

Une première série d'expérimentations de l'algorithme DICCoop (voir Section 4.2) a été réalisée sur base d'une distribution aléatoire uniforme d'instances. Le principe de distribution intelligente proposée dans le projet DisDaMin n'a pas été utilisée pour ces expérimentations de manière à étudier le comportement de l'algorithme sur une distribution telle que celles utilisées dans les travaux existants.

Les expérimentations ont été réalisées sur base d'implémentations java effectuant seulement en partie le recouvrement de communications, de manière à permettre une observation des comportements engendrés par les apports d'informations extérieures. Les communications sont toutefois non synchronisées donc non bloquantes.

Les expériences ont été réalisées sur une grappe homogène de stations (Pentium 4 2.66Ghz, 256Mo, système d'exploitation Windows 2000, type salle de travaux pratiques universitaires). Ce type d'architecture est non dédiée. Lors des expérimentations, les processeurs des machines étaient réservés au traitement, mais pas le réseau.

Trois méthodes de communication entre le fédérateur F et les sites T_i ont été comparées (voir Section 4.2.5) : **Aucune communication**, **Tout communiquer**, **des communications limitées et paramétrées**.

C.1.2 Synthèse des observations effectuées

La version sans **aucune communication** consiste à effectuer des traitements indépendants par l'algorithme DIC classique, et amène donc aux problèmes observés dans la pré-version utilisant des traitements indépendants sur base de l'algorithme Apriori (voir Section 2.3.1), à savoir : les "parasites" locaux et le problème des cas limites (incomplé-

tude des supports).

La version avec **communications paramètrées** permet d'effectuer des échanges d'informations quand la quantité d'informations à envoyer est *suffisante*. Tous les sites T_i travaillant sur des instances a priori similaires (répartition aléatoire uniforme), les traitements locaux ne sont pas spécialisés sur un sous-ensemble d'items et donc la taille de l'espace de recherche visité n'est donc pas diminué.

Lorsque l'on utilise ce mode de communications, il s'avère que, les sites T_i travaillant sur des données similaires, l'apport des informations engendré par les communications est limité dans les traitements locaux. En effet, les sites travaillant sur des quantités de données similaires et de même complexité (en rapport avce le nombre d'items considéré sur chaque site), ils travaillent à des vitesses comparables. Les envois d'informations locales vers le fédérateur puis des informations globales du fédérateur F aux sites T_i ne permettent pas une exploitation optimale localement, car ces informations arrivent souvent trop tard (après qu'un travail inutile ait été réalisé ou après qu'une communication inutile ait été réalisée). Ainsi, les informations reçues ne permettent pas d'ajouter des traitements locaux (traitements de spécialisation pour anticiper certains résultats), ni de supprimer des communications vers le fédérateur (informations déjà identifiées comme inintéressantes globalement). Ceci correspond à un « pire cas », dans lequel le fédérateur a simplement un rôle d'« accumulateur ».

La version **tout communiquer** permet de contourner ce problème de retard d'arrivée des informations, en admettant des communications fréquentes (et toujours NON synchronisées). Un site plus rapide permet ainsi de venir enrichir ces voisins.
En particulier, cette méthode apparaît plus apte (que la méthode de communications paramétrées) *à traiter des données distribuées de manière aléatoire* (donc non fragmentées de manière "intelligente").

Ces expérimentations confirme le rôle d'accumulateur du fédérateur qui permet la validation des résultats au fur et à mesure plutôt qu'a posteriori.
Ces expérimentations permettent également de constater un problème de dépassement des capacités locales de traitement. En effet, malgré la distribution, les fragments de données à traiter sur les sites T_i peuvent être de trop grande taille, et donc se confrontent à la limite de capacités de traitement **ET** de stockage sur les sites.

C.2 Impact du taux de parallélisme

Les figures C.1 et C.2 présentent respectivement les temps minimum et maximum observés sur les noeuds lors d'exécutions de l'algorithme DICCoop sur 4, 8 et 16 noeuds (voir Section 4.4.6 pour les autres résultats).

Comme indiqué dans le chapitre présentant les expérimentations complètes, les déséquilibres de charge augmentent entre les sites lorsque le taux de parallélisme augmente. Les

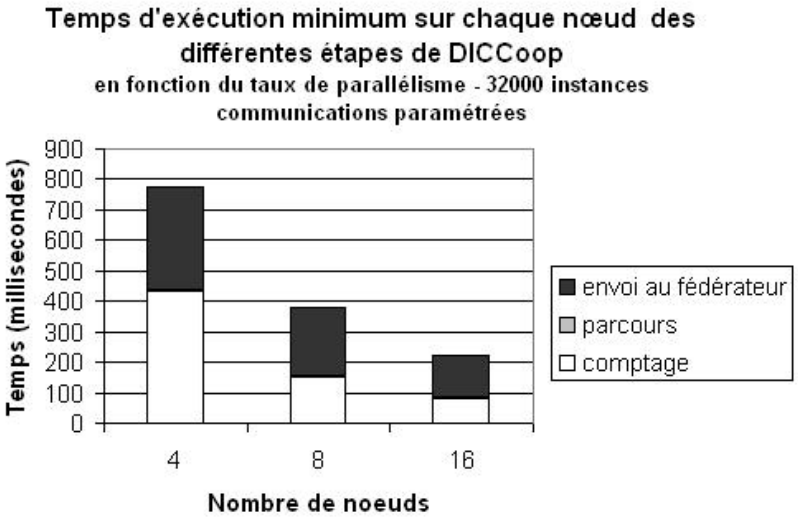


FIG. C.1 – Comparaison des temps minimum d'exécution sur chaque site en fonction du taux de parallélisme utilisé pour l'algorithme DICCoop (30 attributs - 32000 instances - communications paramétrées)

noeuds retardataires augmentent leurs retards, les noeuds plus rapides augmentent leurs avances.
L'accélération sur les noeuds les plus rapides s'explique par le nombre moins important d'instances à traiter (tâche de **comptage**), du fait de la distribution des instances sur un plus grand nombre de noeuds.
Les **envois** moins nombreux vers le fédérateur proviennent du fait que le noeud étant en avance, il n'est pas questionné par le fédérateur (il a anticipé ces questions par des envois d'informations).
Le retard sur les noeuds plus lents provient de l'évolution de la tâche d'envois vers le fédérateur. La tâche de comptage possède un temps quasiment constant. L'augmentation des communications vers le fédérateur correspond au phénomène inverse de celui observé sur les noeuds en avance, le fédérateur pose des questions auxquelles les noeuds doivent répondre (il doit donc effectuer des envois).

On notera qu'il existe un rapport de plus ou moins 4 entre les temps de traitement sur les noeuds les plus rapides et ceux sur les noeuds les plus lents, avec un gain de près de 75% sur l'avance des noeuds plus rapides (entre un exécution sur 4 noeuds et sur 16 noeuds) contre une augmentation de seulement 25% du retard des noeuds plus lents (toujours entre un exécution sur 4 noeuds et sur 16 noeuds).

L'augmentation des quantités d'informations communiquées par et au fédérateur correspond aux augmentations observées sur les temps cumulés et moyens de communications sur les noeuds (voir Section 4.4.6).
Le fédérateur reçoit plus d'informations d'un plus grand nombre de sources (noeuds de traitement). Il doit envoyer les informations globales générées à un plus grand nombre de destinataires (noeuds). L'envoi d'une même information globale à n noeuds est compta-

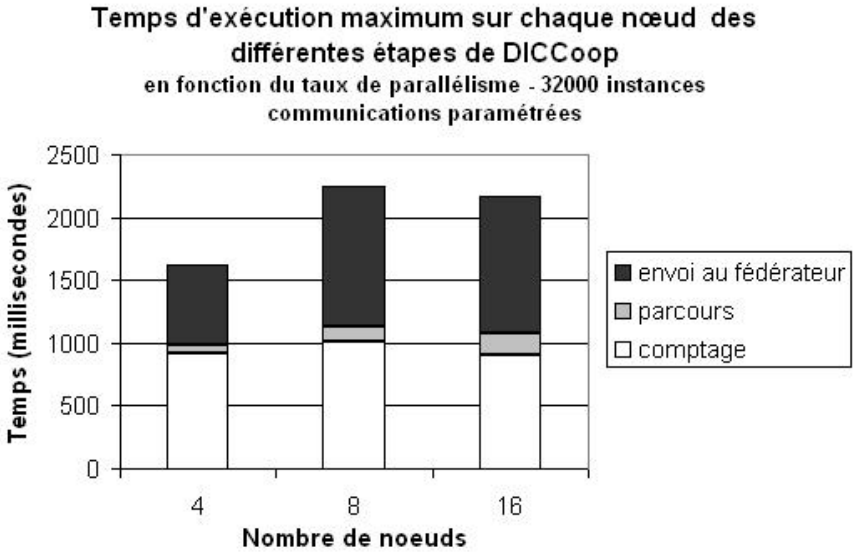


FIG. C.2 – Comparaison des temps maximum d'exécution sur chaque site en fonction du taux de parallélisme utilisé pour l'algorithme DICCoop (30 attributs - 32000 instances - communications paramétrées)

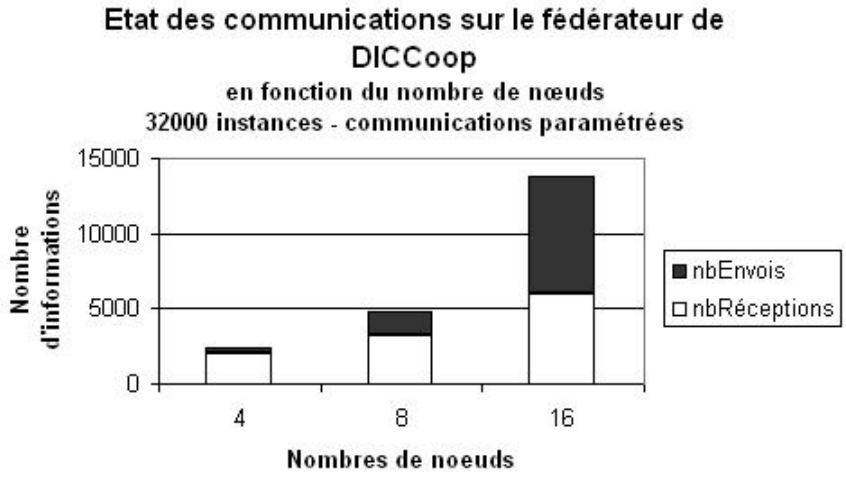


FIG. C.3 – Comparaison des quantités d'informations reçues et envoyées par le fédérateur en fonction du taux de parallélisme utilisé pour l'algorithme DICCoop (30 attributs - 32000 instances - communications paramétrées)

bilisé n fois dans les mesures puisque cela donne lieu à des communications vers chacun des n destinataires.

C.3 Impact du nombre d'instances

Les figures C.4 et C.5 présentent respectivement les temps minimum et maximum observés sur les noeuds lors d'exécutions de l'algorithme DICCoop sur 32000 et 64000 instances (voir Section 4.4.6 pour les autres résultats).

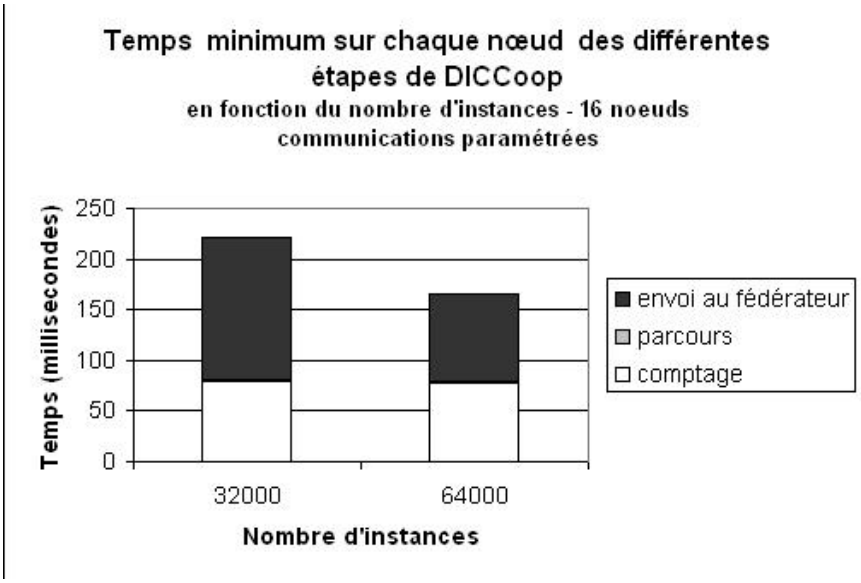


FIG. C.4 – Comparaison des temps minimum d'exécution sur chaque site en fonction du nombre d'instances pour l'algorithme DICCoop (30 attributs - 16 noeuds - communications paramétrées)

Comme indiqué dans le chapitre présentant les expérimentations complètes, les déséquilibres de charge augmentent faiblement entre les sites lorsque le nombre d'instances augmente. Les noeuds retardataires augmentent leurs retards, les noeuds plus rapides augmentent leurs avances.

La faible accélération sur les noeuds les plus rapides (280 millisecondes pour 32000 instances contre 165 millisecondes pour 64000 instances) provient d'une faible diminution des communications effectuées (140 millisecondes pour 32000 instances contre 90 millisecondes pour 64000 instances), et non pas cette fois-ci d'une diminution du temps reservé au comptage grâce à l'anticipation des traitements utiles globalement (ce qui limite les questions du fédérateur à ces sites).
Le temps de **comptage** est stable sur ces sites malgré la diminution du nombre d'instances obtenus par le plus grand nombre de noeuds (environ 80 millisecondes dans les deux cas).

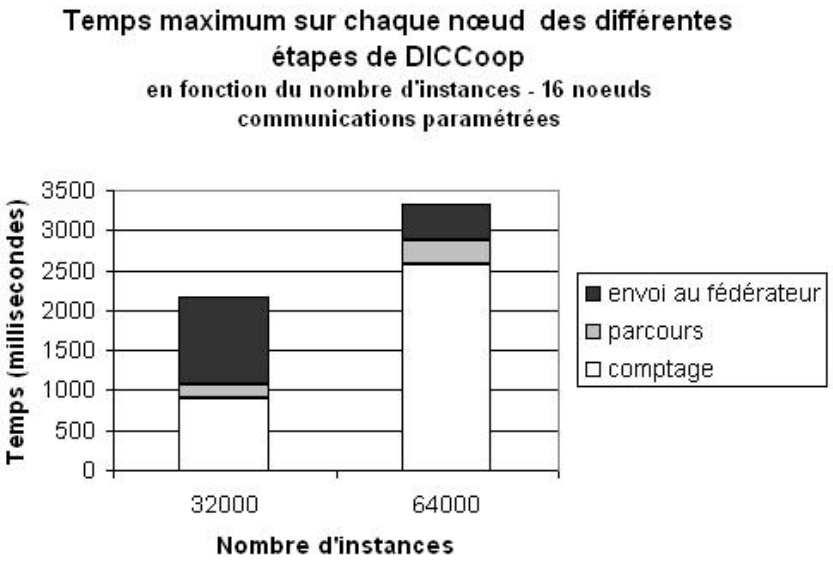


FIG. C.5 – Comparaison des temps maximum d'exécution sur chaque site en fonction du nombre d'instances pour l'algorithme DICCoop (30 attributs - 16 noeuds - communications paramétrées)

L'augmentation du retard sur les noeuds les plus lents (2170 millisecondes pour 32000 instances contre 3300 millisecondes pour 64000 instances) provient d'une augmentation des temps de **comptage** des supports (900 millisecondes pour 32000 instances contre 2600 millisecondes pour 64000 instances), ce qui est cohérent avec l'augmentation du nombre d'instances par noeud.
Le temps de **parcours** de la structure de données augmente également (170 millisecondes pour 32000 instances contre 300 millisecondes pour 64000 instances), mais en proportion plus faible.
Par contre, le temps de **communications** diminue (1100 millisecondes pour 32000 instances contre 450 millisecondes pour 64000 instances), mais en proportion plus faible, par la limitation des envois (grâce aux informations reçues du fédérateur et amenées par les autres noeuds).

La diminution des quantités d'informations communiquées par et au fédérateur correspond aux diminutions observées sur les temps cumulés et moyens de communications sur les noeuds (voir Section 4.4.6).
On peut cependant considérer que le nombre de réceptions d'informations restent stables : 6000 informations reçues pour 32000 instances contre 4300 pour 64000 instances (qui correspondent à la faible augmentation d'envois observée sur chaque noeud multipliée par le nombre de noeuds),
Par contre, le nombre d'informations émises diminuent fortement (7900 informations envoyées pour 32000 instances contre 7500 pour 64000 instances), mais avec toujours une même information comptabilisée plusieurs fois si elle est envoyée à plusieurs noeuds.
Cette diminution des envois s'expliquent par les différences importantes entre les temps de traitement des noeuds retardataires et des noeuds en avance (un rapport de plus ou

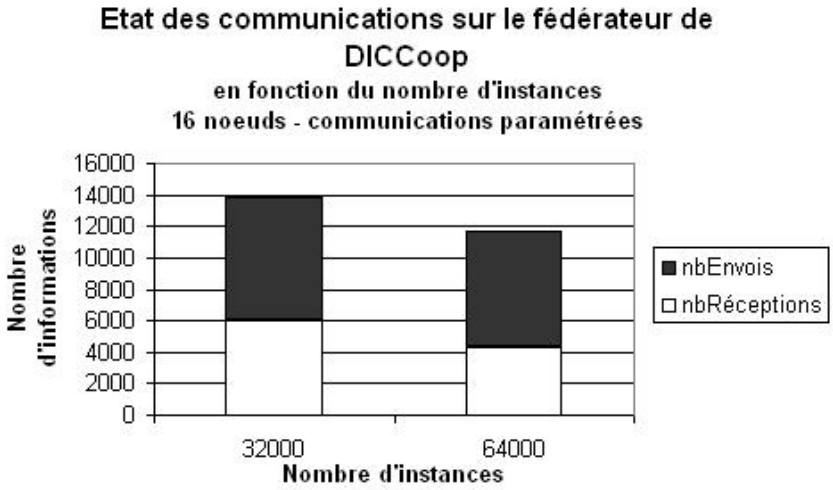


FIG. C.6 – Comparaison des quantités d'informations reçues et envoyées par le fédérateur en fonction du nombre d'instances pour l'algorithme DICCoop (30 attributs - 16 noeuds - communications paramétrées)

moins 10 entre les temps de traitement sur les noeuds les plus rapides et ceux sur les noeuds les plus lents), avec un gain de 20% sur l'avance des noeuds plus rapides contre une augmentation de 50% du retard des noeuds plus lents.

D Grid 5000

D.1 Le projet GRID'5000

Grid'500 est une plateforme expérimentale qui répond aux critères suivants, elle est :

- configurable,
- contrôlable,
- monitorable.

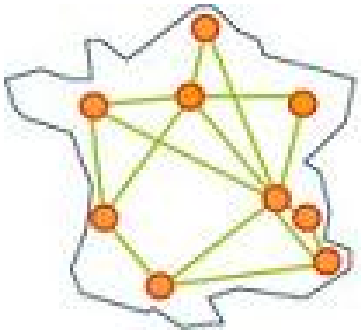


FIG. D.1 – Répartition des sous-grappes de Grid5000 dans l'hexagone.

La plateforme est constituée de 9 sites géographiquement distribués en France (Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia-Antipolis, Toulouse).

Chaque site dispose d'un cluster comportant de 100 à 1000 noeuds, pour un total de 5000 CPUs.
Les clusters sont interconnectés par le réseau d'éducation et de recherche *Renater* avec un débit de 1Gb/s (10Gb/s prochainement).

E Générateur de fragments "intelligents"

Pour valider l'algorithme DICCoop (DIC Coopératif voir Section 4.2), et afin de ne pas interférer dans les tests avec la méthode de clustering proposé pour obtenir une distribution intelligente, nous utilisons dans les expérimentations de validation de la méthode (voir Section 4.4) un générateur de fragments d'instances intelligents.

Le générateur créé k fragments d'instances, le nombre d'instances de chaque fragment est choisi aléatoirement entre une borne minimale et une borne maximale fixée.
Le nombre d'attributs (items) considéré pour le jeux de données est également choisi aléatoirement entre deux bornes fixées ou paramétrées.

On considère 4 catégories d'attributs :

- les attributs suffisamment présents sur le fragment,
- les attributs faiblement présents (voir absents),
- les attributs fortement présents (triviaux),
- les autres.

Pour chaque fragment d'instances, un pourcentage d'attributs suffisamment présents est choisi pour le fragment (toujours entre deux bornes fixées), de même un pourcentage d'attributs fortement absents et un pourcentage d'attributs triviaux sont générés.
Pour chaque attribut, un taux de probabilité est généré (en accord avec le type de l'attribut, voir exemple ci-dessous).

Les instances sont générées en respectant les probabilités sélectionnées.

Exemple :

On souhaite générer $k = 3$ fragments d'instances de m attributs (supposons m fixé arbitrairement à 10).

On fixe les bornes pour la génération du nombre d'instances dans chaque groupe à :

`MIN_NB_INST_PER_GROUP=100 et  MAX_NB_INST_PER_GROUP=5000`

On fixe les bornes pour la génération du pourcentage d'attributs de chaque type (présents, fortement absents, triviaux) respectivement à :

```
MIN_PRCTG_ATT_PRES=25 et  MAX_PRCTG_ATT_PRES=70,
MIN_PRCTG_ATT_ABS=25 et  MAX_PRCTG_ATT_ABS=70,
MIN_PRCTG_ATT_TRIV=5 et  MAX_PRCTG_ATT_TRIV=20;
```

Les taux de probabilité associé à chaque type d'attribut sont :

```
MIN_PRCTG_INST_PRES=20 et  MAX_PRCTG_INST_PRES=60,
MIN_PRCTG_INST_ABS=0 et  MAX_PRCTG_INST_ABS=15,
MIN_PRCTG_INST_TRIV=90 et  MAX_PRCTG_INST_TRIV=100.
```

Le pseudocode du générateur est fourni Table E.1.

---

pour chaque groupe d'instances
generation aléatoire du nombre instance pour ce groupe (nbInst dans [MIN_NB_INST_PER_GROUPS, MAX_NB_INST_PER_GROUPS[)
generation du nombre d'attributs de chaque type (pour ce groupe) pourcentage d'attributs fortement présents sur ce groupe (rateAttFortPres dans [MIN_PRCTGE_ATT_PRES, MAX_PRCTGE_ATT_PRES[) pourcentage d'attributs restants fortement absents sur ce groupe (pAttFortAbs dans [MIN_PRCTGE_ATT_ABS, MAX_PRCTGE_ATT_ABS[rateAttFortAbs = pAttFortAbs*(1-rateAttFortPres)) pourcentage d'attributs restants triviaux sur ce groupe (pAttTriv dans [MIN_PRCTGE_ATT_TRIV, MAX_PRCTGE_ATT_TRIV[rateAttTriv = pAttTriv* (1-rateAttFortPres-rateAttFortAbs)) pourcentage d'attributs autres (le reste) (rateOther=1-rateFortPres-rateFortAbs-rateTriv)
pour chaque attribut génération probabilisée d'appartenance à un type d'attributs (fréquent, absent, trivial, autre) calcul du pourcentage (probabilité) d'instances associées à ce type pour cet attribut si fréquent rateFreq dans [MIN_PRCTGE_INST_FREQ, MAX_PRCTGE_INST_FREQ[si absent rateFreq dans [MIN_PRCTGE_INST_ABS, MAX_PRCTGE_INST_ABS[si trivial rateFreq dans [MIN_PRCTGE_INST_TRIV, MAX_PRCTGE_INST_TRIV[sinon (autre) rateFreq = 50 génération des instances pour chaque instance du groupes génération probabiliste (selon probabilité rateFreq fixée ci-dessus) de l'appartenance de l'item à l'instance

TAB. E.1 – PseudoCode du générateur de fragments intelligents d'instances .

F Diffusion scientifique des travaux

F.1 Articles publiés

F.1.1 Articles scientifiques sélectionnés après avis d'un comité de lecture et à large diffusion dans les milieux scientifiques spécialisés

- « Distributed Data Mining » V. Fiolet, B. Toursel.
  Journal Scalable Computing : Practice and Experiences (SCPE), Special Issue : Internet-based Computing,
  Vol. 6, Number 1, ISSN 1097-2803, march 2003, (p99-109).

F.1.2 Abstracts et actes de congrès, colloques, symposium, etc

- « Optimizing Distributed Data Mining Applications Based on Object Clustering Methods» V. Fiolet, E. Laskowski, R. Olejnik, L. Masko, B. Toursel, M. Tudruj.
  In Proceedings of International Symposium on Parallel Computing in Electrical Engineering (Parelec'06),
  IEEE Computer Society, ISBN : 0-7695-2554-7, pages 257-262, Poland, september 2006.
- « Optimal Grid Exploitation Algorithms for Data Mining» V. Fiolet, G. Lefait, R. Olejnik, B. Toursel.
  In Proceedings of the 5th International Symposium on Parallel and Distributed Computing (ISPDC'06),
  IEEE Computer Society, ISBN : 0-7695-2638-1, pages 246-252, Romania, july 2006.
- « Progressive Clustering for Database Distribution on a Grid » V. Fiolet, B. Toursel.
  In Proceedings of the 4th International Symposium on Parallel and Distributed Computing (ISPDC'05),
  IEEE Computer Society, ISBN : 0-7695-2434-6, pages 282-289, France, july 2005.
- « Intelligent Database Distribution on a Grid using Clustering » V. Fiolet, B. Toursel.
  In Proceedings of Atlantic Web Intelligence Conference (AWIC'05), Workshop Knowledge and Data Mining on Grid (KDMG'05), Lectures Notes In Computer Sciences. Proc. of AWIC 2005,
  LNAI 3528-2005, ISBN : 3-540-26219-9, pages 466, Springer-Verlag , Poland, june 2005.
- « DisDaMin : algorithmes de Data Mining distribués » V. Fiolet, B. Toursel.

Actes du workshop Fouille de données complexes (FDC),
conférence Extraction et de Gestion de Connaissances (EGC'04), Clermont-Ferrand, janvier 2004.

- « Distributed Data Mining » V. Fiolet, B. Toursel.
  In Proceedings of the 1st International Symposium on Parallel and Distributed Computing (ISPDC'02),
  Scientific Annals of "Alexandro Ioan Cuza" University of Iasi, Computer Science Section, TOME XI, pages 349-365, Romania, july 2002.

F.2 Articles en attente de publication

F.2.1 Articles scientifiques sélectionnés après avis d'un comité de lecture et à large diffusion dans les milieux scientifiques spécialisés

- «A Clustering Method to Distribute a Database on a Grid » V. Fiolet, B. Toursel.
  The International Journal of Grid Computing : Theory, Method and Application, Special Issue FGSC (Future Generation Computer Systems).

F.3 Exposés dans des congrés, colloques, symposiums, etc

- Exposé de travaux lors du symposium ISPDC'06 (5th International Symposium on Parallel and Distributed Computing)
  Timisoara (ROUMANIE), Juillet 2006.
- Exposé de travaux "Data Mining sur Grille",
  au Centre Cetic (Centre d'Excellence En Technologies de l'Information et de la Communication)
  Gosselies (BELGIQUE), Mars 2006.
- Exposé de travaux lors du symposium ISPDC'05 (4th International Symposium on Parallel and Distributed Computing)
  Lille (FRANCE), Juillet 2005.
- Exposé de travaux au Workshop Knowledge Data Mining on Grid,
  conférence AWIC'05 (Atlantic Web Intelligence Conference)
  Lodz (POLOGNE), Juin 2005.
- Exposé de travaux Groupe de travail EGPDC (Extraction et Gestion Parallèles Distribuées de connaissances),
  conférence EGC'05 (Extraction et Gestion de connaissances)
  Paris (FRANCE), Janvier 2005.
- Exposé de travaux au Sun HPC Consortium (High Parallel Computing),
  Heidelberg (ALLEMAGNE), Juin 2004.
- Exposé de travaux Groupe de travail FGC (Fouille et Gestion de données complexes),
  conférence EGC'04 (Extraction et Gestion de connaissances)
  Clermont-Ferrand (FRANCE), Janvier 2004.
- Exposé de travaux lors du symposium ISPDC'02 (1st International Symposium on Parallel and Distributed Computing)

Iasi (ROUMANIE), Juillet 2002.