UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

RECHERCHE DE MOTIFS DANS UN CONTEXTE FORMEL TRIADIQUE

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN SCIENCES ET TECHNOLOGIES DE L'INFORMATION

PAR

KAMAL BENMOUSSA

AVRIL 2020

UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

Département d'informatique et d'ingénierie

Ce Mémoire intitulé :

RECHERCHE DE MOTIFS DANS UN CONTEXTE FORMEL TRIADIQUE

présenté par
Kamal Benmoussa

pour l'obtention du grade de maîtrise ès science (M.Sc.)

a été évalué par un jury composé des personnes suivantes :

Prof. Rokia Missaoui ..... Directrice de recherche
Prof. Karim El Guemhioui ..... Président du jury
Prof. Omar Abdul Wahab ..... Membre du jury

Mémoire accepté le :

À mon épouse Najla qui m'a toujours soutenu tout le long de ce programme. À mes deux garçons, Jad et Youssef, qui m'inspirent à me dépasser constamment. Leurs petits encouragements ont eu un énorme impact sur ma détermination à continuer et à ne pas abandonner. Merci !

Remerciements

Je tiens à remercier Mme Rokia Missaoui, Professeur à l'Université Québec en Outaouais, qui m'a encadré tout au long de ce projet et qui m'a fait partager ses brillantes intuitions. Qu'elle soit aussi remerciée pour sa gentillesse, sa disponibilité permanente et pour les nombreux encouragements qu'elle m'a prodigués.

J'adresse tous mes remerciements à Monsieur Karim El Guemhioui, Professeur à l'Université de Québec en Outaouais, ainsi qu'à Monsieur Omar Abdul Wahab, Professeur à l'Université de Québec en Outaouais, de l'honneur qu'ils m'ont fait en acceptant d'être membres du jury de ce mémoire ainsi de leurs commentaires pertinents.

Enfin, je tiens à remercier l'ensemble du personnel et professeurs du département informatique et d'ingénierie de l'université de Québec en Outaouais qui m'ont fourni tout l'appui nécessaire pour la réalisation de ce projet.

Table des matières

Liste des figures

Liste des tableaux

Liste des abréviations, sigles et acronymes

AFC Analyse formelle de concepts

ATC Analyse triadique de concepts

AJAX *Asynchronous JavaScript and XML*

ConExp *Concept Explorer*

CSV *Comma Separated Values*

FCA *Formal concept analysis*

TCA *Triadic concept analysis*

FCC *Frequent Closed Cube*

Résumé

L'analyse triadique de concepts porte sur l'analyse des données tridimensionnelles qu'on représente par un contexte triadique $\mathbb{K} = (K_1, K_2, K_3, Y)$ où K_1 décrit un ensemble d'objets, K_2 un ensemble de propriétés (attributs), K_3 un ensemble de conditions et Y est une relation ternaire reliant ces trois ensembles. L'analyse triadique de concepts est une technique de regroupement conceptuel qui a la particularité de générer à la fois (i) un ensemble de concepts triadiques (*clusters*) reliés entre eux par une relation de quasi-ordre et (ii) des règles d'association et des implications entre des attributs sous des conditions spécifiques ou inversement. L'analyse triadique de concepts peut faire ses preuves dans plusieurs projets de découverte de connaissances. Néanmoins, l'avènement des données massives (*big data*) interpelle la capacité d'expansion et la performance des outils actuels d'une telle analyse.

Le but de ce travail est de développer de nouvelles procédures de calcul des concepts et des implications triadiques adaptées aux enjeux du *big data* se basant sur le paradigme de traitement parallèle et distribué. La démarche est comme suit :

1. on décompose le contexte formel triadique en autant de contextes dyadiques qu'il y a de conditions ou d'attributs selon le type d'implications désiré,
2. on génère toutes les implications triadiques de deux formes distinctes dont l'une d'elles est $A \xrightarrow{C} B$ signifiant que A implique B sous tout sous-ensemble de C, où A et B sont des groupes d'attributs et C est un ensemble de conditions,
3. on récupère un ensemble non nécessairement exhaustif de concepts triadiques à partir des résultats de l'étape 2.

Notre solution est implémentée et validée à l'aide du cadre de développement *Apache Spark*.

Abstract

Triadic concept analysis (TCA) focuses on the study of three-dimensional data represented by a triadic context $\mathbb{K} = (K_1, K_2, K_3, Y)$ where K_1 describes a set of objects, K_2 a set of properties (attributes), K_3 a set of conditions, and Y a ternary relation between these three sets. Triadic concept analysis is a conceptual clustering technique which has the particularity of generating both (i) a set of triadic concepts *(clusters)*) linked by a quasi-order relation and (ii) a set of association and implications rules between attributes under specific conditions or vice versa. Triadic concept analysis can show its usefulness in several knowledge discovery projects. Nevertheless, the advent of *big data* questions the scalability and the performance of current tools for such analysis.

The goal of this research work is to develop new procedures for computing triadic concepts and implications that are better adapted to the challenges of big data using the parallel and distributed computing paradigm. The proposed methodology is as follows :

1. The triadic formal context is split into as many dyadic contexts as there are conditions or attributes, depending on the kind of implications to compute
2. We extract all triadic implications of two distinct forms, including $A \xrightarrow{C} B$, which means that A implies B under any subset of C, where A and B are sets of attributes and C is a set of conditions,
3. We generate a non-necessarily exhaustive set of triadic concepts from the results of Step 2.

Our solution is implemented and validated using the *Apache Spark* environment.

Chapitre 1

Introduction

1.1 Introduction

Depuis une trentaine d'années, l'analyse formelle de concepts (AFC) [17] s'est établie comme un puissant outil d'analyse et d'extraction de connaissances dans les tables de données bidimensionnelles. L'AFC est souvent sollicitée dans de nombreuses et importantes applications de découverte de connaissances dans des bases de données (*Knowledge discovery in databases - KDD*) telles que la conception des ontologies, l'exploration des règles d'association ou encore l'apprentissage machine (*machine learning*) [16, 26]. Les modèles de représentation, d'exploration et traitement de connaissances basés sur l'AFC ont été intensivement conçus et utilisés dans divers projets académiques ou industriels à travers le monde. On peut ainsi énumérer plusieurs revues de littérature abordant les différentes applications de l'AFC, notamment la revue de Priss et al. [41, 42] sur les applications de l'AFC pour le traitement automatique du langage naturel, celle de Carpineto et Romano [6] puis celle de Poelmans et al [39] sur les applications de l'AFC pour la recherche d'information (*Information retrieval IR*) et la fouille de texte (*text mining*). On peut aussi citer la revue de Tilley et al. [47] sur l'utilisation de l'AFC dans les activités de génie logiciel, particulièrement pour les activités de maintenance logiciel ou les tâches d'identification des classes orientées objet. On peut continuer ainsi en citant d'autres domaines

d'utilisation tels que l'analyse des réseaux sociaux [32], l'appariement des ontologies biomédicales [54] ou l'analyse des performances des capteurs dans les environnements intelligents [12].

Cependant, l'avènement des données massives (*Big Data*) constitue un double défi pour les chercheurs. D'une part, les approches utilisées pour l'AFC devront faire face à des quantités gigantesques de données sans précédent. Les outils utilisés doivent être optimaux afin de rendre les résultats avec des performances acceptables et surtout éviter de produire de la connaissance redondante sans réelle valeur pour les utilisateurs. Malheureusement, la majorité des algorithmes développés jusqu'ici, restent inefficaces et pas suffisamment adaptés pour les données massives [23], [24].

Par ailleurs, bien que les approches dyadiques sur les données bidimensionnelles ont été satisfaisantes pour plusieurs applications, il y avait aussi bien des situations qui suggéraient l'extension de l'analyse formelle de concept par d'autres dimensions (données tridimensionnelles ou plus). L'analyse triadique de concept (ATC), une extension de l'AFC, commence ainsi à prendre de l'essor et attire de plus en plus de chercheurs. Bien entendu, le présent contexte des données massives ajoute son lot de complexité à une tâche certainement plus ardue que l'AFC.

Ce projet de recherche s'attache donc à proposer une approche pour l'analyse triadique de concepts adaptée au contexte actuel des données massives. Dans ce qui suit, nous expliquons le contexte de ce travail, l'approche préconisée ainsi que les objectifs spécifiques que nous visons à l'issue de ce projet.

1.2 Contexte

L'expression "données massives" (*Big Data*) est un concept qui recouvre globalement toutes les opportunités, les défis ainsi que les techniques mises en œuvre pour le traitement du déluge de données auquel on assiste depuis quelques années. Selon le magazine Forbes [1], la quantité de données créées à travers le monde chaque jour

1. https ://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/66ec362860ba. Page consultée en juin 2019

atteint les $2,5$ trillions ($2.5\ 10^{18}$) d'octets. Le phénomène ne fait que s'accélérer avec le développement de l'internet des objets (*Internet of things-IoT)*. Ainsi, 90% des données dans le monde aujourd'hui ont été générées seulement au cours des deux dernières années.

Le groupe *Gartner* définit les données massives (*Big Data*) comme « des ressources informationnelles (données) de **V**olume, **V**élocité et / ou **V**ariété élevés exigeant des formes novatrices et efficientes de traitement afin d'obtenir une meilleure compréhension, prise de décision et automatisation des processus» [30]. Bien qu'importante, la notion du *Big Data* va au-delà de la seule propriété du **V**olume. La **V**ariété réfère à une prolifération de types de données provenant de différentes sources comme les réseaux sociaux, les interactions machine-machine, les senseurs ou autres terminaux mobiles, créant ainsi une très grande diversité au-delà des données transactionnelles traditionnelles et rendant par conséquent la gestion beaucoup plus complexe. La **V**élocité selon *Gartner* désigne à la fois "la rapidité de production des données et la rapidité avec laquelle elles doivent être traitées pour répondre à la demande". La réalité de plusieurs entreprises est que, très rapidement, les données peuvent devenir obsolètes. Traiter efficacement les données massives «nécessite d'effectuer des analyses prenant en charge le **V**olume et la **V**ariété des données pendant qu'elles sont toujours en mouvement, pas juste après »[30]. Depuis cette définition, d'autres caractéristiques ainsi que d'autres V tels que la **V**isualisation, la **V**éracité ou la **V**aleur ont été rajoutées par la suite.

Une autre définition plus subtile désigne des collections de données qui atteignent des volumes tellement faramineux qu'elles dépassent toute capacité humaine d'analyse ou de perception. Elles dépassent même celles des outils informatiques classiques de gestion de base de données ou de l'information. Pour illustration, les problèmes informatiques séquentiels les plus courants sont des problèmes de classe P, c'est le cas par ailleurs de la majorité des problèmes liés à l'AFC quand les données d'entrée ne sont pas denses. Ces problèmes peuvent donc être résolus en un temps polynomial (*efficace*) sur toute machine de Turing déterministe. Cependant, dans le contexte des données massives, le temps polynomial devient tout simplement inacceptable. Par conséquent, les problèmes de classe P deviennent pratiquement insolubles. Certes,

ils demeurent théoriquement solubles, mais les temps de résolution deviennent excessivement longs que dans la pratique ces problèmes sont considérés insolubles. Considérant un ensemble de données de taille égale à un pétaoctet (10^{15} *Octets*), une taille tout à fait envisageable pour plusieurs géants du commerce en ligne ou des réseaux sociaux. Le simple fait de balayer cet ensemble de données, même sur le plus moderne des disques SSD, devrait prendre 166666 secondes à une vitesse de balayage de 6 Gbps [14], soit environ deux jours !

Par ailleurs, les opportunités liées au *big data* présentent des enjeux économiques et sociaux d'une grande importance. Selon un dernier rapport de l'IDC (*International Data Corporation*) [2], le marché mondial du *big data analytics* étant estimé à 166 milliards de dollars américain en 2017, devrait enregistrer une croissance annuelle soutenue de 11,7 % entre 2017-2022 pour atteindre les 260 milliards de dollars en 2022. Il n'est donc pas étonnant de voir les géants de l'industrie être les meneurs des efforts entrepris depuis quelques années pour l'exploitation efficace des données massives face à de nombreux défis. Ces derniers concernent tous les aspects de traitement comme la collecte, le stockage et principalement la valorisation de ces données. Ces efforts se poursuivent selon quatre axes principaux qui sont eux-mêmes répartis en plusieurs disciplines [48]:

Un premier axe touche à l'informatique distribuée et parallèle dont l'un des précurseurs fut l'introduction par *Google* de *MapReduce* [11] en 2004. *MapReduce* est un paradigme de programmation pour des calculs parallèles et distribués sur les données massives permettant une «possibilité d'extension» (*scalability*) optimale par l'utilisation de centaines, voire des milliers de serveurs *(nœuds)*, collectivement appelés des grappes (*clusters*). Depuis, de nombreuses implémentations ouvertes (*open source*) de ce paradigme ont vu le jour comme Hadoop (Yahoo, Fondation Apache) inspiré du système de fichiers distribués *GFS*, lui-même issu des recherches de Google *« The Google File System »* en 2003 [18], ou encore Apache Spark qui est en train de prendre le relais ces dernières années.

Un deuxième axe touche au *supercalcul* ou *calcul haute performance* (*High-performance computing - HPC*). Le terme HPC réfère à l'utilisation agrégée de ressources informa-

2. https ://www.idc.com

tiques haut de gamme (ou superordinateurs), combinées à des puissants algorithmes de traitement parallèle ou simultané pour résoudre des problèmes de calculs intensifs. Pour des raisons évidentes, une évolution se dessine, caractérisée par la convergence du *big data* et du *HPC*. Les spécialistes font référence alors à *« high performance data analytics » (HPDA)*.

Un troisième axe porte principalement sur les aspects mathématiques et algorithmiques afin de développer des techniques plus rapides qui peuvent être implémentées de manière efficace dans les environnements parallèles et distribués.

Finalement, le quatrième axe touche au développement des bases de données dites *Not only SQL*. *NoSQL* est une famille de systèmes de gestion de base de données (SGBD) qui s'écarte du paradigme classique des bases de données relationnelles pour permettre la manipulation de volumes gigantesques de données avec un stockage distribué à très large échelle facilitant du même coup la scalabilité horizontale qui reste encore problématique pour les bases de données relationnelles.

1.3 But et méthodologie

Le présent mémoire de recherche s'intéresse à l'analyse triadique de concepts. Nous avons comme but de proposer une nouvelle approche de calcul de concepts dans les contextes triadiques qui soit plus appropriée à la réalité du *big Data*. Notre démarche s'inscrit dans les récentes tendances en matière de traitement et d'analyse des données massives. Nous allons donc proposer un algorithme basé sur le paradigme *MapReduce* qui sera implémenté dans l'environnement Apache Spark, un des cadres de développement parallèle et distribué les plus utilisés pour les données massives.

1.4 Contributions

Dans cette section, nous présentons brièvement les contributions principales de ce travail. Ces contributions sont reflétées particulièrement dans certains aspects de notre approche que nous résumons dans les points suivants :

- Nous proposons une nouvelle manière de calculer des implications et des concepts dans un contexte triadique en décomposant ce dernier en autant de contextes dyadiques qu'il y a des attributs ou des conditions. Il s'agit là d'une nouveauté de ce travail car nous cherchons d'abord les implications dyadiques qui servent par la suite à la production des implications triadiques.
- Nous procédons ensuite par une sorte de rétro-ingénierie qui représente une particularité importante de ce travail et qui consiste à obtenir les concepts triadiques à partir des implications triadiques à l'inverse du cheminement habituel adopté par les autres méthodes de fouille de données.
- Nous adoptons une implémentation basée sur un calcul parallèle et distribué utilisant le paradigme *MapReduce* car nous considérons que des approches parallèles et distribuées peuvent être une alternative intéressante lors de l'analyse de contextes triadiques massifs nécessitant une puissance combinatoire souvent non disponible avec les outils traditionnels.

L'organisation du présent document est comme suit : au chapitre 2, nous rappelons les fondements théoriques de l'analyse formelle de concepts de même que l'analyse triadique de concepts. Le chapitre 3 résume les principaux algorithmes développés jusqu'ici pour le calcul de concepts dans les contextes dyadiques et triadiques. Nous passons en revue les différentes catégories d'algorithmes et nous nous arrêtons sur les derniers travaux en matière de calcul parallèle et distribué. Le chapitre 4 présente le *framework* Apache Spark puis détaille les étapes de notre approche. Le chapitre 5 expose les grandes lignes de la mise en oeuvre de notre procédure sur l'environnement Apache Spark ainsi que les résultats des simulations effectuées sur des contextes triadiques. La conclusion ainsi que les travaux futurs sont fournis dans le chapitre 6.

Chapitre 2

Analyse triadique de concepts

Ce chapitre passe en revue quelques-unes des notions les plus importantes de l'analyse formelle de concepts puis de l'analyse triadique de concepts. Ces notions seront utiles pour la suite de ce travail.

2.1 Analyse formelle de concepts

Il est généralement accepté que l'analyse formelle de concepts fut introduite par Rudolf Wille en 1982 [51, 49], bien que quelques travaux précurseurs existent auparavant, notamment les travaux des mathématiciens Barbut et Monjardet sur les treillis de Galois [1]. L'AFC concilie la notion de concept telle que développée par les approches de philosophie classique avec les mathématiques, précisément à travers la théorie des treillis fournissant ainsi une méthode mathématique pour l'extraction, la formalisation et la hiérarchisation des concepts.

Cette approche commence par la définition de la notion de contexte formel $\mathbb{K} := (G, M, I)$ où G est un ensemble d'objets, M est une collection d'attributs et I une relation binaire entre G et M. La notation (g, m) ou encore gIm indique qu'un objet g de G possède un certain attribut m de M.

Un contexte formel de taille réduite peut être simplement décrit par un tableau croisé bidimensionnel. Les entêtes des lignes représentent des objets, ceux des co-

lonnes représentent des attributs et les croix indiquent la relation binaire I entre les objets et les attributs.

Anorexie	m1 :Trop sensible	m2 :Renfermé	m3 :Confiant	m4 :Respectueux	m5 :Cordial	m6 :Difficile	m7 :Attentif	m8 :Facilement offensé	m9 :Calme	m10 :Inquiet	m11 :Bavard	m12 :Superficiel	m13 :Sensible	m14 :Ambitieux
g1 :Moi-même	×	×	×		×	×	×		×	×			×	×
g2 :Mon idéal	×		×	×	×		×		×				×	×
g3 :Père	×	×		×	×	×	×	×	×	×		×	×	×
g4 :Mère	×	×		×	×	×		×	×	×		×	×	×
g5 :Sœur	×	×		×	×	×	×		×	×			×	×
g6 :Gendre			×	×	×		×				×	×		×

Tableau 2.1: Exemple de contexte formel

L'exemple du tableau 2.1 provient d'un cas d'étude authentique sur des patients souffrant d'anorexie mentale. Cet exemple fut l'un des premiers exemples cités par Wolff pour illustrer la puissance de l'AFC comme outil de visualisation des données [44]. Le tableau 2.1 présente bien un contexte formel dont les objets sont les patients participant à cette étude tandis que les attributs sont leurs traits de personnalité.

Pour deux sous-ensembles $A \subseteq G$ et $B \subseteq M$, on définit A' comme l'ensemble des attributs communs aux éléments de A et B' comme l'ensemble des objets partageant les attributs dans B. Les opérateurs de dérivation $(.)'$ peuvent être ainsi formellement définis pour ($A \subseteq G$ et $B \subseteq M$) comme suit :

$$A' := \{m \in M |\ gIm \quad \forall g \in A\} \quad B' := \{g \in G |\ gIm \quad \forall m \in B\}$$

Exemple 2.1. Pour l'exemple, considérons le sous-ensemble d'objets *(patients)* constitué de la sœur et du gendre, soit $A = \{g5, g6\}$. L'ensemble des attributs (*les traits de*

personnalité) partagés par ces deux personnes est donné par $A' := \{m4, m5, m7, m14\}$. Ces deux personnes sont respectueuses, cordiales, attentives et ambitieuses.

Dans l'autre sens, si on prend le sous-ensemble constitué des deux attributs $m3$ *(confiant)* et $m8$ *(facilement offensé)*, nous pouvons voir que $\{m3, m8\}' = \emptyset$. Aucun patient ne possède ses deux traits de personnalité en même temps.

Les opérateurs de dérivation {(.)',(.)'} sont aussi appelés des opérateurs de formation de concepts (*concept forming operator*).

Théorème 1. *La paire (.)" constituée des opérateurs de dérivation induits par* (G, M, I) *forment une* correspondance de Galois *entre* G *et* M*. Les deux opérateurs de fermeture (ou simplement fermetures) sur* G *et* M *qui en découlent sont tous deux notés* $('')$.

Définition 2.1. (***Correspondance de Galois***) [2] On rappelle qu'une correspondance de Galois entre deux ensemble X et Y est une paire d'applications $\langle f, g \rangle$ avec $f : 2^X \to 2^Y$ et $g : 2^Y \to 2^X$ respectant les conditions suivantes pour tout $A, A_1, A_2 \subseteq X$ et $B, B_1, B_2 \subseteq Y$:

$$A_1 \subseteq A_2 \Rightarrow f(A_2) \subseteq f(A_1) \tag{2.1}$$

$$B_1 \subseteq B_2 \Rightarrow g(B_2) \subseteq g(B_1) \tag{2.2}$$

$$A \subseteq g(f(A)) \tag{2.3}$$

$$B \subseteq f(g(B)) \tag{2.4}$$

Plus intéressant, si $\langle f, g \rangle$ est une correspondance de Galois entre deux ensembles X et Y, alors les applications suivantes $C_X = f \circ g$ et $C_Y = g \circ f$ sont des fermetures sur X et Y respectivement.

Définition 2.2. Une ***fermeture*** C sur un ensemble X est une application $C : 2^X \to 2^X$ qui satisfait les trois conditions suivantes :

$$A \subseteq C(A) \tag{2.5}$$

$$A_1 \subseteq A_2 \Rightarrow C(A_1) \subseteq C(A_2) \tag{2.6}$$

$$C(A) = C(C(A)) \tag{2.7}$$

Pour une fermeture $C : 2^X \to 2^X$, les sous-ensembles A de X tels que $C(A) = A$ sont appelés **les points fixes** ou **les fermés** par C.

Par ailleurs, la notion de "***fermeture***" est étroitement liée à une autre notion, celle des systèmes fermés (*closure system*). Un système fermé sur un ensemble M est un ensemble de sous-ensembles de M qui est stable par l'intersection :

Définition 2.3. Un ***système fermé*** sur un ensemble M est un ensemble $\mathcal{C} \subseteq \mathfrak{P}(M)$ où $\mathfrak{P}(M)$ dénote l'ensemble de tous les sous-ensembles de M qui satisfait ces deux conditions :

- $M \in \mathcal{C}$
- pour tout $D \subseteq \mathcal{C}$, alors $\bigcap D \in \mathcal{C}$

Les systèmes fermés et opérateurs de fermeture peuvent découler l'un de l'autre de manière assez simple. En fait, l'ensemble des fermés (*closure*) d'un opérateur de fermeture constitue un système fermé. Inversement, si on considère un système fermé $\mathcal{C}$ sur M, on peut définir un opérateur de fermeture reliant tout sous-ensemble X de M au plus petit sous-ensemble $C \in \mathcal{C}$ contenant X, soit :

$$\bigcap_{X \subseteq D \in \mathcal{C}} D$$

Nous arrivons désormais à la notion fondamentale au cœur de l'AFC : la notion de concept formel.

2.1.1 Concept formel

Définition 2.4. Un ***concept formel*** c d'un contexte formel $\mathbb{K} = (G, M, I)$ est défini par une paire de sous-ensembles (A, B) avec $A \subseteq G$ et $B \subseteq M$ tel que $A = B'$ et $B = A'$. Les sous-ensembles A et B sont appelés respectivement l'extension et l'intention du concept c.

Si on veut une définition verbale, on peut dire que $c := (A, B)$ est un concept formel si et seulement si A contient uniquement les objets qui possèdent tous les attributs dans B, de même que B ne contient que les attributs partagés par tous les objets dans A.

Mathématiquement, $c := (A, B)$ est un concept formel si et seulement si la paire (A, B) est un point fixe de la fermeture $(.)''$ définie précédemment en théorème 1.

$\mathbb{K}$	a	b	c	d
1	×		×	×
2		×	×	×
3	×		×	×
4	×		×	×
5		×		×
6	×		×	

Tableau 2.2: Un contexte formel $\mathbb{K} := (G, M, I)$

Exemple 2.2. Pour la table 2.2 ci-dessus, le rectangle grisé représente le concept formel $\langle A, B\rangle = \langle \{1, 2, 3, 4\}, \{c, d\}\rangle$. On voit bien que $A' = \{1, 2, 3, 4\}' = \{c, d\} = B$ de même que $B' = \{c, d\}' = \{1, 2, 3, 4\} = A$.

On peut générer un concept formel à partir d'un seul objet g en prenant la paire de la forme $(\{g\}'', \{g\}')$. Ce genre de concept est appelé un concept objet. Par analogie, on peut générer un concept formel à partir d'un seul attribut m en prenant la paire $(\{m\}', \{m\}'')$. On parle dans ce cas de concept attribut. Si le contexte $\mathbb{K} := (G, M, I)$

est décrit par un tableau croisé et en adoptant un arrangement adéquat des lignes et des colonnes, on peut voir que les concepts formels représentent des rectangles pleins maximaux.

2.1.2 Treillis de concepts

Définition 2.5. La collection de tous les concepts résultant d'un contexte formel $\mathbb{K} := (G, M, I)$ constitue le ***treillis de concepts*** de $\mathbb{K}$ noté $\underline{\mathfrak{B}}(G, M, I)$. Cet ensemble est structuré par la relation d'ordre partiel suivante :

$$(A_1, B_1) \leq (A_2, B_2) \Leftrightarrow A_1 \subseteq A_2 (\Leftrightarrow B_1 \subseteq B_2) \tag{2.8}$$

(A_1, B_1) est alors dit un sous-concept ou prédécesseur de (A_2, B_2) alors que ce dernier est appelé un successeur de (A_1, B_1). Lorsqu'une relation d'ordre existe entre les concepts (A_1, B_1) et (A_2, B_2), on dit alors que ce sont des concepts comparables. Sinon, ils ne le sont pas. (A_1, B_1) est dit prédécesseur immédiat de (A_2, B_2) si en plus de l'inégalité précédente, il n'existe aucun concept entre (A_1, B_1) et (A_2, B_2). Autrement dit, s'il existe un concept (C, D) tel que $(A_1, B_1) \leq (C, D) \leq (A_2, B_2)$ alors $(A_1, B_1) = (C, D)$.

Remarque : $(A_1, B_1) \leq (A_2, B_2)$ signifie que le concept (A_1, B_1) est plus spécifique que le concept (A_2, B_2).

Nous pouvons désormais introduire le théorème fondamental de l'analyse formelle de concepts présenté par Rudolf Wille en 1982 [49] intitulé le théorème principal des treillis de concepts (*the basic theorem of concept lattices*).

Théorème 2. *(**Théorème principal des treillis de concepts**) Soit $\mathbb{K} := (G, M, I)$ un contexte formel. Alors le treillis de concepts de $\mathbb{K}$, noté $\underline{\mathfrak{B}}(G, M, I)$, est un treillis complet qui admet un **infimum** et un **supremum** qu'on peut décrire par les formules*

suivantes :

$$\bigwedge_{i\in J}\langle A_i, B_i\rangle = \langle \bigcap_{i\in J} A_i, (\bigcup_{i\in J} B_i)''\rangle \tag{2.9}$$

$$\bigvee_{i\in J}\langle A_i, B_i\rangle = \langle (\bigcup_{i\in J} A_i)'', \bigcap_{i\in J} B_i\rangle \tag{2.10}$$

Un ensemble partiellement ordonné $\underline{\mathfrak{B}}(G, M, I)$ est dit un treillis complet si toute partie (*sous-ensemble de concepts*) S dans $\underline{\mathfrak{B}}$ admet à la fois une borne inférieure (*infimum*) et une borne supérieure (*supremum*) dans $\underline{\mathfrak{B}}$.

Également ici, les notions de treillis complet et de système fermé sont étroitement liées. De toute évidence, un système fermé $\mathcal{C}$ sur M peut être considéré comme un treillis complet. *L'infimum* de toute sous-famille $\mathcal{D} \subseteq \mathcal{C}$ est simplement l'intersection des ces éléments $\bigcap \mathcal{D}$ alors que le *supremum* est simplement la fermeture de $\bigcup \mathcal{D}$. Inversement, on peut trouver pour tout treillis complet $\mathfrak{L}$ un système fermé isomorphe à $\mathfrak{L}$.

Plus important, l'ensemble des intentions de tous les concepts formels d'un contexte $\mathbb{K} := (G, M, I)$ est stable par intersection. Cela signifie que l'intersection de deux intentions est aussi une intention d'un concept formel. Cet ensemble constitue donc un système fermé sur M. Par analogie, l'ensemble de toutes les extensions est aussi un système fermé sur G.

La figure 2.1 présente *le diagramme de Hasse* correspondant au contexte formel 2.2. Dans un tel diagramme, les nœuds représentent les concepts alors que les arcs, représentent la relation d'ordre partiel. Les concepts sont placés plus bas que leurs successeurs immédiats.

2.2 Règles d'association et implications dyadiques

La génération de règles d'association est un autre aspect important de l'analyse formelle de concepts. Les règles d'association sont des assertions de type *(si..., alors...)* qui permettent de mettre en évidence des relations entre des données sans aucun lien apparent à priori. Ces données peuvent provenir de bases de données re-

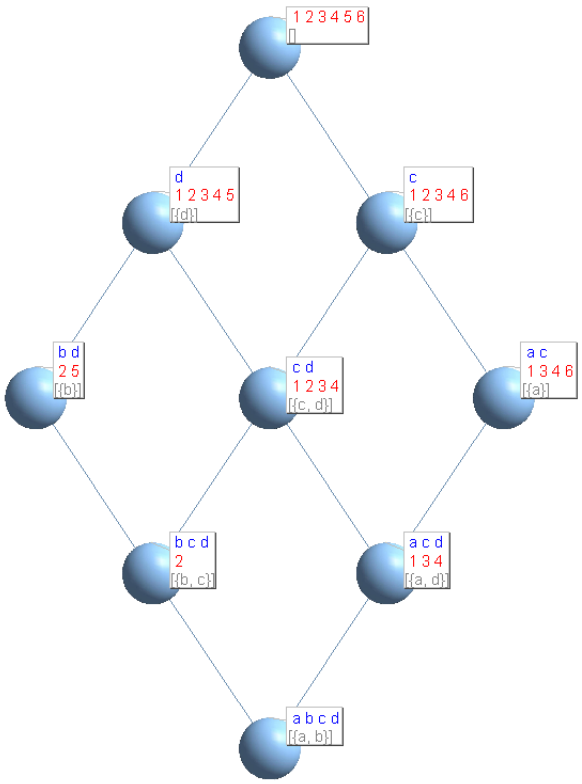

FIGURE 2.1: Treillis de concepts du contexte formel tableau 2.2

lationnelles tout aussi bien de n'importe quel autre type de structures de données. Un exemple de règle d'association serait "*Si un client achète une douzaine d'œufs, il est susceptible à 80% d'acheter également du lait.*"

Définition 2.6. Soit un contexte formel dyadique $\mathbb{K} := (G, M, I)$, une règle d'association est une expression de type :

$$r : B \to C \quad s, c$$

B et C sont des sous-ensembles disjoints de l'ensemble des attributs M. Le sous-ensemble B est appelé la prémisse de la règle d'association r alors que le sous-ensemble C désigne sa conclusion. Les paramètres s et c sont appelés respectivement *le support* et *la confiance* de la règle d'association r et sont régulièrement exprimés en pourcentage. La valeur de s représente la proportion des objets ayant simultanément les attributs B et C. On l'obtient par la formule suivante :

$$s = supp(r) = \frac{|B' \cap C'|}{|G|} \tag{2.11}$$

La valeur de c d'autre part, représente la probabilité conditionnelle *Prob(C/B)* qui indique la probabilité qu'un objet puisse posséder les attributs dans C lorsqu'il possède tous les attributs dans B.

$$c = conf(r) = \frac{|B' \cap C'|}{|B'|} \tag{2.12}$$

Les implications dyadiques en revanche sont un simple cas particulier des règles d'association. Leur définition en découle de celles-ci comme suit :

Définition 2.7. Soit un contexte formel dyadique $\mathbb{K} := (G, M, I)$, une ***implication*** entre attributs $B \to C$ est une règle d'association dont la confiance est égale à 100%.

2.2.1 Les bases d'implications

Le nombre d'implications qui sont vraies pour une situation donnée peut être excessivement grand. À titre d'exemple, si un contexte $\mathbb{K} := (G, M, I)$ a pour seul fermé l'ensemble M, alors toute implication est vraie dans ce cas. Si en plus, M est constitué de n éléments, on peut alors énumérer au moins 2^{2^n} implications pour cet exemple, ce qui peut apparaître tout à fait absurde sachant que toutes ces implications peuvent être tout simplement inférées de la seule implication $(\emptyset \to M)$. À ce propos, les règles d'inférence pour les implications dyadiques sont les mêmes que les axiomes d'inférence d'Armstrong pour les dépendances fonctionnelles.

Ainsi, dans certains cas, l'ensemble des implications peut être très redondant. En conséquence et tout naturellement, plusieurs travaux se sont intéressés à développer des approches pour une génération et représentation concise des implications à travers des listes réduites $\mathcal{L}$ (*bases d'implications*), à partir desquelles toute autre implication peut être inférée. Une base $\mathcal{L}$ doit respecter les trois conditions suivantes [16] :

- **Correcte**, c-à-d toutes les implications dans la base sont vraies dans $\mathbb{K} := (G, M, I)$,
- **Complète**, c-à-d toute implication vraie dans $\mathbb{K}$ peut être inférée de $\mathcal{L}$,
- **Non redondante**, c-à-d aucune implication de $\mathcal{L}$ ne peut être inférée des autres implications dans $\mathcal{L}$.

Parmi ces travaux, on peut citer les travaux de Guigues et Duquenne sur les familles minimales des implications [19], les travaux de Pasquier et al sur les bases génériques [36] ou encore les travaux de Luxenburger sur les bases d'implications partielles [29]. Dans ce qui suit, on fait un bref rappel de quelques notions fondamentales pour ces travaux, notamment les notions de pseudo-intention et de générateurs.

Définition 2.8. (**Pseudo-intention**) Un ensemble fini $P \subseteq M$ est une *pseudo-intention* de (G, M, I) si et seulement si

- $P \neq P''$ (autrement P n'est pas une intention de concept) et,
- $Q \subseteq P, Q \neq P$ est une pseudo-intention alors $Q'' \subseteq P$.

Théorème 3. *L'ensemble des implications* $\mathcal{L} := \{P \rightarrow P''\backslash P \quad | P$ *est une pseudo-intention}* *est non redondant et complet. On appelle cet ensemble la base de Guigues-Duquenne.*

Exemple 2.3. [19] Considérant le contexte de l'exemple du tableau 2.2. L'ensemble vide $\emptyset$ étant un fermé ne peut pas être une pseudo-intention, non plus $\{c\}$ et $\{d\}$ pour la même raison ($\{c\}'' = \{c\}$ et $\{d\}'' = \{d\}$). Cependant $\{a\}$ et $\{b\}$ sont bien des pseudo-intentions puisqu'il ne s'agit pas de fermés ($\{a\}'' = \{a, c\}$ et $\{b\}'' = \{b, d\}$). La deuxième condition est bien évidement respectée pour le seul sous-ensemble de $\{a\}$ ou $\{b\}$, c-à-d l'ensemble vide $\emptyset$. En revanche, cette condition n'est pas respectée par exemple pour $\{a, b\}$ qui bien qu'il contienne les pseudo-intentions $\{a\}$ et $\{b\}$, il ne contient pas leurs fermés $\{a, c\}$ ou $\{b, d\}$.

Finalement, on peut démontrer que $\{a\}$ et $\{b\}$ sont les seules pseudo-intentions pour le contexte de l'exemple, ainsi la base de Guigues-Duquenne, dite aussi *stem base* est composée des deux implications suivantes :

$$\{a\} \rightarrow \{a\}''\backslash\{a\} \Leftrightarrow \{a\} \rightarrow \{c\}$$
$$\{b\} \rightarrow \{b\}''\backslash\{b\} \Leftrightarrow \{b\} \rightarrow \{d\}$$

Examinons à présent la notion de **générateur** : Dans un contexte $\mathbb{K} := (G, M, I)$, un itemset (*sous-ensemble d'attributs*) Z est appelé un *générateur* de l'intention B d'un concept $c = (A, B)$ si $Z'' = B$. De plus, Z est appelé un *générateur minimal* de l'intention B si pour tout $Y \subsetneq Z$, alors $Y'' \subsetneq Z'' = B$. Pour une même intention, on peut avoir plus d'un générateur minimal.

Exemple 2.4.

Concept	extension	intention	générateur minimal
1	$\{1,2,3,4,5\}$	$\{d\}$	$\{d\}$
2	$\{1,2,3,4,6\}$	$\{c\}$	$\{c\}$
3	$\{1,2,3,4\}$	$\{c,d\}$	$\{c,d\}$
4	$\{1,3,4,6\}$	$\{a,c\}$	$\{a\}$
5	$\{2,5\}$	$\{b,d\}$	$\{b\}$
6	$\{2\}$	$\{b,c,d\}$	$\{b,c\}$
7	$\{1,3,4\}$	$\{a,c,d\}$	$\{a,d\}$
8	$\emptyset$	$\{a,b,c,d\}$	$\{a,b\}$

Tableau 2.3: Les concepts formel, extensions et intentions avec leur générateurs minimaux pour le contexte formel du tableau 2.2

Pasquier et al. [37] proposent une base générique pour une représentation relativement concise et non redondante des implications puis des règles d'association.

Définition 2.9. (**Base générique pour les implications**)[34]

Soit c un concept formel et $G_c = \{g_1, \ldots, g_i, \ldots g_n\}$ l'ensemble des générateurs minimaux de son intention $Int(c)$. Une implication r ou règle exacte a la forme

$g_i \rightarrow Int(c) \backslash g_i$. On a aussi, $supp(r) = support(c)$ et $conf(r) = 100\%$. $support(c)$ est la proportion d'objets contenus dans l'extension de c.

Soit $\mathfrak{B}$ l'ensemble des concepts formels d'un contexte $\mathbb{K} := (G, M, I)$. La base générique pour les implications *The Min-max Exact Basis* est donnée par :

$$MinMaxExact = \{g \rightarrow (Int(c) \backslash g) \quad | \quad c \in \mathfrak{B} \wedge g \in G_c \wedge g \neq Int(c)\}$$

où G_c est l'ensemble des générateurs minimaux de $Int(c)$.

La condition $g \neq Int(c)$ est nécessaire, autrement nous aurons des implication de la forme $g \rightarrow \emptyset$ qui n'appartiennent pas à l'ensemble des règles d'association valides (*règles non informatives*).

Exemple 2.5. La base générique pour les implications extraites du contexte 2.2 pour un support minimal de 0 % :

Générateur	Fermeture	Implication	Support
$\{d\}$	$\{d\}$		
$\{c\}$	$\{c\}$		
$\{c, d\}$	$\{c, d\}$		
$\{a\}$	$\{a, c\}$	$\{a\} \rightarrow \{c\}$	66%
$\{b\}$	$\{b, d\}$	$\{b\} \rightarrow \{d\}$	33%
$\{b, c\}$	$\{b, c, d\}$	$\{b, c\} \rightarrow \{d\}$	16%
$\{a, d\}$	$\{a, c, d\}$	$\{a, d\} \rightarrow \{c\}$	50%
$\{a, b\}$	$\{a, b, c, d\}$	$\{a, b\} \rightarrow \{c, d\}$	0%

Tableau 2.4: Base générique extraite du contexte du tableau 2.2 pour un support minimal de 0 %

Définition 2.10. (**Bases informatives pour les règles association**) [34]

Soit $\mathfrak{B}$ l'ensemble des concepts formels d'un contexte $\mathbb{K} := (G, M, I)$ et $\mathcal{G}$ l'ensemble des générateurs minimaux de leurs intentions. La base **générique des règles**

approximatives (*the min-max approximate basis*) est donnée par :

$$MinMaxApprox = \{g \rightarrow (Int(c) \backslash g) \quad | \quad c \in \mathfrak{B} \wedge g \in \mathcal{G} \wedge g'' \subset Int(c)\}$$

Exemple 2.6. La base informative pour les règles d'association extraites du contexte du tableau 2.2 pour un support minimal de 50 % est donnée ci-après.

Générateur t	Fermeture	Sur-ensemble fermé	règle d'association	confiance
$\{d\}$	$\{d\}$	$\{c, d\}$	$\{d\} \rightarrow \{c\}$	80%
$\{c\}$	$\{c\}$	$\{c, d\}$	$\{c\} \rightarrow \{c\}$	80%
$\{c\}$	$\{c\}$	$\{a, c\}$	$\{c\} \rightarrow \{a\}$	80%
$\{c, d\}$	$\{c, d\}$	$\{a, c, d\}$	$\{c, d\} \rightarrow \{a\}$	75%
$\{a\}$	$\{a, c\}$	$\{a, c, d\}$	$\{a\} \rightarrow \{c, d\}$	75%
$\{b\}$	$\{b, d\}$	$\{b, c, d\}$	$\{b\} \rightarrow \{c, d\}$	50%

Tableau 2.5: Base informative extraite du contexte du tableau 2.2 pour un *minsupport* =50%

2.3 Analyse triadique de concepts

L'analyse triadique de concepts (ATC) a été introduite en 1995 par Fritz Lehmann et Rudolf Wille [28] comme une extension de l'analyse formelle de concepts pour le traitement des données tridimensionnelles qu'on peut représenter par un groupe de trois ensembles G (objets), M (attributs) et B (conditions) liés par une relation ternaire $Y \subseteq G \times M \times B$. Le quadruplé (G, M, B, Y) représente ainsi ce qu'on appelle un contexte formel triadique et le triplet $(g, m, b) \in Y$ peut être interprété comme suit : l'objet g possède l'attribut m sous la condition b.

Les conditions dans B doivent être interprétées au sens large. Elles peuvent en effet couvrir plusieurs aspects tels que les relations, les conditions, les médiations, les modalités, le sens, les raisons, les objectifs, ou toutes autres caractéristiques qui décrivent la connexion entre les objets et attributs.

Pour les contextes triadiques, il est plus fréquent d'adopter la notationK_1, K_2, K_3 au lieu de G, M, B. On adopte alors cette notation à partir de ce point pour le reste de ce document.

À titre d'exemple, le tableau 2.6 extrait de [33] représente des données tridimensionnelles (Clients, Fournisseurs et Produits) provenant d'un groupe de clients K_1 notés de 1 à 5 qui achètent auprès de cinq fournisseurs constituant le groupe K_2 : **P**ierre, **N**elson, **R**ichard, **K**evin et **S**imon. Finalement, les produits forment le groupe K_3 composé de **a**ccssoires, livres *(**b**ooks)*, ordinateurs *(**c**omputers)* et caméras *(**d**igital cameras)*.

$\mathbb{K}$	P				N				R				K				S			
	a	b	c	d	a	b	c	d	a	b	c	d	a	b	c	d	a	b	c	d
1	×	×		×	×	×		×	×		×		×	×			×			
2	×			×		×	×	×	×	×		×	×			×				×
3	×	×		×				×	×	×			×	×			×			
4	×	×		×		×		×	×	×			×	×						×
5	×			×	×			×	×	×		×	×	×	×		×			

Tableau 2.6: Contexte triadique *(Clients-Fournisseurs-Produits)* avec l'exemple de concept triadique (345, RK , ab) [1]

2.3.1 Opérateurs de dérivation

Comme pour les contextes dyadiques, la notion de *opérateurs de dérivation* est utile pour la construction des concepts triadiques. Ce pendant, dans le cas de l'ATC, on distingue deux types d'opérateurs de dérivation :

Définition 2.11. (**Les opérateurs de dérivation-**(i)) Soit $\mathbb{K} = (K_1, K_2, K_3, Y)$ un contexte triadique et $\{i, j, k\} = \{1, 2, 3\}$ tel que $j < k$. Pour $X_i \subseteq K_i$ et $Z =$

1. ici, nous utilisons la notation 345 , RK et ab pour représenter les sous-ensembles $\{3, 4, 5\}, \{R, K\}$ et $\{a, b\}$ respectivement

$(X_j, X_k) \subseteq K_j \times K_k$, on définit les opérateurs de dérivation-(i) comme suit :

$$X_i \to X_i^{(i)} = \{(a_j, a_k) \in K_j \times K_k \quad | \quad (a_i, a_j, a_k) \in Y \quad \forall a_i \in X_i\}$$

$$Z \to Z^{(i)} = \{a_i \in K_i \quad | \quad (a_i, a_j, a_k) \in Y \quad \forall (a_j, a_k) \in Z\}$$

Cette définition permet de retrouver exactement les opérateurs de dérivation des contextes dyadiques définis par :

$$\mathbb{K}^{(1)} := (K_1, K_2 \times K_3, Y^{(1)})$$
$$\mathbb{K}^{(2)} := (K_2, K_1 \times K_3, Y^{(2)})$$
$$\mathbb{K}^{(3)} := (K_3, K_1 \times K_2, Y^{(3)})$$

avec $(g, (m, b)) \in Y^{(1)} \Leftrightarrow (m, (g, b)) \in Y^{(2)} \Leftrightarrow (b, (g, m)) \in Y^{(3)} \Leftrightarrow (g, m, b) \in Y$.

Ainsi, à titre d'exemple, la dérivation-*(1)* dans le contexte triadique $\mathbb{K} = (K_1, K_2, K_3, Y)$ du tableau 2.6 est équivalente à la dérivation dans le contexte dyadique $\mathbb{K}^1 := (K_1, K_2 \times K_3, Y^{(1)})$ suivant :

$\mathbb{K}^{(1)}$	$P-a$	$P-b$	$P-c$	$P-d$	$N-a$	$N-b$	$N-c$	$N-d$	$R-a$	$R-b$	$R-c$	$R-d$	$K-a$	$K-b$	$K-c$	$K-d$	$S-a$	$S-b$	$S-c$	$S-d$
1	×	×		×	×	×		×	×		×		×	×			×			
2	×			×		×	×	×	×	×		×	×			×				×
3	×	×		×				×	×	×			×	×			×			
4	×	×		×		×		×	×	×			×	×						×
5	×			×	×			×	×	×		×	×	×	×		×			

Tableau 2.7: Contexte dyadique $\mathbb{K}^{(1)} := (K_1, K_2 \times K_3, Y^{(1)})$ extrait de $\mathbb{K}$

En raison de la complexité des contextes triadiques, nous avons besoin de définir davantage d'opérateurs de dérivation pour construire les concepts triadiques. Les opérateurs de dérivation internes notés -(i, j, X_k) sont définis ainsi :

Définition 2.12. (**Les opérateurs de dérivation -**(i, j, X_k)) Soit $\mathbb{K} = (K_1, K_2, K_3, Y)$ un contexte triadique et $\{i, j, k\} = \{1, 2, 3\}$ tel que $j < k$. Pour $X_i \subseteq K_i, X_j \subseteq K_j$ et $X_k \subseteq K_k$, les opérateurs de dérivation -(i, j, X_k) sont définis par :

$$X_i \to X_i^{(i,j,X_k)} = \{a_j \in K_j \quad | \quad (a_i, a_j, a_k) \in Y \quad \forall (a_i, a_k) \in X_i \times X_k\}$$
$$X_j \to X_j^{(i,j,X_k)} = \{a_i \in K_i \quad | \quad (a_i, a_j, a_k) \in Y \quad \forall (a_j, a_k) \in X_j \times X_k\}$$

Dans ce cas aussi, les opérateurs de dérivation -(i, j, X_k) sont équivalents aux opérateurs de dérivation sur les contextes dyadiques $\mathbb{K}_{X_k}^{(i.j)}$ définis par :

$$\mathbb{K}_{X_k}^{i,j} := (K_i, K_j, Y_{X_k}^{i,j})$$

avec $(a_i, a_j) \in Y_{X_k}^{i,j}$ si et seulement si $(a_i, a_j, a_k) \in Y$ pour tout $a_k \in X_k$. Dans ce cas, la relation $(a_i, a_j) \in Y_{X_k}^{i,j}$ peut être interprétée comme l'objet a_i possède l'attribut a_j sous toute les conditions $a_k \in X_k$.

En prenant le contexte triadique du tableau 2.6 et en considérant $i = 1$, $j = 2$ et $k = 3$. Pour un sous-ensemble de conditions $A_3 = \{a, b\}$, on obtient le contexte dyadique $\mathbb{K}_{A_3}^{1,2} := (K_1, K_2, Y_{A_3}^{1,2})$ suivant :

	P	N	R	K	S
1	×	×		×	
2			×		
3	×		×	×	
4	×		×	×	
5			×	×	

Tableau 2.8: Contexte dyadique $\mathbb{K}_{A_3}^{1,2} := (K_1, K_2, Y_{A_3}^{1,2})$ avec $A_3 = \{a, b\}$

Nous arrivons maintenant à la définition de concept triadique introduite par Lehmann et Wille [28] :

2.3.2 Concept triadique

Définition 2.13. (**Concept triadique**) Un concept triadique (appelé aussi un triset fermé) d'un contexte triadique $\mathbb{K} = (K_1, K_2, K_3, Y)$ est un triplet (A_1, A_2, A_3) tel que $A_i \in K_i$ pour $i = 1, 2, 3$ et $A_i = (A_j \times A_k)^{(i)}$ pour $\{i, j, k\} = \{1, 2, 3\}$ avec $j < k$. Les sous-ensembles A_1, A_2 et A_3 sont appelés respectivement l'extension, l'intention et le mode (modus) du concept triadique (A_1, A_2, A_3).

Une autre définition fournie par Wille [50] permet d'avoir une illustration plus visuelle des concepts triadiques :

Théorème 4. *les concepts triadiques d'un contexte triadique* $\mathbb{K} = (K_1, K_2, K_3, Y)$ *sont exactement les triplets* (A_1, A_2, A_3) *maximaux avec* $A_1 \in K_1$, $A_2 \in K_2$, $A_3 \in K_3$ *et* $A_1 \times A_2 \times A_3 \subseteq Y$.

En d'autres termes, aucun des trois sous-ensembles A_i ne peut être augmenté sans violer la condition $(A_1 \times A_2 \times A_3) \subseteq Y$. Graphiquement, Les triplets (A_1, A_2, A_3) représentent aussi les cubes maximaux pleins.

Si pour les contextes dyadiques, la génération d'un concept arbitraire ne nécessite qu'un seul sous-ensemble d'objets $X \subseteq G$ ou alternativement un sous-ensemble d'attributs $Y \subseteq M$ *(en formant les paires* (X'', X') *ou* (Y', Y'')*)*, la génération des concepts triadiques en revanche requiert au minimum deux sous-ensembles de $\mathbb{K}_i$ et $\mathbb{K}_j, i \neq j$ sur lesquels on applique les opérateurs de dérivation discutés au début de la section.

2.4 Implications et règles d'association triadiques

2.4.1 Implications triadiques

À l'instar de l'AFC, les travaux sur l'analyse triadique de concepts se sont naturellement élargis vers la génération des règles d'association et implications. À ce propos, Biedermann reste à notre connaissance le premier à avoir proposé une définition des implications triadiques [3, 4] :

Définition 2.14. Soit $\mathbb{K} := (K_1, K_2, K_3, Y)$ un contexte triadique, A et D deux sous-ensembles d'attributs de K_2 et $C \subseteq K_3$ un sous-ensemble de conditions. Une implication triadique de la forme $(A \rightarrow D)_C$ est vraie dans $\mathbb{K}$ si pour tout objet $x \in K_1$, la condition suivante est satisfaite :

$$\{x\} \times A \times C \subseteq Y \Rightarrow \{x\} \times D \times C \subseteq Y$$

Autrement dit, chaque fois que l'objet x possède tous les attributs dans A sous toutes les conditions dans C, alors il possède aussi tous les attributs dans D sous les mêmes conditions dans C.

Exemple 2.7. Dans le cas du contexte triadique du tableau 2.6, on peut voir que l'implication $(N \rightarrow P)_{abd}$ est vraie puisque à chaque fois qu'un client achète auprès du fournisseur N l'ensemble des produits $\{a, b, d\}$, alors il fait de même après du fournisseur P.

Sur la base des travaux de Biedermann, Ganter et Obiedkov, proposent trois nouveaux types d'implications [15] :

- Les implications de type **Attributs** $\times$ **Conditions** $A \times CIs$,
- Les implications **entre attributs sous conditions** notées **CAI** pour *(conditional attribute implications)*,
- Les implications **entre conditions pour attributs** notées **ACI** pour *(attributional condition implications)*.

Définition 2.15. Une implication de type **Attributs** $\times$ **Conditions** prend la forme de $A \rightarrow D$ avec A, D des sous-ensembles de $K_2 \times K_3$. Ces implications sont en fait les mêmes implications dyadiques qu'on peut extraire à partir du contexte dyadique $\mathbb{K}^{(1)} := (K_1, K_2 \times K_3, Y^{(1)})$.

Exemple 2.8. Prenons le contexte du tableau 2.6. Une implication de type **Attributs** $\times$ **Conditions** est simplement une implication du contexte dyadique $\mathbb{K}^{(1)} := (K_1, K_2 \times K_3, Y^{(1)})$ qu'on peut extraire avec un outil de FCA comme *Lattice Miner* [31]. $K_2 \times K_3$ dans ce cas représente des paires *(fournisseurs , produits)*. Une implication comme $K - a \rightarrow N - d, P - a, P - d, R - a$ signifie que tous les clients

ayant acheté le produit **a**ccessoire auprès du fournisseur Kevin ont également acheté le produit caméras *(**d**igital camera)* auprès des fournisseurs Nelson et Pierre et le produit **a**ccessoire auprès des fournisseurs Pierre et Richard.

Définition 2.16. Une implication **CAI** (**entre attributs sous conditions**) s'écrit sous la forme de $A \xrightarrow{C} D$ avec A, D deux sous-ensembles d'attributs dans K_2, et C un sous-ensemble de conditions dans K_3.

Une telle implication est vraie dans un contexte triadique $\mathbb{K} := (K_1, K_2, K_3, Y)$, si pour toute condition $c \in C$, il existe un objet $\{x\}$ qui a tous les attributs dans A, alors il a forcément tous les attributs dans D. D'une autre manière, on peut dire A implique D sous toutes les conditions dans C et particulièrement, pour tout sous-ensemble de conditions de C. Ganter a démontré la relation entre ce type d'implication et les implications triadiques définies par Biedermann comme suit :

$$A \xrightarrow{C} D \Leftrightarrow (A \to D)_{C1} \quad \forall C_1 \subseteq C$$

Exemple 2.9. L'implication **CAI** suivante $R \xrightarrow{ad} P$ est vraie et signifie qu'à chaque fois que Richard fournit un des produits accessoires ou caméras *(digital cameras)* ou les deux, alors Pierre fait autant.

Cependant, alors que l'implication $(N \to P)_{abd}$ est vraie à la Biedermann, l'implication **CAI** $N \xrightarrow{abd} P$ ne l'est pas. En effet $(N \to P)_C$ n'est pas vraie pour tous $C \subseteq \{a, b, d\}$, et en particulier pour $C = \{b\}$ et $C = \{b, d\}$.

Définition 2.17. Une implication **ACI** (**entre conditions pour attributs**) est de la forme $A \xrightarrow{C} D$ avec A et D sont des sous-ensembles de conditions dans K_3, et C est un sous-ensemble d'attributs dans K_2.

Exemple 2.10. L'implication $b \xrightarrow{PN} d$ est vraie et signifie que chaque fois que les livres *(**b**ooks)* sont fournis par Pierre et Nelson et par chacun d'eux, alors les caméras *(**d**igital cameras)* sont aussi fournies par ces deux fournisseurs.

En raison de l'aspect tri-symétrique des contextes triadiques, on peut arbitrairement échanger les rôles des objets, attributs et conditions et obtenir ainsi d'autres

types d'implications telles que les implications entre objets sous conditions, ou les implications entre attributs pour objets. Seulement jusqu'à présent, peu de travaux se sont intéressés à ce type d'implications, nous nous concentrons dans ce mémoire sur ces quatre types d'implications mentionnées plus haut et en particulier les deux dernières.

2.4.2 Générateurs triadiques et implications

Nous abordons la notion de générateurs triadiques. Les travaux de Missaoui et Leonard [33] ont utilisé cette notion pour la génération des règles d'association, des implications et des concepts triadiques comme nous allons voir un peu plus en détail par la suite : Introduisons d'abord l'expression suivante basée sur les notions d'opérateurs de dérivation et concepts triadiques : $(U_2, U_3)^{1^1} := \{(A_2, A_3) | \quad U_i \subseteq A_i \subseteq K_i$ et $((U_2, U_3)^1, A_2, A_3)$.

Définition 2.18. Soit $\mathbb{K} = (K_1, K_2, K_3, Y)$ un contexte triadique et (U_2, U_3) un couple de $K_2 \times K_3$. La paire (U_2, U_3) est un t-générateur *(générateur triadique)* d'un couple (A_2, A_3) , lui-même constituant une partie d'un concept triadique (A_1, A_2, A_3) si et seulement si $(A_2, A_3) \in ((U_2, U_3)^1)^1$.

Exemple 2.11. Reprenant le tableau 2.6 et la figure 2.2. On peut voir que $(NK, b)^1 = \{1, 4\}$ et $\{1, 4\}^1 = \{(PNK, b)(PN, bd)\}$. Ainsi, $(NK, b)^{1^1} = \{(PNK, b), (PN, bd)\}$ et par conséquent la paire (NK, b) est un t-générateur pour les paires (PNK, b) et (PN, bd) associées à l'extension $\{1, 4\}$.

Pour un contexte triadique $\mathbb{K} := (K_1, K_2, K_3, Y)$, les concepts triadiques et t-générateurs peuvent être obtenus des concepts dyadiques générés à partir du contexte aplati $\mathbb{K}^{(1)} := (K_1, K_2 \times K_3, Y^{(1)})$. Cette approche se base sur les deux propositions suivantes [33] :

Proposition 2.1. Soit (A_1, B) un concept dyadique du contexte dyadique $\mathbb{K}^{(1)} := (K_1, K_2 \times K_3, Y^{(1)})$. Alors, (K_2, K_3, B) est un contexte dyadique.

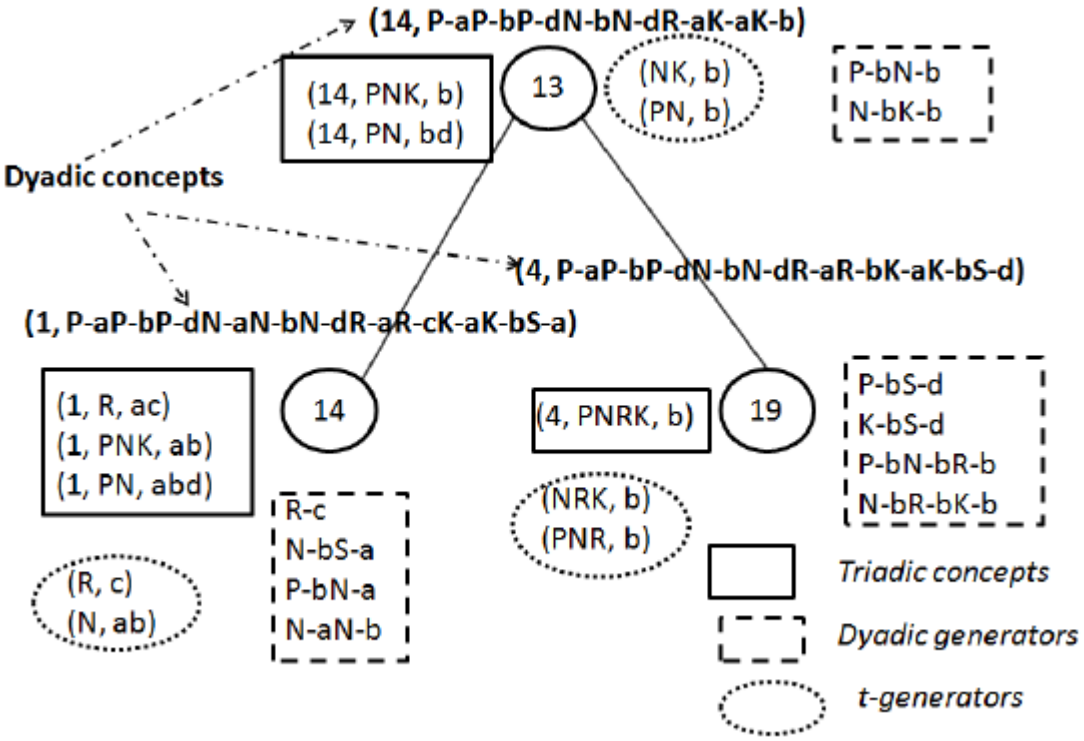

FIGURE 2.2: Concepts triadiques et dyadiques du contexte 2.6

Si on définit $B_2 = \pi(B)$ et $B_3 = \pi(B)$ les projections de B sur K_2 et K_3 respectivement. Le triplet (A_1, A_2, A_3) avec $A_1 \subseteq K_1, A_2 \subseteq B_2$ et $A_3 \subseteq B_3$ est un concept triadique de $\mathbb{K}$ si et seulement si les conditions suivantes sont vraies :

i. (A_2, A_3) est un concept dyadique du sous-contexte (K_2, K_3, B),

ii. $(A_2, A_3)^1 = A_1$.

Proposition 2.2. Soit g un générateur (dyadique) dans $\mathbb{K}^{(1)} := (K_1, K_2 \times K_3, Y^{(1)})$ (et donc un sous-ensemble de $K_2 \times K_3$). Soit $U_2 := \pi_2(g)$ et $U_3 := \pi_3(g)$, alors (U_2, U_3) est un t-générateur si et seulement si $|U_2| \times |U_3| = |g|$.

Exemple 2.12. Soit le contexte dyadique obtenu par l'aplatissement du contexte présenté par le tableau 2.6. La figure 2.3 représente les concepts dyadiques obtenus de ce contexte.

À titre d'exemple, à partir du concept dyadique $(\{1,3,4\}, \{K-a, K-b, N-d, P-a, P-d, R-a\})$ on peut générer quatre triplets. Cependant, seulement les triplets $(134, PK, ab)$ et $(134, P, abd)$ constituent des concepts triadiques. Les deux autres triplets $(134, PKR, a)$ et $(134, PN, d)$ ne sont pas des cubes maximaux puisque dans ces deux cas $(A_2, A_3)^1 = \{1,2,3,4,5\} \neq \{1,3,4\}$.

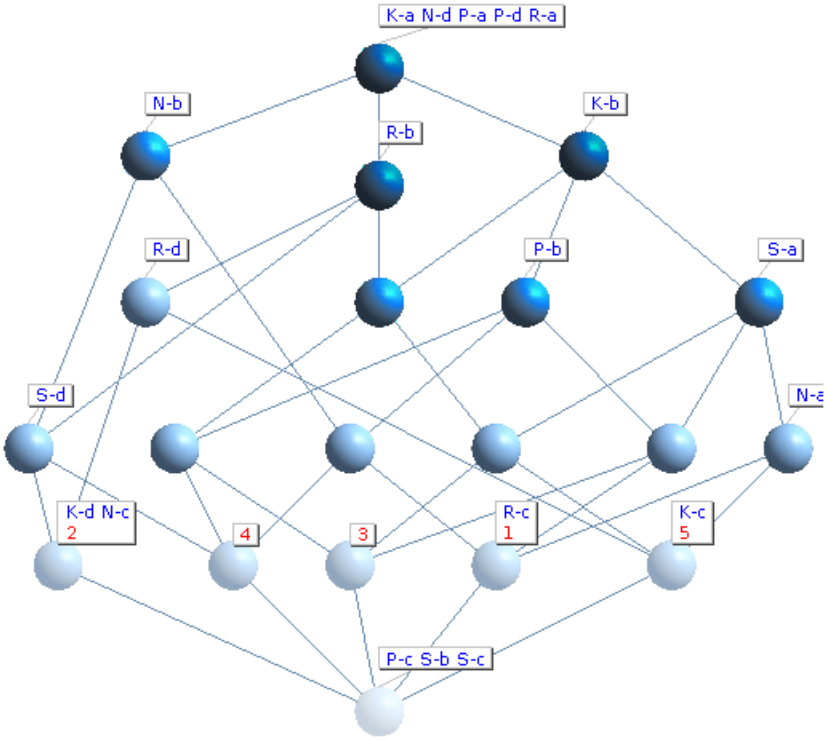

FIGURE 2.3: Concepts dyadiques du contexte $\mathbb{K}_1$ obtenu de 2.6

La notion de générateurs triadiques est bien utile pour générer des implications triadiques comme suit :

Proposition 2.3. Soit (U_2, U_3) un t-générateur du couple (A_2, A_3) dans le concept triadique (A_1, A_2, A_3) avec $A_1 \subseteq K_1$. Alors, l'implication suivante entre attributs

sous conditions dans le sens de Biedermann notée **BCAI** *(Biedermann conditional attribute implication)* : $(U_2 \rightarrow A_2 \backslash U_2)_{U_3}$ est vraie avec un support égal à $|A_1|/|K_1|$ à condition que la condition $A_2 \backslash U_2 \neq \varnothing$.

De manière duale, l'implication de type entre conditions pour attributs dans le sens de Biedermann **BACI** *(Biedermann attributional condition implications)* : $(U_3 \rightarrow A_3 \backslash U_3)_{U_2}$ est vraie avec un support égal à $|A_1|/|K_1|$ à condition que la condition $A_3 \backslash U_3 \neq \varnothing$.

Exemple 2.13. Pour l'exemple, l'implication **BACI** $(b \rightarrow d)_{NK}$ et l'implication **BCAI** $(NK \rightarrow P)_b$ peuvent être extraites à partir du générateur triadique (NK, b).

Chapitre 3

État de l'art

Nous passons en revue dans ce chapitre quelques-uns des principaux algorithmes utilisés pour l'analyse formelle de concepts et l'analyse triadique de concepts. Nous discutons, en particulier dans la deuxième section trois algorithmes de référence dans le domaine triadique et nous finissons dans la troisième section par un résumé des récents travaux adoptant des approches parallèles et distribuées pour l'extraction des concepts.

3.1 Algorithmes pour les contextes dyadiques

3.1.1 Calcul de concepts dyadiques

On peut énumérer dans la littérature une variété d'algorithmes capables de générer et de construire des treillis de concepts dans les contextes dyadiques en un temps polynomial. Le problème de déterminer tous les concepts dans un contexte formel donné étant reconnu comme un problème de classe *P (P complete)* [25].

Rappelons que tout concept formel d'un contexte $\mathbb{K} = (G, M, I)$ peut être écrit sous la forme (X'', X') ou la forme (Y', Y'') où X et Y sont des sous-ensembles d'objets et d'attributs respectivement. L'approche naïve pour générer l'ensemble des concepts consiste tout simplement à énumérer tous les sous-ensembles d'objets $A \subseteq G$ (ou d'attributs $B \subseteq M$) et leur appliquer l'opérateur de dérivation $(.)'$.

Comme le nombre total des concepts formels peut atteindre $2^{min(|G|,|M|)}$ dans le pire cas, cette approche reste bien inefficace même pour de petits contextes. Il existe par ailleurs d'autres algorithmes capables de générer l'ensemble des concepts à partir d'un nombre réduit de sous-ensembles de G tout en utilisant un test de *canonicité* efficace afin d'éviter que les concepts soient générés plus d'une fois.

D'une manière générale, les algorithmes de génération de concepts dans les contextes dyadiques essayent de résoudre deux problèmes principaux :

i. Comment générer tous les concepts formels d'un contexte donné,

ii. Comment éviter la génération répétitive du même concept ou, au moins, déterminer si un concept est généré pour la première fois.

Ces algorithmes peuvent être classés en deux catégories [25] : Les algorithmes incrémentaux et les algorithmes par lot *(batch)*. Les algorithmes par lot traitent d'emblée la totalité du contexte formel et sont plus rapides à identifier les concepts en raison du test de canonicité qui permet d'éviter de générer les mêmes concepts plusieurs fois. Les algorithmes incrémentaux en revanche actualisent le treillis de concepts suite à l'ajout de nouveaux objets. Les algorithmes incrémentaux ne comportent pas de test de canonicité, en conséquence, un même concept peut être généré à plusieurs reprises augmentant ainsi le temps d'exécution. Cependant, les algorithmes incrémentaux existants maintiennent la relation d'ordre entre les concepts et permettent d'obtenir le treillis de concepts.

En outre, tout algorithme par lot *(batch)* adhère à l'une des deux stratégies suivantes : Approche descendante *(top-down)* générant les concepts à partir de l'extension maximale jusqu'à l'extension minimale ou l'approche ascendante *(bottom–up)* qui part de l'extension minimale. Cependant, il est toujours possible d'inverser la stratégie de l'algorithme en considérant les attributs au lieu des objets et vice versa.

3.1.2 Calcul des bases d'implications pour les contextes dyadiques

Nous avons vu au chapitre 2 que les bases génériques sont des implications qui prennent comme prémisses les générateurs minimaux puis les itemsets fermés fré-

quents *(frequent closed itemsets)* comme conclusions. Par itemsets fermés fréquents, on entend des fermés dont le support dépasse un seuil prédéfini par l'utilisateur. Cela correspond aux intentions fréquents. En conséquence, la construction des bases génériques requière à la fois le calcul de tous les intentions de concepts mais aussi leurs générateurs minimaux correspondants. De plus, le calcul des bases informatives des règles d'association nécessite de surcroît l'investigation de liens de précédence entre les concepts, autrement dit, nécessite la construction du *treillis de concepts fréquents* qu'on désigne souvent par *treillis Iceberg* afin de capturer ces deux éléments (fermés fréquents et liens de de précédence) dans la même procédure.

Dans la littérature, on trouve un certain nombre d'algorithmes qui permettent d'obtenir les fermés fréquents en cherchant d'abord leurs générateurs respectifs. De tels algorithmes incluent les algorithmes *Close* [36] puis *A-Close* [35] de même que *Titanic* [45]. D'autres algorithmes par contre adoptent différentes approches, à l'instar de la version améliorée de *CHARM*, *CHARM-L* [53] qui prend le chemin inverse en construisant d'abord le *treillis Iceberg* et en calculant par la suite les générateurs minimaux de chaque fermé fréquent, ou encore *Snow-Touch* [46] développé en combinant *CHARM* avec les nouvelles routines *Talky-G* et *Snow* pour produire simultanément les fermés fréquents, les générateurs minimaux et les liens de précédence.

Pour illustration, l'algorithme **JEN** [27] que nous allons utiliser pour notre approche dans la section suivante, appartient à cette catégorie d'algorithmes qui permettent l'extraction des générateurs à partir du treillis de concepts. La construction des générateurs est basée sur la notion de bloqueur minimal défini un peu plus bas, mais auparavant, nous rappelons deux notions qui y sont rattachées, notamment celle de face et de bloqueur :

Définition 3.1. Face [38] : Soit $c = (X, Y)$ un concept formé de l'extension X et de l'intention Y. Soit $S_i = (X_i, Y_i)$ le $i$ème successeur (parent) immédiat de c dans le treillis de concepts. La $i$ème face F_i du concept c correspond à la différence entre son intention et l'intention de son ième successeur, soit : $F_i = Y - Y_i$.

En notation mathématique, la famille des faces F_C du concept c s'exprime alors par l'expression suivante où N est le nombre de successeurs immédiats du concept c :

$$F_c = \{Y - Intention(S_i(c)), i \in \{1..N\}\}$$

Définition 3.2. Bloqueur [27] : Soit $G = \{G_1, G_2, \ldots G_n\}$ une famille d'ensembles. Un bloqueur de la famille G est un ensemble B dont l'intersection avec tous les ensembles $G_i \in G$ est non vide. À titre d'exemple, si $G = \{AB, AC\}$, alors les ensembles $\{A\}, \{AB\}$ et $\{AC\}$ sont des bloqueurs de la famille G.

Définition 3.3. Bloqueur minimal [27] : Un bloqueur B d'une famille d'ensemble $G = \{G_1, G_2, \ldots G_n\}$ est dit minimal s'il n'existe aucun bloqueur B' de G tel que $B' \in B$. Dans l'exemple précédent, seul l'ensemble $\{A\}$ est un bloqueur minimal de G.

L'importance des bloqueurs minimaux provient du théorème 5 :

Théorème 5. *[27] Soit $c = (X, Y)$ un concept formé de l'extension X et de l'intention Y. Soit F_c la famille de ses faces. L'ensemble des générateurs G associés à l'intention Y du concept c correspondent aux bloqueurs minimaux de la famille des faces F_c.*

L'Algorithme **JEN** parcourt donc les nœuds du treillis de concepts puis calcule au fur et mesure leurs générateurs associés en calculant tout simplement les bloqueurs minimaux de leur famille de faces F_c. Bien entendu, l' algorithme calcule d'abord la famille des faces F_c du nœud en traitement. Il construit par la suite de manière itérative la famille de bloqueurs minimaux. À cet égard, la procédure *calculBloqueursMinimaux* permet d'actualiser la liste JEN_{j+1} représentant les bloqueurs minimaux de la famille des $j+1$ premières faces d'un nœud donné, à partir de la liste JEN_j représentant uniquement les j premières faces.

Le pseudo code pour le traitement d'un nœud avec l'algorithme *JEN* se présente ainsi [27] :

Algorithme 1 : calculGenerateurNoeud(N)

Input : N : Nœud du treillis $\mathfrak{B}$
Output : JEN : ensemble des générateurs du nœud N

ensembleFace ← *calculFace (N)*
JEN ← *attributs de la première face*
if *nombre de successeurs de* $N > 1$ **then**
    On élimine la première face de l'ensemble des faces
    for all *face* ∈ *ensembleFace* **do**
        JEN' ← *calculBloqueursMinimaux (face, JEN)*
        JEN ← *JEN'*
return JEN

3.2 Calcul de concepts dans les contextes triadiques

Les travaux sur la génération de concepts triadiques ont donné lieu jusqu'ici à trois sérieuses propositions qui jouissent d'une bonne popularité au sein de la communauté AFC. Il s'agit notamment du TRIAS [21], *CubeMiner* [22] pour les relations ternaires et *Data-Peeler* [7] qui s'applique aussi pour des relations de $n(\geq 2)$ dimensions. Le principal défi de ces approches est la difficulté de traitement de grandes quantités de données pouvant générer une myriade de possibilités. Ceci et d'autant plus problématique lorsqu'aucune hypothèse n'est faite sur les relations ternaires ou sur la taille du domaine d'attributs.

Pour cela, une des caractéristiques communes de ces approches est de permettre l'activation de contraintes définies par l'utilisateur. Cette fonctionnalité est extrêmement utile afin de permettre de canaliser la recherche des motifs (*les fermés*) vers les centres d'intérêt qui présentent plus de pertinence aux utilisateurs. Par exemple, nous pouvons demander des motifs avec un nombre minimal d'éléments dans certains domaines ou des motifs couvrant au moins un nombre donné d'éléments de la relation Y. Toutefois, ces algorithmes diffèrent par la suite dans leur stratégie d'énumération de concepts. Alors que le TRIAS repose essentiellement sur l'exploration

de concepts formels issus de la conversion du contexte triadique en un contexte dyadique, *CubeMiner* utilise de son côté, une énumération ternaire appliquée de manière récursive en divisant l'ensemble de données original en plus petits sous-ensembles [8]. *Data-Peeler* en revanche adopte une stratégie différente utilisant une nouvelle classe de contraintes et élargissant le traitement vers des relations de n'importe quel ordre ($n \geq 2$).

3.2.1 Trias

Un des premiers algorithmes à traiter les contextes triadiques est Trias[21] proposé par l'équipe de Ganter. Cet algorithme adopte une approche assez intuitive pour construire les cubes maximaux du contexte triadique $\mathbb{K} = (K_1, K_2, K_3, Y)$ en deux étapes. D'abord, en générant les rectangles maximaux (concepts formels) du contexte dyadique $\mathbb{K}^{(1)} := (K_1, K_2 \times K_3, Y^{(1)})$.

En deuxième étape, les extensions A de chaque concept généré $c = (A, I))$ seront testées si elles constituent des cuboïdes maximaux avec les rectangles maximaux (concepts formels) du contexte $\mathbb{K}_I := (K_2, K_3, I)$ construit à partir de I avec $I \subseteq K_2 \times K_3$.

Exemple 3.1. Reprenons le contexte triadique du tableau 2.6 et prenons le cas ou aucune contraintes n'est considérée. Le premier concept retourné est le concept $(A, I) = (\{1, 2, 3, 4, 5\}.\{K - a, N - d, P - a, P - d, R - a\}$.

Le premier contexte $\mathbb{K}_I := (K_2, K_3, I)$ est présenté dans le tableau 3.1 permet de générer les concepts formels suivants : $(KNPRS, \emptyset), (KPR, a), (NP, d)(P, ad)$ et $(\emptyset, abcd)$

Tous ces concepts permet de générer des cuboïdes maximaux en ajoutant $A = \{1, 2, 3, 4, 5\}$ comme extension. A étant déjà maximal ne peut plus être augmenté. Les concepts triadiques générés de cette itération sont alors :

$$(12345, KNPRS, \emptyset), (12345, KPR, a), (12345, NP, d), (12345, P, ad), (12345, \emptyset, abcd)$$

	P	N	R	K	S
a	×		×	×	
b					
c					
d	×	×			

Tableau 3.1: Contexte dyadique $\mathbb{K}_I := (K_2, K_3, I)$

3.2.2 CubeMiner

CubeMiner [22] introduit par Ji et al en 2006 adopte une nouvelle approche au regard du *TRIAS* en opérant directement dans l'espace 3D. Il explore les cubes fermés fréquents **FCC** (*Frequent Closed Cube*) définis comme les cubes fermés $A = (H', R', C')$ respectant trois contraintes de support minimum par rapport aux trois axes suivants : *H-support* $= |R' \times C'| \geq minH$, *R-support*$= |H' \times C'| \geq minR$ et *C-support*$= |R' \times H'| \geq minC$.

CubeMiner saisit l'ensemble des données du contexte $\mathbb{K} = (K_1, K_2, K_3, Y)$ qu'on peut représenter par un cube $3D$ de valeurs booléens "0" ou "1" indiquant l'existence de la relation Y. *CubeMiner* divise de manière récursive le cube de données jusqu'à ce que les cubes résultants soient des cubes entièrement pleins (autrement dit, tous les éléments sont reliés par Y). La division des cubes fait appel à la notion de coupoirs (*"cutters"*) :

3.2.3 Data-Peeler

Contrairement au TRIAS et *CubeMiner*, *Data-Peeler* [7] est un algorithme plus généraliste dans le sens qu'il ne s'applique pas seulement au cas des données tri-dimensionnelles mais les concepteurs voulaient un algorithme qui peut s'appliquer également aux données de dimension supérieure $n > 2$ et qui sont reliées par une relation n-aire $\mathfrak{R} \subseteq D_1 \times ... \times D_n$. L'objectif de *Data-Peeler* est d'extraire les **n-sets fermés** qui respectent une nouvelle classe de contraintes dites contraintes (anti)-

monotones par morceaux. La notion du **n-set fermé** est une généralisation de la notion de concept formel. Intuitivement, un ensemble de n sous-ensembles d'attributs $H = \langle X_1, ..., X_n \rangle$ tel que $x_i \subseteq D_i$ est un ensemble fermé (*closed n-set*) si et seulement si les deux conditions suivantes sont satisfaites :

i. tout élément d'un sous-ensemble X_i est en relation dans $\mathfrak{R}$ avec tous les autres éléments dans les sous-ensembles $X_j, i \neq j$;

ii. Aucun sous-ensemble X_i ne peut pas être élargi sans violer la condition précédente (a).

Formellement, un ensemble $H = \langle X_1, ..., X_n \rangle$ est un (*n-set fermé*) si et seulement s'il respecte les deux contraintes suivantes $C_{connected}$ et C_{closed}, qu'on définit comme suit :

Définition 3.4. **($C_{connected}$)** Un ensemble $H = \langle X_1, ..., X_n \rangle$ satisfait la contrainte $C_{connected}$ si seulement si $\forall U = (x_1, ..., x_n) \in X_1 \times ... \times X_n$, alors $U \in \mathfrak{R}$.

Définition 3.5. **(C_{closed})** Un ensemble $H = \langle X_1, ..., X_n \rangle$ satisfait la contrainte C_{closed} si seulement si $\forall j = 1...n, \forall x_j \in D_j \backslash X_j$, alors l'ensemble $\langle X_1, ..., X_j \cup \{x_j\}, ..., X_n \rangle$ ne satisfait pas la contrainte $C_{connected}$.

3.2.3.1 Déroulement de l'algorithme

Data-Peeler est un algorithme de parcours en profondeur (*Depth-First Search*h) qui progresse le long d'un arbre binaire à partir d'un sommet S en s'appelant récursivement pour chaque nœud voisin. Chaque nœud N de l'arbre d'énumération est une paire (U, V) avec U et V deux $n - sets$ (ensemble de n sous-ensembles). Le nœud $N = (U, V)$ représente tous les $n - sets$ contenant tous les éléments de U et quelques éléments de V. Le sommet S est la paire $(\langle \emptyset_1, ..., \emptyset_n \rangle, \langle D_1, ..., D_n \rangle)$ qui représente tous les n-sets possibles de l'ensemble des données.

Exemple 3.2. Soit le nœud $E = (U, V) = (\langle\ (1), (PR), (a) \rangle, \langle (2), (K, (\emptyset) \rangle)$. Le nœud E représente les quatre 3-sets suivants : $\langle (1), (PR), (a) \rangle$, $\langle (1, 2), (PR), (a) \rangle$, $\langle (1), (PRK), (a) \rangle$ et $\langle (12), (PNR), (a) \rangle$.

Aussi, le nœud $N = (\langle(\emptyset),(\emptyset),(\emptyset)\rangle, \langle\ (12345),(PNRKS),(abcd)\rangle)$ constitue la racine de l'arborescence

Lorsque un nœud $N = (U, V)$ est traité, l'algorithme choisit un élément p de V puis génère deux nouveaux nœuds $N_L = (U_L, V_L)$ et $N_R = (U_R, V_R)$. Le fils gauche N_L représente tout les n-sets de N qui contiennent l'élément p alors que N_R représente à l'opposé ceux qui ne contiennent pas l'élément p. Autrement dit, $(U_L, V_L) = (U \cup \{p\}, V \backslash \{p\})$ alors que $(U_R, V_R) = (U, V \backslash \{p\})$.

Exemple 3.3. Prenons encore l'exemple 3.2. Si on choisit l'élément $p = K$, le nœud $E = (\langle\ (1),(PR),(a)\rangle, \langle(2),(K),(\emptyset)\rangle)$ nous permet d'obtenir les deux nœuds suivants : $E_L = (\langle\ (1),(RRK),(a)\rangle, \langle(2),(\emptyset),(\emptyset)\rangle)$ et $E_R = (\langle\ (1),(PR),(a)\rangle, \langle(2),(\emptyset),(\emptyset)\rangle)$

La contrainte $C_{connected}$ est exploitée lors de cette étape pour réduire la taille de V_L et par conséquent réduire le nombre de candidat p à considérer par la suite. En effet, les éléments de V_L qui ne peuvent pas être ajoutés à U_L sans violer la contrainte $C_{connected}$ peuvent être écartés sans conséquence.

La représentation formelle de ce processus pour un nœud $N = (U, V)$ exprimé par les deux $n - sets$ suivants : $U = \langle U_1, \ldots, U_n\rangle$ et $V = \langle V_1, \ldots, V_n\rangle$ et un élément $p \in V_j$ se décline ainsi :

i. $U_L = \langle U_1, \ldots, U_j \cup \{p\}, \ldots, U_n\rangle$

ii. $V_L = \langle V'_1, \ldots, V'_n\rangle$ tel que $\forall i = 1 \ldots n$,

$$V'_i = \begin{cases} V_j & \text{si } i = j \\ V_i \backslash \{v \in V_i \quad | \langle U_1, \ldots, \{p\}, , \ldots, \{v\}, \ldots, U_n\rangle \quad \text{n'est pas connecté}\} & \text{si } i \neq j \end{cases}$$

iii. $U_R = U$

iv. $V_R = \langle V_1, \ldots, V_j \backslash \{p\}, \ldots, V_n\rangle$

Exemple 3.4. Continuons avec l'exemple 3.2, nous obtenons ultimement les deux nœuds suivants :

$E_L = (\langle\ (1),(PRK),(a)\rangle, \langle(\emptyset),(\emptyset),(\emptyset)\rangle)$ et $E_R = (\langle\ (1),(PN),(a)\rangle, \langle(2),(\emptyset),(\emptyset)\rangle)$.

En effet, l'élément 2 peut être retiré de V_L car il n'est pas connecté à tous les éléments dans U_L, notamment $(2, N, a) \notin Y$

La contrainte de fermeture C_{closed} est traitée également lors du même processus d'énumération. En effet, s'il existe un élément p qui n'est pas couvert par le nœud N c-à-d n'appartenant pas à $U \cup V$ et tel que $\langle U_1 \cup V_1, \ldots, \{p\}, \ldots, U_n \cup V_n \rangle$ est connecté. Il en résulte que tous les n-sets connectés appartenant à N peuvent être élargis par p et par conséquent, ils ne sont pas fermés. Le nœud N peut ainsi être écarté sans conséquence. Cependant, on n'est pas obligé de tester tous les éléments dans l'espace $(D_1 \times ... \times D_n) \backslash U \cup V$. en effet, les éléments qui ont été retirés lors de la vérification de $C_{connected}$ n'ont pas à être testés. La contrainte de fermeture est testée seulement sur le nœud N_R par rapport aux éléments $p_1..p_m$ sélectionnés lors du processus d'énumération

Exemple 3.5. Pour l'exemple 3.2, le nœud $E_R = (\langle\ (1), (PN), (a)\rangle, \langle (2), (\emptyset), (\emptyset)\rangle)$ ne satisfait pas C_{closed} car il peut être augmente par l'élément R.

3.2.4 Comparaisons des algorithmes pour l'ATC

Data-Peeler reste à ce jour l'algorithme le plus performant pour l'extraction des motifs dans le domaine triadique. Sa performance dépasse d'une magnitude allant de 10 à 100 celle de *CubeMiner* ou TRIAS [40]. La figure 3.1 présente une comparaison entre les trois algorithmes. Dans cet article, les trois algorithmes ont été testés sur des ensembles de données provenant de Twitter. Que se soit avec un pré-traitement des données ou sans pré-traitement, *Data-Peeler* dépasse clairement les deux autres algorithmes, particulièrement lorsque la taille des données devient assez importante. Pour cela, dans la suite de ce mémoire, nous focalisons notre travail comparatif sur la comparaison avec *Data-Peeler* qui nous considérons l'algorithme de référence pour l'extraction des motifs dans les contextes triadiques.

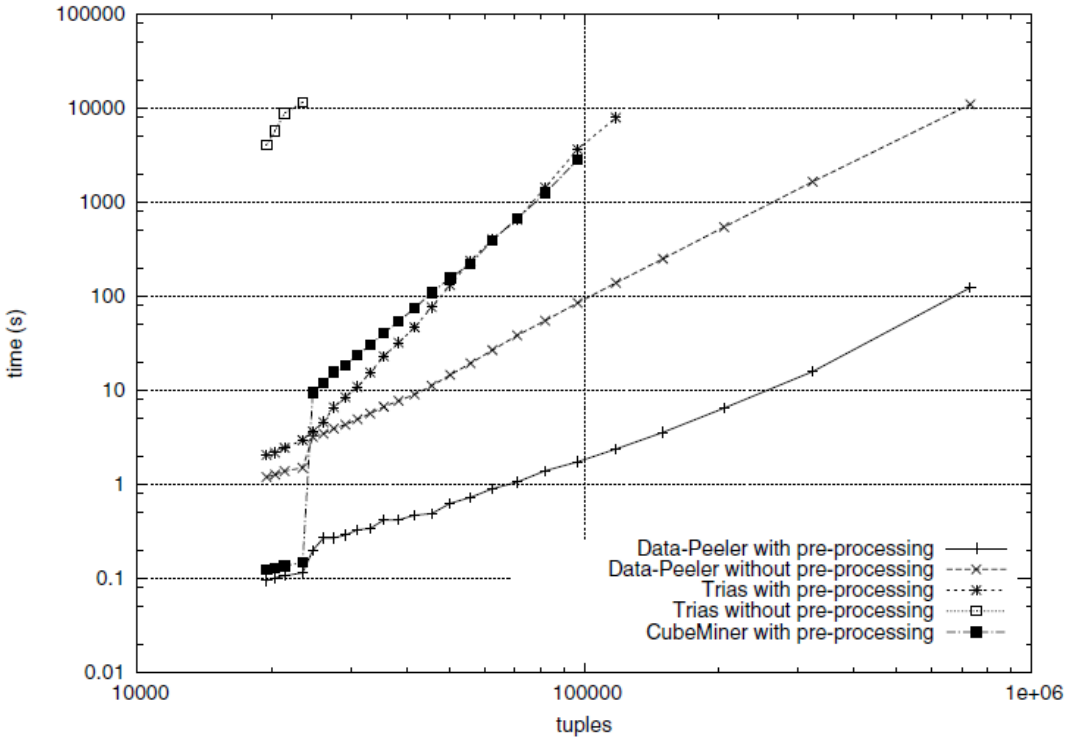

FIGURE 3.1: Temps d'exécution de *Data-Peeler*, TRIAS et *CubeMiner* sur un contexte triadique provenant de données de Twitter [40]

3.3 Approches parallèles et distribuées pour l'AFC

Avec l'avènement du big data, la majorité des algorithmes discutés dans la section 3.1 restent inefficaces et difficilement évolutifs pour traiter les données massives. Ainsi, beaucoup de chercheurs se sont intéressés naturellement à adapter certaines de ces approches aux modèles de programmation parallèle et distribuée utilisées pour les données massives.

Depuis quelques années, on trouve quelques travaux sur la mise en œuvre de MapReduce ou ses variantes pour le calcul itératif comme Haloop et Twister. Xu et al. [52] ont ainsi proposé un algorithme itératif pour la génération de concepts utilisant Twister. Il s'agit d'une implémention des algorithmes de Ganter *Next Clo-*

sure [16] et *CbO*[24] sur le framework MapReduce. Chunduri et al. [9] proposait une autre approche utilisant Haloop. Néanmoins, Beaucoup de ces travaux sont seulement opérationnels pour les données de taille relativement réduite et ne sont pas suffisants pour traiter de large données, en partie, car ces approches ont souvent omis de régler un certain nombre de faiblesse du framework MapReduce comme l'utilisation efficace des mémoires ou le calcul en mémoire (*in-memory computation*) [10]. Il faut aussi dire que la majorité de ces algorithmes calculent l'ensemble des concepts mais pas les liens entre eux.

Dans les deux sous-sections suivantes, nous abordons en détails les version parallèle et distribuée des algorithmes *Next Closure* et *CbO*.

3.3.1 MRGanter

MRGanter[52] représente une version distribuée de *Next Closure*. Cette approche adopte également une implémention itérative du framework MapReduce, cependant, elle a la particularité de diviser les données traitées en plusieurs tranches disjointes. Chaque tranche est traitée de manière complètement indépendante par une opérion *Map*, les concepts formels partiels générés sont ensuite fusionnés par une opération *Reduce* pour obtenir les concepts formels du contexte dyadique initial.

L'intérêt du travail de Xu et al. [52] ne réside pas seulement dans les algorithmes présentés, en l'occurrence *MRGANTER* ou sa version améliorée *MRGANTER+*, les auteurs présentent aussi quelques réflexions intéressantes quant au traitement distribué des contextes dyadiques que nous résumons ci-dessous :

Soit un contexte formel $\mathbb{K} = (G, M, I)$. Le traitement distribué consiste à fractionner l'ensemble des objets G en n sous-ensembles disjoints. Cependant, à titre d'illustration, nous nous limitons sans perte de généralité à $n = 2$. Le contexte formel $\mathbb{K} = (G, M, I)$ peut alors être divisé en deux contextes $\mathbb{K}_1 = (G_1, M, I_1)$ et $\mathbb{K}_2 = (G_2, M, I_2)$. Notons que les deux contextes partagent le même ensemble d'attributs M alors que $G_1 \cap G_2 = \emptyset$ et $G_1 \cup G_2 = G$.

$\mathbb{K}_1$	a	b	c	d
1	×		×	×
2		×	×	×
3	×		×	×

$\mathbb{K}_2$	a	b	c	d
4	×		×	×
5		×		×
6	×		×	

Tableau 3.2: Les partitions $\mathbb{K}_1$ et $\mathbb{K}_2$ dérivées du contexte dyadique du tableau 2.2

Proposition 3.1. Soit $\mathbb{K} = (G, M, I)$ un contexte formel et $\mathbb{K}_1 = (G_1, M, I_1)$ et $\mathbb{K}_2 = (G_2, M, I_2)$ deux partitions de $\mathbb{K}$. Soit $Y \subseteq M$ un sous-ensemble d'attributs, nous avons la propriété suivante : $Y'_{\mathbb{K}} = Y'_{\mathbb{K}_1} \cup Y'_{\mathbb{K}-2}$. Autrement dit, l'union des sous-ensembles d'objets générés par les opérateurs de dérivation $(.)'$ associés aux partitions $\mathbb{K}_1$ et $\mathbb{K}_2$ est équivalent au sous-ensemble d'objets généré par l'opérateur de dérivation associé au contexte global $\mathbb{K}$ [52].

Proposition 3.2. Soit $\mathbb{K}(G, M, I)$ un contexte formel et $\mathbb{K}_1 = (G_1, M, I_1)$ et $\mathbb{K}_2 = (G_2, M, I_2)$ deux partitions de $\mathbb{K}$. Soit $Y \subseteq M$ un sous-ensemble d'attributs, nous avons la propriété suivante : $Y''_{\mathbb{K}} = Y''_{\mathbb{K}_1} \cap Y''_{\mathbb{K}-2}$. Autrement dit, l'intersection des fermetures générées par un sous-ensemble attributs Y par rapport aux partitions $\mathbb{K}_1$ et $\mathbb{K}_2$ est équivalent au fermé du même sous-ensemble par rapport au contexte global $\mathbb{K}$ [52].

Exemple 3.6. Prenons le contexte $\mathbb{K} = (G, M, I)$ de tableau 2.2 et ses deux partitions présentées au tableau 3.2 . Soit $Y = \{a, c\}$, on peut constater que :

$Y'_{\mathbb{K}_1} = \{a, c\}' = \{1, 2\}$, $Y'_{\mathbb{K}_2} = \{a, c\}' = \{4, 5\}$

$Y'_{\mathbb{K}} = \{a, c\}' = \{1, 2, 4, 5\} = \{1, 2\} \cup \{4, 5\} = Y'_{\mathbb{K}_1} \cup Y'_{\mathbb{K}_2}$

Aussi,

$Y''_{\mathbb{K}_1} = \{a, c\}'' = \{1, 2\}' = \{a, c, d\}$, $Y''_{\mathbb{K}_2} = \{a, c\}'' = \{4, 5\}' = \{a, c\}$

$Y''_{\mathbb{K}} = \{a, c\}'' = \{1, 2, 4, 5\}' = \{a, c\}$=$\{$a,c,d $\}\cap\{a, c\} = Y''_{\mathbb{K}_1} \cap Y''_{\mathbb{K}_2}$

Théorème 6. *[52] Soit $Y \subseteq M$ un sous-ensemble d'attributs. Soit $F^Y_{\mathbb{K}_1}$ et $F^Y_{\mathbb{K}_2}$ les deux fermés de Y générés dans les partitions $\mathbb{K}_1$ et $\mathbb{K}_2$ respectivement. Alors le fermé de Y dans le contexte global $\mathbb{K}$ est donné par :*

$$F^Y_{\mathbb{K}} = F^Y_{\mathbb{K}_1} \cap F^Y_{\mathbb{K}_2}$$

Plus généralement, pour n partitions disjointes on peut écrire :

$$F_{\mathbb{K}}^{Y} = F_{\mathbb{K}_1}^{Y} \cap ... \cap F_{\mathbb{K}_n}^{Y}$$

Le théorème 6 nous donne déjà un indice de l'approche préconisée par l'algorithme *MRGanter*. Nous devons observer que l'algorithme *Next Closure* consiste en deux étapes : 1) calculer le fermé du nouveau candidat, puis 2) évaluer si le résultat doit ou non être ajouté à la liste courante des concepts formels. Dans le langage MapReduce, la première étape correspond à la fonction *Map* qui peut être déployée de manière parallèle sur plusieurs grappes (*clusters*), chacun traitant une partition $\mathbb{K}_i$ du contexte formel $\mathbb{K}$. L'intégration et la validation des résultats se font en revanche lors de phase *Reduce*.

3.3.2 Version parallèle et distribuée de CBO *(Distributed and parallel CBO)*

La version originelle de l'algorithme *CBO (Closed-by-one)* introduite par Kuznetsov [24] est un algorithme de parcours en profondeur opérant sur l'espace des concepts formels. 0n peut le caractériser par une procédure récursive, ici notée *GenerateForm((A,B),j)*, qui prend comme arguments un concept formel (A, B) puis génère un nouveau concept (C, D) en lui ajoutant un nouveau attribut $j \in Y$. Le concept (C, D) peut être obtenu de la manière suivante :

$$(C, D) = ((B \cup \{j\})', (B \cup \{j\})'')$$

Il faut noter que la pair (C, D) est toujours un concept formel de sorte que $B \subseteq D$. La prochaine étape pour l'algorithme est de vérifier si le concept (C, D) peut être retenu comme prochain argument de la procédure récursive *GenerateForm* ou il doit être ignoré. Cette validation se fait à travers un *test de canonicité* afin de s'assurer que le concept (C, D) est généré pour la première fois. Ce test est basé simplement

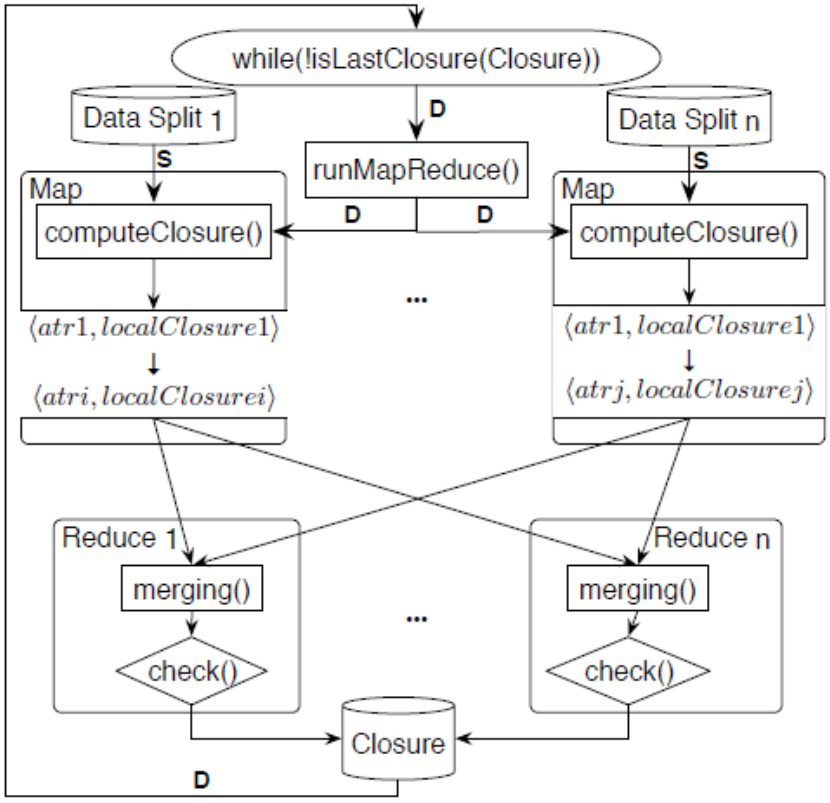

FIGURE 3.2: Processus *(workflow) de la version parallèle et distribue de NextClosure* [52]

sur la comparaison de $B \cap Y_j = D \cap Y_j$ avec

$$Y_j = \{y \in Y | y < j\}$$

L'algorithme prend fin lorsque l'une des deux conditions suivantes est satisfaite : 1) le concept $(A, B) = (Y', Y)$, c-à-d lorsque le concept *infimum* est atteint. 2) $j > n$, ce qui signifie qu' aucun autre attribut ne peut être ajouté. Autrement dit, l'algorithme passe à travers tous les attributs $i \in Y$ et $i >= j$ qui ne sont pas déjà présent dans l'intention du concept B.

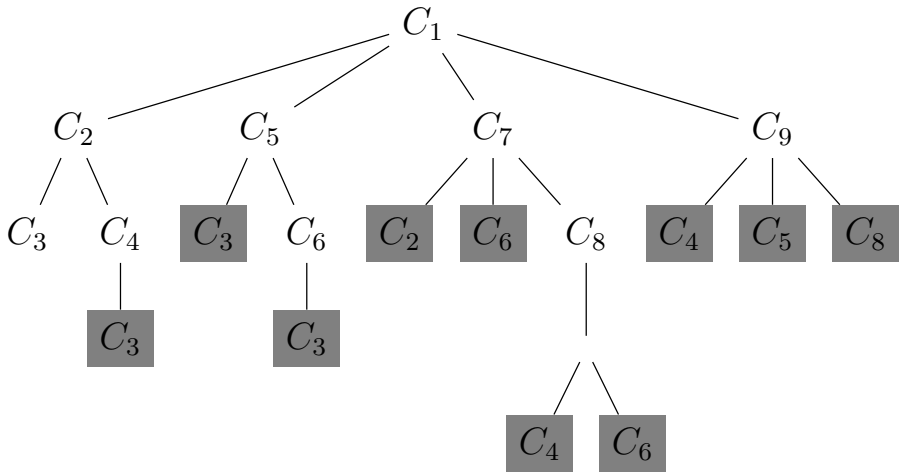

FIGURE 3.3: Arbre en profondeur généré par CBO pour le contexte du tableau 2.2

$C_1 = (\{1,2,3,4,5,6\},\emptyset)$, $C_2 = (\{1,3,4,6\},\{a,c\})$, $C_3 = (\emptyset,\{a,b,c,d\})$, $C_4 = (\{1,3,4\},\{a,c,d\})$, $C_5 = (\{2,5\},\{b,d\})$, $C_6 = (\{2\},\{b,c,d\})$, $C_7 = (\{1,2,3,4,6\},\{c\})$, $C_8 = (\{1,2,3,4\},\{c,d\})$, $C_9 = (\{1,2,3,4,5\},\{d\})$

L'invocation récursive de *GenerateForm* à partir du concept racine $(\emptyset', \emptyset'')$ permet de construire un arbre dont chaque nœud correspond à un appel de *GenerateForm*. Les points noircis représentent les concepts calculés mais qui ne passent pas le test de canonicité.

La version distribuée de CBO introduite par [23] est souvent considérée comme une validation (" *proof-of-concept*") de la possibilité d'utilisation du framework *MapReduce* pour le calcul des concepts formels. En substance, cette approche est basée sur la partition de l'ensemble des concepts formels en plusieurs partitions disjointes qui peuvent être traitées simultanément. Une différence notable à soulever : alors que la version originale adopte une stratégie de parcours en profondeur, la version distribuée adopte une stratégie de parcours en largeur afin d'adapter l'algorithme au framework *MapReduce*.

Aussi, la procédure originale *GenerateForm* est divisée en deux fonctions : une fonction *Map* appelée *MapConcepts* et une fonction *Reduce* appelée *ReduceConcepts*. La fonction *MapConcepts* permet de générer de nouveaux concepts alors que la

fonction *ReduceConcepts* se charge de la validation du test de canonicité. La stratégie de parcours en largeurs consiste à parcourir l'arbre des concepts formels niveau par niveau. Chaque niveau est obtenu par l'application successive de la fonction*MapConcepts* et la fonction *ReduceConcepts* aux concepts générés au niveau précédent.

Exemple 3.7. La figure 3.4 résume le fonctionnement de la version distribuée de *CBO*. Si on reprend le contexte de l'exemple 2.2, la première itération nous permet d'obtenir les concepts $C_2 = (\{1,3,4,6\},\{a,c\})$, $C_5 = (\{2,5\},\{b,d\})$, $C_7 = (\{1,2,3,4,6\},\{c\})$ et $C_9 = (\{1,2,3,4,5\},\{d\})$. Uniquement les concepts passant le test le canonicité $B \cap Y_j = D \cap Y_j$ sont sauvegardés puis retenus pour la deuxième itération, soit tous les concepts dans ce cas. Cependant, lors de la deuxième itération, $C_7 = (\{1,2,3,4,6\},\{c\})$ permet de générer $C_8 = (\{1,2,3,4\},\{c,d\})$ mais aussi $C_2 = (\{1,3,4,6\},\{a,c\})$, $C_6 = (\{2\},\{b,c,d\})$ qui ne passent pas le test de canonicité et ne sont pas donc retenus pour la suite du calcul. On peut noter que l'arbre généré correspond aussi à l'arborescence de la figure 3.3. La fonction *Map* permet de calculer les concepts de chaque niveau, alors que la fonction *Reduce* permet de filtrer les concepts retenus pour les prochaines itérations.

3.4 Synthèse

Depuis l'avènement de l'AFC et par la suite l'ATC, plusieurs travaux se sont intéressés à proposer différentes sortes d'algorithmes ou d'approches pour la découverte de motifs (concepts et implications) dans les contextes formels. Nous pouvons néanmoins dresser les deux constats suivants :

- À ce jour, on compte peu de publications qui se sont penchées sur la problématique du *big data* pour l'analyse formelle de concepts ou l'analyse triadique de concepts ainsi que sur la mise en oeuvre d'approches parallèles et distribuées adéquates. Au moment de la rédaction de ce mémoire, les contributions significatives sur ce sujet sont rares bien qu'on ressente un intérêt palpable pour le sujet au sein de la communauté.

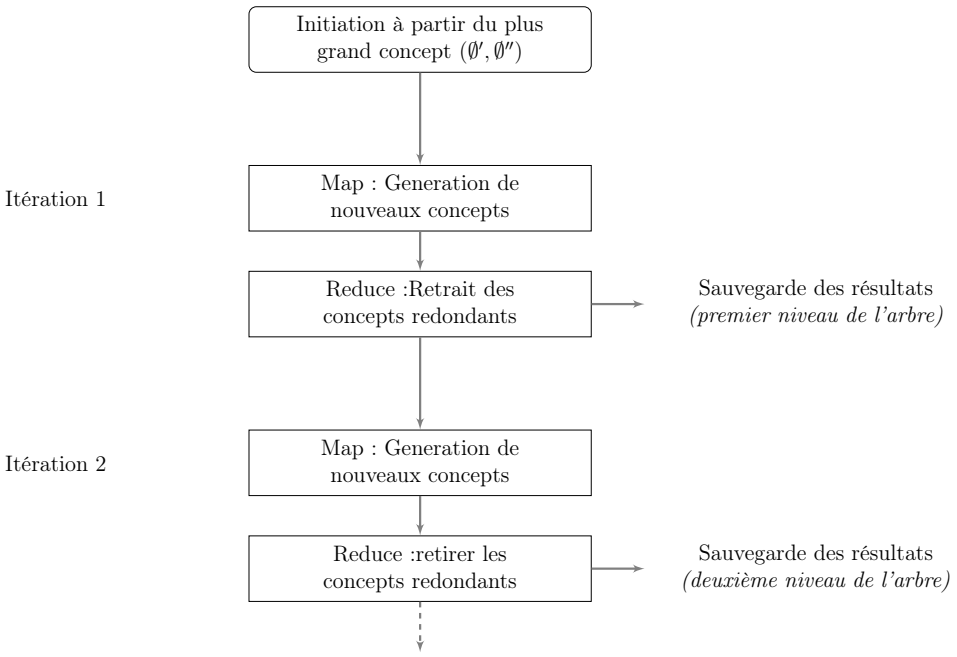

FIGURE 3.4: Exécution itérative de Mapreduce par CBO

- Peu de méthodes [33, 20] en analyse triadique de concepts ont proposé des algorithmes de génération à la fois de concepts et d'implications triadiques. Certaines ont été dédiées exclusivement aux concepts triadiques ou aux implications et plus généralement des règles d'association.

L'intérêt de ce travail réside dans ces deux constats. À travers la méthodologie proposée, nous pensons apporter une réponse à ces préoccupations en adoptant une approche plus adaptée au défi des données massives basée sur le modèle parallèle et distribué et qui permet de générer à la fois les implications triadiques et les concepts triadiques.

Chapitre 4

Calcul parallèle et distribué

Nous rappelons dans l'introduction de ce chapitre quelques caractéristiques du calcul parallèle et distribué, nous justifions notre choix du framework *Apache Spark* en menant une comparaison avec le framework *Apache Hadoop*. Le reste du chapitre est consacré à la méthodologie que nous adoptons pour le calcul des concepts et implications dans les contextes triadiques. Nous concluons ce chapitre avec un exemple illustratif de petite taille avant le déploiement futur de notre solution sur des systèmes adaptés au *big data*.

4.1 Introduction

Le paradigme de calcul distribué (ou réparti), bien qu'il existe depuis déjà quelques décennies (milieu des années cinquante) [5], revient avec force sur la scène comme une alternative aux très onéreux super-ordinateurs afin de répondre aux besoins nouveaux et croissants à la fois des utilisateurs et des applications avec l'émergence des données massives. À l'opposé des super-ordinateurs, les systèmes informatiques distribués sont construits à partir d'un très grand nombre de nœuds ou entités de calcul, souvent peu coûteuses (*commodity hardware*) connectées via un réseau local rapide. Cette architecture permet d'obtenir des capacités de calcul très élevées en joignant ensemble un grand nombre d'unités de calcul via un réseau rapide et en partageant

les ressources entre différents utilisateurs de manière transparente. Avoir plusieurs unités de calcul traitant les mêmes problèmes signifie qu'un dysfonctionnement de l'une de ces unités n'influence pas l'ensemble du processus informatique.

À ce stade, il importe d'apporter la précision suivante : les termes *calcul parallèle* et *calcul distribué* sont souvent utilisés de manière interchangeable. Ils réfèrent en vérité à des concepts légèrement différents. En effet, le terme calcul distribué désigne une classe un peu plus large que celle du calcul parallèle. Plus précisément, le terme calcul parallèle fait référence à un modèle de calcul où le traitement est divisé entre plusieurs processeurs partageant la même ressource mémoire. Cette architecture est souvent caractérisée par une homogénéité des composants : des processeurs similaires accédant à une ressource mémoire partagée. Les programmes parallèles sont décomposés en plusieurs unités d'exécution qui peuvent être allouées à des processeurs différents communiquant entre eux par le moyen de la mémoire partagée. Le calcul distribué, en revanche, désigne une classe plus étendue englobant toute architecture ou tout système qui permet de décomposer le calcul en plusieurs parties qu'on peut exécuter de manière simultanée sur plusieurs éléments informatiques connectés à l'aide d'un réseau de communication. Ces éléments, bien que cela ne soit pas toujours la règle, peuvent être hétérogènes en terme de fonctionnalités matérielles ou logicielles sans partager nécessairement une ressource mémoire commune.

Les approches traditionnelles du calcul parallèle et distribué se basent sur les principes de verrouillage et synchronisation (*synchronization and locking*) [13]. Ces approches abordaient séparément le calcul proprement dit et les données à traiter, en se basant sur l'hypothèse que les données qui sont stockées dans une base de données ou tout autre système de stockage sont accessibles de manière équitable par les nœuds ou unités de calcul. Le traitement parallèle dans ces nœuds a la charge par la suite de verrouiller les données et de les traiter. Bien entendu, les coûts (*overhead*) associés à une telle méthode sont assez importants, non seulement, ils couvrent les coûts de verrouillage, mais surtout les coûts de transfert des données entre les nœuds où se trouvent les données et les nœuds de traitement de ces données. Nous pouvons désormais introduire le modèle de programmation *MapReduce* pour le calcul distribué

et parallèle qui fut développé pour pallier ce genre de limitations. *MapReduce* est basé sur les deux principes suivants [13] :

i. lorsque la répartition des tâches de calcul est basée sur une segmentation des données, de sorte que chaque nœud de traitement traite une partition différente des données, alors dans ce cas, on peut se passer de l'impératif de verrouillage et de synchronisation, ce qui permet d'améliorer les performances du calcul parallèle.

ii. Si le nœud de calcul traite des données qui sont stockées localement sur le nœud lui-même, la surcharge (*overhead*) associé aux transmissions de données d'un nœud vers un autre peut être évité.

MapReduce a été développé en premier lieu dans le cadre du projet *Hadoop ecosystem*. Néanmoins, le modèle reste générique et fut adopté par la suite par une variété de systèmes incluant les systèmes de gestion des bases de données Non-SQL tel que Cassandra ou MongoDB [13]. Comme nous avons adopté dans le cadre de ce projet le framework *Apache Spark*, nous dressons dans les deux sous-sections qui suivent une comparaison entre ce système et le système précurseur *Apache Hadoop* en vue de justifier notre choix.

4.1.1 Apache Hadoop

Apache Hadoop est un framework logiciel ouvert *(open source)* qui fournit des solutions pour la manipulation des données massives permettant à la fois un traitement distribué, fiable et évolutif des données. Créé par Doug Cutting en 2005 alors qu'il travaillait pour Yahoo, Hadoop 1.0 fut proposé au grand public en novembre 2012 dans le cadre du projet Apache, sponsorisé par la fondation Apache Software. Hadoop 1.0 possède deux composantes principales nommées HDFS *(Hadoop Distributed File System)* et le framework *MapReduce*.

Hadoop Distributed File System inspiré du système de fichiers *(GFS)* de Google est un système de fichiers distribués pour un stockage évolutif et efficace basé sur la réplication des données sur divers nœuds faisant partie de grappes. HDFS est basé sur une architecture maître-esclave. Chaque grappe comporte un *NameNode* faisant office

de serveur principal (maître) chargé de la gestion des fichiers. Les grappes comportent également un ou plusieurs nœuds esclaves *DataNode* où se trouvent réellement les données, très probablement des données répliquées. Le facteur de réplication par défaut est de trois mais peut être configuré en fonction des besoins de l'utilisateur et du type d'utilisation.

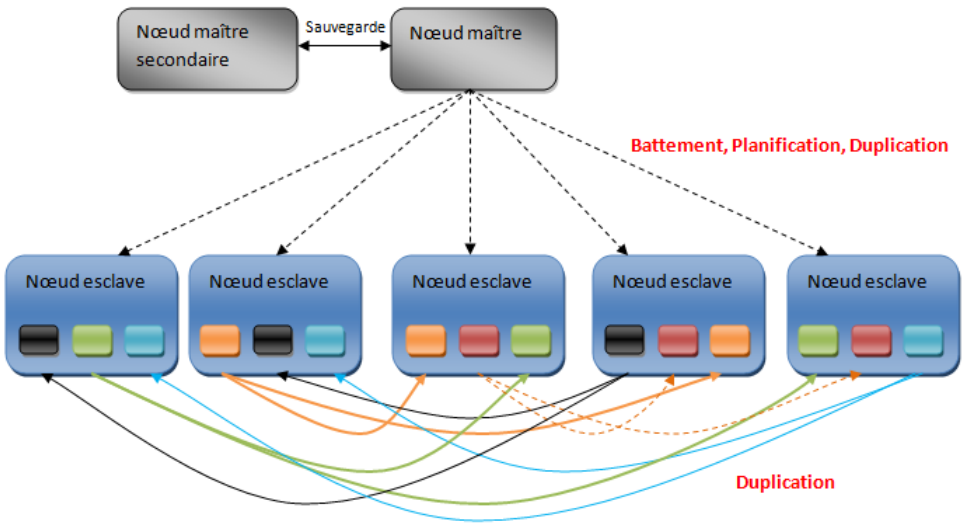

FIGURE 4.1: Répartition des données sur *Hadoop File system*

Le deuxième composant, *MapReduce*, comme fut introduit précédemment est un modèle de programmation qui permet un traitement résilient, parallèle et distribué des données massives sur des grappes d'ordinateurs. Concrètement, ce modèle est basé sur deux fonctionnalités principales : la fonction *Map* permet de distribuer des partitions différentes de données vers une multitude de *"mappeurs"* qui travaillent en parallèle pour accomplir l'essentiel du calcul sur les données massives. Les résultats des différentes grappes sont par la suite récupérés et réorganisés dans une phase intermédiaire désignée par l'opération *Sort/Group/Shuffle*. Finalement, la fonction *Reduce* agrège les résultats intermédiaires pour trouver le résultat final du problème initial.

La figure 4.2 fournit un exemple sur la façon dont *MapReduce* permet d'obtenir la distribution des mots dans un groupe de fichiers. Le *"Input"* de la phase *Map* est une

collection de paires (*clé_ entrée, Valeur*). La sortie de la phase *Map* est également une autre collection de paires (*clé_ sortie, Valeur*). Pour cet exemple, l'entrée de la phase *Map* est un ensemble de fichiers. Chaque fichier est présenté comme une collection de lignes. Les paires (*clé_ entrée, Valeur*) pour la phase *Map* sont constituées alors, de la position de la ligne comme *clé_ entrée* et la ligne elle-même comme *valeur*. La phase *Map* implique le lancement de plusieurs programmes (dans différents fils du même nœud et de plusieurs nœuds), chacun de ces programmes est appelé *mappeur*. L'entrée de chaque *mappeur* est une paire (*clé_ entrée, Valeur*), soit dans ce cas une ligne individuelle, la clé_entrée est souvent omise dans cette phase. Le programme *mappeur* a pour tâche de fractionner la ligne en mots et de créer des paires *(mot,1)*. L'étape *Sort/Group/Shuffle* permet de trier cette collection de sortie en fonction des clés. Elle les regroupe puis elle les transmet vers l'étape *Reduce*.

Un des avantages intéressants de ce framework est sa tolérance aux fautes. Une tâche peut être transférée d'un nœud à l'autre si par exemple un nœud reste silencieux plus longtemps que prévu, le nœud principal peut assigner à nouveau la tâche à un autre nœud. Cette résilience permet la mise en place de cette structure sur des serveurs peu coûteux permettant une baisse substantielle des coûts.

Le framework Hadoop n'a cessé d'être développé et mis à jour depuis sa première apparition. Les versions qui suivirent, incorporaient d'autres composants tel que le gestionnaire des ressources *YARN* ou le module d'apprentissage machine *Submarine* permettant ainsi d'étendre la polyvalence du framework pour le traitement et l'analyse des données. Actuellement, Hadoop est proposé en version 3.2.0.

Toutefois, bien que tout récemment, le monde du *Big Data* connaît un revirement notable par rapport à ce modèle avec l'introduction d'un nouveau framework Apache Spark. Ce dernier fournit une interface de programmation plus conviviale qui réduit les efforts de codage et permet d'obtenir de meilleures performances dans la majorité des applications. Les raisons de ce revirement viennent du fait que pour une bonne classe de problèmes analytiques sur les *big data*, une seule itération *MapReduce* n'est souvent pas suffisante pour effectuer les calculs requis. En réalité, même pour des problèmes analytiques assez banals, on doit avoir recours à plusieurs itérations de *MapReduce* pour obtenir des résultats. À titre d'exemple, le calcul de l'écart-type

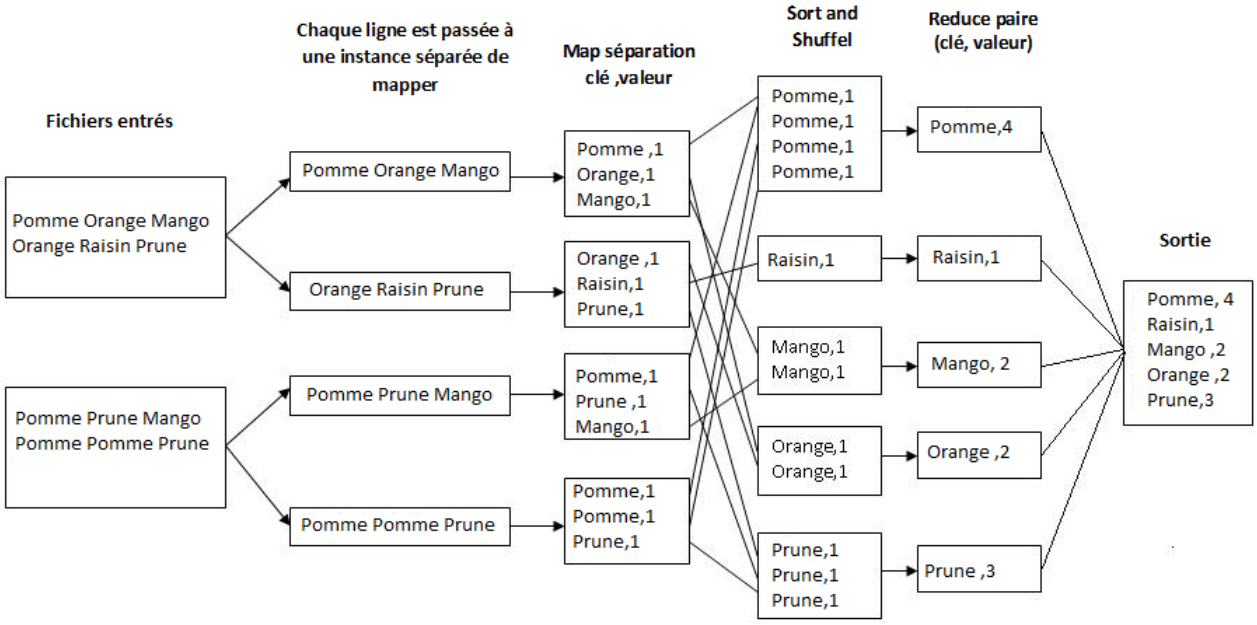

FIGURE 4.2: Comptage de mots par MapReduce

nécessite deux itérations *MapReduce* : la première permet de calculer la moyenne alors que la deuxième se base sur ce résultat pour calculer l'écart-type. Le résultat de chaque itération est sauvegardé sur le HDFS et sert d'entrée pour l'itération suivante . La figure 4.3 montre un processus type à plusieurs itération de *MapReduce*.

Malheureusement, dans bien de cas, cette sorte d'approche s'avère lourde et prend un temps non négligeable en raison des opérations répétitives de lecture et d'écriture sur les différents HDFS. De telles opérations coûtent 90 % de temps global de calcul. En outre, bien que *MapReduce* ait été développé pour exécuter des calculs parallèles distribués sur du matériel informatique standard, de nos jours, ces ordinateurs sont

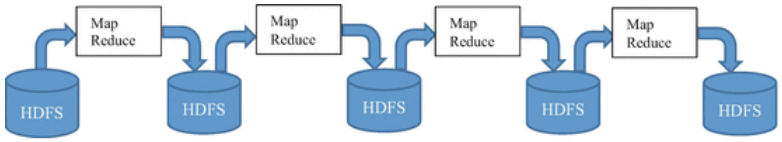

FIGURE 4.3: MapReduce Workflow

dotés de plus en plus de capacités mémoire et de traitement supérieurs et sont assez courants dans les entreprises et facilement disponibles. Ces limitations ont été le principal motif derrière le développement de *Apache Spark* qui essaye de tirer profit de cette capacité de stockage disponible sur les machines actuelles. *Apache Spark* est donc le système choisi pour notre implémentation.

4.1.2 Apache Spark

Apache Spark a commencé comme projet de recherche à l'Université UC Berkeley en 2009 avant de devenir un système ouvert (*open source*) sous forme de projet Apache en 2010. Il n'a pas été prévu pour remplacer Hadoop mais, pour offrir une solution complète et unifiée permettant de prendre en charge une classe d'applications beaucoup plus large que MapReduce, tout en maintenant la tolérance aux pannes du framework. Un des points distinctifs de Spark est son utilisation des RDDs *(Resilient Distributed Datasets)*, une abstraction permettant de réutiliser efficacement les données dans une large famille d'applications. Les RDDs sont conçus pour assurer le stockage de données en mémoire entre les requêtes sans besoin de réplication. La résilience est obtenue en partie en suivant la lignée des transformations appliquées au partitions de données. Les RDDs permettent à Spark de surpasser les modèles existants jusqu'à 100 fois pour les analyses multi-passes. Les RDD peuvent prendre en charge une grande variété d'algorithmes itératifs, ainsi que l'exploration de données interactives.

Les RDD sont donc la structure de données fondamentale de Spark. Il s'agit d'une collection immuable d'objets distribués. Chaque ensemble de données dans RDD est divisé en partitions logiques qui peuvent être calculées sur différents nœuds

du cluster. Les RDD peuvent contenir tout type d'objet Python, Java ou Scala, y compris les classes définies par l'utilisateur.

En langage plus pratique, les RDD considèrent la mémoire de plusieurs ordinateurs comme une seule mémoire contiguë. Les variables RDD typiques sont des collections (telles que *map, array, list*) qui restent en mémoire mais s'étendent sur plusieurs ordinateurs. Cette approche engendre deux avantages : d'une part, la collection RDD étant distribuée sur plusieurs machines, se prête bien au traitement parallèle et chaque machine effectue le calcul localement sur la partition qu'elle héberge. D'autre part, le traitement *MapReduce* distribué étant effectué dans les zones mémoire du RDD, le traitement est beaucoup plus rapide que *MapReduce* basé sur HDFS comme l'illustre la figure 4.4.

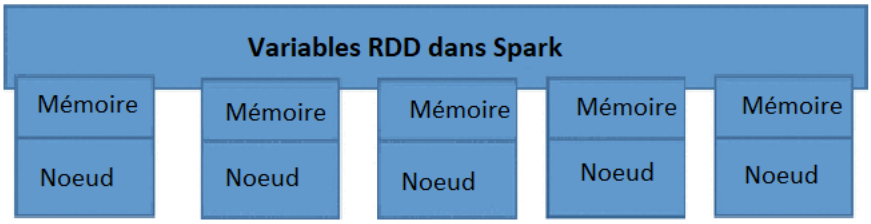

FIGURE 4.4: Variable RDD dans Spark

Dans le cas de Spark, *map* est une transformation qui traite chaque élément du RDD puis retourne un ou plusieurs nouveaux RDD représentant les résultats sans changer le RDD d'entrée. Par exemple, si notre RDD *sc* représente une collection de 1000 nombres entiers, si on veut augmenter chaque élément de 10 points, il suffit d'appliquer une transformation $sc.map(lambda \quad x : x + 10)$. La *Réduction*, en revanche est une *action* qui regroupe tous les éléments d'un RDD, leur applique une fonction puis retourne un résultat final. Pour cet exemple, une réduction va retourner la somme de tous les éléments (nombre entiers) du RDD.

En règle générale, les RDD sont conservés dans la mémoire et ne cessent d'exister qu'une fois l'exécution du programme Spark est terminée. Cependant, il est également

possible de conserver un RDD en mémoire auquel cas Spark conservera les éléments sur la grappe pour un accès beaucoup plus rapide la prochaine exécution. La figure 4.5 illustre les opérations itératives sur un RDD de Spark. Le système stocke les résultats intermédiaires dans une mémoire distribuée au lieu d'un stockage stable (disque) et accélère en conséquence l'exécution :

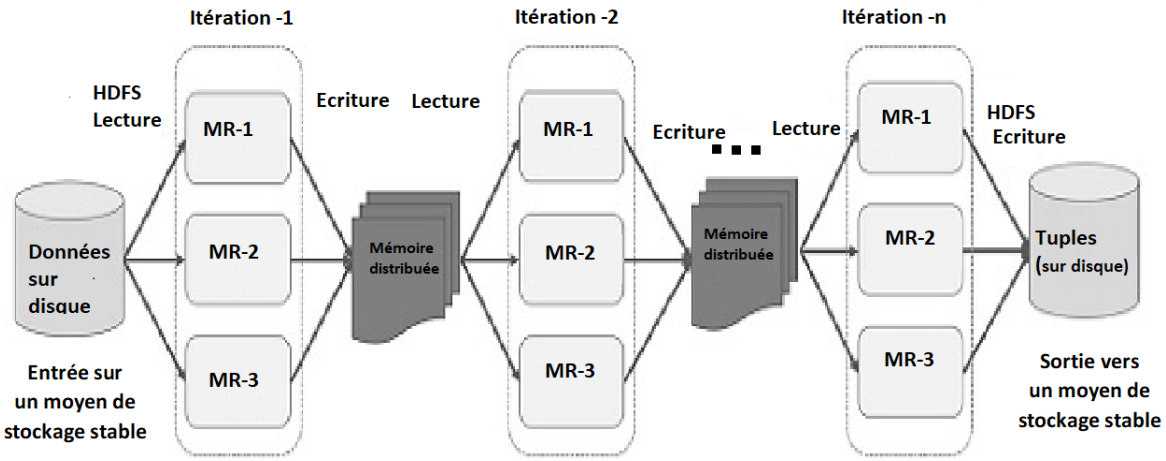

FIGURE 4.5: Exécution itérative de MapReduce sur Spark

4.2 Approche et méthodologie

L'objectif de ce travail est de présenter une nouvelle approche pour le calcul des concepts dans un contexte triadique qui serait plus adaptée aux exigences des données massives en se basant sur le traitement parallèle et distribué, notamment celui offert par le framework MapReduce. La solution sera implémentée dans un environnement Apache Spark pour les avantages qu'il offre pour les exécutions itératives de MapReduce.

Tel que mentionné dans le premier chapitre, la particularité de cette approche est de procéder en quelque sorte à contre-pied des approches traditionnelles. Jusqu'ici, les travaux sur l'AFC ou ultérieurement sur l'ATC ont toujours procédé à l'extraction

des concepts (fermés) et éventuellement des générateurs minimaux dans une première phase avant d'en déduire les bases des implications par la suite. Notre approche adopte justement le chemin inverse en calculant d'abord les implications triadiques avant d'en déduire les concepts triadiques.

Nous nous intéressons en particulier aux implications triadiques selon la définition de *Ganter* présentées au chapitre 2. Ainsi, pour un contexte triadique $\mathbb{K} = (K_1, K_2, K_3, Y)$, les implications **CAI** *(entre attributs sous conditions)* peuvent être obtenues à partir des implications entre attributs calculées pour les contextes dyadiques de la forme $\mathbb{K}^{1,2}_{c_i} := (K_1, K_2, Y^{1,2}_{c_i})$ avec $Y^{1,2}_{c_i}$ représentant la relation binaire entre K_1 et K_2 sous la condition c_i (cf. figure 4.6). De la même manière, les implications **ACI** *(entre conditions pour attributs)* peuvent être générées à partir des implications entre conditions calculées pour les contextes dyadiques de la forme $\mathbb{K}^{1,3}_{a_i} := (K_1, K_3, Y^{1,3}_{a_i})$ où $Y^{1,3}_{a_i}$ est la relation binaire entre K_1 et K_3 pour l'attribut a_i. Le nombre d'attributs peut être assez important et par conséquent l'extraction de tous les contextes dyadiques afférents peut s'avérer extrêmement coûteuse. Nous restons ainsi réticent à l'inclusion de ce type d'implications pour les données massives bien que nous en tenons compte dans notre prototype sur l'exemple courant.

Il est à noter que les implications selon la définition de Biedermann restent difficiles à obtenir directement des contextes dyadiques quand le sous-ensemble de conditions C n'est pas limité à une seule condition à moins qu'elles coïncident avec celles à la **Ganter**. Cependant, l'avantage des implications **CAI** et **ACI** par rapport à celles de Biedermann est le fait qu'elles sont moins nombreuses et plus compactes et qu'elles véhiculent une sémantique plus riche que les secondes.

Rappelons qu'à l'instar des autres méthodes de calcul des concepts triadiques présentées à la section 3.2, nous n'ambitionnons pas de retrouver tous les concepts triadiques du contexte étudié mais, plutôt les concepts les plus significatifs. Ce point fera partie de développements ultérieurs suite à des tests sur des données massives.

Nous pouvons maintenant détailler notre approche qui se décline à travers les étapes suivantes :

i. **Étape 1 :** Génération des contextes dyadiques,

ii. **Étape 2 :** Extraction des bases d'implications génériques pour chaque contexte dyadique,

iii. **Étape 3 :** Extraction des implications triadiques à partir des ensembles des implications dyadiques générées à l'étape 2,

iv. **Étape 4 :** Extraction des concepts triadiques à partir des implications triadiques générées à l'étape précédente.

Examinons maintenant un peu plus en détail ces quatre étapes. Pour illustrer notre approche, nous prenons comme exemple le contexte triadique du tableau 2.6. Pour rappel, ce contexte représente des données tridimensionnelles : les clients 1 à 5, les fournisseurs **P**ierre, Nelson, **R**ichard, **K**evin et **S**imon, et les produits *a(accessories)*, *b(book)*, *c (computer)* et *d(digital camera)*.

4.2.1 Étape 1 : Génération des contextes dyadiques

Soit le contexte triadique $\mathbb{K} = (K_1, K_2, K_3, Y)$ avec K_1 un ensemble d'objets, alors que $K_2 = \{a_1, ...a_m\}$ et $K_3 = \{c_1, ...c_n\}$ sont des ensembles d'attributs et de conditions respectivement. La première étape consiste à produire les $\|K_3\|$ contextes dyadiques $\mathbb{K}^{1,2}_{c_i} := (K_1, K_2, Y^{1,2}_{c_i})$ pour tout $c_i \in K_3$. Ainsi pour notre exemple du tableau 2.6, nous pouvons générer les quatre contextes illustrés sur la figure 4.6.

$\mathbb{K}^{1,2}_a$	P	N	R	K	S
1	×	×	×	×	×
2	×		×	×	
3	×		×	×	×
4	×		×	×	
5	×	×	×	×	×

$\mathbb{K}^{1,2}_b$	P	N	R	K	S
1	×	×		×	
2		×	×		
3	×		×	×	
4	×	×	×	×	
5			×	×	

$\mathbb{K}^{1,2}_c$	P	N	R	K	S
1			×		
2		×			
3					
4					
5				×	

$\mathbb{K}^{1,2}_d$	P	N	R	K	S
1	×	×			
2	×	×	×	×	×
3	×	×			
4	×	×			×
5	×	×	×		

FIGURE 4.6: Les contextes dyadiques extraits de $\mathbb{K}$ pour chaque condition

Comme nous nous intéressons aussi aux implications **ACI**, nous devons aussi générer les contextes $\mathbb{K}^{1,3}_{a_i} := (K_1, K_3, Y^{1,3}_{a_i})$ représentant les relations binaires entre

K_1 et K_3 pour chaque attribut $a_i \in K_2$. Pour notre exemple courant, cette étape permet de générer les cinq contextes illustrés sur la figure 4.7.

$\mathbb{K}_P^{1,3}$	a	b	c	d
1	×	×		×
2	×			×
3	×	×		×
4	×	×		×
5	×			×

$\mathbb{K}_N^{1,3}$	a	b	c	d
1	×	×		×
2		×	×	×
3				×
4		×		×
5	×			×

$\mathbb{K}_R^{1,3}$	a	b	c	d
1	×		×	
2	×	×		×
3	×	×		
4	×	×		
5	×	×		×

$\mathbb{K}_K^{1,3}$	a	b	c	d
1	×	×		
2	×			×
3	×	×		
4	×	×		
5	×	×	×	

$\mathbb{K}_S^{1,3}$	a	b	c	d
1	×			
2				×
3	×			
4				×
5	×			

FIGURE 4.7: Les contextes dyadiques extraits de $\mathbb{K}$ pour chaque attribut

4.2.2 Étape 2 : Génération des implications dyadiques

La deuxième étape consiste à générer pour tous les contextes dyadiques obtenus à l'étape précédente les bases génériques pour les implications de support non nul. Pour les quatre contextes $\mathbb{K}_{c_i}^{1,2}$ avec $c_i \in K_3$, nous obtenons les implications présentées dans le tableau 4.1.

Tableau 4.1: Bases génériques des implications pour les contextes $\mathbb{K}_{c_i}^{1,2}$ avec c_i une condition de K_3.

Contexte $\mathbb{K}_a^{1,2}$	Contexte $\mathbb{K}_b^{1,2}$	Contexte $\mathbb{K}_c^{1,2}$	Contexte $\mathbb{K}_d^{1,2}$
$\{K\} \xrightarrow{a} \{PR\}$	$\{KN\} \xrightarrow{b} \{P\}$		$\{N\} \xrightarrow{d} \{P\}$
$\{P\} \xrightarrow{a} \{KR\}$	$\{NP\} \xrightarrow{b} \{K\}$		$\{P\} \xrightarrow{d} \{N\}$
$\{R\} \xrightarrow{a} \{KP\}$	$\{P\} \xrightarrow{b} \{K\}$		$\{S\} \xrightarrow{d} \{NP\}$
$\{S\} \xrightarrow{a} \{KPR\}$	$\{KNR\} \xrightarrow{b} \{P\}$		$\{R\} \xrightarrow{d} \{NP\}$
$\{N\} \xrightarrow{a} \{KPRS\}$	$\{NPR\} \xrightarrow{b} \{K\}$		$\{K\} \xrightarrow{d} \{NPRS\}$
	$\{PR\} \xrightarrow{b} \{K\}$		$\{RS\} \xrightarrow{d} \{KNP\}$

De manière similaire, on peut calculer les bases génériques des implications pour les contextes dyadiques $\mathbb{K}_{a_i}^{1,3}$ représentant les relations binaires pour chacun des cinq attributs $a_i \in K_2$. Ces implications sont illustrées dans le tableau 4.2.

Tableau 4.2: Bases génériques des implications pour les contextes $\mathbb{K}_{a_i}^{1,3}$ de chaque attribut a_i.

Contexte $\mathbb{K}_P^{1,3}$	Contexte $\mathbb{K}_N^{1,3}$	Contexte $\mathbb{K}_R^{1,3}$	Contexte $\mathbb{K}_K^{1,3}$	Contexte $\mathbb{K}_S^{1,3}$
$\{d\} \xrightarrow{P} \{a\}$	$\{b\} \xrightarrow{N} \{d\}$	$\{b\} \xrightarrow{R} \{a\}$	$\{d\} \xrightarrow{K} \{a\}$	
$\{b\} \xrightarrow{P} \{a, d\}$	$\{c\} \xrightarrow{N} \{b, d\}$	$\{d\} \xrightarrow{R} \{a, b\}$	$\{c\} \xrightarrow{K} \{a, b\}$	
$\{a\} \xrightarrow{P} \{d\}$	$\{a\} \xrightarrow{N} \{d\}$	$\{c\} \xrightarrow{R} \{a\}$	$\{b\} \xrightarrow{K} \{a\}$	
	$\{a, b\} \xrightarrow{N} \{d\}$			

4.2.3 Génération des implications triadiques

La troisième étape de notre approche consiste à construire des implications triadiques à partir des implications dyadiques obtenues à l'étape précédente. Notre procédure commence par regrouper les implications partageant la même partie gauche (la prémisse) en groupes ou baquets distincts (*buckets*).

a) Baquets de taille 2

Nous examinons d'abord le cas simple des baquets formés par seulement deux implications, soit $A \xrightarrow{c_1} B$, $ext1$ et $A \xrightarrow{c_2} D$, $ext2$, où $ext1$ représente les objets vérifiant la première implication. Dans ce cas, nous pouvons construire une troisième implication triadique de la manière suivante :

$$A \xrightarrow{c_1 \cup c_2} B \cap D, \quad ext$$

où $ext = ext1 \cap ext2$ représente une extension, c.à.d. un ensemble d'objets, vérifiant la nouvelle implication et donc son support absolu. Toutefois, cette implication n'est valide que lorsque $ext \neq \emptyset$. Si toutefois, $B \cap D$ est vide, nous gardons une telle implication qui signifie que les objets dans ext possèdent nécessairement les attributs contenus dans A sous les conditions décrites dans $c_1 \cup c_2$.

Par ailleurs, il faut observer que les deux implications dyadiques constituant le baquet initial peuvent former à leur tour des implications triadiques à la *Ganter* avec une conclusion maximale qui seront utiles pour notre calcul des concepts triadiques. Le traitement de ces deux implications dépendra des deux cas de figure suivants :

Cas 1 : $B \subseteq D$ **et** $ext1 \subseteq ext2$**.** Dans ce cas de traitement des deux implications, on obtient $A \xrightarrow{c_2} D, ext2$ et $A \xrightarrow{c_1 \cup c_2} B, ext1$. Cependant, $A \xrightarrow{c_1} B, ext1$ est écartée car elle devient redondante en présence de la nouvelle implication.

Cas 2 : $B \nsubseteq D$ **ou** $ext1 \nsubseteq ext2$**.** Il correspond à la négation du cas 1 et aboutit à l'ajout de la nouvelle implication $A \xrightarrow{c_1 \cup c_2} B \cap D, ext1 \cap ext2$ aux deux implications initiales $A \xrightarrow{c_1} B$ et $A \xrightarrow{c_2} D$.

À titre d'exemple, les deux implications $\{P\} \xrightarrow{b} \{K\}$, $\{134\}$ et $\{P\} \xrightarrow{a} \{KR\}$, $\{12345\}$ correspondent au cas 1 et aboutissent aux deux implications suivantes :
$\{P\} \xrightarrow{ab} \{K\}$, $\{134\}$
$\{P\} \xrightarrow{a} \{KR\}$, $\{12345\}$.

Prenons maintenant l'exemple du baquet formé par la prémisse $\{K\}$ et les deux implications du tableau 4.3 :

$\{K\} \xrightarrow{a} \{PR\}, \{12345\}$
$\{K\} \xrightarrow{d} \{NPRS\}, \{2\}$

Tableau 4.3: Baquet des implications dyadiques dont la prémisse est $\{K\}$

L'implication CAI obtenue dans ce cas est la suivante et indique que le client 2 vérifie cette implication :

$$\{K\} \xrightarrow{ad} \{PR\}, \quad \{2\}$$

Ce baquet correspond aussi au cas 2 puisque $\{PR\} \subseteq \{NPRS\}$ *mais* $\{12345\} \nsubseteq \{2\}$. Nous pouvons garder par conséquent les deux implications du baquet auquelles se rajoute la troisième et nouvelle implication dans le tableau 4.4 :

Implications CAI dont la prémisse est $\{K\}$
$\{K\} \xrightarrow{a} \{PR\}, \{12345\}$
$\{K\} \xrightarrow{d} \{\mathbf{N}PR\mathbf{S}\}, \{2\}$
$\{K\} \xrightarrow{ad} \{PR\}, \quad \{2\}$

Tableau 4.4: Implications triadiques dont la prémisse est $\{K\}$

Nous avons dû garder l'implication initiale $\{K\} \xrightarrow{d} \{\mathbf{N}PR\mathbf{S}\}, \{2\}$ car sa conclusion englobe celle de $\{K\} \xrightarrow{ad} \{PR\}, \{2\}$ pour la même extension. Prenons maintenant l'exemple des implications dont la prémisse est l'attribut $\{S\}$, soit le baquet du tableau 4.5 :

Baquet formé par la prémisse $\{S\}$
$\{S\} \xrightarrow{a} \{KPR\}, \{135\}$
$\{S\} \xrightarrow{d} \{NP\}, \{24\}$

Tableau 4.5: Implications dyadiques dont la prémisse est $\{S\}$

L'implication CAI obtenue dans ce cas est la suivante :

$$\{S\} \xrightarrow{a,d} \{P\}, \emptyset$$

Cette implication a un support nul et donc ne peut pas être retenue. Par contre, le baquet correspond au cas 2 aussi. Par conséquent, les deux implications initiales peuvent être gardées.

$\{S\} \xrightarrow{a} \{KPR\}, \{135\}$
$\{S\} \xrightarrow{d} \{NP\}, \{24\}$

Tableau 4.6: Implications triadiques dont la prémisse est $\{S\}$

b) Baquets de taille 3

Le cas des baquets de taille égale à trois va consister à considérer les implications deux à deux puis les trois implications en même temps. Prenons l'exemple du baquet formé par la prémisse $\{P\}$. Tel qu'indiqué dans le tableau 4.7, ce baquet est composé de trois implications :

Implication 1 :	$\{P\} \xrightarrow{a} \{RK\}, \{12345\}$
Implication 2 :	$\{P\} \xrightarrow{b} \{K\}, \{134\}$
Implication 3 :	$\{P\} \xrightarrow{d} \{N\}, \{12345\}$

Tableau 4.7: Baquet d'mplications dyadiques dont la prémisse est $\{P\}$

Par le même procédé (union des conditions et intersection des conclusions), on peut obtenir de nouvelles implications triadiques par la prise en compte des implications deux à deux puis les trois implications en même temps. Dans le dernier cas, on obtient l'implication triadique suivante :

$$\{P\} \xrightarrow{a,b,d} \emptyset, \{134\}$$

Tel qu'expliqué précédemment, même si cette implication a une conclusion vide, elle reste tout de même valable pour le calcul des concepts triadiques puisqu'elle a un support non nul et signifie que le fournisseur P livre les produits a, b et d aux clients 1, 3 et 4. Cela se traduit par la génération du concept $(134, P, abd)$.

En considérant maintenant les implications (cf. le tableau 4.7) du baquet deux par deux, les calculs suivants sont faits :

Implications 1 et 2 Si on prend les deux premières implications, on se trouve dans le premier cas. Nous aboutissons aux deux implications suivantes :

i. $\{P\} \xrightarrow{a,b} \{K\}, \{134\}$

ii. $\{P\} \xrightarrow{a} \{RK\}, \{12345\}$

iii. ~~$\{P\} \xrightarrow{b} \{K\}, \{134\}$~~

La troisième implication $\{P\} \xrightarrow{b} \{K\}, \{134\}$ n'est en revanche pas retenue car sa conclusion et son support absolu sont les mêmes que ceux de la première implication $\{P\} \xrightarrow{a,b} \{K\}, \{134\}$ de laquelle elle peut être inférée puisque $\{b\} \subset \{a \cup b\}$.

Implications 1 et 3 L'ensemble constitué par *Implication 1* ($\{P\} \xrightarrow{a} \{RK\}$, $\{12345\}$) et *Implication 3* ($\{P\} \xrightarrow{d} \{N\}$, $\{12345\}$) correspond au cas 2. Nous pouvons générer les trois implications valides suivantes :

i. $\{P\} \xrightarrow{a,d} \emptyset, \{12345\}$

ii. $\{P\} \xrightarrow{a} \{RK\}, \{12345\}$

iii. $\{P\} \xrightarrow{d} \{N\}, \{12345\}$

Implications 2 et 3 Finalement, l'ensemble formé par *Implication 2* ($\{P\} \xrightarrow{b} \{K\}$, $\{134\}$) et *Implication 3* ($\{P\} \xrightarrow{d} \{N\}$, $\{12345\}$) correspond aussi au cas 2. Nous pouvons générer à priori les trois implications suivantes :

i. ~~$\{P\} \xrightarrow{b,d} \emptyset, \{134\}$~~

ii. ~~$\{P\} \xrightarrow{b} \{K\}, \{134\}$~~

iii. $\{P\} \xrightarrow{d} \{N\}, \{12345\}$

Néanmoins, on peut faire les deux observations suivantes et en déduire par conséquent le traitement à réserver pour ces situations :

Observation 1 : La première implication i : $(\{P\} \xrightarrow{b,d} \emptyset, \{134\})$, obtenue par l'union des conditions et l'intersection des conclusions n'est pas maximale. En effet, elle peut être déduite de manière triviale de la première implication issue du traitement combiné des trois implications du baquet, soit $\{P\} \xrightarrow{a,b,d} \emptyset, \{134\}$. Pour cela, il suffit de noter qu'elle présente le même support et la même conclusion que cette dernière alors que sa condition en est incluse. Cette implication ne sera pas retenue et on gardera uniquement la toute première implication de ce calcul.

Observation 2 : La deuxième implication ii : $(\{P\} \xrightarrow{b} \{K\}, \{134\})$ n'est pas maximale non plus. En fait, la même implication n'avait pas été retenue lors du traitement de l'ensemble formé par *l'implication 1* et *l'implication 2* qui correspondait au cas 1. Cette implication n'étant pas maximale devrait être écartée. Seulement, si on restreint le traitement à l'ensemble formé par *l'implication 2* et *l'implication 3*, on n'est pas capable de l'écarter.

Finalement, les cinq (5) implications triadiques qui doivent être retenues pour ce baquet sont regroupées dans le tableau 4.8.

$\{P\} \xrightarrow{a,b,d} \emptyset, \{134\}$
$\{P\} \xrightarrow{a,b} \{K\}, \{134\}$
$\{P\} \xrightarrow{a} \{RK\}, \{12345\}$
$\{P\} \xrightarrow{a,d} \emptyset, \{12345\}$
$\{P\} \xrightarrow{d} \{N\}, \{12345\}$

Tableau 4.8: Implications triadiques dont la prémisse est $\{P\}$

c) Baquets de taille $n > 3$

Le traitement des baquets de taille $n > 3$ va suivre le même principe que celui des baquets de taille 3. On va considérer les n implications en même temps pour

éventuellement obtenir une nouvelle implication triadique dont la taille de la condition vaut n. Ensuite, on considère les diverses combinaisons de p parmi n en faisant varier p de $n-1$ à 2. De manière similaire, les implications doivent être scrutées pour écarter celles qu'on peut associer aux *observations* 1 et 2 du paragraphe précédent.

Algorithme 2 : SamePremiseSet (*Buck(A) , FirstImp)*

Input : Un Baquet de taille n : $Buck(\mathrm{A}) = \{A \xrightarrow{c_i} B_i, Ext_i\}, i \in \{1..n\}$ et *FirstImp* : l'implication triadique issue de traitement de toutes les implications

Output : $\sum$ l'ensemble des implications triadiques à la Ganter

```
1: ∑ ← { FirstImp }
2: if Buck(A)≠ ∅ then
3:   for imp ∈ Buck(A) do
4:     Temp ← Buck(A)
5:     Temp ← Temp \{ imp}  {On enlève l'implication imp du baquet}
6:     if Temp ≠ ∅ then
7:       NewImp ← JointAll(temp)  {traitement de toutes les implications du baquet
         Temp}
8:       if Conc(NewImp) ≠ Conc(NewImp) or Ext(NewFirst) ≠ Ext(FirstImp) then
9:         NewFirst ← NewImp
10:      else
11:        NewFirst ← FirstImp
12:      end if
13:      ∑ ← ∑ ∪ SamePremiseSet (Temp, NewFirst)
14:    else
15:      ∑ ← ∑ ∪ SamePremiseSet (Temp, FirstImp)
16:  end for
17: return ∑
```

L'algorithme 2 résume cette approche. La construction de l'ensemble des implications triadiques se fait par le biais d'une fonction récursive *SamePremiseSet* qui prend deux arguments. Le premier est le baquet de taille n : $Buck(A)_n$ qui regroupe les n implications $imp_i, 1 < i < n$ dont la prémisse est le sous-ensemble A. Le deuxième argument est la toute première implication *FirstImp* obtenue par la simple intersection de toutes les extensions, l'intersection des conclusions et l'union de toutes les conditions des implications imp_i. Notons *JointAll* la fonction qui permet de calculer

cette implication à partir d'un baquet *Buck(A)*$_n$. On peut écrire :

$$FirstImp := JointAll(Buck(A)) = A \xrightarrow{\bigcup_1^n c_i} \bigcap_1^n B_i, \bigcap_1^n Ext_i$$

La fonction *SamePremiseSet* est conçue pour écarter les implications associées à l'observation 1. Lors de chaque appel récursif de cette fonction, le deuxième argument *FirstImp* est d'abord testé (ligne 8 de l'algorithme) puis ensuite il est actualisé ou conservé en conséquence. Une illustration de l'exécution partielle de la fonction *SamePremiseSet* pour un baquet constitué de trois implications est présentée dans les tableaux 4.9 et 4.10.

Pour le traitement des implications associées à l'observation 2, nous choisissons de le faire à l'extérieur du bloc de la fonction *SamePremiseSet* afin de bénéficier de la puissance du traitement parallèle. Il suffit pour cela de s'assurer que lorsque deux triplets [1] sont égaux sur deux dimensions, de ne garder que le triplet qui possède "la plus grande " troisième dimension au sens de l'inclusion. Plus précisément, si deux implications ont la même prémisse et la même condition par exemple, on garde seulement l'implication qui a la plus grande conclusion. En notation formelle, si on prend par exemple $ct_1 = (X_1, Y_1, Z_1)$ et $ct_2 = (X_2, Y_2, Z_2)$ deux triplets tels que $X_1 = X_2$, $Y_1 = Y_2$ et $Z_2 \subset Z_1$, notre algorithme ne doit garder que le triplet ct_1 comme concept triadique pour la suite de notre calcul.

En mode parallèle, ce traitement supplémentaire peut être réalisé en deux étapes par une approche *MapReduce*. En première étape, on applique une fonction *Map* au triplet (X, Y, Z) pour créer des paires clé-valeur $((X, Y), Z)$. La *clé* (X, Y) est construite à partir de deux dimensions, le troisième élément du triplet Z est considéré comme la *valeur* de notre paire clé-valeur. La deuxième étape *Reduce* permet de fusionner les valeurs associées à des clés identiques à l'aide d'une fonction de réduction associative qui est dans ce cas l'union. Cette opération retournera forcement le triplet avec la plus grande valeur au sens de l'inclusion. Aussi, ce traitement permet de

1. On peut raisonner aussi bien sur les implications que sur les concepts triadiques.

Tableau 4.9: Illustration d'une exécution partielle de la fonction *SamePremiseSet*

It	*Buck(d)*	*imp*	*BuckTemp*	*NewFirst*	*Test*	*FirstImp*	$\sum$
0	$(i)\ d \xrightarrow{P} a, 12345$ $(ii)\ d \xrightarrow{R} ab, 25$ $(iii)\ d \xrightarrow{K} a, 2$					$d \xrightarrow{PRK} a, 2$	$d \xrightarrow{PRK} a, 2$
1	$(i)\ d \xrightarrow{P} a, 12345$ $(ii)\ d \xrightarrow{R} ab, 25$ $(iii)\ d \xrightarrow{K} a, 2$	(i)	$(ii)\ d \xrightarrow{R} ab, 25$ $(iii)\ d \xrightarrow{K} a$	$d \xrightarrow{RK} a, 2$	*Non*	$d \xrightarrow{PRK} a, 2$	$d \xrightarrow{PRK} a, 2$
2	$(ii)\ d \xrightarrow{R} ab, 25$ $(iii)\ d \xrightarrow{K} a, 2$	(ii)	$(iii)\ d \xrightarrow{K} a, 2$	$d \xrightarrow{K} a, 2$	*Non*	$d \xrightarrow{PRK} a, 2$	$d \xrightarrow{PRK} a, 2$
3	$(iii)\ d \xrightarrow{K} a, 2$	(iii)	$\emptyset$	*non calculé*		$d \xrightarrow{PRK} a, 2$	$d \xrightarrow{PRK} a, 2$
4	$(ii)\ d \xrightarrow{R} ab, 25$ $(iii)\ d \xrightarrow{K} a, 2$	(iii)	$(ii)\ d \xrightarrow{R} ab, 25$	$d \xrightarrow{K} ab, 25$	*oui*	$d \xrightarrow{K} ab, 25$	$d \xrightarrow{PRK} a, 2$ $d \xrightarrow{K} ab, 25$
5	$(ii)\ d \xrightarrow{R} ab, 25$	(ii)	$\emptyset$	*non calculé*		$d \xrightarrow{R} ab, 25$	$d \xrightarrow{PRK} a, 2$ $d \xrightarrow{K} ab, 25$
6	$(i)\ d \xrightarrow{P} a, 12345$ $(ii)\ d \xrightarrow{R} ab, 25$ $(iii)\ d \xrightarrow{K} a, 2$	(ii)	$(i)\ d \xrightarrow{P} a, 12345$ $(iii)\ d \xrightarrow{K} a$	$d \xrightarrow{KP} a, 2$	*Non*	$d \xrightarrow{PRK} a, 2$	$d \xrightarrow{PRK} a, 2$ $d \xrightarrow{K} ab, 25$
7	$(i)\ d \xrightarrow{P} a, 12345$ $(iii)\ d \xrightarrow{K} a, 2$	(i)	$(iii)\ d \xrightarrow{K} a, 2$	$d \xrightarrow{K} a, 2$	*Non*	$d \xrightarrow{PRK} a, 2$	$d \xrightarrow{PRK} a, 2$ $d \xrightarrow{K} ab, 25$
8	$(iii)\ d \xrightarrow{K} a, 2$	(iii)	$\emptyset$	*non calculé*		$d \xrightarrow{PRK} a, 2$	$d \xrightarrow{PRK} a, 2$ $d \xrightarrow{K} ab, 25$
9	$(i)\ d \xrightarrow{P} a, 12345$ $(iii)\ d \xrightarrow{K} a, 2$	(iii)	$(i)\ d \xrightarrow{P} a, 12345$	$d \xrightarrow{P} a, 12345$	*oui*	$d \xrightarrow{P} a, 12345$	$d \xrightarrow{PRK} a, 2$ $d \xrightarrow{K} ab, 25$ $d \xrightarrow{PRK} a, 12345$
10	$(i)\ d \xrightarrow{P} a, 12345$	(i)	$\emptyset$	*non calculé*		$d \xrightarrow{PRK} a, 12345$	$d \xrightarrow{PRK} a, 2$ $d \xrightarrow{K} ab, 25$ $d \xrightarrow{PRK} a, 12345$

Tableau 4.10: Illustration d'une exécution partielle de la fonction *SamePremiseSet*

It	*Buck(d)*	*imp*	*BuckTemp*	*NewFirst*	*Test*	*FirstImp*	$\sum$
11	$(i)\ d \xrightarrow{P} a, 12345$ $(ii)\ d \xrightarrow{R} ab, 25$ $(iii)\ d \xrightarrow{K} a, 2$	(iii)	$(i)\ d \xrightarrow{P} a, 12345$ $(ii)\ d \xrightarrow{R} ab, 25$	$d \xrightarrow{PR} a, 25$	*oui*	$d \xrightarrow{PR} a, 25$	$d \xrightarrow{PRK} a, 2$ $d \xrightarrow{K} ab, 25$ $d \xrightarrow{P} a, 12345$ $d \xrightarrow{PR} a, 25$
12	$(i)\ d \xrightarrow{P} a, 12345$ $(ii)\ d \xrightarrow{R} ab, 25$	(i)	$(ii)\ d \xrightarrow{R} ab, 25$	$(ii)\ d \xrightarrow{R} ab, 25$	*oui*	$(ii)\ d \xrightarrow{R} ab, 25$	$d \xrightarrow{PRK} a, 2$ $d \xrightarrow{K} ab, 25$ $d \xrightarrow{P} a, 12345$ $d \xrightarrow{PR} a, 25$
13	$(ii)\ d \xrightarrow{R} ab, 25$	(ii)	$\emptyset$	*non calculé*		$d \xrightarrow{K} ab, 25$	$d \xrightarrow{PRK} a, 2$ $d \xrightarrow{K} ab, 25$ $d \xrightarrow{P} a, 12345$ $d \xrightarrow{PR} a, 25$
14	$(i)\ d \xrightarrow{P} a, 12345$ $(ii)\ d \xrightarrow{R} ab, 25$	(ii)	$(i)\ d \xrightarrow{P} a, 12345$	$d \xrightarrow{P} a, 12345$	*oui*	$d \xrightarrow{P} a, 12345$	$d \xrightarrow{PRK} a, 2$ $d \xrightarrow{K} ab, 25$ $d \xrightarrow{P} a, 12345$ $d \xrightarrow{PR} a, 25$
15	$(i)\ d \xrightarrow{P} a, 12345$	(i)	$\emptyset$	*non calculé*		$d \xrightarrow{PRK} a, 12345$	$d \xrightarrow{PRK} a, 2$ $d \xrightarrow{K} ab, 25$ $d \xrightarrow{PRK} a, 12345$ $d \xrightarrow{PR} a, 25$

garder uniquement les triplets maximaux sur la dimension Z et doit être répété bien entendu pour les deux autres dimensions pour écarter tous les triplets non maximaux.

Les résultats de cette étape pour l'ensemble des baquets issus du contexte étudié se déclinent au niveau des tableaux 4.11 et 4.12. Le premier tableau 4.11 résume le traitement des baquets issus des implications CAI. Le deuxième tableau est réservé par contre au traitement des implications ACI. La troisième colonne de ces deux tableaux présente en revanche les concepts obtenus lors de l'étape 4 qu'on aborde ci-après.

4.2.4 Étape 4 : Calcul des concepts triadiques

L'extraction des concepts triadiques à partir des implications triadiques se base sur les propositions 2.2 et 2.3 du chapitre 2. Soit $(\{B\} \xrightarrow{C} \{D\}, A)$ une implication **CAI** *(entre attributs sous condition)* issue d'un contexte triadique $\mathbb{K} = (K_1, K_2, K_3, Y)$. Autrement dit, B et D sont deux sous-ensembles d'attributs de K_2, C un sous-ensemble de K_3 et A un sous-ensemble non vide d'objets de K_1 représentant l'ensemble des objets qui vérifient (offrent un support à) cette implication. Il faut noter qu'une implication CAI (dans le sens du Ganter) est aussi une implication dans le sens de Biedermann mais l'inverse n'est pas vrai. Rappelons qu'ayant adopté les bases génériques, les prémisses de nos implications sont formées par les générateurs minimaux dyadiques. On peut déduire que le triplet $(A, B \cup D, C)$ est un concept triadique. A est *l'extension* du concept et coïncide avec les objets qui vérifient l'implication, *l'intention* est l'union de la prémisse et la conclusion de l'implication alors que le *modus* est formé par la condition C.

À leur tour, les implications triadiques **ACI** *(entre conditions pour attribut)* permettent de générer des concepts triadiques de manière légèrement différente. Ainsi pour une implication **ACI** : $(\{b\} \xrightarrow{C}: \{d\}, A)$, on peut obtenir un concept triadique en formant le triplet $(A, C, b \cup d)$.

Ainsi pour l'implication $\{K\} \xrightarrow{a,d} \{P, R\}$, on peut déduire que la paire (KPR, ad) est constituée de l'intention et du modus d'un concept triadique si le support de cette

Tableau 4.11: Traitement des baquets regroupant les implications entre attributs du contexte du tableau 2.6

Baquets	**Implications triadiques calculées**	**Concepts obtenus**
$\{K\} \xrightarrow{a} \{P,R\},\{1,2,3,4,5\}$	$\{K\} \xrightarrow{a,d} \{R,P\},\{2\}$	$(2, KPR, ad)$
$\{K\} \xrightarrow{d} \{N,R,S,P\},\{2\}$	$\{K\} \xrightarrow{a} \{P,R\},\{1,2,3,4,5\}$	$(12345, KPR, a)$
	$\{K\} \xrightarrow{d} \{N,R,S,P\},\{2\}$	$(2, KNPRS, d)$
$\{P\} \xrightarrow{a} \{R,K\},\{1,2,3,4,5\}$	$\{P\} \xrightarrow{a,b,d} \emptyset,\{134\}$	$(134, P, abd)$
$\{P\} \xrightarrow{b} \{K\},\{1,3,4\}$	$\{P\} \xrightarrow{a,b} \{K\},\{134\}$	$(134, PK, ab)$
$\{P\} \xrightarrow{d} \{N\},\{1,2,3,4,5\}$	$\{P\} \xrightarrow{a} \{RK\},\{12345\}$	$(12345, PRK, a)$
	$\{P\} \xrightarrow{a,d} \emptyset,\{12345\}$	$(12345, P, ad)$
	$\{P\} \xrightarrow{d} \{N\},\{12345\}$	$(12345, PN, d)$
$\{R\} \xrightarrow{d} \{N,P\},\{2,5\}$	$\{R\} \xrightarrow{a,d} \{P\},\{2,5\}$	$(25, RP, ad)$
$\{R\} \xrightarrow{a} \{K,P\},\{1,2,3,4,5\}$	$\{R\} \xrightarrow{d} \{N,P\},\{2,5\}$	$(25, NPR, d)$
	$\{R\} \xrightarrow{a} \{K,P\},\{1,2,3,4,5\}$	$(12345, KPR, a)$
$\{S\} \xrightarrow{a} \{P,R,K\},\{1,3,5\}$	$\{S\} \xrightarrow{a} \{P,R,K\},\{1,3,5\}$	$(135, PKRS, a)$
$\{S\} \xrightarrow{d} \{N,P\},\{2,4\}$	$\{S\} \xrightarrow{d} \{N,P\},\{2,4\}$	$(24, NPS, d)$
$\{N\} \xrightarrow{d} \{P\},\{1,2,3,4,5\}$	$\{N\} \xrightarrow{a,d} \{P\},\{1,5\}$	$(15, NP, ad)$
$\{N\} \xrightarrow{a} \{R,K,S,P\},\{1,5\}$	$\{N\} \xrightarrow{a} \{P\},\{1,2,3,4,5\}$	$(12345, NP, d)$
	$\{N\} \xrightarrow{a} \{R,K,S,P\},\{1,5\}$	$(15, NKRSP, a)$
$\{N,P\} \xrightarrow{b} \{K\},\{1,4\}$	$\{N,P\} \xrightarrow{b} \{K\},\{1,4\}$	$(14, NPK, b)$
$\{R,P\} \xrightarrow{b} \{K\},\{3,4\}$	$\{R,P\} \xrightarrow{b} \{K\},\{3,4\}$	$(34, RPK, b)$
$\{R,S\} \xrightarrow{d} \{N,K,P\},\{2\}$	$\{R,S\} \xrightarrow{d} \{N,K,P\},\{2\}$	$(2, RSNKP, d)$
$\{R,K\} \xrightarrow{d} \{N,S,P\},\{2\}$	$\{R,K\} \xrightarrow{d} \{N,S,P\},\{2\}$	$(2, RSNKP, d)$
$\{N,R,K\} \xrightarrow{b} \{P\},\{4\}$	$\{N,R,K\} \xrightarrow{b} \{P\},\{4\}$	$(4, RNKP, b)$
$\{N,R,P\} \xrightarrow{b} \{K\},\{4\}$	$\{N,R,P\} \xrightarrow{b} \{K\},\{4\}$	$(4, RNKP, b)$
$\{N,K\} \xrightarrow{b} \{P\},\{1,4\}$	$\{N,K\} \xrightarrow{b} \{P\},\{1,4\}$	$(14, NKP, b)$

implication est non nul et que la conclusion est maximale, ce qui est bien le cas pour cet exemple où on obtient le concept triadique $(2, KPR, ad)$.

À l'opposé, l'implication $\{c\} \xrightarrow{N,K} \{b\}$ qui a un support nul ne permet pas de générer de concept triadique. Cette approche se résume au niveau du pseudo-code 3.

Tableau 4.12: Traitement des baquets regroupant les implications entre conditions du contexte du tableau 2.6

Baquets	**Implications triadiques calculées**	**Concepts obtenus**
$\{a\} \xrightarrow{P} \{d\}, \{1,2,3,4,5\}$	$\{a\} \xrightarrow{N,P} \}, \{1,5\}$	$(15, NP, ad)$
$\{a\} \xrightarrow{N} \{d\}, \{1,5\}$	$\{a\} \xrightarrow{P} \{d\}, \{1,2,3,4,5\}$	$(12345, P, ad)$
$\{b\} \xrightarrow{P} \{a,d\}, \{1,3,4\}$	$\{b\} \xrightarrow{PNRK} \emptyset, \{4\}$	$(4, PNRK, b)$
$\{b\} \xrightarrow{N} \{d\}, \{1,2,4\}$	$\{b\} \xrightarrow{PN} \{d\}, \{1,4\}$	$(14, PN, bd)$
$\{b\} \xrightarrow{R} \{a\}, \{2,3,4,5\}$	$\{b\} \xrightarrow{P} \{a,d\}, \{1,3,4\}$	$(134, P, abd)$
$\{b\} \xrightarrow{K} \{a\}, \{1,3,4,5\}$	$\{b\} \xrightarrow{N} \{d\}, \{1,2,4\}$	$(124, N, bd)$
	$\{b\} \xrightarrow{N,R} \emptyset, \{2,4\}$	$(24, NR, b)$
	$\{b\} \xrightarrow{R} \{a\}, \{2,3,4,5\}$	$(2345, R, ab)$
	$\{b\} \xrightarrow{P,R,K} \{a\}, \{3,4\}$	$(34, PRK, ab)$
	$\{b\} \xrightarrow{P,N,K} \emptyset, \{1,4\}$	$(14, KNP, b)$
	$\{b\} \xrightarrow{P,K} \{a\}, \{1,3,4\}$	$(134, PK, ab)$
	$\{b\} \xrightarrow{K} \{a\}, \{1,3,4,5\}$	$(1345, K, ab)$
	$\{b\} \xrightarrow{R,K} \{a\}, \{3,4,5\}$	$(345, RK, ab)$
$\{d\} \xrightarrow{P} \{a\}, \{1,2,3,4,5\}$	$\{d\} \xrightarrow{PRK} \{a\}, \{2\}$	$(2, PRK, ad)$
$\{d\} \xrightarrow{R} \{a,b\}, \{2,5\}$	$\{d\} \xrightarrow{P} \{a\}, \{1,2,3,4,5\}$	$(12345, P, a)$
$\{d\} \xrightarrow{K} \{a\}, \{2\}$	$\{d\} \xrightarrow{R} \{a,b\}, \{2,5\}$	$(25, R, abd)$
	$\{d\} \xrightarrow{PR} \{a\}, \{2,5\}$	$(25, PR, ad)$
$\{c\} \xrightarrow{N} \{b,d\}, \{2\}$	$\{c\} \xrightarrow{N} \{b,d\}, \{2\}$	$(2, N, bcd)$
$\{c\} \xrightarrow{R} \{a\}, \{1\}$	$\{c\} \xrightarrow{R} \{a\}, \{1\}$	$(1, R, ac)$
$\{c\} \xrightarrow{K} \{a,b\}, \{5\}$	$\{c\} \xrightarrow{K} \{a,b\}, \{5\}$	$(5, K, adc)$
$\{a,b\} \xrightarrow{N} \{d\}, \{1\}$	$\{a,b\} \xrightarrow{N} \{d\}, \{1\}$	$(1, N, abd)$

Algorithme 3 : *RuleToConcept (imp)*

Input : Une implication triadique $(\{B\} \xrightarrow{C} \{D\}, A)$ et son type T

Output : Concept triadique ct

if T = 'CAI' **then**
    return $ct = (A, B \cup D, C)$
if T = 'ACI' **then**
    return $ct = (A, C, B \cup D)$

Si on parcourt la troisième colonne des tableaux 4.11 et 4.12, on peut noter une certaine redondance au niveau des concepts obtenus. Ainsi par exemple, le concept triadique $(15, NP, ad)$ est obtenu une première fois à partir de l'implication CAI $\{N\} \xrightarrow{a,d} \{P\}, \{1,5\}$ mais il peut aussi être obtenu à partir de l'implication ACI $\{a\} \xrightarrow{N,P} d\}, \{1,5\}$. Cette redondance était prévisible du fait qu'un même générateur et un même concept triadique peuvent générer à la fois une implication CAI et une implication ACI. Elle reste pourtant assez facile à corriger par une simple fonction de filtration qui retourne des résultats distincts.

En supprimant la redondance, les concepts obtenus par cette procédure sont fournis par le tableau 4.13.

Tableau 4.13: Concepts triadiques obtenus

Concepts triadiques			
(1, R, ac)	(12345, KPR, a)	(12345, NP, d)	(12345, P, ad)
(124, N, bd)	(134, P, abd)	(134, PK, ab)	(1345, K, ab)
(135, PKRS, a)	(14, NKP, b)	(14, PN, bd)	(15, NKRSP, a)
(15, PN, ad)	(2, KNPRS, d)	(2, KPR, ad)	(2, N, bcd)
(2345, R, ab)	(24, NPS, d)	(24, NR, b)	(25, NPR, d)
(25, R,abd)	(25, RP, ad)	(345, RK, ab)	(4, PNRK, b)
(5, K, abc)	(34, PRK, ab)	(1 PN, abd)	

4.2.5 Cas particulier

Le cas de l'implication $\{a, b\} \xrightarrow{N} \{d\}, \{1\}$, la dernière du tableau 4.12, est particulièrement intéressant. En effet, si on considère le générateur lié à cette implication, $\{N, ab\}$, on peut constater qu'il n'est pas le *t-générateur* du concept triadique généré par la méthode expliquée au 4.2.4, soit le triplet $(1, N, abd)$. Plus précisément, si on se réfère à la définition de la section 2.4.2, on peut vérifier que $(N, abd) \notin ((N, ab)^1)^1 = \{(PN, abd)\}$. En réalité, le concept triadique en lien avec le *t-générateur* $\{N, ab\}$ est plutôt $(1, PN, abd)$. Ce cas nous enseigne sur la dernière

vérification à effectuer sur les triplets calculés pour s'assurer qu'il s'agit bien d'un concept triadique.

Une première option consiste à vérifier pour chaque concept calculé $c = (A_1, A_2, A_3)$ si $(A_2, A_3) \in ((U_2, U_3)^1)^1$ avec (U_2, U_3) le générateur lié à l'implication qui génère le concept c. Cependant, le coût de calcul de $((U_2, U_3)^1)^1$ peut s'avérer assez important dans le contexte des données massives.

La deuxième option qu'on va adopter dans le cadre de ce projet consiste à parcourir la liste des concepts calculés et vérifier si les triplets peuvent être augmentés. Par exemple, notre liste de triplets contient à la fois les deux triplets $(1, N, abd)$ et $(134, P, abd)$. On peut en déduire qu'on peut augmenter le triplet $(1, N, abd)$ par P puisqu'on voit bien que l'objet 1 possède aussi l'attribut P sous les conditions abd car l'extension du premier concept est incluse dans celle du deuxième concept. Ainsi, c'est bien le triplet $(1, PN, abd)$ qui constitue un concept triadique. D'une manière générale, en présence de deux concepts triadiques $ct_i = (A_1, A_2, A_3)$ et $ct_j = (B_1, B_2, B_3)$ tel que $A_3 = B_3$ et $A_1 \subset B_1$, alors ct_i devient égal à $(A_1, A_2 \cup B_2, A_3)$. Un même raisonnement peut être fait pour une égalité des intentions et une inclusion stricte des extensions.

Ce traitement peut être réalisé encore par une approche *MapReduce*. La fonction *Map* permet de créer des paires clé-valeur avec comme clés les modus des concepts et comme valeur les couples *(extension, intention)*. La fonction *Reduce* permet par la suite d'agréer les valeurs ayant la même clé en une seule liste. Les listes obtenues sont par la suite traitées par la fonction récursive *TestSpecialCase* qui teste lorsque les modus de deux concepts sont identiques si l'intention (ou respectivement l'extension) de l'un est incluse dans l'autre. Dans ce cas, elle augmente l'extension (ou l'intention) de ce dernier par l'union.

Par exemple, soit (x_1, y_1, z_1) et (x_2, y_2, z_2) deux triplets de sorte que $z_1 = z_2$. Dans ce cas, si $y_1 \subset y_2$, alors on peut augmenter x_1 par x_2, soit $(x_1 \cup x_2, y_1, z_1)$ est un concept triadique. Pour cet exemple, la fonction *Reduce* discutée en haut permet de retourner la liste suivante $[(X_1, Y_1), (X_2, Y_2)]$. Le pseudo-code du bloc principal de la fonction *TestSpecialCase* se décline comme suit :

Algorithme 4 : Fonction *TestSpecialCase*($[(X_1, Y_1), (X_2, Y_2)]$)

Input : Liste de paires $[(X_1, Y_1), (X_2, Y_2)]$
Output : Liste de paires maximales

if $X_1 \subset X_2$ **then**
  return $[(X_1, Y_1 \cup Y_2), (X_2, Y_2)]$
if $Y_1 \subset Y_2$ **then**
  return $[(X_1 \cup X_2, Y_1), (X_2, Y_2)]$
if $X_2 \subset X_1$ **then**
  return $[(X_1, Y_1), (X_2, Y_2 \cup Y_1)]$
if $Y_2 \subset Y_1$ **then**
  return $[(X_1, Y_1), (X_2 \cup X_1, Y_2)]$

Pour les listes de taille supérieure à deux, la fonction *TestSpecialCase* va s'appeler de manière récursive jusqu'à ce que la liste produite ne subisse aucun changement.

4.3 Discussion

L'approche adoptée permet de générer une bonne partie des concepts triadiques. Ainsi 27 concepts ont pu être calculés sur un total de 30 concepts autres que les trois concepts ayant comme valeur de l'extension, de l'intention ou du modus l'ensemble viide. Il s'agit de $(\emptyset, NKPRS, abcd)$, $(12345, \emptyset, abcd)$, et $(12345, NKPRS, \emptyset)$. Les concepts manquants sont les suivants :

Tableau 4.14: Concepts triadiques non calculés

Concepts triadiques non calculés		
$(1, PNK, ab)$	$(2, NR, bd)$	$(5, PNR, ad)$

Cependant, nous constatons que les concepts manquants sont ceux issus d'implications que nous pouvons déduire par l'un des axiomes d'inférence définis dans [43],

notamment, l'axiome d'inférence suivant :

$$\begin{cases} X \xrightarrow{c_1} Y \\ Z \xrightarrow{c_2} W \end{cases} \vDash X \cup Z \xrightarrow{c_1 \cup c_2} Y \cap W$$

En appliquant cet axiome aux deux implications $\{R\} \xrightarrow{d} \{NP\}$ et $\{N\} \xrightarrow{a} \{KRPS\}$, on obtient l'implication suivante : $\{RN\} \xrightarrow{ad} \{P\}$, ce qui nous permet d'obtenir le concept $(5, PNR, ad)$.

Ce même axiome peut être aussi appliqué aux implications $\{NK\} \xrightarrow{b} \{P\}$ et $\{N\} \xrightarrow{a} \{KRPS\}$, ce qui permet de générer l'implication suivante $\{NK\} \xrightarrow{ab} \{P\}$ et par conséquent le concept $(1, PNK, ab)$. Toutefois, le concept $(2, NR, bd)$ ne peut pas être trouvé à partir d'une implication "à la Ganter" qui ne correspond pas à une implication "à la Biedermann".

À l'issue de cette première phase de ce projet de recherche, nous pouvons conclure que notre nouvelle approche de calcul des motifs à partir d'un contexte formel triadique nous permet de produire en premier lieu des implications triadiques *à la Ganter* pour ensuite déterminer les concepts triadiques. Cette approche convertit d'abord un contexte triadique initial en autant de contextes dyadiques qu'il y a de conditions en vue d'un traitement parallèle pour ensuite obtenir des implications dyadiques qui permettent de former les implications triadiques. L'approche proposée est opposée à la pratique courante puisqu'elle permet d'obtenir les concepts à partir des implications et non pas l'inverse. Nous pensons être en mesure de retrouver la plupart des concepts triadiques et si possible caractériser formellement ceux (généralement peu nombreux) qui ne peuvent être obtenus à partir de la base des implications ou par application d'axiomes d'inférence. Nous avons besoin néanmoins de tester cette approche sur des collections de données volumineuses avant de tirer toute conclusion. La suite des travaux continue avec la finalisation de la programmation des algorithmes sur l'environnement PySpark.

Ce travail trouve son application dans plusieurs domaines comme l'analyse de réseaux sociaux tridimensionnels, les folksonomies représentant des usagers et les mots-clés qu'ils affectent à des ressources sur le Web ou l'affectation de privilèges

d'accès (ex. lecture et écriture) à des ressources (document ou base de données) aux utilisateurs.

Chapitre 5

Implémentation

Ce chapitre expose les grandes lignes de la mise en oeuvre de notre procédure sur l'environnement *Apache Spark* ainsi que les résultats des simulations effectuées sur des contextes triadiques.

5.1 Implémentation et algorithmes

Nous abordons dans la première section de ce chapitre les détails de la mise en œuvre de notre procédure sur l'environnement *Apache Spark*. La figure 5.1 représente la lignée RDD générée par cette implémentation.

Dans la terminologie *Apache Spark*, la lignée RDD, appelée aussi le graphe acyclique orienté ou simplement **DAG** *(directed acyclic graph)*, n'est rien d'autre qu'un graphe représentant tous les RDD parents d'un RDD. Dans un DAG, les nœuds représentent les RDD ou les résultats. Les connexions entre les nœuds représentent en revanche les transformations ou les actions. Ces connexions sont dites orientées car elles ne permettent de passer d'un RDD à un autre que dans un seul sens. Le graphe est dit aussi acyclique car aucun RDD ne permet de se transformer en lui-même via une série d'actions. En d'autres termes, la lignée RDD illustre les principaux procédés ou transformations que subissent les données jusqu'à l'obtention des résultats recherchés.

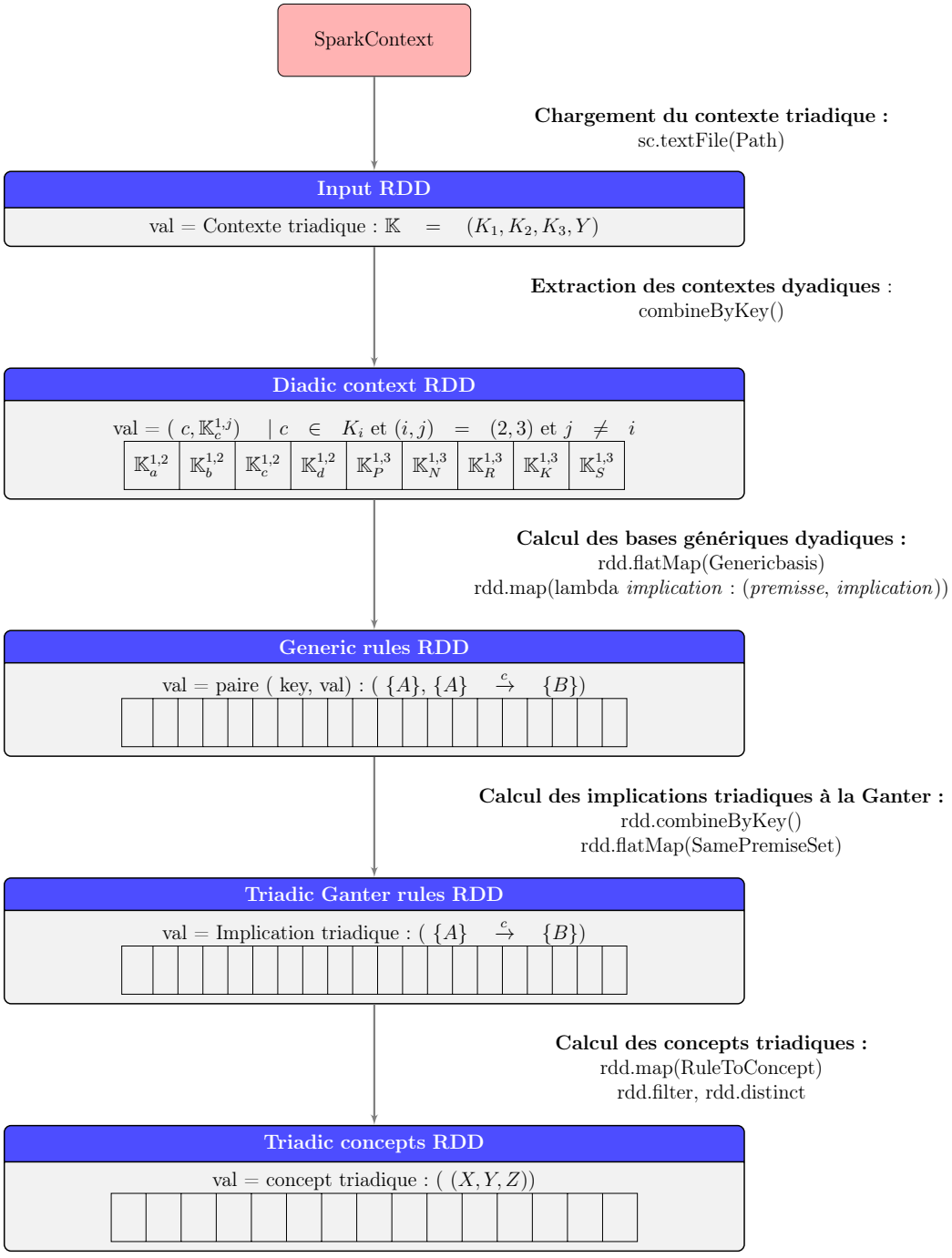

FIGURE 5.1: Graphique de la lignée RDD *(RDD Lineage Graph)*. Les RDD sont représentés par des rectangles alors que les flèches représentent les transformations

Dans le cas de ce projet, ce parcours peut être divisé en trois phases. La première phase effectue un prétraitement qui permet de prendre en charge les données représentant le contexte triadique et d'en extraire les contextes dyadiques par décomposition. La deuxième phase de traitement a pour objectif de traiter ces contextes dyadiques afin d'en déduire leurs bases génériques. Cette phase permet aussi d'intégrer ces bases d'implications dyadiques pour le calcul des implications triadiques et par la suite des concepts triadiques. La troisième phase de ce parcours consiste à épurer les résultats obtenus dans la phase précédente pour éliminer les concepts erronés ou redondants selon les observations 4.2.3 et 4.2.5 du chapitre 4.

5.1.1 Prétraitement

Lors de cette phase, on procède à la saisie des données représentant le contexte triadique. On en extrait par la suite les contextes dyadiques en vue du traitement ultérieur. Les deux premières lignes de notre code ci-dessous permettent d'instancier un objet du type *SparkContext* qui sera également chargé de la lecture de données et produire en sortie un RDD.

```
1 from pyspark import SparkContext
2 sc = SparkContext()
```

SparkContext est le point d'entrée principal pour une application *Spark* et représente la connexion à un *cluster* Spark en cours d'utilisation. *SparkContext* gère aussi les propriétés globales de l'application, telles que le niveau de logging ou le niveau de parallélisation par défaut.

Les données peuvent être chargées à partir d'une grande variété de sources différentes telles que les fichiers textes, des données au format **JSON**, des fichiers binaires ou autres. Cependant, pour les tests sur les données massives, nous adoptons des fichiers textes de même syntaxe des données que celle adoptée par les concepteurs de *Data-peeler*. Les données d'entrée sont composées de lignes où chacune d'elles contient trois champs représentant une dimension du contexte triadique. Les champs sont séparés par un espace. Les deux premiers champs représentent un objet et un attribut. Le troisième champ peut, lui-même, rassembler plusieurs éléments ou

conditions. Les conditions sont séparées quant à elles par des virgules. Les éléments peuvent être n'importe quelle chaîne de caractères dans la mesure où ils n'incluent aucune virgule.

Prenons l'exemple de cette ligne qui peut faire partie d'un ensemble de données d'entrée : **client1 Peter accessoires,livre**

Cet ensemble de données comporte 3 dimensions : la première étant l'objet (le client 1), la deuxième représente l'attribut (le fournisseur Peter) et la dernière représente les conditions (les produits accessoires et livres).

Dans le cas de fichiers textes, le fichier d'entrée est directement chargé puis transformé en RDD par la méthode *sc.textFile(path)*, où *path* désigne le chemin vers le fichier.

Un premier traitement du fichier d'entrée permet d'enlever les séparateurs (virgule) entre les éléments des lignes et d'en extraire les objets, les attributs puis les conditions. La procédure *condition(L)* permet de regrouper toutes les conditions sur une ligne dans un seul *set*.

```
Input RDD = textFile
                          .map(lambda x : x.replace(',', ' ')
                          .split())
                          .map(lambda x: (x[1],(x[0],condition(x))))
```

La dernière partie du code *.map(lambda x : (x[1],(x[0],condition(x))*, permet de préparer notre phase réduction *Reduce* et de transformer notre RDD régulier en un RDD apparié *(paired RDD)*.

Il y a plusieurs méthodes pour créer des RDD appariés à éléments clé-valeur à partir d'un RDD régulier. La façon la plus simple est d'exécuter une fonction *map()* qui renvoie des paires clé-valeur. Dans notre cas, les éléments du *RDDTriadicContext* sont des paires clé-valeur dont la clé représente un attribut a_i et la valeur désignant une ligne dont l'entête est l'objet o suivi de toutes les conditions sous lesquelles l'objet o possède l'attribut a_i.

L'extraction des contextes dyadiques $\mathbb{K}_{a_i}^{1,3}$ pour chaque attribut $a_i \in K_2$ est relativement simple et s'obtient par transformation de ce premier RDD par la méthode combineByKey:

```
rdd_ContextPerAttribute = rdd_Triadic_Context.combineByKey(
                                lambda row: [row],
                                lambda rows, row: rows + [row],
                                lambda rows1, rows2: rows1 + rows2
                                                                )
```

la méthode *combineByKey* est une fonction générique de *Apache Spark* dont l'objectif principal est de transformer tout RDD appariés d'éléments $[(K, V)]$ en un autre RDD d'éléments $[(K, C)]$ où C est le résultat de toute agrégation de toutes les valeurs sous la clé K. Dans notre cas, la méthode *combineByKey* permet d'assembler toutes les lignes associées à un attribut a_i pour construire le tableau représentant la relation binaire entre K_1 et K_3 pour a_i et ainsi obtenir le contexte $\mathbb{K}_{a_i}^{1,3}$.

L'extraction des contextes dyadiques $\mathbb{K}_{c_i}^{1,2}$ pour chaque condition nécessite en revanche quelques transformations supplémentaires pour regrouper les attributs par condition sur des mêmes lignes *(RddLinePerCondition)*. Nous pouvons finalement construire ces contextes par la même méthode *combineByKey* et les regrouper dans un nouveau RDD:

```
rdd_ContextPerCondition= rdd_LinePerCondition.combineByKey(
                                lambda row: [row],
                                lambda rows, row: rows + [row],
                                lambda rows1, rows2: rows1 + rows2
                                                                )
```

Finalement, ces deux RDDs seront ensuite fusionnées dans un même RDD contenant les contextes dyadiques $\mathbb{K}_{c_i}^{1,2}$ et $\mathbb{K}_{a_i}^{1,3}$ obtenus du contexte triadique initial.

```
Diadic Context RDD = sc.union([
            rdd_Condition_ContextPerCondition,
            rdd_Attribute_ContextPerAttribute
                                    ])
```

5.1.2 Traitement

À ce stade, notre nouveau RDD est composé de $|K_2| \times |K_3|$ éléments. Chaque élément représentant un contexte dyadique parmi les contextes calculés. Nous pou-

vons ainsi exécuter la principale transformation *Map* de notre approche qui permet de traiter chaque élément du RDD par la fonction *GenericBasiswithCondition* :

```
Generic rules RDD = rdd_Split_Triadic_Context.flatMap(GenericBasis)
```

La fonction *GenericBasis* prend comme arguments la paire $(c_k, \mathbb{K}_{c_k}^{1,j})$, puis elle calcule la base générique de $\mathbb{K}_{c_k}^{1,j}$. Chaque implication $A \rightarrow B$ de cette base sera par la suite *" rehaussée"* pour obtenir une implication triadique à la Ganter de la forme $A \xrightarrow{c_k} B$.

Il faut mentionner ici que nous avons opté pour la méthode *flatMap* au lieu de la méthode *Map*. La principale différence entre ces deux méthodes est que la transformation *Map* préserve le nombre d'éléments du RDD traité en retournant un seul élément par élément en entrée. La fonction *flatMap* par contre prend aussi un seul élément en entrée mais peut renvoyer 0 élément ou plus à la fois selon le code spécifié par le développeur. Autrement dit, *flatMap* transforme un RDD de longueur N en un autre RDD de longueur M. Dans notre cas précisément, chaque élément du RDD *imput* retournera un nombre d'éléments correspondant aux implications de la base générique calculée.

Le calcul de la base générique proprement dit est basé sur l'algorithme **JEN** qui a été abordé un peu en détail dans la section 3.1.2.

Ce dernier RDD sera transformé davantage en un RDD apparié dont la paire clé-valeur est constituée par une clé représentant la prémisse de l'implication et la valeur représente l'implication elle-même. Cette transformation nous permet d'effectuer une transformation *reduce* en invoquant de nouveau la méthode *combineByKey*. Cette transformation nous permet de regrouper les implications partageant la même prémisse dans le même baquet. Chaque élément du nouveau RDD obtenu *(rddImpliactionOfSamePremise)* représente un baquet d'implications ayant la même prémisse.

```
rdd_PremiseAsKey = Generic_rules_RDD
            .map(lambda x: (tuple(x._premise), x))

rdd_ImpliactionOfSamePremise = rdd_PremiseAsKey
            .combineByKey(
```

```
            lambda row: [row],
            lambda rows, row: rows + [row],
            lambda rows1, rows2: rows1 + rows2
            ).map(lambda x: x[1])
```

Désormais, on peut traiter tous les baquets *(ensemble d'implication partageant la même prémisse)* de manière parallèle. On invoque de nouveau une transformation *flatMap* pour appliquer à chacun de ces baquets la fonction *SamePremiseSet* discutée au chapitre 4 et qui permet de calculer les implications triadiques à la Ganter.

```
Triadic_Ganter_rules_RDD = rdd_ImplicationOfSamePremise.flatMap(
   SamePremiseSet).filter(
    lambda x: len(x.ObjectSupport) != 0).distinct()
```

Finalement, une transformation *Map* supplémentaire permet d'appliquer la fonction *ruleToconcept* à chaque implication triadique et d'en déduire un concept triadique.

```
TriadicConcepts RDD = Triadic_Ganter_rules_RDD.map(RuleToConcept).
   distinct()
```

Les méthodes *rdd.filter()* et *rdd.distinct()* permettent de filtrer les implications de support non nul ou encore les éléments RDD redondants.

5.1.3 Épuration des résultats

Il reste une dernière étape qui consiste à enlever les triplets non maximaux calculés tout de même par la fonction *SamePremiseSet* car ils correspondent aux observations 1 et 2 expliquées au paragraphe 4.2.3. Comme on l'a constaté souvent avec cette implémention, ce traitement supplémentaire se fait en deux étapes, d'abord on crée un RDD apparié dont la paire clé-valeur est constituée dans ce cas par la paire (X_1, Y_1) comme clé, puis de Z_1 comme valeur. Les éléments de notre nouveau RDD ainsi constitué se présentent comme suit: $((X_1, Y_1), Z_1)$. La phase réduction, cette fois, est réalisée par la méthode *reduceByKey*. La fonction *reduceByKey* permet de fusionner les valeurs de chaque clé à l'aide d'une fonction de réduction associative qui est pour cette occurrence l'union :

```
TriadicConcepts_RDD_check_modus = TriadicConcepts_RDD
    .map(lambda x: ((x._extent, x._intent), x._modus))
    .reduceByKey(frozenset.union)
```

Ce traitement n'étant pas suffisant, il faut s'attaquer maintenant aux triplets qui peuvent correspondre au cas particulier discuté au point 4.2.5. Ce traitement consiste en une autre transformation *Map* appliquant la fonction *TestSpecialCase* à l'ensemble de concepts partageant le même modus.

```
TriadicConcepts_RDD_Check_SpecialCases =
   TriadicConcepts_RDD_Check_specialCase.map(
    lambda x: (x[0], TestSpecialCase(x[1])))
```

5.2 Conclusion

La mise en oeuvre de notre approche sur un nombre de clusters réduit nous permet de tester notre approche sur des contextes relativement de petite taille. Cette mise en oeuvre valide notre analyse du chapitre précédent. La figure 5.2 est une illustration de cette mise en oeuvre.

Néanmoins, comme fut mentionné précédemment, nous n'obtenons pas la totalité des concepts mais nous pensons en revanche avoir obtenu les concepts les plus significatifs. En effet, notre approche nous permet d'obtenir les deux types d'implications triadiques proposées par Ganter et Obiedkov, soit les implications entre attributs sous conditions **CAI** et les implications entre conditions pour attributs **ACI**. Ces implications sont plus restrictives et plus compactes que celles définies par Biedermann. Ce travail devrait se poursuivre par des tests sur des données massives lorsqu'une plate-forme adéquate sera disponible. Aussi, la production des implications triadiques à la Biedermann devrait être le prochain objectif des travaux de programmation pour couvrir l'ensemble des concepts triadiques.

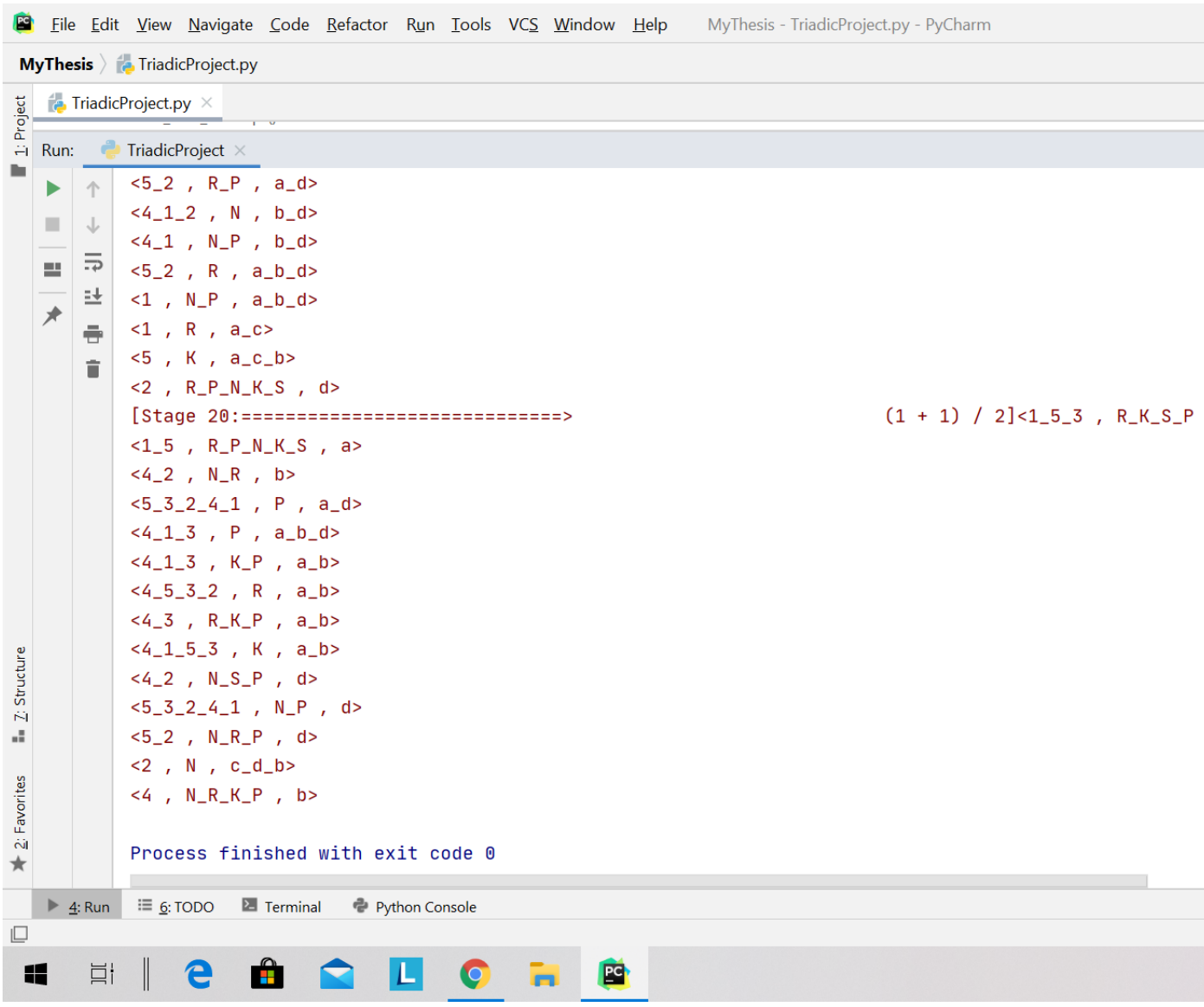

FIGURE 5.2: Résultat d'exécution pour le contexte PNRKS 2.2

Chapitre 6

Conclusion

Dans ce mémoire de maîtrise, nous avons développé de nouvelles procédures de calcul de concepts et d'implications triadiques adaptées aux enjeux du *big data* en se basant sur le paradigme de traitement parallèle et distribué. La démarche est comme suit : (i) on décompose le contexte formel triadique en autant de contextes dyadiques qu'il y a de conditions ou d'attributs selon le type d'implications désiré, (ii) on génère les implications triadiques de la forme $A \xrightarrow{C} B$ signifiant que A implique B sous tout sous-ensemble de C, où A et B sont des groupes d'attributs (respectivement des conditions) et C est un ensemble de conditions (respectivement des attributs) lorsqu'il s'agit d'implications entre attributs sous des conditions (respectivement entre conditions pour des attributs), et finalement (iii) on récupère un ensemble non nécessairement exhaustif de concepts triadiques à partir des résultats de l'étape 2.

Les contributions principales de ce travail sont de démontrer que les approches parallèles et distribuées basées sur le modèles *MapReduce* peuvent être une alternative intéressante en vue du calcul des implications dyadiques, des implications triadiques, et finalement des concepts triadiques lors de l'analyse de contextes émanant de données massives nécessitant une puissance combinatoire souvent non disponible avec les outils traditionnels.

La nouveauté dans ce travail de recherche réside également dans le fait que le contexte triadique initial est analysé en le scindant d'abord en autant de contextes

dyadiques que de conditions en recherchant les motifs (concepts et implications) contenus dans les contextes dyadiques. Cette première phase est utilisée pour extraire les implications triadiques dans le contexte initial et puis par ricochet les concepts triadiques. À ce propos, il s'agit là d'une autre particularité importante de ce travail qui réside dans le fait de procéder par une sorte de rétro-ingénierie consistant à obtenir les concepts triadiques à partir des implications triadiques à l'inverse du cheminement habituel adopté par les autres méthodes de fouille de données.

Notre solution est implémentée et validée à l'aide du cadre de développement *Apache Spark*.

Nous sommes conscients du fait que la génération d'un ensemble exhaustif des concepts triadiques n'est pas possible à l'aide de notre solution tel qu'illustré dans le chapitre 4 car les implications que nous produisons sont des formes plus strictes et plus compactes que celles de Biedermann. Ces dernières auraient fourni l'ensemble complet de concepts triadiques mais leur calcul en parallèle n'est pas évident et fera l'objet de travaux ultérieurs.

Bibliographie

[1] Barbut, M., and Monjardet, B. Ordre et classification, algèbre et combinatoire,(2 tomes), paris, hachette, 1970.

[2] Belohlavek, R. Introduction to formal concept analysis. *Palacky University, Department of Computer Science, Olomouc 47* (2008).

[3] Biedermann, K. How triadic diagrams represent conceptual structures. In *Conceptual Structures : Fulfilling Peirce's Dream, Fifth International Conference on Conceptual Structures, ICCS '97, Seattle, Washington, USA, August 3-8, 1997, Proceedings* (1997), D. Lukose, H. S. Delugach, M. Keeler, L. Searle, and J. F. Sowa, Eds., vol. 1257 of *Lecture Notes in Computer Science*, Springer, pp. 304–317.

[4] Biedermann, K. *A foundation of the theory of trilattices.* PhD thesis, Darmstadt Univ., 1998.

[5] Buyya, R., Vecchiola, C., and Selvi, S. T. *Mastering cloud computing : foundations and applications programming.* Newnes, 2013.

[6] Carpineto, C., and Romano, G. *Concept data analysis : Theory and applications.* John Wiley & Sons, 2004.

[7] Cerf, L., Besson, J., Robardet, C., and Boulicaut, J.-F. Data-peeler : Constraint-based closed pattern mining in n-ary relations. In *proceedings of the 2008 SIAM International conference on Data Mining* (2008), SIAM, pp. 37–48.

[8] Cerf, L., Besson, J., Robardet, C., and Boulicaut, J.-F. Closed patterns meet n-ary relations. *ACM Transactions on Knowledge Discovery from Data (TKDD) 3*, 1 (2009).

[9] Chunduri, R. K., and Cherukuri, A. K. Haloop approach for concept generation in formal concept analysis. *Journal of Information & Knowledge Management 17*, 03 (2018), 1850029.

[10] Chunduri, R. K., and Cherukuri, A. K. Scalable formal concept analysis algorithms for large datasets using spark. *Journal of Ambient Intelligence and Humanized Computing* (2018), 1–21.

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

[12] Derguech, W., Bhiri, S., Hasan, S., and Curry, E. Using formal concept analysis for organizing and discovering sensor capabilities. *The Computer Journal 58*, 3 (2015), 356–367.

[13] Dutta, K. Distributed computing technologies in big data analytics. In *Distributed Computing in Big Data Analytics*. Springer, 2017, pp. 57–82.

[14] Fan, W., Geerts, F., and Neven, F. Making queries tractable on big data with preprocessing : through the eyes of complexity theory. *Proceedings of the VLDB Endowment 6*, 9 (2013), 685–696.

[15] Ganter, B., and Obiedkov, S. Implications in triadic formal contexts. In *International Conference on Conceptual Structures* (2004), Springer, pp. 186–195.

[16] Ganter, B., and Obiedkov, S. *Conceptual exploration*. Springer, 2016.

[17] Ganter, B., and Wille, R. *Formal concept analysis : mathematical foundations*. Springer Science & Business Media, 2012.

[18] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system.

[19] Guigues, J.-L., and Duquenne, V. Familles minimales d'implications informatives résultant d'un tableau de données binaires. *Mathématiques et Sciences humaines 95* (1986), 5–18.

[20] Ignatov, D. I., Gnatyshak, D. V., Kuznetsov, S. O., and Mirkin, B. G. Triadic formal concept analysis and triclustering : searching for optimal patterns. *Mach. Learn. 101*, 1-3 (2015), 271–302.

[21] Jaschke, R., Hotho, A., Schmitz, C., Ganter, B., and Stumme, G. Trias–an algorithm for mining iceberg tri-lattices. In *Sixth International Conference on Data Mining (ICDM'06)* (2006), IEEE, pp. 907–911.

[22] Ji, L., Tan, K.-L., and Tung, A. K. Mining frequent closed cubes in 3d datasets. In *Proceedings of the 32nd international conference on Very large data bases* (2006), VLDB Endowment, pp. 811–822.

[23] Krajca, P., and Vychodil, V. Distributed algorithm for computing formal concepts using map-reduce framework. In *International Symposium on Intelligent Data Analysis* (2009), Springer, pp. 333–344.

[24] Kuznetsov, S. O. A fast algorithm for computing all intersections of objects from an arbitrary semilattice. *Nauchno-Tekhnicheskaya Informatsiya Seriya 2-Informatsionnye Protsessy i Sistemy*, 1 (1993), 17–20.

[25] Kuznetsov, S. O., and Obiedkov, S. A. Comparing performance of algorithms for generating concept lattices. *Journal of Experimental & Theoretical Artificial Intelligence 14*, 2-3 (2002), 189–216.

[26] Kuznetsov, S. O., and Poelmans, J. Knowledge representation and processing with formal concept analysis. *Wiley Interdisciplinary Reviews : Data Mining and Knowledge Discovery 3*, 3 (2013), 200–215.

[27] Le Floc'h, A., Fisette, C., Missaoui, R., Valtchev, P., and Godin, R. Jen : un algorithme efficace de construction de générateurs pour l'identification des règles d'association. *Numéro spécial de la revue des Nouvelles Technologies de l'Information 1*, 1 (2003), 135–146.

[28] Lehmann, F., and Wille, R. A triadic approach to formal concept analysis. In *International Conference on Conceptual Structures* (1995), Springer, pp. 32–43.

[29] Luxenburger, M. Implications partielles dans un contexte. *Mathématiques et Sciences Humaines 113* (1991), 35–55.

[30] Mazumder, S., Bhadoria, R. S., and Deka, G. C. Distributed computing in big data analytics. *AG : Springer International Publishing* (2017).

[31] Missaoui, R. Lattice miner.

[32] Missaoui, R., Kuznetsov, S. O., and Obiedkov, S. A., Eds. *Formal Concept Analysis of Social Networks*. Lecture Notes in Social Networks. Springer, 2017.

[33] Missaoui, R., and Kwuida, L. Mining triadic association rules from ternary relations. In *International Conference on Formal Concept Analysis* (2011), Springer, pp. 204–218.

[34] Pasquier, N. *Data mining : algorithmes d'extraction et de réduction des règles d'association dans les bases de données*. PhD thesis, 2000.

[35] Pasquier, N., Bastide, Y., Taouil, R., and Lakhal, L. Discovering frequent closed itemsets for association rules. In *International Conference on Database Theory* (1999), Springer, pp. 398–416.

[36] Pasquier, N., Bastide, Y., Taouil, R., and Lakhal, L. Efficient mining of association rules using closed itemset lattices. *Information systems 24*, 1 (1999), 25–46.

[37] Pasquier, N., Taouil, R., Bastide, Y., Stumme, G., and Lakhal, L. Generating a condensed representation for association rules. *Journal of intelligent information systems 24*, 1 (2005), 29–60.

[38] Pfaltz, J., and Taylor, C. Scientific discovery through iterative transformations of concept lattices. In *Proceedings Workshop on Discrete Mathematics and Data Mining, 2nd SIAM International Conference on Data Mining, April*, pp. 11–13.

[39] Poelmans, J., Ignatov, D. I., Viaene, S., Dedene, G., and Kuznetsov, S. O. Text mining scientific papers : a survey on fca-based information retrieval research. In *Industrial Conference on Data Mining* (2012), Springer, pp. 273–287.

[40] Poesia, G., and Cerf, L. A lossless data reduction for mining constrained patterns in n-ary relations. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases* (2014), Springer, pp. 581–596.

[41] Priss, U. Formal concept analysis in information science. *Annual review of information science and technology 40*, 1 (2006), 521–543.

[42] Priss, U., and Old, L. J. Modelling lexical databases with formal concept analysis. *J. UCS 10*, 8 (2004), 967–984.

[43] Rodríguez-Lorenzo, E., Cordero, P., Enciso, M., Missaoui, R., and Mora, Á. An axiomatic system for conditional attribute implications in triadic concept analysis. *International Journal of Intelligent Systems 32*, 8 (2017), 760–777.

[44] Spangenberg, N., and Wolff, K. E. Comparison of biplot analysis and formal concept analysis in the case of a repertory grid. In *Classification, data analysis, and knowledge organization*. Springer, 1991, pp. 104–112.

[45] Stumme, G., Taouil, R., Bastide, Y., Pasquier, N., and Lakhal, L. Computing iceberg concept lattices with titanic. *Data & knowledge engineering 42*, 2 (2002), 189–222.

[46] Szathmary, L., Valtchev, P., Napoli, A., Godin, R., Boc, A., and Makarenkov, V. A fast compound algorithm for mining generators, closed itemsets, and computing links between equivalence classes. *Annals of Mathematics and Artificial Intelligence 70*, 1-2 (2014), 81–105.

[47] Tilley, T., Cole, R., Becker, P., and Eklund, P. A survey of formal concept analysis support for software engineering activities. In *Formal concept analysis*. Springer, 2005, pp. 250–271.

[48] VIALLE, S. *Note de cours : Big Data Informatique pour les données et calculs massifs*. Supélec - Campus de Metz, Avril 2019.

[49] Wille, R. Restructuring lattice theory : an approach based on hierarchies of concepts. In *Ordered sets*. Springer, 1982, pp. 445–470.

[50] Wille, R. The basic theorem of triadic concept analysis. *Order 12*, 2 (1995), 149–158.

[51] Wille, R. Formal concept analysis, mathematic foundations, 1999.

[52] Xu, B., de Fréin, R., Robson, E., and Foghlú, M. Ó. Distributed formal concept analysis algorithms based on an iterative mapreduce framework. In *International conference on formal concept analysis* (2012), Springer, pp. 292–308.

[53] Zaki, M. J., and Ramakrishnan, N. Reasoning about sets using redescription mining. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining* (2005), ACM, pp. 364–373.

[54] Zhao, M., Zhang, S., Li, W., and Chen, G. Matching biomedical ontologies based on formal concept analysis. *Journal of biomedical semantics 9*, 1 (2018), 11.