HAL
open science

Fouille de motifs évolutifs dans un contexte big data pour l'exploration de données cliniques

Tariq Lambachri

To cite this version:

Tariq Lambachri. Fouille de motifs évolutifs dans un contexte big data pour l'exploration de données cliniques. Autre. Université Bourgogne Franche-Comté, 2019. Français. NNT : 2019UBFCA025 . tel-02500711

HAL Id: tel-02500711

https://theses.hal.science/tel-02500711

Submitted on 6 Mar 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE DE DOCTORAT DE L'ETABLISSEMENT UNIVERSITE BOURGOGNE FRANCHE-COMTE

PREPAREE A L'UNIVERSITE DE TECHNOLOGIE BELFORT-MONTBELIARD

École doctorale N°37

SPIM - Sciences Physiques pour l'Ingénieur et Microtechniques

Pour obtenir le
Grade de Docteur de
L'Université Bourgogne Franche-Comté

Spécialité : **Informatique**

Par : LAMBACHRI TARIQ

Fouille de motifs évolutifs pour l'exploration dans un contexte Big Data

Thèse présentée et soutenue publiquement le 16 Décembre à l'**UTBM**

Composition du Jury :

Younes JABRANE	*Rapporteur*	*Professeur des universités (Université Cadi Ayyad Marrakech Maroc)*
Michel HASSENFORDER	*Rapporteur*	*Professeur des universités (ENSISA)*
Stéphane GALLAND	*Président*	*Professeur des universités (UTBM)*
Amir HAJJAM EL HASSANI	*Directeur de thèse*	*Maître de conférences HDR (UTBM)*

REMERCIEMENTS

Tout d'abord, je tiens à vous informer que j'éprouve un blocage depuis plusieurs semaines par crainte d'omettre de remercier quelqu'un. Je commence donc par m'en excuser d'avance auprès de ces personnes.

Je dois beaucoup aux nombreuses personnes qui m'ont encouragé, soutenu et conforté au long de toutes ces années. Qu'elles trouvent dans ce travail l'expression de mes plus sincères remerciements.

Je remercie chaleureusement M. Amir HAJJAM EL HASSANI de m'avoir donné la chance pour faire ma thèse au sein de leur équipe, et aussi d'avoir dirigé ma thèse pendant trois ans. Leur rigueur et leur savoir m'ont permis de progresser et surtout d'appréhender une nouvelle philosophie de travail. Puissent-t-ils trouver par ces quelques mots l'expression de mon profond respect et de toute ma reconnaissance.

Je ne manquerais pas de remercier mes collègues pour leurs disponibilités et leurs encouragements. Nous avons eu ensemble des discussions très enrichissantes qui m'ont orientée et aidée dans mes travaux de recherche.

Je remercie vivement les rapporteurs de cette thèse pour tout le temps qui ont consacré à sa lecture. Je tiens à remercier les membres du Jury pour leur intérêt à mon travail.

Merci à tout mon entourage, famille et amis, qui m'ont soutenu et supporté durant ces trois dernières années.

Et pour finir, je tiens à dédier cette étape à mes chers parents ainsi qu'à mon frère Aymane qui a toujours été là pour moi.

À tous mes proches

J'espère qu'ils trouveront dans ce travail toute ma reconnaissance.

Soit A un succès dans la vie. Alors A = x + y + z, où x = travailler, y = s'amuser, z = se taire.

Albert EINSTEIN

SOMMAIRE

REMERCIEMENTS ii

SOMMAIRE 1

1 INTRODUCTION 4

2 ÉTAT DE L'ART 8

2.1 INTRODUCTION 9

2.2 EXTRACTION DE MOTIFS SÉQUENTIELS 10

2.2.1 EXTRACTION DE MOTIFS SÉQUENTIELS DANS DES BASES DE DONNÉES TEMPORELLES 10

2.2.1.1 DÉFINITIONS 11

2.2.1.2 TYPES DE MOTIFS TEMPORELS 11

2.2.1.3 ALGORITHMES D'EXTRACTION DES MOTIFS SÉQUENTIELS 13

i) ALGORITHMES BASES SUR APRIORI, DONNEES CLASSEES HORIZONTALEMENT 14

ii) ALGORITHMES BASES SUR APRIORI, DONNEES CLASSEES VERTICALEMENT 17

iii) ALGORITHMES BASES SUR FP-GROWTH 19

2.2.1.4 EXPÉRIMENTATIONS ET RÉSULTATS 25

2.2.2 FOUILLE DE FLOT DE DONNÉES (DATA STREAM MINING) 27

2.2.2.1 LE MODELE DE FLOT DE DONNÉES 27

2.2.2.2 EXTRACTION DES MOTIFS FREQUENTS DANS LES FLOTS DE DONNÉES 29

2.2.2.3 APPROCHES INCRÉMENTALES 30

i) ALGORITHME ISE 30

ii) ALGORITHME ISM 32

iii) *APPROCHES ε-APPROXIMATIFS* 35

iv) *ALGORITHMES BASES SUR LE COMPTAGE DES ITEMS (COUNT BASED ALGORITHMS)* 35

v) ALGORITHMES SKETCH 36

2.2.3 FOUILLE DES DONNÉES DANS UNE ARCHITECTURE DISTRIBUÉE 38

2.3 POSITIONNEMENT PAR RAPPORT A L'ETAT DE L'ART, CONTRIBUTIONS ENVISAGEES 42
3 CONTRIBUTIONS 43
3.1. INTRODUCTION 44
3.2. UNE PREMIERE FORMALISATION DES MOTIFS EVOLUTIFS 44
3.2.1. MOTIFS ET BASES DE TRACES 44
3.2.2. ÉVOLUTION D'UN MOTIF 46
3.2.3. PARTITIONNEMENT DES SEQUENCES 50
3.3. L'ARCHITECTURE LAMBDA 53
3.3.1. LA COUCHE BATCH 55
3.3.2. LA COUCHE TEMPS REEL 56
3.3.3. LA COUCHE SERVICE 57
3.3.4. CONCLUSION : 57
3.4. FRAMEWORK SPARK 59
3.4.1. COMPOSANTS ET L'ARCHITECTURE DU FRAMEWORK SPARK 60
3.4.2. L'ARCHITECTURE DU SPARK RUNTIME 61
3.4.3. SPARK STREAMING 64
3.4.4. CONCLUSION 65
3.5. LA STRUCTURE DE HACHAGE CUCKOO 66
3.6. LES FILTRES CUCKOO 68
3.7. UNE IMPLEMENTATION EFFICIENTE D'APRIORI DANS SPARK 70
4 IMPLEMENTATIONS 72
4.1 INTRODUCTION 73
4.2 PHASE I 73
4.3 PHASE II 76
4.3.1 COMPARAISON DE LA COMPLEXITE TEMPORELLE DE LA DEUXIEME ITERATIONS DES DEUX VERSIONS D'APRIORI 77
4.3.1.1 ESTIMATION DE LA COMPLEXITE SPATIALE DE LA DEUXIEME ITERATION DE L'IMPLEMENTATION CLASSIQUE D'APRIORI DANS SPARK 77
4.3.1.2 ESTIMATION DE LA COMPLEXITE SPATIALE DE LA DEUXIEME ITERATION DE NOTRE IMPLEMENTATION D'APRIORI DANS SPARK 78
4.3.1.3 BENCHMARK 79

4.3.2 COMPARAISON DE LA COMPLEXITE SPATIALE DE LA DEUXIEME ITERATIONS DES DEUX VERSIONS D'APRIORI 80
4.3.2.1 ESTIMATION DE LA COMPLEXITE SPATIALE DE LA DEUXIEME ITERATION DE L'IMPLEMENTATION CLASSIQUE D'APRIORI DANS SPARK 80
4.3.2.3 BENCHMARK 81
4.4 PHASE III 84
4.5 ANALYSE DES CHANGEMENTS DES MOTIFS 85
5 RESULTATS ET EXPERIMENTATIONS 87
5.1 INTRODUCTION 88
5.2 ÉCHANTILLON DE DONNÉES 88
5.3 ANALYSES DES PERFORMANCES 89
5.4 CONCLUSION 91
6 CONCLUSION ET PERSPECTIVES 92
6.1 CONCLUSION GENERALE 93
6.2 PERSPECTIVES 95
BIBLIOGRAPHIE 96
TABLE DES FIGURES 103
LISTE DES TABLES 104

1 INTRODUCTION

Les croissances de l'intelligence artificielle et du Big Data sont sur le point de se rencontrer, permettant à chacune de se développer encore plus rapidement. Le rapport Global Data Protection Index de Dell EMC révèle que la quantité de données a augmenté depuis 2016 de plus 500%, dans le monde, et de de 700% en France. Les chiffres prévisionnels d'IDC montrent que le Big Data devrait rapidement faire ses preuves dans les secteurs du retail, de la manufacture, des administrations publics et particulièrement de la santé.
Ces données massives représentent une source fantastique de nouvelles connaissances et de progrès médicaux pour l'identification de facteurs de risque de maladies, l'aide au diagnostic de cancers, au choix et au suivi de l'efficacité des traitements, à la surveillance des agents pathogènes, etc.

Dans le domaine de la télésurveillance de patients atteints de maladies chroniques, la plateforme E-care, développée dans le cadre des investissements d'avenir, a démontré qu'un suivi quotidien permettait de détecter une décompensation de manière précoce. Une décompensation cardiaque, par exemple, peut-être ainsi détectée jusqu'à douze jours avant sa survenue.
L'amélioration de la précocité de la détection s'est faite dans le temps, au fur et à mesure des expérimentations et des données collectées. Les techniques de fouille de données, de l'apprentissage automatique ou de l'intelligence artificielle ont permis d'améliorer efficacement le traitement des données collectées. Cependant, la masse de données collectées devient trop massive et nécessite à la fois des algorithmes de traitement puissant mais aussi une architecture efficace.

Nos travaux se placent donc dans le cadre de la fouille de données temporelles, en s'intéressant à la fouille de motifs dits évolutifs dans un contexte Big Data. Le travail de thèse visera à définir la notion de motif évolutif, à étudier les algorithmes d'extraction des motifs temporels évolutifs, à proposer des principes d'interaction pour piloter de tels algorithmes, à concevoir et implémenter un ou plusieurs systèmes, enfin à valider ceux-ci du point de vue des performances. Notre contribution se situe à deux niveaux :

- Une étude des algorithmes d'extraction poussée qui nous a mené à implémenter et à comparer les différents algorithmes de la littérature ;
- Une proposition d'approche pour implémenter ces algorithmes dans un contexte Big Data.

Dans le domaine de l'extraction de motifs, qui nous intéresse dans le cadre de ce travail de thèse, il existe un grand nombre d'algorithmes pour résoudre une large variété de problèmes ou sous problèmes sensiblement identiques. Ces algorithmes possèdent des défauts communs, qui se

résument dans le fait de ne pas prendre en compte les écarts temporels entre les transactions, l'absence de recherche de motifs inter transactions et la dépendance par rapport à la hiérarchie dans les données (impossibilité de rechercher à différents niveaux de celle-ci). Ces défauts sont accentués lorsque la masse de données traitée deviennent importante.

Pour une grande masse de données, nécessitant des applications parallèles/distribuées, l'approche hybride de l'architecture Lambda semble particulièrement pertinente. Elle mélange de batch et de temps réel. En utilisant le traitement batch, cette architecture permet d'équilibrer la latence, le débit, et la tolérance aux pannes de systèmes en fournissant des vues précises sur les données qui sont simultanément confrontées à de la donnée en temps réel pour avoir des résultats plus précis. Le framework Apache Spark et le patron MapReduce consolide l'architecture Lambda. MapReduce [1] est une des meilleures solutions garantissant la tolérance aux pannes. Apache Spark présente plusieurs avantages par rapport à Hadoop [2], l'un des Frameworks les plus populaires basé sur le modèle de programmation MapReduce. Ce dernier est connu pour le stockage distribué et le traitement de grands ensembles de données à l'aide de clusters autonomes.

L'approche proposée est conçue autour de deux choix techniques à savoir Spark et les structures de hachage Cuckoo qui viennent améliorer les algorithmes d'extraction. Cela nous permettra réduire le nombre total des calculs effectués, ce qui implique une amélioration des performances importante notamment dans un contexte Big Data. En outre, l'utilisation de Spark nous permet de profiter de la puissance d'un environnement de calcul hautement évolutif et parallèle.

La suite de ce manuscrit est organisée comme suit : le chapitre 2 constitue un état de l'art par rapport aux techniques d'extraction des motifs fréquents. Ce chapitre est un tour d'horizon des différents algorithmes et approches existants, nous permettant ainsi de cerner le contexte dans lequel se placent nos travaux.
Le chapitre 3 présente notre approche d'extractions des motifs. En effet, cette partie du manuscrit permet de détailler la partie théorique de nos contributions en présentant les différents éléments caractérisant l'évolution de motifs et sa représentation par des motifs évolutifs. Aussi, elle permet de présenter les choix architecturaux proposés pour concevoir notre approche d'extraction des motifs.

Enfin, le chapitre 4 présente les différentes étapes de notre approche ainsi que l'expérimentation que nous avons menée sur les benchmark, comparant la complexité temporelle et spatiale de notre approche vis-à-vis la version classique d'Apriori.

La dernière partie est consacrée aux expérimentations (chapitre 5). Le but de cette section est d'évaluer la performance de notre algorithme par rapport aux algorithmes existants. Pour finir, les conclusions et les perspectives de nos travaux sont discutées dans le dernier chapitre.

.

2 ÉTAT DE L'ART

2.1 INTRODUCTION

L'extraction de données à partir de vastes ensembles de données a toujours été considérée comme un des sujets majeurs du datamining.

L'extraction des motifs séquentiels est l'une des méthodes les plus utilisées pour cet objectif. En effet, l'extraction d'itemsets fréquents recherche les combinaisons fréquentes d'items. De nombreux algorithmes d'exploration des items fréquents existent, mais ces derniers ne sont pas adaptés à la quantité massive des données que l'on a de nos jours.

Par conséquent, pour explorer des volumes de données indéfiniment croissant, il y a un besoin d'algorithme d'exploration des motifs qui peuvent gérer de grands volumes de données tout en étant performant. Pour ce type d'applications parallèles/distribuées, MapReduce est une des meilleures solutions garantissant la tolérance aux pannes. Hadoop est l'un des Frameworks les plus populaires basés sur le modèle de programmation MapReduce. Ce dernier est connu pour le stockage distribué et le traitement de grands ensembles de données à l'aide de clusters autonomes.

Cependant, les opérations d'E/S de disque sont considérées comme étant lourdes dans le cas d'un algorithme itératif. Cela rend Hadoop inefficient.

Au cours de ces dernières années, plusieurs plates-formes basées sur des architectures distribuées ont vu le jour. Spark [3] est l'une des solutions qui a attiré beaucoup d'attention en raison de son support natif des calculs distribués.

Dans ce sens, nous proposons une nouvelle approche d'extraction des items fréquents. Le challenge par rapport à notre approche est d'exploiter des techniques d'extraction traditionnelles dans un environnement distribué.

La principale originalité de notre méthode d'extraction est de tirer profit d'un algorithme existant performant et de l'adapter pour qu'il soit utilisable dans une structure de données distribuée. Cette structure va permettre de maintenir les motifs séquentiels fréquents couplés tout en offrant une stratégie d'élagage rapide.

En d'autres termes, notre approche prépare des plans d'exécutions avant chaque itération et choisit le meilleur plan pour minimiser le temps et les ressources à utiliser.

Dans cet état de l'art, nous allons nous intéresser aux différents types des motifs temporels séquentiels ainsi qu'aux principaux algorithmes et approches permettant l'extraction de ces derniers. Nous allons nous intéresser également aux travaux concernant la fouille dans les flots de données (Data Stream Mining), ainsi que le changement et l'évolution des motifs.

Finalement, nous allons présenter les travaux existants permettant l'extraction des motifs dans des structures de données distribuées.

2.2 EXTRACTION DE MOTIFS SÉQUENTIELS

2.2.1 EXTRACTION DE MOTIFS SÉQUENTIELS DANS DES BASES DE DONNÉES TEMPORELLES

L'extraction de motifs dans des bases de données temporelles a fait l'objet de plusieurs recherches dans le domaine de la fouille de données. De nombreux travaux existent dans ce domaine (Mooney et Roddick [4], Srikant & Agrawal [5] ; Zaki, [6] ; Pei et al. [7] ; Yan et al., [8] ; Nanni & Rigotti, [9] ; Gomariz et al., [10]).

Selon la structure des données utilisées, deux catégories peuvent être distinguées :

- La catégorie des motifs séquentiels, qui consiste à rechercher les motifs fréquents (qui se répètent) au sein de plusieurs suites d'événements.
- La catégorie basée sur les épisodes, qui consiste à rechercher des motifs fréquents au sein d'une suite d'événements.

Dans le paragraphe suivant, nous allons définir les différents concepts dans le domaine de l'extraction de motifs.

2.2.1.1 DÉFINITIONS

Définition 1 (Événement). *Soit E un ensemble d'éléments, nommés types d'événements. Un événement est un couple (e, t) où e ∈ E et t correspond à la date d'apparition de l'événement.*

Définition 2 (Séquence). *Une séquence est une suite non vide d'événements ordonnée selon leur date d'apparition. Une séquence est générée par une source.*

Définition 3 (Motif). *Un motif est une suite non vide de types d'événements.*

Définition 4 (Occurrence d'un motif). *Étant donnée une séquence S et un motif m, une occurrence de m dans S est une sous-séquence de S qui vérifie :*

- *Les types d'événements de m sont présents dans S et dans le même ordre*
- *Le premier élément de S a pour type le premier élément de m*
- *Le dernier élément de S a pour type le dernier élément de m*
- *La durée de l'occurrence est inférieure à un seuil w (nommée taille de fenêtre) fixé par l'analyste.*

La taille de l'occurrence est le nombre d'événements qu'elle contient.

Définition 5 (Support d'un motif). *Étant donnée une séquence, le support d'un motif est le nombre d'occurrences de ce motif dans la séquence, selon une méthode de comptage.*

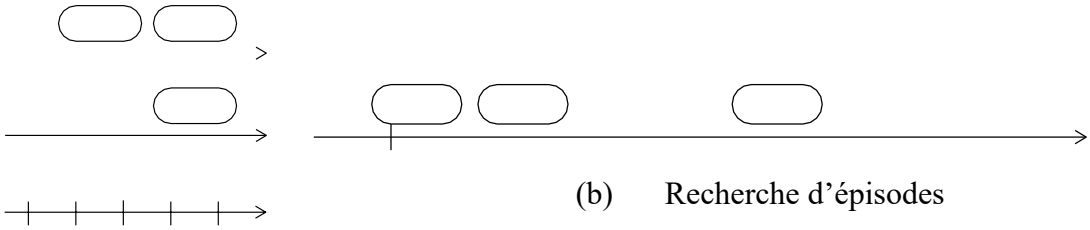
2.2.1.2 TYPES DE MOTIFS TEMPORELS

Au regard des définitions évoquées précédemment, deux types de motifs temporels se précisent. Ils se différencient principalement par la structure des données auxquelles ils s'appliquent. Leurs caractéristiques principales sont les suivantes :

- Motifs séquentiels : introduite par Srikant et Agrawal en 1996, l'extraction de motifs séquentiels s'effectue sur des données composées d'un ensemble de séquences. Le support d'un motif séquentiel correspond au nombre de séquences dans lesquelles le motif se produit au moins une fois.

- Épisodes : Définis par Mannila et Collab [11], la recherche d'épisodes est effectuée sur les données composées d'une seule séquence. Si l'on considère que deux événements ne peuvent se produire simultanément, le support d'un motif correspond au nombre de ses occurrences dans la séquence. Plusieurs types d'épisodes sont identifiés, selon que l'on conserve la relation d'ordre entre les événements (épisodes en série) ou non (épisodes parallèles).

Ces deux types de motifs sont illustrés dans la figure 2.1, où l'on peut observer la recherche d'un motif (B, C) selon les deux types de motifs.



(b) Recherche d'épisodes

(a) Recherche des motifs séquentiels

La manière de compter les occurrences est primordiale pour toute recherche de motif, en particulier lorsque l'on ne souhaite obtenir que les motifs fréquents par rapport à un seuil donné.

De nombreux travaux ont été réalisés à ce sujet, dont ceux de Gan et Dai [12]. Ils y définissent deux propriétés que toute mesure de fréquence doit respecter pour être considérée comme fiable. Ces propriétés sont décrites comme suit :

- L'anti-mono tonicité : qui garantit qu'aucun motif fréquent n'est oublié.

- Fréquence maximale : qui garantit l'absence de doublons dans les résultats de la recherche.

Au fil du temps, chaque type de motif temporel a été enrichi, selon les différents points de vue des chercheurs s'étant intéressés au domaine. On remarque cependant que les motifs séquentiels ont donné lieu à beaucoup plus de travaux que les épisodes. On note également que, quel que soit le type de motif considéré, la majorité des recherches sont effectuées sur des domaines

précis, voir directement en vue de données spécifiques, plutôt que dans un cas général.

2.2.1.3 ALGORITHMES D'EXTRACTION DES MOTIFS SÉQUENTIELS

Comme résumé dans l'état de l'art de Mooney et Roddick [13], le processus d'extraction des motifs peut être guidé par plusieurs contraintes dont on peut citer celles concernant le type des items recherchés, la longueur des motifs, le respect d'une hiérarchie donnée, la durée sur laquelle s'étend un motif ou sur l'écart entre deux événements d'un motif (gap).

Dans cette partie, nous allons présenter une partie des algorithmes utilisés pour extraire des motifs. Ceux-ci sont regroupés selon la façon dont l'extraction est effectuée et selon la structure des données requise. La première sous-partie sera consacrée aux deux structures de données dites classées verticalement ou horizontalement. La deuxième sous-partie s'intéresse aux algorithmes de type Apriori [14], d'abord à partir de données classées horizontalement puis à partir de données classées verticalement. Finalement, la troisième sous- partie sera consacrée aux algorithmes de type FP-growth [15].

Dans ce qui suit, nous allons introduire les structures de données dites classées horizontalement ou verticalement . Ces structures de données constituent un ensemble de séquences contenant elles-mêmes des événements. La principale différence entre ces dernières réside dans les critères utilisés pour trier les données.

- Les données classées horizontalement : Pour chaque séquence, on ordonne les événements par date.

- Les données classées verticalement : Pour chaque séquence, on ordonne les événements par type. Cette structure permet généralement de générer et de compter les candidats d'une façon plus rapide (Yang et collab., 2002) [16].

Pour illustrer ces deux structures, nous allons nous baser sur la séquence de la figure 2.2.

Choisir de classer les données horizontalement revient à lire la table ligne par ligne, ce qui donne l'ensemble de données (t1, {A}), (t2, {B}), (t3, {C}), (t4, {B}), (t5, {C}), (t6, {A}), (t7, {A}), (t8, {C}), (t9, {B}), (t10, {C}).

Le classement des données verticalement revient à lire la table colonne par colonne, nous obtenons cette fois l'ensemble de données {(A, {t1, t6, t7}), (B, {t2, t4, t9}), (C, {t3, t5, t8, t10})}.

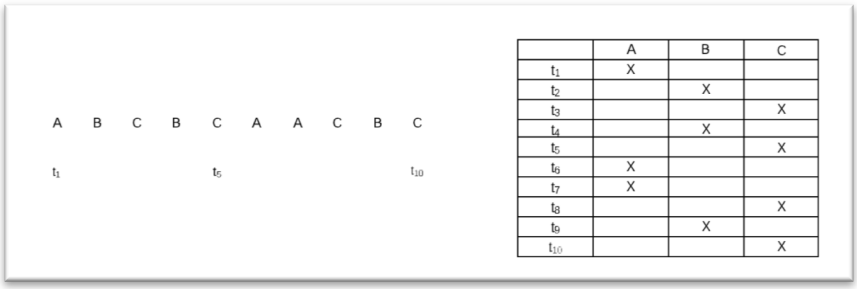


Figure 2.2 : Une séquence et sa représentation sous forme de table

i) ALGORITHMES BASES SUR APRIORI, DONNEES CLASSEES HORIZONTALEMENT

Il existe plusieurs algorithmes pour l'extraction de motifs séquentiels, entre autres AprioriAll, AprioriSome et DynamicSome (Agrawal et Srikant)). Ils se différencient par leur manière de compter les occurrences des motifs retournés.

Ces algorithmes suivent tous le même cheminement. La première étape consiste à récupérer les événements fréquents qui vont être utilisés pour générer des candidats pour la première passe sur les données. A la fin de cette première passe, seuls les candidats fréquents sont conservés, et servent à générer les candidats pour la passe suivante. Le cycle se déroule ainsi, chaque passe générant les motifs fréquents d'une taille supérieure d'un élément à ceux générés lors de la passe précédente. L'algorithme s'arrête lorsqu'aucun candidat ne peut être généré, ou que l'ensemble des motifs fréquents est vide à l'issue d'une passe.

Tous ces algorithmes possèdent des défauts communs, qui se résument dans le fait de ne pas prendre en compte les écarts temporels entre les transactions, l'absence de recherche de motifs inter transactions et la dépendance par rapport à la hiérarchie dans les données (impossibilité de rechercher à différents niveaux de celle-ci).

- **GSP** (Generalized Sequential Patterns) (Srikant et Agrawal,) est un algorithme qui se base sur Apriori et qui permet l'extraction de motifs séquentiels. Pour pallier le défaut de ce dernier, GSP inclut plusieurs paramètres supplémentaires : les contraintes temporelles (gap minimum et maximum entre les transactions) qui permettent de supprimer directement des règles triviales, les fenêtres glissantes qui améliorent les contraintes de temps ainsi que la gestion des taxonomies qui permet d'extraire des motifs séquentiels contenant des éléments présents à plusieurs niveaux de la hiérarchie définie par l'utilisateur [17].

  L'algorithme GSP effectue plusieurs passes sur les données pour rechercher les motifs fréquents. Lors de la première passe, le support de chaque item est calculé, et correspond au nombre de séquences où il apparaît. À partir de cette passe, les items fréquents par rapport au seuil donné forment les motifs de taille 1. À partir de la deuxième itération, les motifs fréquents de l'itération précédente sont utilisés pour générer l'ensemble des motifs candidats pour l'itération en cours. Les candidats d'une itération ont tous le même nombre d'items, et possèdent 1 item de plus que les ceux de l'itération précédente. Pour résumer, le support des candidats est calculé à chaque itération, et seuls ceux qui passent le seuil donné sont conservés et utilisés pour générer les candidats de l'itération suivante. La recherche se termine lorsqu'aucun motif fréquent n'est trouvé à l'issue d'une passe (cf figure 2.3)

```
F1 = chaque item dans la base est un candidat de longueur 1
k=2,
do while F (k-1) ! = Null;
Générer les candidats Ck (ensemble de candidats de longueur k) ;
Pour chaque niveau (c.-à-d., sequences de longueur k) faire
do
Scanner la base pour déterminer le support de chaque séquence dans la base
Garder les séquences fréquentes
Fk = {a ∈ Ck de sorte que son support ≥min_support}
k = k+1 ;
Result = l'ensemble des éléments fréquents est l'union de tous les FK
End do
End do
```

Figure 2.3 : Exemple d'implémentation de l'algorithme GSP

- **PSP (Prefix Tree for Sequential) [18]** constitue une version améliorée du GSP où les auteurs proposent d'utiliser une structure d'arbre préfixé. En effet, le principe de cette structure consiste à factoriser les séquences candidates en fonction de leur préfixe. De plus, pour prendre en compte le changement d'itemset, l'arbre est doté de deux types de branches. Le premier type, entre deux items, signifie que les items sont dans le même itemset alors que le second signifie qu'il y a un changement d'itemset entre ces deux items. Ainsi, l'arbre de préfixes ne stocke plus les candidats dans les feuilles, mais permet de retrouver les candidats de la façon suivante : tout chemin de la racine à une feuille représente un candidat et tout candidat est représenté par un chemin de la racine à une feuille. Cette nouvelle structure de données permet ainsi une nette amélioration des performances lors de l'extraction de motifs séquentiels [13].

- **SPIRIT [19]** est une famille d'algorithmes Apriori qui utilise des expressions rationnelles pour améliorer le processus d'extraction. Le noyau de l'algorithme est similaire à celui du GSP et la principale différence réside dans l'étape de génération des candidats.

  Étant donné que la plupart des expressions rationnelles ne sont pas anti-monotones, nous sommes contraints d'utiliser des relaxations de contraintes lors de la génération des candidats pour construire des séquences valides.

La similitude de cette procédure avec GSP dépend principalement de la relaxation. Si cette dernière est anti-monotone, SPIRIT peut alors adopter la même philosophie que le GSP (générer une séquence k à partir de deux séquences k -1), sinon SPIRIT doit étendre un (k-1pattern) fréquent avec un élément fréquent (1pattern), générant ainsi plus de séquences non fréquentes que ce qui est nécessaire. Des travaux ultérieurs ont démontré que les expressions rationnelles peuvent être remplacées par des expressions sans contexte et cela sans compromettre la performance de l'algorithme. Ce qui est utile parce que les expressions sans contexte sont plus expressives que les expressions rationnelles, tout en étant capables de représenter des connaissances qui sont plus intéressantes.

- **MFS (Maximal Frequent Sequences) [20]** consiste à améliorer la manière utilisée par GSP lors la génération des candidats dans l'optique de réduire le nombre d'accès nécessaires aux données. Pour ce faire, une première passe est réalisée sur un échantillon des données avec GSP. Les séquences fréquentes maximales ainsi obtenues sont ensuite utilisées pour générer les candidats, jusqu'à ce qu'aucun nouveau candidat ne soit généré lors d'une passe. Le gain vient majoritairement du fait que des séquences longues peuvent être vérifiées plus tôt que ne le fait GSP.

ii) ALGORITHMES BASES SUR APRIORI, DONNEES CLASSEES VERTICALEMENT

Plusieurs algorithmes existent et se différencient principalement par leur façon de compter les candidats et les structures de données utilisées.

- **SPADE (Sequential Pattern Discovery using Equivalence classes)** et sa variante cSPADE (Constrained SPADE) (Zaki) [21] est le premier algorithme à avoir utilisé une représentation verticale de la base de données (représentation sous forme d'un treillis) pour extraire des motifs séquentiels. Cette représentation consiste à inverser la méthode d'indexation de la base. En outre, l'algorithme SPADE représente les motifs séquentiels différemment en les regroupant selon des classes d'équivalences qui prennent en compte les préfixes des séquences.

Ces classes d'équivalences permettent de décomposer ainsi le problème initial en sous-problèmes plus faciles à gérer en mémoire.

Les classes d'équivalences de l'algorithme SPADE sont définies à partir d'une relation d'équivalence sur le préfixe. Ainsi, deux séquences de taille k sont dans la même classe d'équivalence si elles ont un préfixe commun de taille k − 1. Plus formellement, une classe d'équivalence est définie de la manière suivante :

$$[\rho \in FSeqs\ (D, \sigma)k\text{-}1\ ] = \{s \in FSeqs(D, \sigma)\ k \mid Pre\ f\ k\text{-}1\ (s) = \rho\ \}$$

Tel que Pre f k-1 (s) est la sous séquence de taille k-1qui constitue le préfix de s alors que FSeqs (D, σ) k-1 représente les séquences fréquentes de taille k. La génération de candidats se fait par une opération de jointures entre les différents éléments des classes d'équivalences. Quant au comptage de support, il est effectué en calculant les intersections possibles dans la base de données inversée.

Ci-dessous une pseudo implémentation de l'algorithme SPADE, (cf figure 2.6)

```
SPADE (SDB, minsup)
1.  Scan SDB to create V(SDB) and identify F1 the list of frequent items,
2.  ENUMERATE(F1)
```

```
ENUMERATE(an equivalence class F)
1.    FOR each pattern Ai ∈ F
2.        Output Ai
3.        Ti :=∅.
4.        FOR each pattern Aj ∈ F, with j ≥ i
5.            R = MergePatterns(Ai , Aj)
6.            FOR each pattern r ∈ R
7.                IF sup(R) ≥ minsup THEN
8.                        Ti:=Ti ∪ {R}
9.        ENUMERATE(Ti)
```

Figure 2.4 : Pseudo implémentation de l'algorithme SPADE

- **SPAM (Sequential Patterns Mining)** [22], cet algorithme se différencie des autres approches par l'utilisation d'une représentation par vecteurs de bits de la base de données. Lors de la première passe sur la base de données, un vecteur binaire est construit pour chaque item. Dans ce vecteur, chaque bit représente une séquence de la base, si l'item est présent dans la séquence, le bit correspondant est activé (valeur à 1). Cette représentation

permet d'optimiser le processus de comptage de supports puisque cette opération revient en fait à utiliser la fonction booléenne AND. Cependant, l'implémentation publique de cet algorithme souffre de certaines limitations qui l'empêchent d'être appliqué sur des bases de données réelles.

- **LAPIN-SPAM (Yang et Kitsuregawa)** se base sur SPAM (Ayres et collab. Plusieurs nouveaux algorithmes ont vu le jour à partir des travaux de J. Pei et J. Han [HPMa+00, PHMa+01] sur les méthodes par projections (Pattern Growth). Dans [YWK05], les auteurs proposent l'algorithme LAPIN pour extraire des séquences fréquentes. Cet algorithme se base sur l'idée simple que seule la dernière position d'un item α dans une séquence s permet de juger si une k-séquence fréquente peut être étendue avec l'item α pour former une k +1 — séquence. Cette approche permet en fait de limiter encore plus l'espace de recherche parcouru lors de l'extraction de motifs puisque l'algorithme ne s'intéresse qu'à construire une table d'index des items pour chaque séquence dans la base de données. Une amélioration de l'algorithme LAPIN est proposée avec l'algorithme PAID [YKW06] qui s'appuie sur une optimisation du précédent algorithme LAPIN en utilisant une représentation verticale de la base de données [23].

iii) ALGORITHMES BASES SUR FP-GROWTH

FP-growth (Frequent Pattern growth) [24] est une famille d'algorithmes qui consiste d'abord à compresser la base de données en une structure compacte appelée FP-tree (Frequent Pattern tree), puis à diviser la base de données compressée en sous-projections de la base de données appelées bases conditionnelles.

Chacune de ces projections est associée à un item fréquent. L'extraction des itemsets fréquents se fera sur chacune des projections séparément.

L'algorithme FP-growth apporte ainsi une solution au problème de la fouille de motifs fréquents dans une grande base de données transactionnelle. En stockant l'ensemble des éléments fréquents de la base de transactions dans une structure compacte, on a supprimé la nécessité de

devoir scanner de façon répétée la base de transactions. De plus, en triant les éléments dans la structure compacte, on accélère la recherche des motifs.

Dans ce qui suit nous allons présenter les algorithmes les plus importants faisant parties à la famille FP-growth.

- **FreeSpan** [Han et al. « frequent pattern-projected sequential pattern mining » SIGKDD] [25] est une nouvelle approche qui part du principe « diviser pour conquérir ». En effet elle consiste à utiliser des éléments fréquents pour projeter de manière récursive une base de données de séquence en un ensemble de bases projetées plus petit pour faire ainsi croître le nombre des sous-séquences. Ce processus permet de partitionner et limiter à la fois les données et l'ensemble des modèles à tester.

Exemple :

SID	*Séquences*
10	<a (abc) (ac) d (cf)>
20	<(ad) c (bc) (ae)>
30	<(ef) (ab) (df) cb>
40	<(eg (af) cbc>

Tableau 2-1 : Base de séquences S ayant un min supp = 2

Soit la base de séquence S ayant un Min_support égal à 2, (cf. Tableau 2.1).

- Étape 1 : Trouver les motifs séquentiels de longueur 1 et les ordonner d'une façon décroissante selon leur support

    - f_list : a : 4, b : 4, c : 4, d : 3, e : 3, f : 3 ; g : 1

- Étape 2 : diviser l'espace de recherche. L'ensemble des motifs séquentiels peut être partitionné en six sous-ensembles. Le partitionnement se fera comme suit :

  - Ensemble contenant que l'item a,
  - Ensemble contenant b, mais aucun item qui se situe après b dans la liste f_list,
  - Ensemble contenant c, mais aucun item qui se situe après c dans la liste f_list,
  - Ensemble contenant d, mais aucun item qui se situe après d dans la liste f_list,
  - Ensemble contenant e, mais aucun item qui se situe après e dans la liste f_list.
  - Ensemble contenant que l'item f

Les motifs séquentiels liés aux six sous-ensembles partitionnés peuvent être extraits en construisant six bases de données projetées (obtenues par une analyse supplémentaire de la base de données d'origine). Les éléments peu fréquents, tels que *g* dans cet exemple, sont supprimés des bases de données projetées. Le processus d'extraction de chaque base de données projetée est détaillé comme suit.

L'exploration des motifs séquentiels ne contenant que *a*

- La base de données projetée ⟨a⟩ est {⟨aaa⟩, ⟨aa⟩, ⟨a⟩, ⟨a⟩}. En explorant cette base de données projetée, un seul motif séquentiel supplémentaire contenant seulement l'élément a est trouvé qui est ⟨aa⟩ : 2.

L'exploration des motifs contenant *b*, mais aucun item qui se situe après b dans la liste f_list

- En explorant la base de données projetée ⟨b : {⟨a (ab) a⟩, ⟨aba⟩, ⟨(ab) b⟩, ⟨ab⟩}, quatre motifs séquentiels supplémentaires contenant l'élément b, mais aucun élément après b dans la liste f sont trouvés. Ces éléments sont {⟨ab⟩ : 4, ⟨ba⟩ : 2, ⟨(ab)⟩ : 2, ⟨aba⟩ : 2}.

L'exploration des motifs contenant *c*, mais aucun item qui se situe après c dans la liste f_list

L'exploration de la base de données projetée ⟨c⟩: {⟨a (abc) (ac) c⟩, ⟨ac (bc) a⟩, ⟨(ab) cb⟩, ⟨acbc⟩}, suit le processus suivant : un balayage de la base de données projetée génère l'ensemble des séquences fréquentes de longueur 2 qui sont {⟨ac⟩: 4, ⟨(bc)⟩: 2, ⟨bc⟩: 3, ⟨ cc⟩: 3, ⟨ca⟩ : 2, ⟨ cb⟩: 3}.

L'exploration de la base de données projetée ⟨ac⟩: {⟨ a (abc) (ac) c⟩, ⟨ ac (bc) a⟩, ⟨(ab) cb⟩, ⟨acbc⟩} génère l'ensemble des motifs de longueur 3 suivant : {⟨acb⟩: 3, ⟨acc⟩: 3, ⟨(ab) c⟩: 2,

⟨aca⟩: 2}. Quatre bases de données projetées seront générées à partir de celles-ci. Si on prend le cas de la base de données ⟨acb⟩— projetée : {⟨ac (bc) a⟩, ⟨(ab) cb⟩, ⟨acbc⟩}, cette dernière ne génère pas de pattern de longueur 4. D'une façon similaire, on peut démontrer que l'exploration des autres bases projetées ne générera pas de pattern de longueur 4.

L'exploration des autres motifs : D'autres sous-ensembles de modèles séquentiels peuvent être extraits de la même façon. Récursivement on obtiendra l'ensemble complet des motifs séquentiels.

- **PrefixSpan : (Prefix-Projected Sequential Patterns Mining)** [26] est une méthode qui permet d'améliorer l'algorithme FreeSpan. En effet, pour éviter de vérifier toutes les combinaisons possibles d'une séquence candidate potentielle, on peut d'abord fixer l'ordre des éléments à l'intérieur de chaque séquence. Si l'on choisit de fixer l'ordre des éléments selon l'ordre alphabétique, la séquence_id 1 sera <a (abc) (ac) d (cf)> au lieu ⟨a (bac) (ca) d (fc). En définissant une telle convention, l'expression de la séquence sera unique.

  PrefixSpan propose également (à l'instar de PSP) d'analyser les préfixes communs que présentent les séquences de données de la base à traiter. À partir de cette analyse, l'algorithme construit des bases de données intermédiaires qui sont des projections de la base d'origine déduites à partir des préfixes identifiés. Ensuite, dans chaque base obtenue, PrefixSpan cherche à faire croître la taille des motifs séquentiels découverts en appliquant la même méthode de manière récursive. Deux sortes de projections sont alors mises en place pour réaliser cette méthode : la projection dite « niveau par niveau » et la « bi-projection ». Au final, les auteurs proposent une méthode d'indexation permettant de considérer plusieurs bases virtuelles à partir d'une seule, dans le cas où les bases générées ne pourraient être maintenues en mémoire en raison de leurs tailles.

  PrefixSpan fonctionne également avec une écriture de la base différente de celle utilisée par GSP. En effet cet algorithme requiert un format qui présente sur une ligne le numéro de client suivi de toutes ses transactions sous forme de séquence de données.

  Ce format nécessite une réécriture de la base de données avant de procéder à l'étape de fouille de données. Nous illustrons, à présent, un exemple de fouille avec PrefixSpan sur la base de données S présentée précédemment.

La méthode de projection préfixée va permettre de procéder à une extraction des motifs séquentiels avec un support minimum de deux clients, en appliquant les étapes suivantes :

- Étape 1 : Trouver les items fréquents. Pour cela, une passe sur la base de données va permettre de collecter le nombre de séquences supportant chaque item rencontré et donc d'évaluer le support des items de la base. Les items trouvés sont (sous la forme <item> : support) : <a> : 4, <b> : 4, <c> : 4, <d> : 3, <e> : 3 ,<f> : 3.

- Étape 2 : Diviser l'espace de recherche. L'espace de recherche complet peut être divisé en six sous-ensembles, puisqu'il y a six préfixes de taille 1 dans la base (c.-à-d. les six items fréquents). Ces sous-ensembles seront : (1) les motifs séquentiels ayant pour préfixe <a>, (2) ceux ayant pour préfixe <b>,… et (6) ceux ayant pour préfixe <f>.

- Étape 3 : Trouver les sous-ensembles de motifs séquentiels. Les sous-ensembles de motifs séquentiels peuvent être trouvés en construisant les projections préfixées des bases obtenues et en réappliquant l'algorithme de fouille de manière récursive.

Le processus de découverte des motifs séquentiels fréquents, sur les bases projetées, se déroule alors de la manière suivante :

Tout d'abord, PrefixSpan cherche les sous-séquences des séquences de données ayant pour préfixe <a>. Seules les séquences contenantes <a> sont prise en compte. De plus dans chacune de ces séquences, seul le suffixe doit être considéré. Par exemple avec la séquence < (e f) (a b) (d f) (c) (b) >, seule la sous-séquence (le suffixe) < (_b) (d f) (c) (b) > sera prise en compte [dans cette séquence le caractère « _ » signifie que le préfixe était contenu dans le même itemset que « b ».

Les séquences de DBspan qui contiennent <a> sont alors projetées pour former DBspan<a>, qui contient quatre suffixes : <(abc) (ac) (d) (cf)> <(_d) (c) (bc) (ae)> <(_b) (df) (c) (b)> et <(_f) (c) (b) (c)>. Une passe sur DBspan<a> permet alors d'obtenir les motifs séquentiels de longueur 2 ayant <a> pour préfixe commun : <(a) (a)> : 2, <(a) (b)> : 4, <(ab)> : 2, <(a) (c)> : 4, <(a) (d)> : 2 et <(a) (f)> : 2.

De façon récursive, toutes les séquences ayant pour préfixe <a> peuvent être partitionnées en 6 sous-ensembles : (1) celles qui ont pour préfixe <(a) (a)>, (2) celles

qui ont pour préfixe <(a) (b)>… et (6) celles qui ont pour préfixe <(a) (f)>. Ces motifs peuvent alors former de nouvelles bases projetées, et chacune de ces bases peut être utilisée en entrée de l'algorithme, toujours de manière récursive.

- **CloSpan : (Mining Closed Frequent Sequential Patterns)** Les algorithmes d'exploration séquentielle de modèles développés jusqu'à présent ont de bonnes performances dans des bases de données constituées de séquences courtes et fréquentes. Malheureusement, lors de l'exploitation de longues séquences fréquentes ou en utilisant des seuils de support très faibles, la performance de tels algorithmes se dégrade souvent de façon spectaculaire. En effet si l'on considère une base de données ne contenant qu'une seule longue séquence fréquente ⟨ (a1) (a2). . . (a100)⟩, cela va générer 2100-1 sous-séquences fréquentes si le support minimum est de 1. CloSpan peut produire un nombre de séquence nettement moins élevé que les méthodes traditionnelles tout en préservant la même performance expressive de ces derniers. En effet CloSpan est un algorithme basé sur le principe depth-first et implémente l'algorithme PrefixSpan. En fait, il s'agit d'une optimisation de ce dernier, destinée à élaguer l'espace de recherche en évitant de parcourir certaines branches dans le processus de divisions récursives (en détectant par avance les motifs séquentiels non clos). Le principe de CloSpan repose sur deux éléments essentiels : l'ordre lexicographique des séquences et la détection de liens systématiques entre deux items [i.e." β apparaît toujours avant γ dans la base de données »] [27].

2.2.1.4 EXPÉRIMENTATIONS ET RÉSULTATS

Afin d'évaluer l'efficacité et l'efficience des différents approches et algorithmes présentés précédemment, nous avons effectué des expérimentations comparatives sur un système équipé de 16 GB de mémoire central et d'un processeur Intel® Core™ i7-4712HQ à 2.30GHZ. Les algorithmes ont été écrits en Java. Le code source a été récupéré depuis le SPMF (librairie open source dédiée à la fouille de données).

Les jeux de données utilisés sont des données fictives générées depuis un générateur de données IBM. La taille des fichiers utilisés varie entre 3,5 Mb et 14 Mb.

Le tableau ci-dessous présente les résultats obtenus lors de l'exécution des algorithmes Apriori, GSP, SPADE, Prefix Span, Clospan, Lapin ainsi que SPAM en utilisant plusieurs jeux de données. Les détails techniques par rapport aux tests que l'on a effectués sont présents au niveau de l'annexe 1 et l'annexe 2.

Algorithme	***Taille de data set***		***Temps d'exécution en sec***	***Consommation de la mémoire en Mégabytes***
AprioriAll (1995) [agrawal and skrikant] (Utilisé jusqu'à maintenant pour la découverte des règles d'association)	Moyenne	Petit min-supp	Très lent	Énorme
		Grand min-supp	Lent	Énorme
	Large	Petit min-supp	Échec	Échec
		Grand min-supp	Très lent	Énorme
GSP (1996) [skrikant et agrawal]	Moyenne	Petit min-supp	>3600	800
		Grand min-supp	2126	687
	Large	Petit min-supp	Échec	Échec
		Grand min-supp	Échec	Échec
SPADE (2001) [Zaki]	- moins performant de SPAM de facteur de 2,5 dans les séquences courtes - plus performant dans les longues séquences.			
SPAM (2002) [Ayers et al.]	Moyenne	Petit min-supp	Échec	Échec
		Grand min-supp	136	574
	Large	Petit min-supp	Échec	Échec
		Grand min-supp	674	1052
PSP (1999) [Masseglia et al.]	Similaire au GSP, mais plus efficace			

FreeSpan (2002) [han et al.]	- Moins performant que PrefixSPan. - Consommation plus petite de mémoire que prefixSpan en cas de « disk based projection », la consommation de prefixSpan est plus efficace en cas de « memory based projection »			
PrefixSpan (2001) [Pei et al.] publié en 2004	Moyenne	Petit min-supp	31	13
		Grand min-supp	5	10
	Large	Petit min-supp	1958	525
		Grand min-supp	798	320
LAPIN (2007) [Yang. Et al.]	Moyenne	Petit min-supp	>3600	Échec
		Grand min-supp	7	8
	Large	Petit min-supp	Échec	Échec
		Grand min-supp	201	300
CM- SPADE (2014) [Fournier-viger et al.]	- 8 fois plus performant que SPADE - Un « memory overhead plus petit avec un grand nombre d'items »			

Tableau 2-2 : Tableau comparatif des différents algorithmes d'extraction des motifs séquentiels

2.2.2 FOUILLE DE FLOT DE DONNÉES (DATA STREAM MINING)

L'ensemble des algorithmes décrits précédemment considère que le processus de fouille s'effectue dans des bases de données qui seront accessibles sans contraintes. En effet dans le cadre de notre projet de thèse, les données seront produites par des capteurs d'activité et de sommeil ce qui constitue un flux de données continu, rapide et sans fin. Par conséquent, le large volume des données générées peut difficilement être stocké dans une base de données classique. Dans cette partie, nous allons présenter la différence entre les flots de données (Data Stream) et les bases de données classiques. Nous allons également présenter les approches et algorithmes existants permettant l'extraction des motifs fréquents dans des flux de données continus.

2.2.2.1 LE MODELE DE FLOT DE DONNÉES

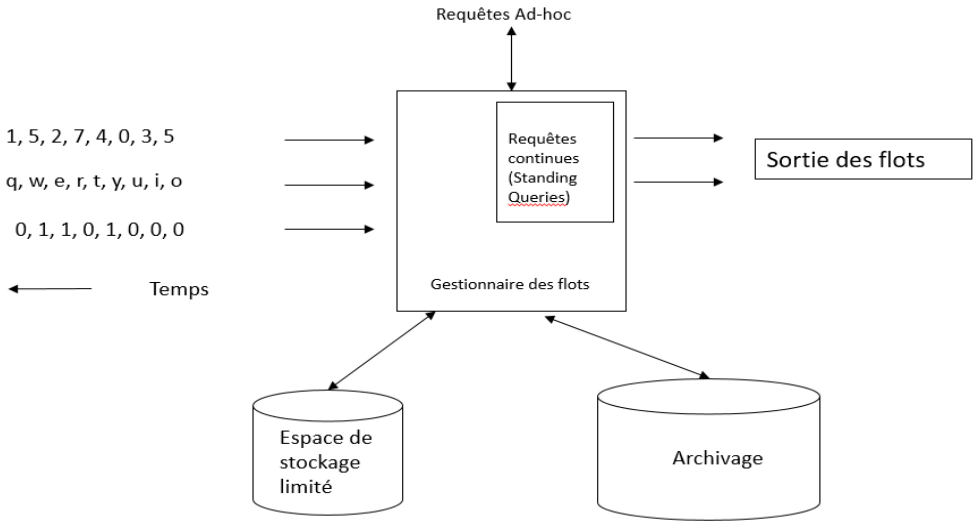


Figure 2.5 : Système de management des flots de données

Par analogie avec un système de gestion de base de données, nous pouvons considérer un gestionnaire de flots de données comme un système de gestion de données dans le cadre de base de données classiques. Les flots de données sont considérés comme des flux de données continu et sans fin. Ces derniers n'ont pas besoin d'avoir les mêmes débits de données ou types de données pour qu'ils soient traités, en outre le temps entre les éléments d'un flux n'a pas besoin d'être uniforme.

Le système de management du flot de données possède deux espaces de stockage, le premier est sous forme d'un entrepôt de données permettant d'archiver les flots de données traités, le deuxième constitue un espace de stockage de travail « Working Storage », ce dernier est utilisé pour stocker les flots de données permettant de répondre aux requêtes. Cet espace de stockage peut être un disque ou une mémoire, tout dépend des besoins en termes des temps de réponse.

Quant aux requêtes utilisées, on peut distinguer deux types : des requêtes continues et des requêtes ad hoc. Pour expliquer le fonctionnement de ces deux types de requêtes, nous allons considérer l'exemple suivant :

Dans le cadre de la télésurveillance médicale avec la plateforme E-care, si nous avons besoin d'une requête continu permettant de lever une alerte si la fréquence cardiaque atteint un seuil min ou max. Dans ce cas de figure, la requête requise est assez simple à mettre en place si l'exécution est effectuée sur le flot de données le plus récent.

Supposant maintenant que l'on ait besoin de connaitre la fréquence cardiaque maximale enregistrée chez un patient. Dans ce cas, il ne serait pas nécessaire d'enregistrer le flot de données dans sa totalité. En effet on aurait besoin juste de calculer le maximum des fréquences cardiaques enregistrées. Pour ce faire, il va falloir calculer la fréquence cardiaque maximale dans chaque flot de données pour qu'ensuite la comparer avec le maximum déjà calculé.

En ce qui concerne les requêtes ad hoc, elles constituent une demande directe d'informations. Ces requêtes sont utilisées pour réduire l'espace de stockage et améliorer les performances. En effet lorsque l'on définit le champ de recherche d'une requête, nous sommes amenés à ne stocker que les parties du flot permettant à répondre à cette requête.

Dans le cas où il serait nécessaire de répondre à plusieurs types de requêtes ad hoc, on utilise une approche nommée « Fenêtre glissante ou Sliding window ». La fenêtre glissante peut être les n derniers éléments d'un Stream comme elle peut être tous les éléments stocker durant

l'unité de temps t : par exemple un jour, une semaine, etc. Ainsi, si nous considérons chaque élément du flux comme un tuple, nous pouvons traiter la fenêtre comme une relation et donc l'interroger avec n'importe quelle requête SQL.

Exemple : Parmi les requêtes qui reviennent souvent dans le cadre de notre projet de thèse, nous avons besoin de connaitre le nombre d'utilisateurs uniques au cours du mois écoulé.

Si nous considérons que chaque connexion en tant qu'élément de flux, nous pouvons définir une fenêtre qui regroupe toutes les connexions dans le mois le plus récent. Nous devons également associer l'heure d'arrivée à chaque connexion, pour pouvoir filtrer les connexions non incluses dans la fenêtre. Si l'on considère la fenêtre comme une relation Logins (nom, heure), une requête SQL simple pourrait nous suffire pour obtenir le résultat souhaité.

SELECT COUNT (DISTINCT [name])
FROM Logins
WHERE time>= t ; // avec t est une constante de temps qui représente une période du mois passé.

2.2.2.2 EXTRACTION DES MOTIFS FREQUENTS DANS LES FLOTS DE DONNÉES

Soit le data Stream DS = Bb_{iai}, Bb_i+1a_i+,…, Bb_{nan} une séquence infinie de batch ; où chaque batch est associé à une période [a_k, b_k]. On considère que Bb_{nan} est le batch le plus récent. Tout Bb_{kak} est un flux de séquence de données tel que Bb_{kak} = [S_1,S_2, S_3, ..., S_j]. Chaque séquence de données S, dispose d'une liste d'itemsets. Nous allons également considérer L= |Bb_{iai}|+| Bb_i+1a_i+1 | … + |Bb_{nan}| la longueur du data Stream tel que |Bb_{kak}| Constitue la cardinalité de l'ensemble Bb_{kak}.

B_0^1	*Sa* *Sb*	(1) (2) (3) (4) (5) (8) (9)
B_1^2	*Sc*	(1) (2)
B_2^3	*Sd* *Se* *Sf*	(1) (2) (3) (1) (2) (8) (9) (2) (1)

Tableau 2-3 : Ensemble des batch B01, B12 et B23

Nous allons définir le support des motifs séquentiels comme suit : le support d'une séquence S à un intervalle de temps [a_i, b_i] est défini par le ratio entre le nombre de (clients, patients, etc.) ayant une séquence *S* appartenant à l'intervalle de temps courant, et le nombre total des (clients, patients, etc.). Cependant, ayant défini un min sup, le problème d'extraction des motifs séquentiels dans des flots de données serait de trouver tous les motifs fréquents S_k dans un intervalle de temps arbitraire tel que :

$$\sum_{t=a_i}^{b_i} support_t\,(S_k) \geq minsupp \times |B_{a_i}^{b_i}|$$

Pour illustrer cela, nous allons considérer l'ensemble de batch du tableau 2.3.

Le tableau 2.4 représente les motifs séquentiels fréquents selon un intervalle de temps [a_i, b_i] et un min support de 50 %.

Intervalles	Batches	Motifs fréquents
[0, 1]	B_0^1	< (1) (2) (3) (4) (5) > < (8) (9) >
[0,2]	B_0^1, B_1^2	< (1) (2) >
[0,3]	B_0^1, B_1^2, B_2^3	< (1) (2) >, < (1) >,< (2) >

Tableau 2-4 : Motifs fréquents dans l'intervalle [a_i, b_i] avec un min sup de 50 %

À partir de cet exemple, on peut remarquer que le support des motifs séquentiels peut varier considérablement selon l'intervalle de temps. D'où la nécessité d'avoir des approches nous permettant de gérer ce fait tout en respectant tous les critères de performances.

2.2.2.3 APPROCHES INCRÉMENTALES

i) ALGORITHME ISE

ISE [28] est l'un des algorithmes les plus efficaces proposés dans le cadre des approches incrémentales. L'ISE réutilise les informations pré calculés concernant les items fréquents à partir des anciennes données afin de réduire la taille du jeu de données à fouiller.

Pour mieux comprendre le fonctionnement de cet algorithme, nous allons considérer les éléments suivants :

- DB une base de données à l'instant t
- Minsupp en tant que support minimum
- L^{DB} l'ensemble des séquences fréquentes dans la base DB
- Db, un incrément de bases de données contenant les nouvelles transactions effectuées sur DB à un instant t+n tel que n>0
- U=DB ∪ db une base de données updatée contenant toutes les séquences de DB et db.
- Nous allons également considérer que toutes les transactions stockées dans db sont triées par id et par temps.

ID	***Itemsets***		
P1	10 20	20	50 70
P2	10 20	30	40
P3	10 20	40	30
P4	60	90	

Itemsets	
50 60 70	80 100
50 60	80 90

Figure 2.6 : Une base de données DB et un incrément de base db contenant de nouvelles transactions

Soit E^{DB} l'ensemble des items fréquents extrait de DB. La difficulté dans une approche de fouille de données incrémentale est d'extraire les motifs fréquents de U, tout en respectant le minSupp. En outre, cette approche doit réutiliser les résultats antérieurs afin de ne pas avoir réexaminer l'ensemble des données.

Premièrement, nous allons traiter le cas où de nouvelles transactions sont ajoutées aux données préalablement existantes. Afin de mieux illustrer ce problème, soit DB la base présentée dans la figure 2.9 de sorte que les transactions soient triées par horodatage (timestamp). En fixant le support à 50 %, le L^{DB} serait {< (10 20) (30)>, < (10 20) (40)>}. Considérant maintenant db l'incrément de la base contenant les nouvelles transactions qui seront ajoutées aux données de P2 et P3. En gardant la même valeur du support, les séquences < (60) (90)> et < (10 20) (50 70)> deviennent fréquentes.

ID	***Itemsets***		
P1	10 20	20	50 70
P2	10 20	30	40
P3	10 20	40	30
P4	60	90	
P5			

Itemsets	
50 60 70	80 100
50 60	80 90
10 40	70 80

Figure 2.7 : Une base de données DB et un incrément de base db contenant de nouvelles transactions

Considérons maintenant le cas où de nouveaux ajouts (patients pour notre exemple) et de nouvelles transactions sont ajoutés aux données préalablement existantes (cf.figure 2.9). En considérant toujours que le support est fixé à 50 %, L^{DB} serait {< (10 20)>} du moment que {< (10 20) (30)>, < (10 20) (40)>} n'apparaissent que pour P2 et P3. Néanmoins, la séquence < (10 20)> reste fréquente si elle apparaît pour P1, P2 et P3. En introduisant l'incrément db, la séquence fréquente L^{U} devient {< (10 20) (50)>, < (10) (70)>, < (10) (80)>, < (40) (80)>, < (60) >}.

Si l'on regarde de près la séquence < (10 20) (50)>, cette dernière aurait pu être détectée dans DB pour P1 or elle ne devient fréquente qu'après l'introduction des données en provenance de l'incrément db. La même chose peut être constatée pour la séquence < (10) (70)>.

Pour résumer, l'algorithme ISE commence par rechercher toutes les nouvelles séquences de taille j $\leq$(k + 1).
L'objectif de cette étape est de rechercher parmi les séquences de DB si celles qui étaient fréquentes le reste en considérant l'incrément mais également de rechercher celles qui n'étaient pas fréquentes précédemment et qui le deviennent en ajoutant des données.
Bien entendu, lors de cette phase, nous recherchons également les séquences qui seront fréquentes sur l'incrément. A l'issue de cette phase, nous obtenons donc toutes les séquences de DB qui deviennent fréquentes, i.e. le nombre d'apparition de ces séquences en prenant en compte DB et db est tel qu'il est supérieur au support, celles qui restent fréquentes avec l'incrément, les séquences fréquentes contenues dans l'incrément et enfin les extensions des fréquentes de DB auxquelles on ajoute un item de db.

ii) ALGORITHME ISM

ISM [29] est un algorithme proposé par S. Parthasarathy, M. Zaki, M. Ogihara, S. Dwarkadas, Il représente une extension de l'algorithme SPADE présenté précédemment et permet de conserver l'ensemble de motifs séquentiels pendant les opérations de mise à jour de la base de données. En effet ISM maintient toutes les séquences fréquentes ainsi que leurs supports dans

la base de données et dans un treillis[11] tandis que les séquences sont contenues dans la bordure négative (selon leurs supports).

Le tableau 2.5 ci-après est un exemple d'une base de données et de sa version mise à jour où l'on observe que 3 éléments ont été mise à jour (éléments en gras).

La première étape de l'algorithme ISM consiste à élaguer les séquences non fréquentes issues des séquences fréquentes après la mise à jour. En effet, un seul scan de la base est suffisant afin de mettre à jour le treillis ainsi que la bordure négative. La deuxième étape prend en compte les nouvelles séquences fréquententes, et ainsi les conserver dans le treillis en utilisant le processus de génération SPADE.

Patient	***ItemSet***	***Items***
1	10	AB
	20	B
	30	AB
	100	**AC**
2	20	AC
	30	ABC
	50	B
3	10	A
	30	B
	40	A
	110	**C**
	120	**B**
4	30	AB
	40	A
	50	B
	140	**C**

Tableau 2-5 : DBspade, une base de données ainsi que sa mise à jour

Exemple : Considérant l'item "C" dans la DBspade (cf tableau 2.5), en effet et d'après SPADE, cet item à un support de 1. Après la mise à jour de la base (éléments en gras du tableau 2.5),

[1] 1 : C'est un ensemble partiellement ordonné dans lequel chaque paire d'éléments admet une borne supérieure et une borne inférieure.

ISM considérera que le support de ce dernier est 4. De ce fait "C" passera de BN (bordure négative) à l'ensemble des séquences fréquentes. Ce cheminement résume la première étape de cet algorithme. Quant à la seconde étape, elle vise à générer les candidats à partir de l'ensemble des séquences fréquentes obtenues lors de la première étape. Pour les séquences <(A) (A) (B)> et <(A) (B) (B)> peuvent générer le candidat <(A) (A) (B) (B)> qui aura un support 0 et sera ainsi ajouté à la BN. Après la mise à jour de la base, l'ensemble des éléments fréquents sera comme suit :

A,B,C,<(A)(A)>,<(B)(A)>,<(AB),<(A)(B)>,<(B)(B)>,<(A)(C)>,<(B)(C)>,<(A)(A)(B)>,<(A B)(B)>,<(A)(B)(B)>, <(A)(A)(C)>,<(A)(B)(C)>.

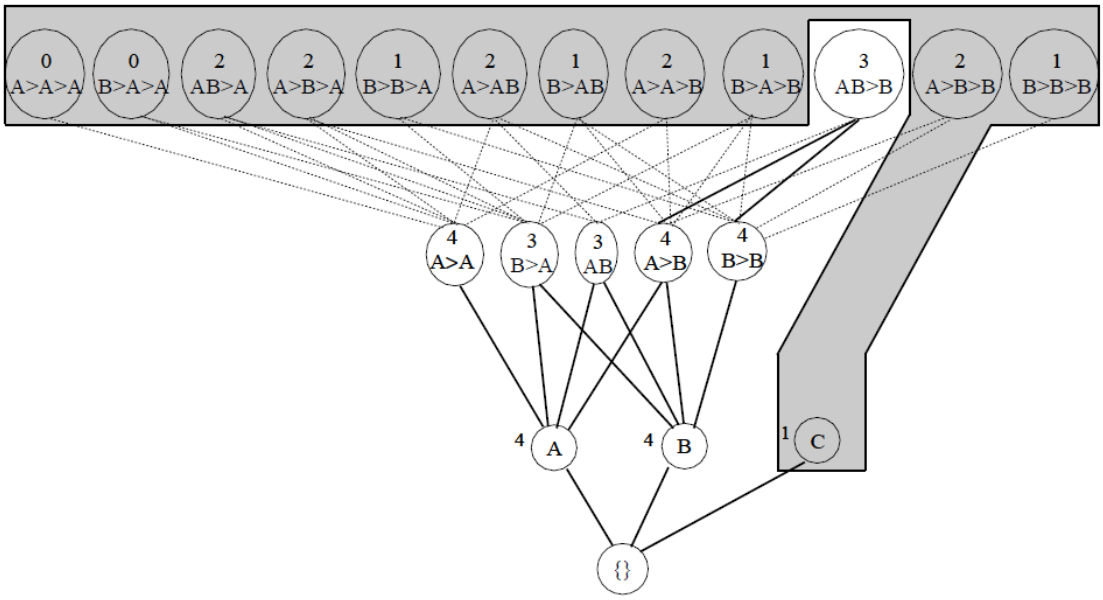


Figure 2.8 : La bordure négative considérée par ISM après l'exécution du SPAD sur la base de données du tableau 2.6 avant la mise à jour

Après la fin de la seconde et dernière étape de l'ISM, le treillis est mis à jour et l'on obtient un nouvel ensemble de séquences fréquentes ainsi qu'une nouvelle bordure négative, permettant à l'algorithme de prendre en compte les nouvelles mises à jour. Comme nous pouvons le constater dans la figure 2.10, le treillis stockant les items fréquents ainsi que la bordure négative peuvent être assez importants et consommateurs de mémoire et de ressources.

iii) *APPROCHES ε-APPROXIMATIFS*

Définition *:*

Considérant un Stream de n items $S_1, S_2 \ldots S_n$. Le support d'un item k est définit par : $f_k = |\{r | s_j = k\}|$. Le problème des exacts items Φ fréquents se résume à ce que ces derniers appartiennent à l'ensemble $\{i | f_i > \phi_n\}$.

On définit le problème des items fréquents ε-approximatifs comme suit :

Afin d'extraire l'item R fréquent, on relaxe les conditions de sorte qu'il soit compris dans l'ensemble $\{i | f_i > (\phi - \varepsilon) n\}$ et que tous éléments vérifiant $f_i > \phi n$ est fréquent.

Notons que tous les algorithmes présentés par la suite sont considérés comme étant ε-approximatifs.

iv) *ALGORITHMES BASES SUR LE COMPTAGE DES ITEMS (COUNT BASED ALGORITHMS)*

Les algorithmes basés sur le comptage des items conservent un sous ensemble constant contenant les items du Stream ainsi que leurs occurrences.

- **L'algorithme Misra-Gries** [30] : Afin d'extraire les k fréquents items, l'algorithme conserve les k-1 (item, nb d'occurrences) couples. Quand un nouvel item apparaît, il est comparé avec les items stockés précédemment. Si ce dernier ne fait pas partie des items existants et s'il y'a moins de k-1 éléments stockés, le nouvel item est donc stocké et son occurrence est ainsi incrémentée à 1. Dans le cas où l'on a déjà k-1 items stockés, leurs occurrences sont décrémentées de 1 par la suite on supprime tous les éléments ayant une occurrence 0. Cet algorithme est assez simple à implémenter et il est considéré comme peu consommateur de mémoire.

- **Lossy counting** [31] : Manku and Motwani sont les auteurs de cet algorithme. Dans leurs approches, ils ont considéré que les items sont stockés dans le tuple (item, f, Δ) tel que Δ est le seuil maximum d'erreur possible et f l'occurrence actuelle de l'item. Comme convenu, les Stream sont devisés en Batch dont la taille est w= [1/ε] et des id démarrant à 1. Soit l'item I l'ième item, l'algorithme vérifie s'il est déjà stocké et si c'est

le cas, son occurrence f sera incrémentée. Sinon, on crée un nouvel enregistrement ayant comme identifiant (I, 1, bc -1) tel que bc est l'id du bucket courant. L'algorithme supprime également tous les items vérifiant $f+\Delta \leq bc$.

v) ALGORITHMES SKETCH

Les algorithmes sketch ont la particularité d'utiliser des structures de données nommées sketch (résumé). En effet, cela permet d'améliorer les performances liées à l'estimation des supports des items.

Il existe plusieurs algorithmes utilisant la notion de sketch, parmi lesquels, nous allons présenter l'algorithme Count Sketch et l'alogirthme Count Min sketch.

- **Count Sketch [Moses, Kevin Chen and Martin Farach Colton Charikar] [32]**

  Cet algorithme utilise une structure de données nommée count sketch. Count sketch est une matrice de compteur C dont la taille est t*b (les valeurs de b et t seront définies ci-après).
  Soit $h_1,\ldots, h_t$ une fonction de hachage mappant les items à $\{1,\ldots, b\}$ et $s_1,\ldots, s_t$ mappant les items à $\{-1,+1\}$. Tout item i est haché via tous les s_j, les valeurs correspondantes seront ajoutées à la matrice $[C (j, h_j [i]]$ où $1 \leq j \leq b$.

  L'algorithme maintien une file d'item de taille fixe contenant les items les plus fréquents ainsi que leurs occurrences. Le processus d'ajout début lorsqu'un nouvel item est ajouté à la file. La première étape consiste à vérifier si ce dernier existe dans la file afin que son compteur soit incrémenté. Sinon on utilise le count Sketch pour estimer l'occurrence de ce nouvel item et voir ainsi s'il vérifie le min Count défini dans la file. Si c'est le cas, on supprime le min item de la file est on y ajoute le nouvel item.

  Ainsi on détermine les valeurs de t et b tel que $t=\log 4/\delta$ où δ est la probabilité d'échec de l'algorithme et $b=e/\varepsilon^2$

- **Count-Min Sketch [G. Cormode, S. Muthu Krishnan] [33]**

  Soit un Stream St $=< a_1, a_2\ldots a_t >$ d'objets $o_1, o_2\ldots o_n$, dont l'occurrence est consécutivement $f(o_1)$, $f(o_2)$, …, $f(o_n)$. L'algorithme Count-Min Sketch permet

d'obtenir une valeur approximative de $f(a)$, c'est-à-dire que l'on peut obtenir le nombre d'occurrences d'un élément a à un instant t quelques qu'il soit.

Soit $\widehat{f(a)}$défini comme suit : $f(a) \leq \widehat{f(a)} \leq f(a)+\varepsilon t$. La structure des données du Count-Min Sketch est une matrice de taille d*w tel que w = $[(\frac{e}{\varepsilon})]$ et d=$[\ln(\frac{1}{\delta})]$. Elle inclut également la fonction de hachage d tel que $h_1, \ldots, h_d : \{1\ldots n\} \rightarrow \{1 \ldots w\}$ où n est le nombre des différents objets contenus dans notre Stream.

Chaque fonction de hachage est associée à une ligne de la matrice alors que toutes les fonctions de hachage sont considérées comme étant indépendantes par paire (cf figure 2.11).

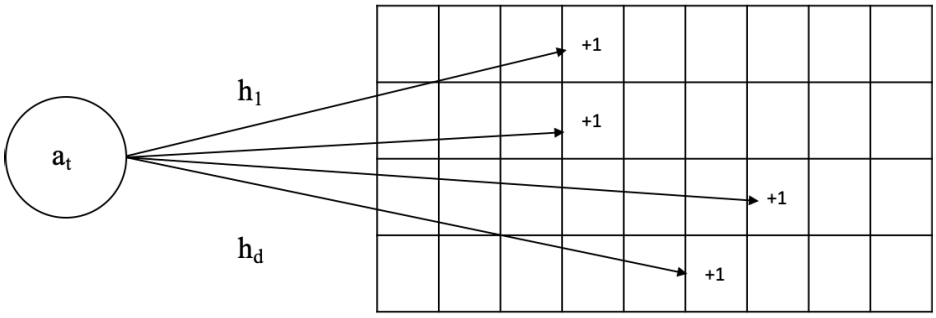


Figure 2.9 : Exemple de matrice utilisée par l'algorithme Count-min Sketch

Soit (i_d , c_d) une mise à jour de la matrice. Cette dernière se traduit comme suit : C [j, h_j (i_t)] = C [j, h_j (i_d)] +cd $\forall 1 \leq j \leq d$

Une estimation de la fréquence d'un item est ainsi obtenue par le min (C [j,h_j(i)]) tel que $1 \leq j \leq b$.

Pour résumer ce qui a été expliqué précédemment concernant cet algorithme, le problème ici est de stocker le nombre d'occurrences de tous les éléments dans un flux. Les paramètres de la structure de données peuvent être choisis de manière à obtenir une précision souhaitée.

L'algorithme Count min sketch (CM) fonctionne comme suit : nous avons k différentes fonctions de hachage et k différentes tables qui sont indexées par les sorties de ces fonctions. Les champs dans les tables sont désormais des valeurs entières. Initialement, nous avons initialisé tous les champs à 0. Lorsque nous augmentons le nombre

d'occurrences d'un élément, nous incrémentons tous les champs k correspondants dans les différentes tables (données par les valeurs de hachage de l'élément). Si une opération de diminution est autorisée (ce qui rend les choses plus difficiles), nous soustrayons également une valeur de tous les éléments k.

Pour obtenir le compte d'un élément, nous prenons le minimum des k champs qui correspondent à cet élément. Ce minimum risque de ne pas retourner une valeur correcte s'il y a eu une collision des valeurs dans la table de hachages.

Les caractéristiques des méthodes incrémentales peuvent, dans un premier temps, le principal point commun est la nécessité de maintenir une connaissance qui est devenue obsolète suite à des mises à jour dans les données. Les connaissances sont calculées au temps t sur la base des données disponibles et doivent être révisées au temps t + n suite à des modifications. Cependant les différences entre le calcul incrémental et le calcul sur les flux sont plus importantes que les ressemblances

La version incrémentale d'une méthode résout le même problème que la version statique, mais en prenant en compte la mise à jour. Dans un problème de flux de données, il est généralement admis qu'on ne peut pas résoudre le même problème que sur les données statiques à cause des contraintes liées au flux. Par exemple l'extraction d'itemsets sur un flux impose une approximation dont on peut s'affranchir avec les données statiques. La définition d'un problème connu sur les données statiques se verra donc adaptée dans sa version flux de données.

2.2.3 FOUILLE DES DONNÉES DANS UNE ARCHITECTURE DISTRIBUÉE

Au cours des trois dernières décennies, de nombreux algorithmes d'extraction des motifs fréquents ont été proposés par les chercheurs. Ces algorithmes étaient rapides et efficaces jusqu'à l'émergence du Big Data durant la dernière décennie. À l'ère du Big Data, les données sont produites à une vitesse telle que ces algorithmes sont incapables de les exploiter. Pour remédier à cela, des algorithmes d'exploration basés sur des architectures parallèles et

distribuées ont vu le jour. La majorité de ces derniers sont basés sur le paradigme MapReduce ; Apriori est considéré comme étant un des algorithmes les plus simples à paralléliser.

De ce fait, la plupart des chercheurs utilisent l'approche Apriori combinée avec le Framework MapReduce pour la mise en œuvre d'algorithmes d'exploration de motifs fréquents.

En 2008, Li [34] a proposé un algorithme d'extraction des motifs fréquents intitulé PFP (Parallel Frequent Pattern). Cet algorithme est une implémentation parallèle de l'algorithme FP-Growth (Frequent Pattern-Growth) basé sur le paradigme MapReduce. Il élimine les exigences de répartition des données et de charge en utilisant le paradigme MapReduce. Cette version scalable était tout à fait adaptée à l'exploration de données Web. PFP recherche des top-k paternes au lieu de se baser sur le support minimum spécifié par l'utilisateur, ce qui le rend efficace pour l'exploration de données Web. L'auteur l'appliquait principalement sur le journal de requête pour proposer des recommandations de recherche. La technique d'équilibrage de charge (load balancing [35]) PFP n'était pas aussi efficace, c'est pourquoi Zhou [36] a proposé un nouvel algorithme appelé BPFP (Balance Parallel FP-Growth). L'algorithme BPFP dispose d'une meilleure technique d'équilibrage de charge pour rendre le PFP plus rapide et efficace.

En 2010, Yang [37] a proposé un algorithme très simple, d'exploration de motifs fréquents basé sur MapReduce. Cet algorithme était l'implémentation très simple de l'algorithme Apriori sur Hadoop. Il utilise une seule fois la fonction map et réduit les phases pour obtenir des items fréquents. En 2011, Li [38] a proposé un nouvel algorithme d'exploration de motifs basé sur le cloud computing, utilisant l'implémentation MapReduce d'Apriori en une phase. Ils ont utilisé de meilleures techniques de distribution des données pour améliorer l'efficacité et la rapidité d'Apriori.
Yu [39] utilise une approche totalement différente pour extraire les motifs fréquents. Il a remplacé la base de données transactionnelle par une matrice booléenne. Ainsi, de nombreuses opérations AND avec d'autres opérateurs logiques sont appliquées à la matrice pour trouver des modèles fréquents. Il utilise Hadoop pour le calcul parallèle de la matrice en la scindant en plusieurs parties.

Certains chercheurs se sont concentrés sur l'utilisation de plates-formes dans le cloud telles que EC2 et S3 pour l'exploration rapide et efficace des motifs fréquents.

En 2012, Li [40] a proposé un algorithme d'exploration des motifs fréquents basé sur MapReduce et il l'a exécuté sur un cluster Amazon EC2. Cet algorithme utilise l'approche Apriori. Le déploiement d'Apriori dans une telle structure lui a permis d'obtenir des résultats plus rapides en raison de la puissance de calcul élevée disponible sur EC2. Il utilise Amazon S3 pour le stockage de données. Plus tard, lin [41] a proposé trois algorithmes d'exploration d'extraction basés sur MapReduce, appelés SPC (comptage unique), FPC (comptage combiné passe fixe) et DPC (comptage dynamique).

L'algorithme SPC est la simple implémentation d'Apriori sur MapReduce. L'algorithme FPC fonctionne de la même manière que SPC pour rechercher jusqu'à 2 ensembles d'éléments. Il combine des ensembles de candidats pour les itérations restantes afin d'obtenir des résultats en une seule phase. Cela était utile dans certains cas où la taille de l'ensemble des candidats est petite après deux itérations et que de nombreuses machines du cluster Hadoop restent inactives lors des étapes suivantes. Par défaut, FPC combine un ensemble de candidats de 3 itérations, à savoir un ensemble de candidats de 3 éléments, de 4 éléments et de 5 éléments. FPC présente également l'inconvénient de combiner un nombre fixe d'itérations et de provoquer un crash si le candidat défini pour des itérations plus élevées est volumineux. L'algorithme DPC résout ce problème assez efficacement en combinant des itérations en fonction de la taille du candidat et de la puissance de calcul des machines. Il offre un meilleur équilibrage de la charge pour une efficacité accrue. La même année, Li [42] et Yahya (MRApriori) [43] ont proposé un autre algorithme d'exploration des motifs fréquents basé sur un paradigme MapReduce, qui est une implémentation parallèle de l'algorithme Apriori.

Certains chercheurs utilisent des échantillons aléatoires de base de données pour trouver des motifs fréquents de manière approximative. PARMA [44] fournit des échantillons aléatoires de base de données à diverses machines du cluster. Chaque machine trouve des modèles fréquents pour son échantillon et le réducteur combine les résultats. Cet algorithme ne fournit pas de résultats précis, mais permet à l'utilisateur de définir ses choix pour le pourcentage d'erreur autorisé. Un échantillon aléatoire est de taille fixe et dépend du taux d'erreur autorisé par l'utilisateur. L'inconvénient est qu'il est moins précis que les algorithmes existants basés sur MapReduce. En 2013, Kovacs [45] a utilisé une approche différente en calculant les singletons ainsi que les paires d'éléments fréquents lors de la première itération de MapReduce à l'aide d'une matrice triangulaire. L'ensemble des candidats est généré dans la phase de réduction au lieu de la phase de map dans tous les algorithmes existants basés sur MapReduce. Plus tard,

Oruganti [46] a utilisé l'implémentation parallèle d'Apriori en utilisant le paradigme MapReduce pour explorer les capacités Hadoop. La même année, Yong [47] a proposé un algorithme d'exploration des motifs fréquents basé sur le paradigme MapReduce, qui utilise le cloud pour exécuter une implémentation parallèle de FP-Growth. Il présente deux avantages majeurs par rapport à l'algorithme conventionnel parallèle FP-Growth. Premièrement, il réduit le nombre de lectures de la base de données et, deuxièmement, il optimise la consommation des ressources par rapport à l'algorithme conventionnel parallèle FP-Growth.

Plus tard, Moens [48] a proposé deux algorithmes d'extraction des motifs appelés Dist-Eclat (Distributed-Eclat) et BigFIM. Dist-Eclat est axé sur le concept de l'algorithme Eclat. Il s'agit d'une implémentation de l'algorithme Eclat basée sur le paradigme MapReduce. Il s'est concentré davantage sur la vitesse et convient pour des résultats plus rapides. Quant au BigFIM, il se base sur une approche hybride. Il utilise à la fois l'approche mixte des algorithmes Apriori et Eclat. Il est plus approprié de gérer des bases de données volumineuses de manière optimisée. La même année, Farzanayar [49] utilise l'algorithme IMRApriori (Apriori basé sur MapReduce amélioré) pour analyser d'énormes données sur les réseaux sociaux.

Barkhordari [50] a également proposé un algorithme d'exploration des motifs basé sur MapReduce appelé ScaDiBino (algorithme d'exploration des motifs évolutif et distribuable), qui convertit chaque ligne de transactions en entrée au format binomial. Les données binomiales peuvent être traitées plus efficacement sur MapReduce.

Cet algorithme génère directement un ensemble de motifs fréquents. Ils ont utilisé cet algorithme pour recommander aux clients des services à valeur ajoutée en analysant le trafic réseau des opérateurs de téléphones mobiles.

Certains chercheurs utilisent diverses structures de données pour améliorer l'efficacité des algorithmes d'exploration des motifs fréquents. Singh [51] essaie d'utiliser une table de hachage pour le stockage de candidats dans une implémentation basée sur Apriori MapReduce. Ils constatent que la table de hachage est plus efficace que d'autres dans le contexte MapReduce alors qu'elle n'est pas très efficace dans une approche séquentielle. Sur la plate-forme Hadoop, les résultats sont stockés dans HDFS (système de fichiers distribué hadoop) après chaque itération. Ils sont à nouveau générés à partir de HDFS en tant qu'entrée pour l'itération suivante, ce qui diminue les performances en raison du temps d'entrée/sortie élevé. Mais Spark [n], une nouvelle plate-forme de flux de données distribuées en mémoire, résout ce problème en utilisant

son architecture RDD. Les RDD stockent les résultats dans la mémoire principale à la fin d'une itération et les rendent disponibles pour la prochaine itération. L'implémentation traditionnelle d'Apriori sur la plate-forme Spark donne des résultats beaucoup plus rapides sur des jeux de données standard, ce qui fait de Spark l'un des meilleurs outils pour l'implémentation d'Apriori.

Zhang [52] a proposé un algorithme d'exploration des motifs appelé DFIMA (Algorithme d'extraction d'items fréquents distribués) qui est implémenté sur Spark. L'algorithme DFIMA utilise une technique d'élagage à base de matrices pour réduire la taille du candidat. L'auteur affirme qu'il surpasse l'algorithme PFP lorsque les deux sont implémentés sur Spark. Récemment, notre nouvel algorithme d'exploration des motifs fréquent a utilisé une approche réduite pour la 2e itération afin de réduire les calculs. Notre approche a surpassé tous les algorithmes d'extraction des motifs en termes de précision et de performances.

2.3 POSITIONNEMENT PAR RAPPORT A L'ETAT DE L'ART, CONTRIBUTIONS ENVISAGEES

Dans nos travaux de thèse nous avons envisagé la mise en place d'une approche d'exploration et d'extraction des données préservant l'expérience existante au sein de la plateforme E-care existante. L'approche que l'on a proposée s'articule autour de trois éléments essentiels que sont l'algorithme d'extraction Apriori, le Framework Spark ainsi que les structures de hachage Cuckoo. Le filtre Cuckoo, particulièrement performant, permet d'assurer une faible probabilité de faux positifs.

La combinaison de ces derniers nous a permis de concevoir une solution performante permettant l'extraction des motifs évolutifs.

3 CONTRIBUTIONS

3.1 INTRODUCTION

À travers ce chapitre, nous allons proposer une définition de l'évolution des motifs en les situant par rapport à notre modèle de données. Par la suite, nous allons présenter notre approche d'extraction des motifs. Pour ce faire, nous allons procéder comme suit :

- Présenter l'architecture LAMBDA
- Présenter le Framework Spark
- Présenter les bases du Cuckoo filter
- Présenter notre proposition permettant l'extraction des motifs
- Proposer une définition de l'évolution de motifs, exprimable à travers le concept de motif évolutif.
- Mettre la définition proposée à l'épreuve de données réelles, pour valider sa pertinence.

En d'autres termes, le but de chapitre est de détailler la partie théorique de nos contributions en présentant les différents éléments caractérisant l'évolution de motifs et sa représentation par des motifs évolutifs ainsi que les choix architecturaux proposés pour concevoir notre approche d'extraction des motifs.

3.2 UNE PREMIERE FORMALISATION DES MOTIFS EVOLUTIFS

3.2.1 MOTIFS ET BASES DE TRACES

Nous reprenons ici les définitions données en section 2.2.1.1 concernant les notions d'événement, de type d'événement, de séquence et de motif.

Définition 1 (Événement). *Soit E un ensemble d'éléments, nommés* ***types d'événements***. *Un* ***événement*** *est un couple (e, t) où e ∈ E et t est la date d'apparition de l'événement.*

Définition 2 (Séquence). *Une* ***séquence*** *est une suite non vide d'événements ordonnée selon leur date d'apparition. Une séquence est générée par une source.*

***Définition 3 (Motif)**. Un **motif** est une suite non vide de types d'événements.*

***Définition 4 (Occurrence d'un motif)**. Étant donnée une séquence S et un motif m, une **occurrence** de m dans S est une sous-séquence de S qui vérifie :*

- Les types d'événements de m sont présents dans S et dans le même ordre
- Le premier élément de S a pour type le premier élément de m
- Le dernier élément de S a pour type le dernier élément de m
- La durée de l'occurrence est inférieure à un seuil w (nommée taille de fenêtre) fixé par l'analyste.
- La taille de l'occurrence est le nombre d'événements qu'elle contient.

***Définition 5 (Support d'un motif)**. Étant donnée une séquence, le support d'un motif est le nombre d'occurrences de ce motif dans la séquence, selon une méthode de comptage.*

***Définition 6 (Taille, durée et pureté d'une occurrence de motif).** La **taille** d'une occurrence d'un motif est équivalente au nombre d'événements la constituant, et sa **durée** est le temps écoulé entre le premier et le dernier événement. La **pureté** d'une occurrence est le ratio entre le nombre de types d'événements du motif et le nombre d'événements de l'occurrence.*

***Définition 7 (Support d'un motif).** Étant donnée une base de séquences, le **support** d'un motif est le nombre d'occurrences du motif dans la base, selon une méthode de comptage de l'état de l'art. Un motif est **fréquent** si son support est supérieur ou égal à un seuil de support défini pour la recherche effectuée.*

Ces définitions sont illustrées sur la figure 3.1, où on considère un ensemble de types d'événements *A, B, C, D* et une base de traces[2] [53] *B* composée de deux séquences s_1 et s_2. Sur cette base, on observe trois occurrences du motif *M* = (*A, B, B, C)*. La première est de taille 4, sa durée est de 6 secondes, et sa pureté est de 100 %, la deuxième est de taille 4, sa durée est des 4 secondes et sa pureté 100 %, alors que la troisième est de taille 6 avec une durée de 12 secondes et une pureté de 67 %. Selon la façon de compter les occurrences, le support du motif peut varier. Par exemple, si l'on considère que le support correspond au nombre de séquences où le motif est présent, il

2 : Le terme Trace est employé comme un concept et non dans son sens commun.

vaut 2, alors que si l'on compte toutes les occurrences, il vaut 3.

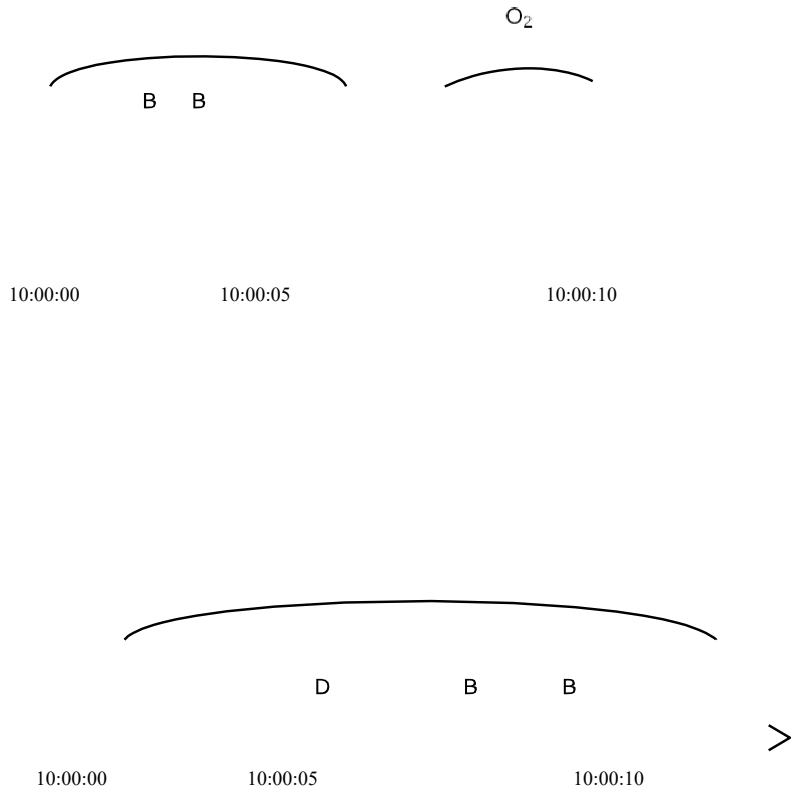


Figure 3.1 : Trois occurrences d'un motif (A, B, B, C)

3.2.2 ÉVOLUTION D'UN MOTIF

Le concept de motif qui évolue dans le temps revêt un caractère contradictoire. En effet, pour pouvoir être détecté dans les données, un motif est nécessairement une structure stable, c.-à-d. capable de se répéter à l'identique. Extraire un motif évolutif dans des données suppose donc d'identifier deux aspects complémentaires :

- Une stabilité dans le motif, qui indique qu'on observe bien une seule évolution plutôt que des éléments indépendants, en respectant une contrainte d'identité.

- Un changement dans le motif, qui met en évidence la différence entre deux étapes de l'évolution, tout en restant circonscrit à un périmètre maximal d'évolution.

Par *étape de l'évolution*, on réfère aux différents éléments identifiés comme liés au sein de l'évolution d'un même motif. Ensemble, ces deux aspects forment un cadre d'évolution pour les motifs, que nous définissons de la manière suivante :

Définition 8 (Cadre d'évolution). *Un* ***cadre d'évolution*** *détermine dans quelle mesure un motif peut évoluer au fil du temps. Il se compose de deux parties :*

- *Une* ***contrainte d'identité****, qui indique les conditions que doivent respecter les différentes étapes de l'évolution*
- *Un* ***périmètre d'évolution****, qui indique la mesure dans laquelle les différentes étapes de l'évolution peuvent se différencier les unes des autres*
- *Il est également nécessaire d'identifier les aspects du motif pouvant évoluer. Pour cela, nous considérons plusieurs* ***types d'évolutions*** *sur les motifs :*
    - *Évolution des caractéristiques des occurrences du motif : durée, pureté.*
    - *Évolution des caractéristiques du motif : fréquence, syntaxe, propriétés des événements.*

La liste de ces types d'évolution se base sur celle établie par Tsai et Shieh [54] qui ne considéraient que les évolutions sur la fréquence et la syntaxe.

Pour chacun de ces types d'évolution, nous déterminons ce qui constitue le cadre d'évolution.

Ces résultats sont présentés dans le tableau 3.1

Type d'évolution	Contrainte d'identité	Périmètre d'évolution
Évolution de la durée	Les occurrences sont celles d'un même motif	Les occurrences n'ont pas toutes la même durée
Évolution de la pureté	Les occurrences sont celles d'un même motif	Les occurrences n'ont pas toutes la même pureté
Évolution de la fréquence	Les occurrences sont celles d'un même motif	Les occurrences n'ont pas toutes la même fréquence locale. Elles apparaissent par - fois en étant concentrées dans le temps, parfois en étant isolées. La diminution de la fréquence peut aller jusqu'à la disparition complète du motif sur une période.
Évolution de la syntaxe	Les différentes syntaxes sont semblables, au regard d'un seuil minimal de similarité, car l'évolution est progressive	La syntaxe peut changer, donnant naissance à plusieurs motifs extraits des données
Évolution des propriétés des événements	Les occurrences sont celles d'un même motif	Les propriétés des événements qui forment les occurrences peuvent changer

Tableau 3-1 : Cadre d'évolution de chaque type d'évolution

L'évolution de la syntaxe est particulière par rapport aux autres évolutions, puisqu'elle fait intervenir plusieurs motifs (tels qu'ils peuvent être extraits par un algorithme standard de l'état de l'art) qu'il faut comparer.

On peut également noter que les périmètres d'évolution et les contraintes d'identité présentés dans le tableau 3.1 sont formulés de la façon la plus générale possible. Selon les besoins spécifiques à un cadre applicatif, on peut envisager de les exprimer de façon plus

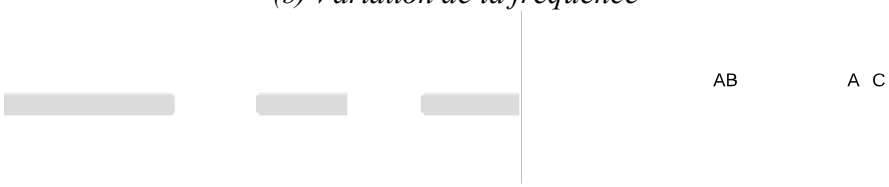
précise. Par exemple, si on s'intéresse aux évolutions de la syntaxe d'un motif, mais uniquement lorsque sa taille ne change pas, on peut ajouter cette condition à la contrainte d'identité qui devient ainsi « Les différentes syntaxes sont semblables, au regard d'un seuil minimal de similarité, car l'évolution est progressive, et la taille ne change pas ».

Ces différents types d'évolution sont illustrés dans la figure 3.2.

- Une évolution au niveau de la syntaxe des motifs (3.2a). Cette évolution s'opère de façon régulière par un enchaînement de petites modifications (un élément à la fois).
- Une évolution au niveau de la fréquence des motifs (3,2 b), où un motif *A, D, C* est peu fréquent, avant d'apparaître plusieurs fois sur un intervalle réduit
- Une évolution au niveau de la durée des motifs (3,2 c), où un motif voit la durée de ses occurrences décroître au fil du temps.

(a) Variation syntaxique

(b) Variation de la fréquence



(c) *Variation de la durée*

3.2.3 PARTITIONNEMENT DES SEQUENCES

L'évolution n'étant pas un phénomène observable à un instant fixe, il est nécessaire de déterminer la façon dont le temps va être représenté. On entre ici dans une autre des tâches définies par Böttcher et collab. [**55**], la spécification du modèle de temps. Pour cela, nous proposons de partitionner les séquences en intervalles de temps, afin de rechercher les différents types d'évolution.

Nous définissons une partition de la façon suivante :

Définition 9 (Partition). *Étant donné une base de séquences $B = \{s_j\}$ et un ensemble $T = \{t_i\}$ de dates présentent sur l'intervalle de temps couvert par B, la* ***partition*** *de B selon T est l'ensemble des sous-séquences issues de B en utilisant les dates t_i comme points de coupure.*

Le choix de la partition utilisée lors d'une recherche d'évolution est déterminant, puisqu'il a un impact sur les résultats obtenus. Cette importance est illustrée dans la figure 3.3, où on voit que dans la première partition, le motif M_1 est présent de façon constante sur les deux intervalles. En revanche, pour les mêmes données, on observe sur la deuxième partition que ce motif est présent sur le premier intervalle, disparaît sur le deuxième, puis réapparaît sur le troisième.

Figure 3.3 : Deux partitions possibles sur des données identiques

Comparer deux intervalles de temps différents permet de détecter une évolution monotone. Pour généraliser à une combinaison non monotone, il suffit de combiner des comparaisons monotones deux à deux. Dans les cas les plus simples, on se retrouve ainsi face à deux cas de figure, selon que l'on veuille comparer des intervalles consécutifs ou non. Pour le cas où les intervalles sont consécutifs, une bipartition permet de les identifier. Pour le second cas, l'utilisation d'une tripartition permet d'identifier les deux intervalles que l'on souhaite comparer, ainsi que l'intervalle les séparant. Cette tripartition peut être obtenue en combinant deux bipartitions.

De façon plus générale, toute partition de taille supérieure à 2 peut être obtenue en considérant une combinaison de bipartitions. Ainsi, toute évolution présente dans une séquence peut être considérée comme identifiable à partir d'évolutions considérées sur des bipartitions.

Une illustration de ces deux cas de comparaison d'intervalles (bipartition et partition de taille supérieure) pouvant être réalisée avec chaque type d'évolution identifié, on présente ici le cas de l'évolution de la fréquence d'un motif. L'augmentation du nombre de points de partition rendant la lisibilité des données plus compliquée, nous illustrons ici les partitions de taille supérieure à 2 par leur cas le plus simple, la tripartition. Nous utilisons le cadre d'évolution le plus général, c'est-à-dire que la contrainte d'identité implique que la fréquence d'un même motif est étudiée, et le périmètre d'évolution demande un changement de cette fréquence pour pouvoir identifier une évolution.

Le premier cas, illustré par la figure 3.4, représente l'étude de la variation du motif sur deux intervalles consécutifs. On a donc besoin pour chaque séquence d'une **bipartition**. On observe ainsi le passage d'un intervalle sur lequel un motif M1 est très fréquent à un intervalle sur lequel ce même motif n'est pas fréquent, un motif M2 étant plus présent. On est donc en présence d'une diminution de la fréquence de M1.

:

Figure 3.4 : Évolution de la fréquence d'un motif sur une tripartition

Le deuxième cas, illustré par la figure 3.5, représente l'étude de la variation du motif sur deux intervalles non consécutifs. On utilise donc une tripartition. On observe ainsi le passage d'un intervalle où le motif M1 est très fréquent à un intervalle où il ne l'est plus, avant de revenir sur un intervalle où M1 est fréquent (il est même le seul motif présent). La fréquence du motif M1 présente donc une décroissance puis une augmentation au fil du temps.

>

La combinaison de motifs évolutifs servant à observer des évolutions sur un enchaînement de bipartitions, celles-ci doivent être combinées par la même occasion. Pour ce faire, il est nécessaire que les intervalles qu'elles décrivent soient raccordés les uns aux autres, de façon à étudier une évolution sur une période continue. Chaque bipartition étant identifiée par un point de partition situé sur une sous-base BI d'une base B, il est nécessaire qu'une fois ces sous-bases ordonnées chronologiquement, le second intervalle d'une sous-base soit identique au premier intervalle de la sous-base suivante.

À partir de ces observations, et afin de couvrir les deux types d'ordre identifiés, on propose la définition suivante pour l'ordre d'un motif évolutif :

Définition 10 (Ordre d'un motif évolutif). *L'**ordre d'un motif évolutif** caractérise une hiérarchie entre les motifs évolutifs. Sa valeur est déterminée de la façon suivante :*

- *Si le motif évolutif est identifié sur une bipartition, l'ordre est de 1*
- *Si le motif évolutif est obtenu à partir d'un ensemble de motifs évolutifs identifiés sur des bipartitions, celles-ci doivent une fois ordonnées chronologiquement former une séquence continue, sur laquelle chaque bipartition a pour second intervalle le premier intervalle de la bipartition suivante. Si cette condition est remplie, l'ordre du motif évolutif est égal à la taille de l'ensemble de bipartitions.*

Au vu de tous les éléments présentés précédemment, on peut dire que l'on est en mesure de catégoriser l'ensemble des motifs évolutifs les uns par rapport aux autres, selon une relation d'imbrication entre motifs d'ordre différents. On est ainsi en mesure de décrire une évolution sur un intervalle de temps divisé en un nombre quelconque de partitions en

combinant des bipartitions.

Après avoir proposé une définition de la notion de l'évolution d'un motif, nous allons présenter dans ce qui suit les éléments que nous avons utilisés pour concevoir notre solution d'extraction des dits motifs.

3.3 L'ARCHITECTURE LAMBDA

L'architecture Lambda est une approche hybride de traitement des données qui permet un traitement par lots et en quasi-temps réel. Cette dernière est née de l'idée qu'un seul outil ou application ne peut résoudre tous les problèmes du Big Data. Cependant, on pourrait penser à utiliser plusieurs outils pour construire un Framework contenant des couches. Chaque couche servira à satisfaire un besoin bien défini.

Dans ce qui suit, nous allons présenter l'architecture Lambda en mettant en avant ses différents éléments.

L'architecture Lambda satisfait les trois modèles ci-dessous :

- Tolérance aux pannes
- Immutabilité des données
- Recomputation

L'immuabilité des données signifie que le système stockera les données, mais ne les mettra jamais à jour une fois qu'elles auront été écrites. Ainsi, parmi les quatre fonctions de CRUD1, nous effectuons la création, la lecture et la suppression, mais aucune mise à jour. Ceci mène à la troisième action, ce qui signifie qu'il sera possible de récupérer les résultats à tout moment.

La théorie du CAP est un élément moteur pour choisir la bonne plate-forme de données partagées pour chaque couche de l'architecture Lambda. Cette théorie indique qu'il est

impossible pour un système de calcul distribué de gérer toutes les propriétés ci-dessous :

- Cohérence
- Disponibilité
- Tolérance au partitionnement
- Cohérence (C) : tous les nœuds du système voient exactement les mêmes données au même moment.
- Disponibilité (A) : garantit que toutes les requêtes reçoivent une réponse.
- Tolérance au partitionnement (P) : aucune panne moins importante qu'une coupure totale du réseau ne doit empêcher le système de répondre correctement.

En partant du fait qu'il est impossible de conserver les trois propriétés dans un même système, les conceptions viables pour une plate-forme de données distribuée sont AP et CP (la CA n'est pas une option cohérente).

La figure ci-après illustre les composants de l'architecture Lambda, que l'on présentera en détaille plus loin.

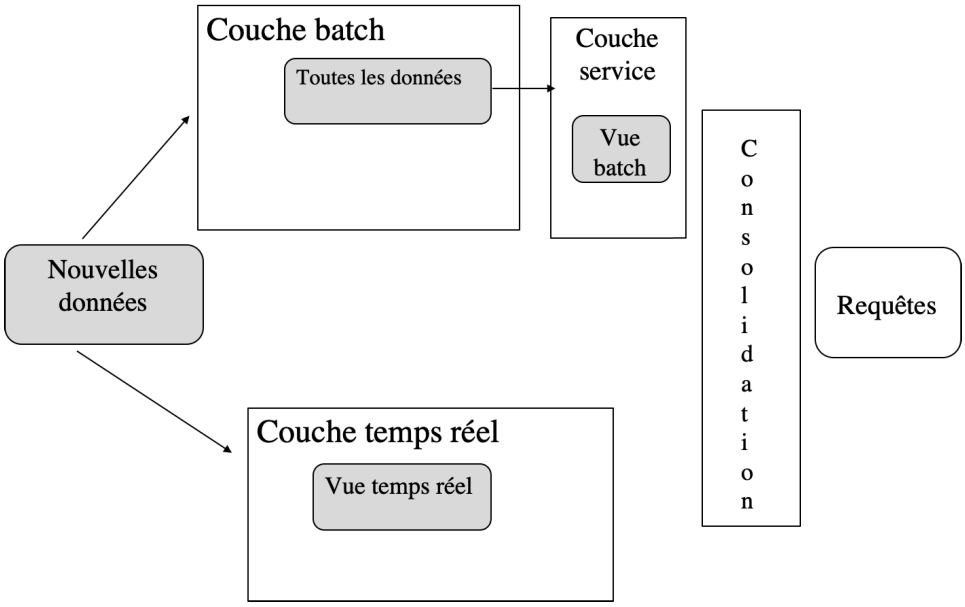


Figure 3.6 : Les composants de l'architecture Lambda [56]

3.3.1 LA COUCHE BATCH

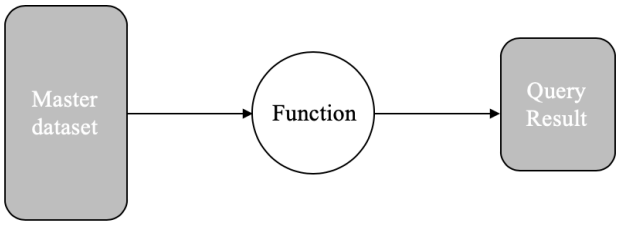


Figure 3.7 : La couche Batch [56]

La couche batch constitue le cœur de l'architecture Lambda. Cette couche étant la couche de traitement par lots, elle devrait idéalement avoir une latence élevée. Cela permettra de réaliser des analyses plus approfondies et des calculs plus lourds sur les données, ainsi que d'assurer la cohérence totale et la gestion des défaillances.

La vue Batch = fonction (toutes les données)

Cette formule définit la couche batch. Elle définit que la couche batch gère la couche maîtresse des données et calcule les vues batch. En d'autres termes, le jeu de données maître est le fondement l'architecture Lambda.

Cette couche a les caractéristiques suivantes :

Elle permet une précision parfaite en permettant de traiter toutes les données disponibles lors de la génération de vues. Cela signifie qu'elle peut corriger les erreurs en récupérant des données sur l'ensemble des données, puis en mettant à jour les vues existantes.

La sortie est généralement stockée dans une base de données en lecture seule, avec des mises à jour remplaçant facilement les vues pré exécutées précédemment.

En ce qui concerne notre sujet, nous avons ici deux algorithmes de calcul appropriés :

- Algorithmes de récupération : élimine les anciennes vues par lots et ré effectue les calculs par rapport à l'ensemble des jeux des données.
- Algorithmes incrémentiels : un algorithme incrémentiel met directement à jour les vues lorsque de nouvelles données arrivent.

3.3.2 LA COUCHE TEMPS REEL

La couche Batch a la capacité de fournir la plupart des caractéristiques souhaitées d'un système Big Data, à l'exception des mises à jour qui ont généralement une faible latence. Cela est principalement dû au temps nécessaire pour que les nouvelles données soient traitées et mises à disposition.

En ce qui concerne la couche performances (temps réel), cette dernière possède deux propriétés essentielles : activer les vues en temps réel et utiliser le flux de données intégré pour mettre à jour ces vues.

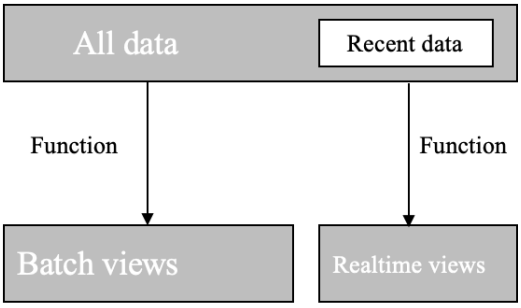


Figure 3.8 : Vue temps réel= fonction (données récentes)

Comme indiqué précédemment, les vues en temps réel de la couche performances sont mises à jour au fur et à mesure de la réception de nouvelles données.

Ceci est significativement différent du processus de la couche Batch. On peut ainsi affirmer que la couche performances effectue une incrémentation accrue.

Pour des raisons de performances et d'optimisation la couche performance se contente de fournir des approximations qui sont remplacées/validées après l'exécution de la couche Batch. En reprenant la théorie du CAP, les propriétés caractérisantes de la couche performances sont la disponibilité et la tolérance au partitionnement (AP).

En vue de la nature des échantillons de données utilisés dans le cadre de notre thèse, nous associons l'aspect de la couche performance au paradigme de la gestion des flux (stream processing). Ceci est développé plus loin dans les sections ci-dessous.

3.3.3 LA COUCHE SERVICE

Le résultat de la couche Batch est un ensemble de fichiers plats contenant les vues pré calculées. La couche service est responsable de l'indexation et du développement des vues afin qu'elles puissent être interrogées.

Dans l'architecture Lambda, la couche service fournit un accès à faible temps de latence aux résultats des calculs effectués sur le jeu de données. Les vues de la couche service sont légèrement obsolètes en raison du temps requis pour la création du lot.

3.3.4 CONCLUSION :

Après avoir donné un aperçu des composants de l'architecture Lambda, nous allons expliquer pourquoi nous avons eu recourt à une architecture de ce type dans le cadre de notre travail de thèse. En effet, l'architecture Lambda palie aux dilemmes rapidité, volumes et précision.

L'idée fondamentale est d'avoir deux voies de données séparées : une couche de traitement rapide pour fournir des résultats à faible latence en utilisant des technologies de traitements de flux et une couche de traitement par Batch pour exécuter les tâches lourdes et fournir des résultats précis sur les données dans leur ensemble. Si l'on revient aux objectifs de notre travail de recherche, notamment à l'extraction des motifs évolutifs. Construire une solution au tour de l'architecture Lambda nous a permis de profiter de la puissance des différentes couches de cette dernière pour pouvoir extraire et suivre le changement des motifs fréquents, voir figure 3 .9 ci-après.

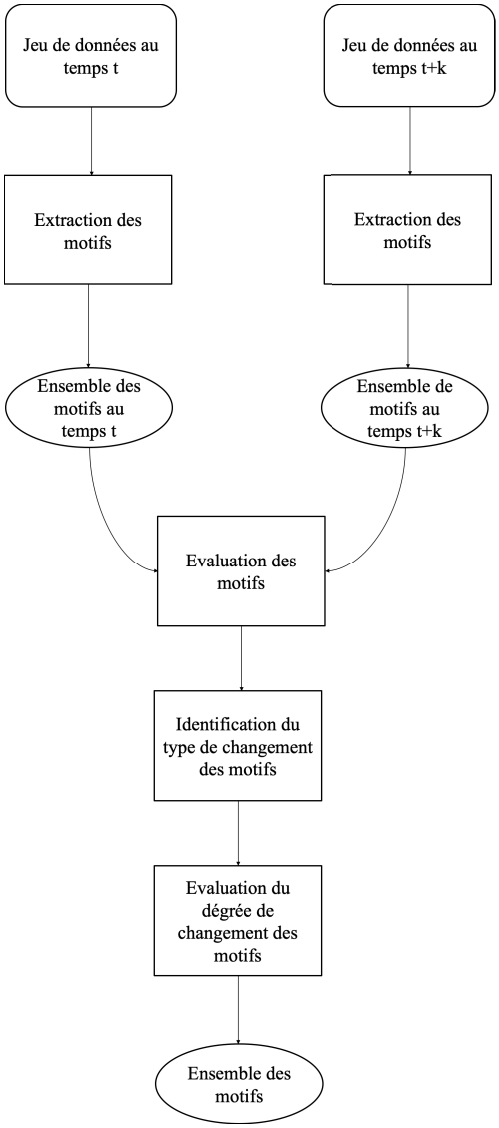


Figure 3.9 : Processus d'extraction des motifs évolutifs

La figure 3.9 schématise le rôle d'une architecture Lambda dans l'extraction des motifs évolutifs. En effet, deux ensembles de motifs sont générés respectivement à l'instant t et t+k par la couche performance et la couche Batch.

Les paires de motifs issues de chaque ensemble sont comparées au niveau de la couche service afin d'en extraire tous les motifs ayant subi un changement selon les critères prédéfinis.

Les motifs résultants sont classés selon le taux de changement du support. On catégorise également l'ensemble des motifs fréquents ayant subi un changement dit inattendu.

En d'autres termes l'architecture Lambda nous permet par le biais de ses deux couches Batch et temps réel de suivre l'évolution des motifs extraits. Elle nous permet également de centraliser les traitements de manière à ce que l'on utilise la même approche pour extraire les motifs que cela soit au niveau de la couche Batch ou performance.

3.4 FRAMEWORK SPARK

Apache Spark est un est un Framework open source de calcul distribué construit autour de la faible latence. Ce Framework a été développé en 2009 au UC Berkeley's AMP Lab.

Spark est un moteur rapide permettant la gestion de données à grande échelle, qui présente plusieurs avantages par rapport à d'autres plates-formes Big Data telles que Storm et Hadoop.

En effet, Spark regroupe plusieurs API tels que SQL, machine learning et l'exploitation des graphes pour le traitement des données à grande échelle. En outre d'être scalable, Spark propose plusieurs Frameworks assurant la distribution et le parallélisme des traitements.

Pour résumer, Spark propose des API pour la collecte de données, l'ETL, l'apprentissage automatique (machine learning), l'exploitation de graphes, la diffusion en continu, la gestion interactive et la gestion par lots. Autrement dit, il peut remplacer plusieurs systèmes d'analyse SQL, de transmission en continu et de traitement Batch par un seul environnement unifié.

3.4.1 COMPOSANTS ET L'ARCHITECTURE DU FRAMEWORK SPARK

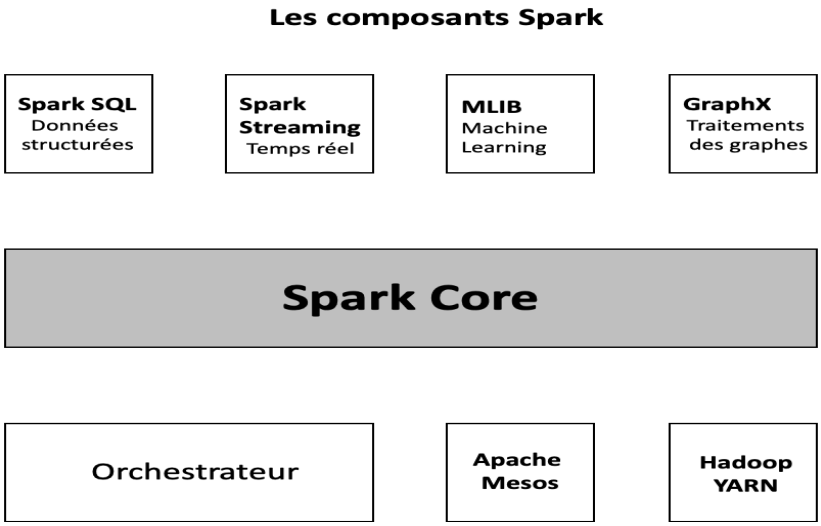


Figure 3.10 : L'écosystème Spark [57]

Le Spark Core constitue comme son nom l'indique le moteur du Framework. Il est en charge des taches ci-dessous :

- Fournir la répartition des tâches distribuées
- L'orchestration et des fonctionnalités lectures/écritures de base
- Le management des applications à travers les clusters
- Gestion de la mémoire
- La tolérance aux pannes
- L'API RDD est implémentée (Resilient Distributed Datasets) sur Spark Core, qui est une collection logique de données partitionnées sur le cluster

En addition à cela, Spark met à disposition des bibliothèques supplémentaires faisant partie du système Spark et fournit des capacités supplémentaires dans les domaines de l'analyse de données à grande échelle et de l'apprentissage automatique.

Lesdites bibliothèques sont les suivantes :

- Spark SQL : Assure le traitement des données structurées
- Spark streaming : Traite les données en temps réel.
- Spark MLlib : Une bibliothèque assurant la partie machine learning.
- GraphX : l'exploitation des graphes pour le traitement des données à grande echelle.
- SparkR : Offre un client léger permettant l'utilisation le Framework Spark avec le langage R.

Spark contient également des composants permettant le management des clusters. Ces derniers sont responsables de l'allocation des ressources nécessaires pour qu'une application donnée puisse s'exécuter dans un cluster. Il existe trois types de ces composants :

- Standalone : un gestionnaire de cluster inclus dans Spark permettant une mise en place facile des clusters.
- Apache Mesos : un gestionnaire de cluster généraliste qui tourne également sous Hadoop.
- Had oop YARN : c'est un gestionnaire de cluster incluant hadoop.

3.4.2 L'ARCHITECTURE DU SPARK RUNTIME

Dans cette partie nous allons introduire l'architecture du Spark Runtime.

Spark a une architecture maître/esclave où le maître est le pilote et les esclaves sont les exécuteurs. Les pilotes et les exécuteurs s'exécutent dans leurs propres processus Java. Voir la figure 3.12.

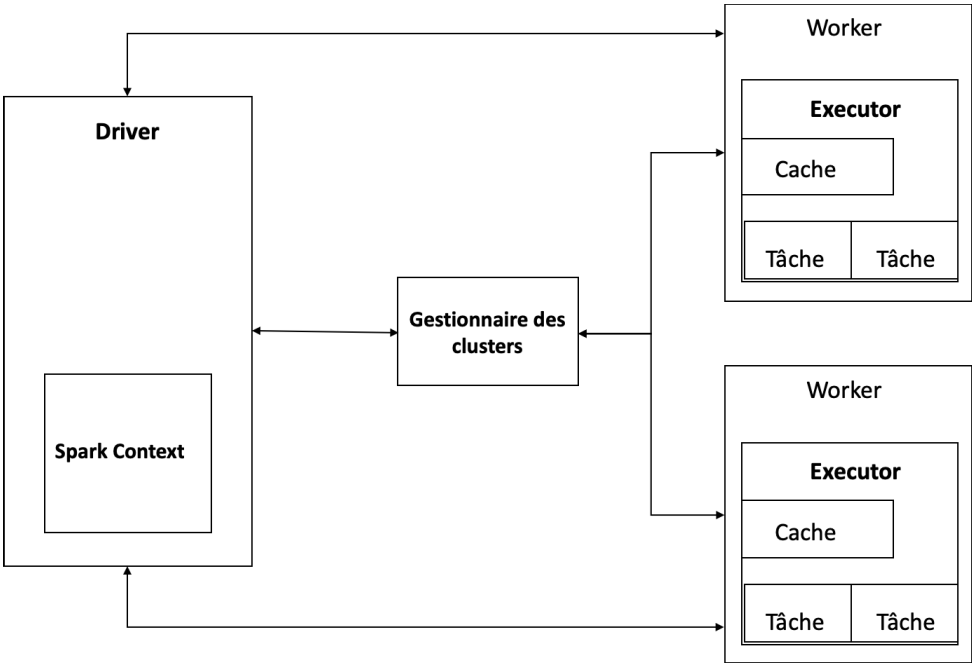


Figure 3.11 : Architecture du Spark Runtime [57]

L'architecture du Spark Runtime est composée du SparkContext, du Driver, d'un gestionnaire de cluster ainsi que d'un Exécuteur (Executor).

Le tableau ci-dessous décrit le rôle de chacun de ces éléments.

Composant	***Rôles***
SparkContext	• Point d'entrée principal à toutes les bibliothèques de Spark • Défini dans le programme pilote • Indique à Spark comment et où accéder à un cluster • Connecte les gestionnaires de clusters • Coordonne les processus Spark en offrant la possibilité d'utiliser différents nœuds de clusters • Utilisé pour créer des RDD et des variables partagées sur le cluster
Driver	• Le processus où la principale (main []) méthode est exécutée • Responsable pour convertir un programme utilisateur en tâches • Le Driver planifie les tâches des exécutants • Les résultats de ces tâches sont renvoyés au Driver.

Executors	• Lancé au début de l'application et généralement durant toute la durée de vie de cette dernière. • Les exécutants se déclarent automatiquement auprès du Driver permettant ainsi à ce dernier de planifier des tâches au niveau des exécuteurs disponibles. • Les Workers exécutent les tâches qui leur sont attribuées et retournent le résultat au Driver. • Assure une stabilisation en mémoire des RDD, ainsi que des disques de stockage.
Cluster Manager	• Un service permettant la gestion des clusters

Tableau 3-2 : Rôles des différents composants Spark

Pour terminer cette partie, nous allons résumer le déroulement d'une exécution en utilisant le Framework Spark.

- Étape 1 : l'utilisateur soumet une application qui lance le programme du Driver.
- Étape 2 : Le Driver appelle la méthode main () spécifiée par l'utilisateur.
- Étape 3 : Le Driver demande au gestionnaire de clusters des ressources pour lancer des exécutables.
- Étape 4 : le gestionnaire de cluster lance des exécutants
- Étape 5 : le Driver divise le programme de l'utilisateur en tâches et les envoie aux exécutants.
- Étape 6 : Les exécutants exécutent les tâches, calculent et enregistrent les résultats qu'ils renvoient par la suite au Driver.
- Étape 7 : lorsque la méthode main du Driver se termine ou que SparkContext.stop est appelé, les exécuteurs sont arrêtés et le gestionnaire des clusters fini par libérer les ressources.

3.4.3 SPARK STREAMING

Spark Streaming utilise le planificateur du Spark Core pour l'analyse en continu des données en temps réel. En effet Spark streaming regroupe les traitements en des mini-lots de données.

La fonctionnalité du traitement des flux de données de Spark permet d'absorber des données provenant de nombreuses sources telles que Kafka [58], Flume [59], Twitter, ZeroMQ [60], Kinesis [61], ou des partages TCP. Les données peuvent en outre être exploitées à l'aide de MLlib ou du GraphX.

Comme décrit dans la figure 3.13, le Spark Streaming suit les étapes ci-dessous :

- La diffusion en continu commence par la collecte des flux de données et la division des derniers en mini lots de données.
- Ces lots de données sont traités par le moteur Spark pour produire des lots de résultats.
- DStream est une API très performante est mise à disposition permettant l'exploitation des flux de données en continu.
- DStream est représenté en interne comme une séquence de RDD
- Des flux de données peuvent être créés à partir de flux de données d'entrée provenant de sources telles que Kafka, Flume et Kinesis, ou en appliquant des expériences de haut niveau sur d'autres flux.

Figure 3.12 : Spark streaming [57]

3.4.4 CONCLUSION

Nous avons opté pour utiliser l'Apache Spark qui peut être considéré comme une solution intégrée pour le traitement de toutes les couches de l'architecture Lambda. Il permet d'exploiter les méthodes de traitement par lots et par flux afin d'équilibrer la latence, le débit ainsi que la précision. La logique dans les deux couches peut être légèrement différente, car chaque couche vise à satisfaire des exigences spécifiques. La couche de traitement par lots crée en continu des vues de traitement par lots des données. Il s'agit d'une opération à latence élevée, car elle exécute un ensemble d'opérations sur toutes les données. Ainsi, pendant que les vues par lots sont exécutées, la couche performance se charge de créer des vues en temps réel des données reçues entre-temps.

Les résultats sont fusionnés au niveau de la couche service qui permet à l'utilisateur de visualiser ce qui a été produit (pour plus de détails, voir le chapitre 4).

3.5 LA STRUCTURE DE HACHAGE CUCKOO

Les tables de hachage Cuckoo fournissent un dictionnaire de données performant. En effet, ces tables de hachage peuvent être considérées comme étant une collection de « sous-tables » et chacune de ces sous-tables est composée de b compartiments (buckets). Ces compartiments contiennent à leur tour des cellules que l'on nommera c.

Une table de hachage utilise des fonctions de hachages « d'h1, h2 h3, …, hd, tel que le périmètre est l'ensemble des entrés possibles dans l'intervalle [0, b]. On considère que toutes ces fonctions sont uniformes et indépendantes.

Un élément x donné peut être placé dans le compartiment hi(x) de la ième sous-table. Chaque élément est stocké dans un seul compartiment.

Le stockage est effectué de telle sorte à ce que l'on ne dépasse pas ces éléments par compartiment. Cette limitation conditionne l'insertion des nouveaux éléments dans les compartiments. Les valeurs associées à un élément peuvent être stockées dans le même emplacement que ce dernier si besoin. Alternativement, les pointeurs vers les valeurs associées stockées dans une mémoire externe peuvent également être stockés.

La structure assure les opérations listées ci-dessous :

- Lookup : Soit x un élément, le compartiment résultant de hi(x) est examiné pour vérifier si l'élément x fait partie de la table.

- Insertion : Soit x un élément, on vérifie premièrement l'existence de cet élément via un lookup. Si l'élément n'existe pas on examine systématiquement le compartiment hi(x). En effet, la première étape consiste à vérifier l'existence d'un emplacement vacant afin d'insérer l'élément x dans le premier espace libre trouvé. Si aucun emplacement n'est disponible, une valeur j est choisie indépendamment et aléatoirement de l'intervalle [1, d]. Ainsi, x sera stocké dans une j sous-table au niveau du compartiment hj(x). Cela entrainera le déplacement d'un élément y du même compartiment et déclenchera un processus d'insertion récursif de ce dernier.

- Suppression : Soit x un élément, on vérifie l'existence de cet élément par le biais d'un lookup et il sera supprimé s'il existe.

Les tables de hachage Cuckoo se basent également sur la notion de seuil ; asymptotiquement, si le facteur de charge, qui correspond au nombre d'éléments divisé par le nombre de cellules est inférieur à un certain seuil (qui dépend de c et d), tous les éléments peuvent être stockés avec une probabilité élevée.

La description de l'insertion ci-dessus est incomplète, parce qu'il y a une marge de variation.

Par exemple, l'élément déplacé y est souvent choisi de manière uniforme au hasard dans le compartiment, comme il est interdit de remettre y immédiatement dans le même compartiment. En outre, la description ne suggère pas d'action en cas d'échec.

Généralement, après un certain nombre de tentatives de stockage récursives, un échec se produit et dans ce cas les éléments peuvent être re hachés ou stockés dans une structure supplémentaire appelée cache.

Une autre implémentation de la méthode de hachage Cuckoo utilise une seule table de compartiments, de sorte que chaque fonction de hachage peut retourner n'importe quels compartiments. Les deux implémentations offrent les mêmes performances en termes de consommation de la mémoire.

Comme indiqué précédemment, le seuil de charge dépend des valeurs de c et d, du nombre de cellules par compartiment et du nombre de fonctions de hachage. En pratique, même pour des systèmes de taille raisonnable, on obtient des performances proches du seuil de charge asymptotique. À titre d'exemple, si l'on considère que d = 2 et c = 4, nous obtenons un seuil supérieur à 98 %. Alors que si l'on considère d = 4 et c = 1, on obtient un seuil supérieur à 97 %.

Le choix de l'implémentation à utiliser dépend essentiellement de la taille de l'élément à stocker et de la taille de cellule.

3.6 LES FILTRES CUCKOO

Un filtre Cuckoo est défini comme étant une structure de données probabiliste basée sur la structure de hachage Cuckoo. Le filtre Cuckoo permet de vérifier l'appartenance d'un élément à un ensemble. La particularité du filtre Cuckoo réside dans le fait qu'au lieu de stocker physiquement les éléments, il se contente de stocker uniquement les empreintes de ces derniers que l'on obtient à l'aide d'une fonction de hachage supplémentaire.

De cette manière, il utilise moins d'espace, tout en assurant une faible probabilité de faux positifs.

Dans l'implémentation suggérée, chaque élément est haché à d = 2 compartiments possibles, avec un maximum de c = 4 éléments par compartiment.

Pour obtenir un espace réduit, le filtre Cuckoo ne stocke pas les éléments d'origine. Il est difficile de déplacer les empreintes lorsque les compartiments sont pleins, comme l'exige la table de hachage Cuckoo.

Le filtre Cuckoo procède comme suit :

Si l'empreinte d'un élément x est f(x), ses deux emplacements seront définis par h1(x) et h1(x) h2 (f [x]).

Notez que, compte tenu de l'emplacement du compartiment et de l'empreinte digitale, l'autre emplacement du compartiment peut être déterminé en calculant h2 (f [x]) et en notant le résultat avec le numéro de compartiment en cours.

Dans la suite, nous supposons qu'une insertion n'est jamais effectuée pour un élément déjà présent dans la structure. La structure prend en charge les opérations suivantes :

- Lookup : à partir d'un élément x, on calcule son empreinte f(x) et les emplacements de compartiment h1(x) et h1(x) h2 (f [x]). Les empreintes stockées dans ces compartiments sont comparées à f(x). Si des empreintes correspondent à f(x), un résultat positif est renvoyé, sinon un résultat négatif est renvoyé.

- Insertion : à partir d'un élément x, on calcule les deux emplacements de

compartiment ainsi l'empreinte f(x). Si une cellule est libre dans l'un de ces compartiments, on place f(x) dans la première cellule disponible. Sinon, une empreinte f (y) dans l'un des compartiments sera déplacée, puis placée de manière récursive, de la même manière que dans une table de hachage de Cuckoo. Nous utilisons ici ces deux compartiments pour pouvoir tracer les déplacements de f (y).

- Suppression : à partir d'un élément x, x est recherché via la fonction Lookup. Si l'empreinte f(x) se trouve dans l'un des compartiments, une copie de l'empreinte est supprimée.

La probabilité de faux positifs d'un filtre Cuckoo peut être approximativement estimée comme suit.

Si l'on utilise n bits pour le stockage des empreintes et que la charge de la table de hachage est égale à l, la probabilité de faux positif sera d'environ 8 l/2n. En effet, la fonction Lookup comparera en moyenne 81 empreintes en supposant que l'on ne vérifie que 2 compartiments des 5 et que chaque compartiment ne contient que 4 cellules. En outre, chacune de ces empreintes à une probabilité 2-n de produire un faux positif.

3.7 UNE IMPLEMENTATION EFFICIENTE D'APRIORI DANS SPARK

Avant de nous plonger dans la présentation de notre approche d'extraction des motifs que l'on a proposée pour extraire les motifs fréquents, nous allons revenir à l'algorithme Apriori que l'on a présenté un peu avant dans le chapitre 2.2.1.4 de l'État de l'art

Apriori est un algorithme itératif permettant l'extraction des motifs fréquents dans une base de données transactionnelle. Il sert à reconnaître des propriétés qui reviennent fréquemment dans un ensemble de données et d'en déduire une catégorisation. Il se base sur le fait qu'un itemset n'est fréquent que si tous ses sous-ensembles (subsets) non vides le sont.

Chaque ième itération d'apriori génère un des patterns de longueur i. La première itération trouve tous les motifs fréquents dont la longueur est 1.

Maintenant, un ensemble de candidats est généré avec toutes les combinaisons possibles de longueur 2. À la 2e itération, tous les itemsets fréquents de longueur 2 sont découverts. Ces itemsets de longueur 2 seront utilisés pour créer un ensemble candidat de longueurs 3.

Cette procédure est répétée jusqu'à ce qu'il n'y ait plus d'itemsets fréquents. Apriori implémenté par R. Agarwal et al. [62] a une phase unique où chaque itération génère initialement un ensemble de candidats à partir des résultats de l'itération précédente. Ensuite, on analyse l'ensemble de ces données pour extraire les itemsets dont l'occurrence est supérieure au support défini par l'utilisateur. Il existe plusieurs implémentations d'Apriori sur Hadoop combiné avec du MapReduce. Hadoop rend Apriori plus évolutif et fiable en fournissant un environnement de stockage et de calcul distribué.

YAFIM (Yet Another Frequent Itemset Mining) [63] est une implémentation d'Apriori sur Spark qui surpasse de manière significative les implémentations d'apriori sur Hadoop. Les jeux de données transactionnels sous forme d'HDFS sont initialement chargés dans des RDD Spark (ensembles de données résilients et distribués), qui sont des collections partitionnées d'enregistrements en lecture seule ne pouvant être créée que par des opérations déterministes. L'utilisation des RDD permet d'exploiter au maximum les ressources disponibles au niveau des clusters. YAFIM possède deux phases : la première

phase consiste à générer tous les singletons d'items fréquents. Lors de la deuxième phase, il génère itérativement tous les itemsets fréquents de longueur k+1 à partir des itemsets fréquents de longueur k.

L'approche que nous proposons à travers notre travail de recherche est une implémentation parallèle d'Apriori sur Spark basée sur trois phases pour améliorer la génération des itemsets fréquents de longueur 2, particulièrement pénalisantes.

Nous apportons des modifications pour réduire le nombre de calculs à effectuer afin de générer les itemsets fréquents de longueur 2. La recherche des itemsets fréquents de longueur 2 est l'étape la plus consommatrice de temps et d'espace dans Apriori, une étape simplifiée par notre approche.

4 IMPLEMENTATIONS

4.1 INTRODUCTION

Dans cette partie nous allons présenter d'une façon détaillée les différentes étapes de l'implémentation de notre approche. Cette dernière représente une nouvelle implémentation de l'algorithme Apriori dans un environnement SPARK.

Nous allons présenter les trois phases de notre approche et l'expérimentation que nous avons menée sur les benchmark, comparant ainsi complexité temporelle et spatiale de notre approche vis-à-vis la version classique d'Apriori.

4.2 PHASE I

Les jeux de données transactionnels sous forme d'HDFS sont initialement chargés dans des RDD Spark. L'utilisation des RDD permet d'exploiter au maximum les ressources disponibles au niveau des clusters et profiter également de la tolérance aux pannes.

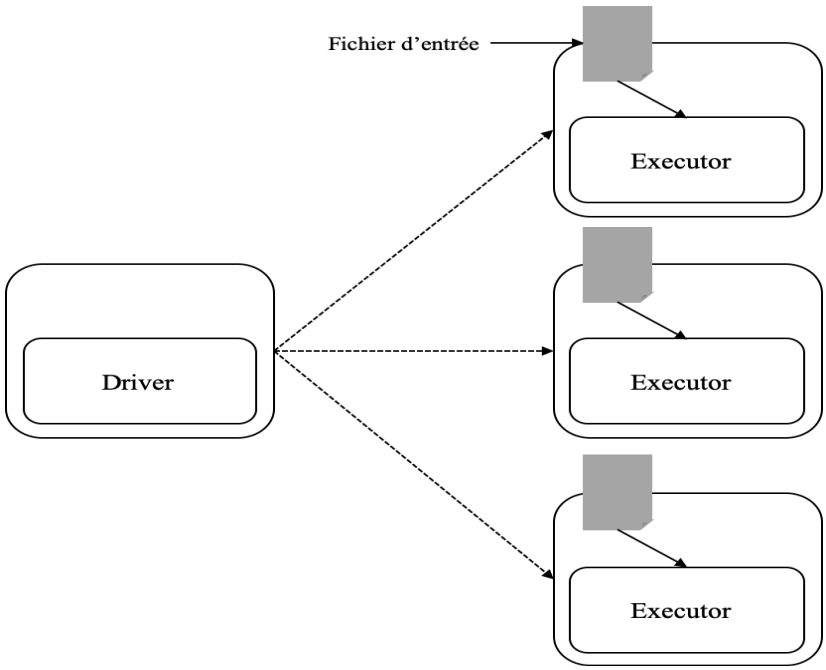


Figure 4.1 Diffusion du fichier d'entrée sur les nœuds du cluster

Le fichier d'entrée est diffusé sur chaque nœud de travail pour le rendre disponible localemet à chaque Executor (Exécuteur) (cf figure 4.1).

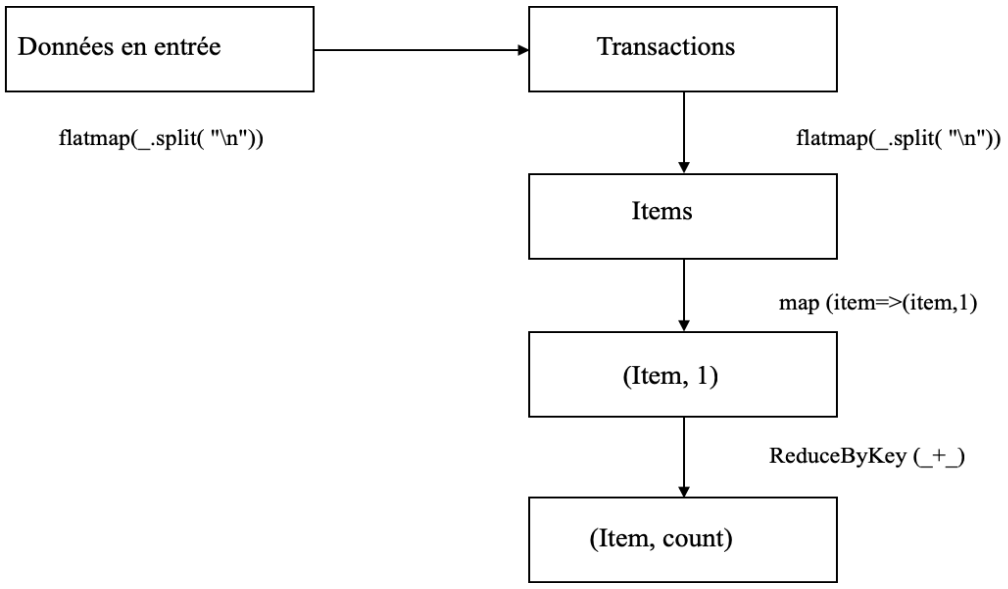
Ensuite, une fonction **flatMap** est appliquée sur le jeu de données pour lire les transactions, cette même fonction sera appelée à chaque transaction.

Une fois le processus **flatMap** terminé, une fonction **map** convertit chaque item en couple clé/valeur (item, 1).

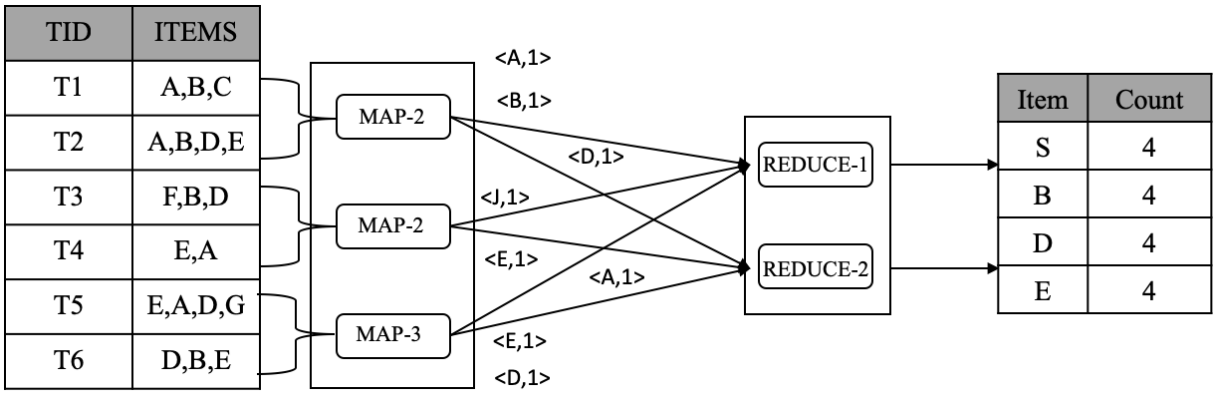
Enfin, la fonction **reductionByKey** est appelée pour calculer la fréquence de chaque élément et élague les éléments dont la fréquence est inférieure au nombre de supports prédéfini. Les éléments restants sont stockés dans le jeu d'éléments fréquents sous forme de singleton. La figure 4.2 (a) représente l'algorithme utilisé pour calculer les singletons fréquents dans la phase I. Un schéma logique illustrant le flux de données ainsi que les différents contrôles effectués durant la phase I est représenté dans la figure 4 .2 (b).

```
foreach Transaction T in Ri
{
        flatMap(line offset, T)
        {
                foreach item I in T
                {
                        Yield(I,1)
                }
        }
}
reduceByKey(I,count)
{
        sum=0
        while (Item I in partition)
        {
                sum+=count
        }
        if(sum>=min_supp)
        {
                Yield(I,sum)
        }
}
```

(a)



(b)



(c)

Figure 4.2 : Phase 1 de notre approche (a). L'algorithme utilisé lors de la phase (b). Exemple d'exécution de la phase 1 (c) où T : {T_1,T_2,…} est un ensemble de transaction

La figure 4.2 (c), représente un exemple d'exécution de la phase 1 après l'application des méthodes **map** et **reductionByKey**.

4.3 PHASE II

Lors de la 2e itération de l'algorithme Apriori classique exécuté dans un environnement Spark, un ensemble de candidats est généré à partir des singletons fréquents. Cet ensemble de candidats est stocké dans une arborescence de hachage pour accélérer les recherches. Chaque mappeur prend une transaction et en génère à travers toutes les combinaisons possibles des candidats de longueur 2. Les réducteurs calculent toutes les occurrences des itemsets de longueur 2 créés par le mappeur et ne garde que ceux ayant une occurrence supérieure au min support.

La 2ème itération est l'itération la plus consommatrice de temps et d'espace dans l'algorithme Apriori en raison de l'importance du nombre des candidats générés. À titre d'exemple, pour 10^3 de singletons fréquents, on en générera près de 10^6 de candidats de longueur 2. Ainsi, vérifier un nombre aussi important de candidats pour chaque transaction prend beaucoup de temps.

Dans l'approche que nous proposons, nous avons réussi à améliorer significativement les performances de la seconde itération. Pour y parvenir, nous avons procédé comme suit :

- Élimination de l'étape de génération des candidats.
- Utilisation du Cuckoo filtre à la place de l'arborescence de hachage.

En effet, nous avons fait le choix de stocker les items fréquents de longueur 1 au niveau du filtre Cuckoo pour des raisons de performances (voir chapitre 3.6). La fonction map élague chaque transaction de sorte qu'elle ne contienne que des items qui existent dans le filtre Cuckoo. Ensuite elle en génère toutes les combinaisons possibles des items de longueur 2. La fonction de réduction **reductionByKey** considère chaque item de longueur 2 comme étant une clé. Cette clé est utilisée pour le calcul des occurrences et retourner tous les items de longueur 2 dont l'occurrence est supérieure au support défini par l'utilisateur.

Le résultat obtenu après l'exécution des fonctions **map** et **reductionByKey** est similaire pour la version classique d'Apriori sur Spark et notre approche. Cependant, l'élimination de l'étape de génération de l'ensemble des candidats et l'utilisation du filtre Cuckoo rend notre implémentation plus performante en termes de temps de réponse ainsi qu'en termes

de consommation de ressource.

4.3.1 COMPARAISON DE LA COMPLEXITE TEMPORELLE DE LA DEUXIEME ITERATIONS DES DEUX VERSIONS D'APRIORI

Notre approche peut être plus efficace que la version classique d'Apriori dans Spark si et seulement si le nombre de singletons fréquents est conséquent.

Notre fonction de réduction reçoit les mêmes données en entrée et produit le même résultat que dans le cas d'un Apriori conventionnel, de sorte que la complexité temporelle est la même pour la fonction de réduction des deux algorithmes. Cependant, la complexité temporelle de l'exécution de la fonction **map** doit être analysée et comparée entre les deux versions d'Apriori.

Supposons que l'on ait X transactions, M mappeur, g comme nombre moyen d'éléments par transaction et f le nombre d'items fréquent après la première itération. Aussi, soit t le temps nécessaire pour rechercher un élément dans un arbre de hachage et C le temps nécessaire à la recherche d'un élément dans un filtre Cuckoo..

A travers les sous chapitres 4.3.1.1 et 4.3.1.2, nous allons estimer la complexité temporelle de la deuxième itération des implémentations d'Apriori, à savoir notre implémentation et l'implémentation classique d'Apriori sous Spark.

4.3.1.1 ESTIMATION DE LA COMPLEXITE SPATIALE DE LA DEUXIEME ITERATION DE L'IMPLEMENTATION CLASSIQUE D'APRIORI DANS SPARK

Le temps nécessaire pour l'exécution de la deuxième itération de l'implémentation conventionnelle d'Apriori est la somme des tâches ci-dessous :

- Le temps de générer un ensemble de candidats (Tg) qui correspond à la somme des temps nécessaires pour effectuer les jointures et l'élagage.

- Le temps de stocker l'ensemble des candidats dans l'arbre de hachage (Th).
- Le temps nécessaire à l'exécution des mappeurs.

De ce fait, on peut déduire les résultats suivants :

- Le temps de génération d'un ensemble de candidats (Tg) peut être calculé comme suit :

$$\frac{f(f-1)}{2} + \frac{2f(f-1)}{2} = \frac{3f(f-1)}{2}$$

- La complexité temporelle pour stocker un item dans un arbre de hachage

$$\frac{2f(f-1)}{2} \approx O(f^2)$$

- La complexité temporelle pour rechercher et générer tous les candidats possibles de longueur 2.

$$\frac{X}{M} * \frac{tg(g-1)}{2} \approx O({}^{X}/_{M} \cdot g^2)$$

ESTIMATION DE LA COMPLEXITE SPATIALE DE LA DEUXIEME ITERATION DE NOTRE IMPLEMENTATION D'APRIORI DANS SPARK

Le temps nécessaire pour l'exécution de la deuxième itération de notre implémentation d'Apriori dans Spark est la somme des tâches ci-dessous :

- Le temps nécessaire pour stocker un item de longueur 1 au niveau du filtre Cuckoo (T_{cf}).
- Le temps nécessaire pour effectuer l'élagage (T_{pr}).
- Le temps nécessaire pour générer tous les candidats possibles de longueur 2 (T_{mr}).

De ce fait, on peut déduire les résultats suivants :

- Le temps nécessaire pour stocker un item de longueur 1 au niveau du filtre Cuckoo

est équivalent à f.

- La complexité temporelle pour effectuer l'élagage peut être calculée comme suit :

$$\frac{X}{M} * g \approx O({}^{X}/_{M} . g)$$

- Le temps nécessaire pour générer tous les candidats possibles de longueur 2 dans le pire des cas peut être calculé comme suit :

$$\frac{X}{M} * \frac{Cg(g-1)}{2} \approx O({}^{X}/_{M} . g^2 . C)$$

Ainsi la complexité temporelle de notre approche est la suivante :

$$f + \frac{X}{M} * \frac{Cg(g-1)}{2} \approx O(f + {}^{X}/_{M} . g^2 . C)$$

4.3.1.2 BENCHMARK

Nous allons maintenant revenir à l'estimation de la complexité temporelle de l'implémentation classique d'Apriori dans Spark pour démontrer que notre approche est théoriquement plus performante.

En effet, si on fait la soustraction entre la complexité de l'implémentation classique d'Apriori et celle de notre approche nous obtenons le résultat suivant :

$$= \frac{5f(f-1)}{2} + \frac{X}{M} * \frac{tg(g-1)}{2} - f + \frac{X}{M} * \frac{cg(g-1)}{2}$$
$$= \frac{f(5f-7)}{2} + \frac{X}{M} * \frac{g(g-1)}{2} * (t-b) \approx O(f^2 + {}^{X}/_{M} . (t-b) . g^2)$$

Cette analyse nous démontre que l'approche classique d'Apriori dans Spark ajoute un coût quadratique additionnel alors que notre approche reste linéaire.

Comme le temps de recherche d'un élément dans un filtre Cuckoo est estimé à un cycle.

Notre approche sera plus efficiente dans le cas où l'on traite un grand volume de données.

4.3.2 COMPARAISON DE LA COMPLEXITE SPATIALE DE LA DEUXIEME ITERATIONS DES DEUX VERSIONS D'APRIORI

La complexité spatiale de notre approche est inférieure à celle de l'algorithme classique Apriori sur Spark.

Le réducteur reçoit la même entrée et produit la même sortie pour notre approche que dans le cas d'Apriori conventionnel. Par conséquent, la complexité de l'espace est la même pour les réducteurs des deux algorithmes. Cependant, la complexité de l'espace lors de l'exécution du mappeur est primordiale.

4.3.2.1 ESTIMATION DE LA COMPLEXITE SPATIALE DE LA DEUXIEME ITERATION DE L'IMPLEMENTATION CLASSIQUE D'APRIORI DANS SPARK

Pour l'implémentation classique Apriori, la complexité spatiale de la 2e itération est la somme de :

- La complexité spatiale pour stocker l'ensemble des candidats
- La complexité spatiale pour créer un arbre de hachage pour stocker l'ensemble des candidats.

Supposons qu'il reste f fréquemment items après la première itération. De plus, supposons que b et c soient respectivement les octets requis pour stocker un candidat de longueur 2 et un item de longueur 1 dans l'arbre de hachage. Alors, on peut déduire les résultats ci-dessous :

- La complexité spatiale pour stocker un ensemble de candidats serait comme suit :

$$\frac{bf(f-1)}{2} \approx O(\,bf^2)$$

- La complexité spatiale pour créer un arbre de hachage est la suivante :

$$\frac{bf(f-1)}{2} + \frac{cf(f-1)}{2} \approx O(\,b{+}c)f^2)$$

Ainsi la complexité spatiale du mapper de l'implémentation classique d'Apriori dans Spark est la suivante :

$$\frac{f(f-1)(2b+c)}{2} \approx O(\,b{+}c)f^2)$$

4.3.2.2 ESTIMATION DE LA COMPLEXITE SPATIALE DE LA DEUXIEME ITERATION DE NOTRE IMPLEMENTATION D'APRIORI DANS SPARK

Pour notre approche, la complexité spatiale de la 2e itération est la somme de :

- La complexité spatiale pour stocker un item de longueur 1 dans un filtre Cuckoo qui est équivalente à *cf*.
-

4.3.2.3 BENCHMARK

D'après les éléments présenter dans les sous chapitre 4.3.2.1 et 4.3.2.2 nous pouvons conclure que la complexité spatiale de notre approche est largement inférieure à celle de la version conventionnelle d'Apriori dans Spark. En effet, en profitant des caractéristiques du filtre Cuckoo ainsi des performances de Spark en termes de puissances de calcul et d'analyse, en outre de la faible complexité spatiale de notre approche, tous ces éléments nous permettent de dire que notre solution d'extraction est grandement plus efficace notamment dans le cas où l'on très des grands volumes de données.

Remarque : *Notre algorithme est plus performant dans une configuration multi core.*

Le réducteur reçoit la même entrée et produit la même sortie pour notre approche que dans le cas d'Apriori conventionnel. Cependant, le mappeur effectue moins de calculs dans le cas de notre approche.

Pour l'implémentation classique Apriori, la complexité temporelle totale de la 2e itération est la somme de :

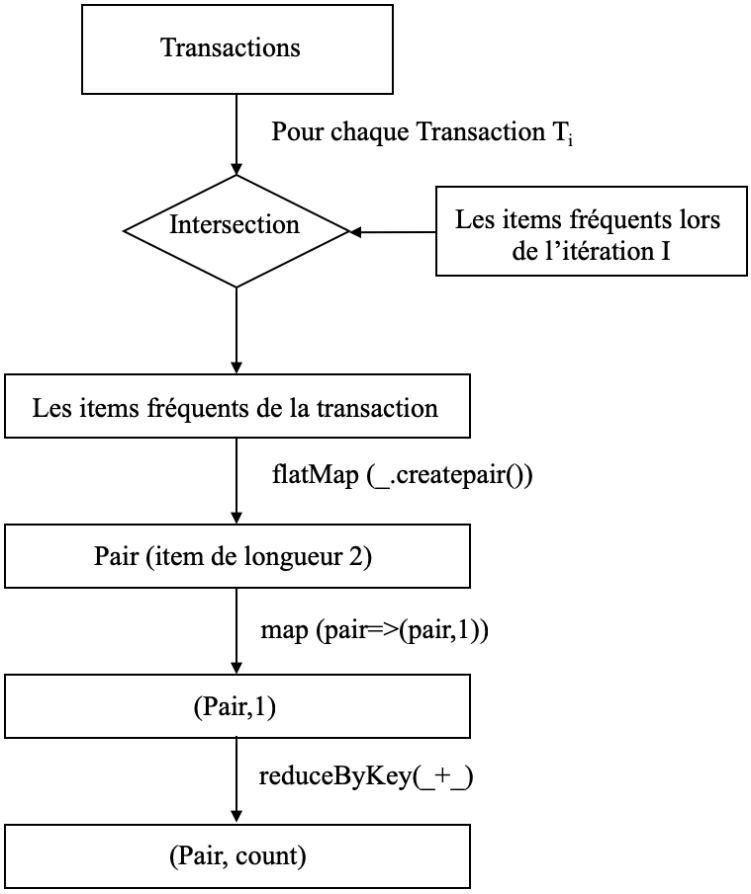
- Le temps de générer un ensemble de candidats (Tg)
- Le temps de stocker le jeu de candidats dans un arbre de hachage (Th)
- Le temps nécessaire à l'exécution des mappeurs (Tm)

Dans notre implémentation, le temps nécessaire pour générer un ensemble de candidats (Tg) devient égal à zéro, tandis que le temps nécessaire pour stocker un ensemble de candidats dans un arbre de hachage (Th) correspond au temps nécessaire pour stocker des singletons fréquents dans le filtre Cuckoo (Tc). Ce temps (Tc) est considéré comme étant négligeable.

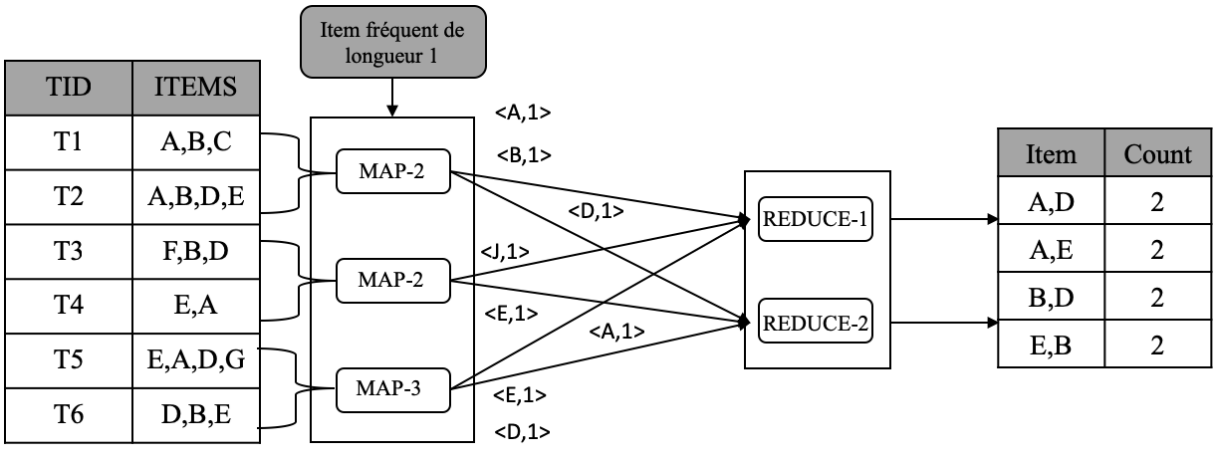
En outre, le temps pris par le mappeur dans notre implémentation est également moindre, car le nombre de comparaisons est réduit. Par conséquent, le temps total pris par les mappeurs diminuera rapidement si le nombre de mappeurs (ou de cœurs) est important.

La figure 4.3 (a) présente le graphique de lignage pour la 2ème phase de notre algorithme. Ici, une fonction **flatMap** est utilisée pour lire chaque paire. Ensuite, une fonction **map** produit (Item, 1) pour chaque Item.

Le diagramme graphique (cf figure 4.3 (b)) illustre comment le mappeur et le réducteur gèrent les données en entrées pour en produire des résultats. Il illustre également le fonctionnement de la fonction **reductionByKey** qui calcule l'occurrence de chaque élément et élague les éléments dont l'occurrence est inférieure au support minimum. Les éléments restants sont stockés dans 2 ensembles d'éléments fréquents.



(a)



(b)

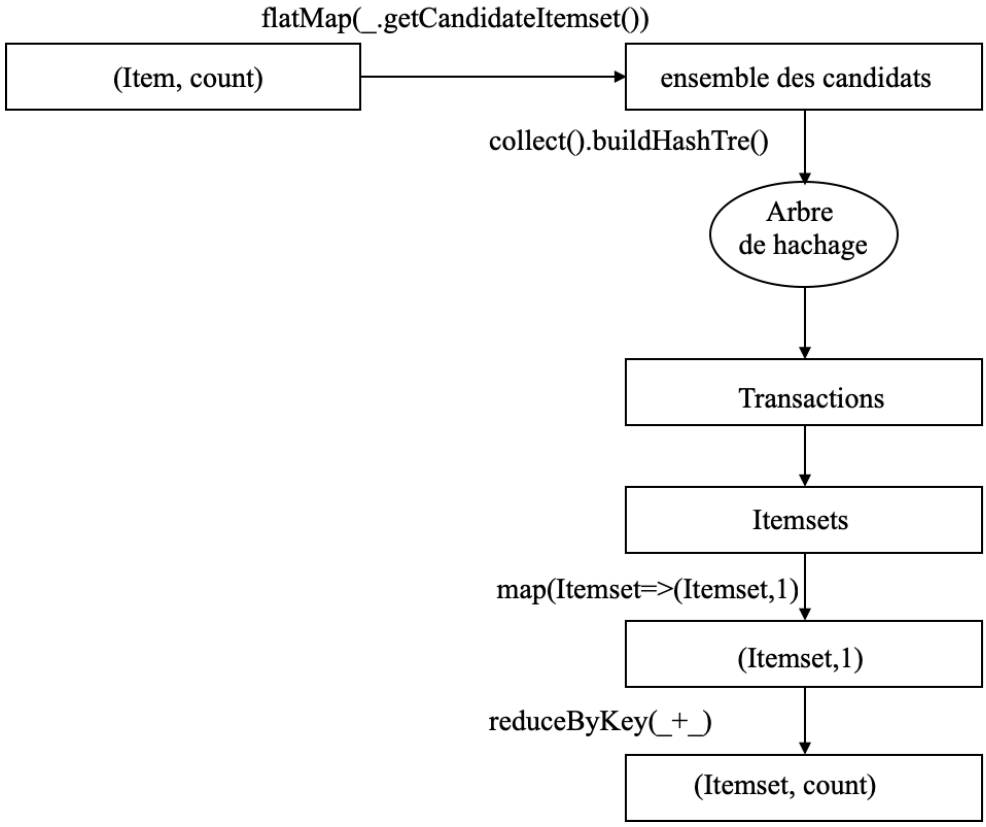
Figure 4.3 : Phase 1 de notre approche (a). L'algorithme utilisé lors de la phase (b). Exemple d'exécution de la phase 1 (c) où T : {T_1,T_2,…} est un ensemble de transaction

4.4 PHASE III

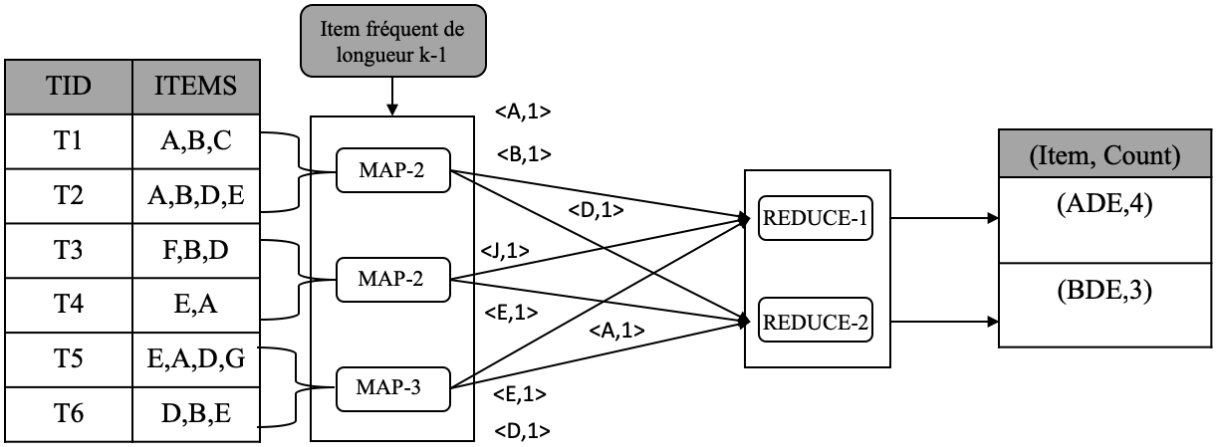
Nous utilisons des itemsets K-fréquents pour en générer des itemsets (K + 1) — fréquents. Initialement, les ensembles d'éléments K fréquents Lk1 sont insérés dans les RDD et l'ensemble candidat Ck + 1 est généré et stocké dans un filtre Cuckoo afin d'augmenter la vitesse de recherche de l'ensemble d'éléments K + 1.

Ensuite, une fonction **flatMap** est utilisée pour obtenir chaque transaction et vérifier les ensembles (K + 1) — fréquents possibles.

La fonction **map** quant à elle, permet de générer un (Item, 1) pour chaque élément. Finalement, la fonction **reductionByKey** calcule l'occurrence de chaque élément et élague les éléments dont l'occurrence est inférieure au support. Les éléments restants sont stockés dans l'ensemble d'items K-fréquent. Le graphique de lignage (cf figure 4.4 (a)), ci-dessous, illustre le flux de données et le contrôle tout au long des différentes étapes de l'étape 3.



(a)



(b)

Figure 4.4 : L'algorithme utilisé lors de la phase 3 (a). Exemple d'exécution de la phase 3 (b) où T : {T1,T2,...} est un ensemble de transaction

4.5 ANALYSE DES CHANGEMENTS DES MOTIFS

Une fois l'extraction des motifs réalisée, une analyse est effectuée selon un intervalle défini par l'utilisateur pour en déduire des cas pouvant présenter des motifs évolutifs. L'objectif de cette étape est de tester les différents types d'évolutions que l'on a identifiés.

N.B : Notre objectif est d'explorer les motifs extraits pour vérifier l'existence de situation où notre définition de l'évolution est applicable.

L'analyse peut être scindée en deux parties :

- Analyse de l'évolution de la durée des occurrences :
  - Cette analyse permet de rechercher les évolutions de la durée des occurrences des motifs dans l'intervalle de temps prédéfini. Cette analyse concerne tous les motifs. Une catégorisation par taille est effectuée par la suite.

- Analyse de l'évolution de la fréquence des occurrences des motifs :
  - Dans cette analyse, on s'intéresse à la fréquence des occurrences des motifs.

Cette analyse consiste à étudier la fréquence locale des occurrences des motifs selon plusieurs sous-intervalles (inclus dans l'intervalle de temps prédéfini).

Ces deux analyses ont pour objectif de donner une vision de la tendance générale observée dans les données pour les questions de durée et de fréquence des motifs. L'intérêt est d'étudier ces phénomènes à une échelle plus réduite, en particulier dans le cas où l'on aurait des résultats allant à l'encontre de cette tendance.

5 RESULTATS ET EXPERIMENTATIONS

5.1 INTRODUCTION

Dans cette section, nous évaluons la performance de notre algorithme en le comparant avec la version standard d'Apriori implémenté sous Hadoop (MRApriori [64]) et la version d'apriori implémenté sous Spark (YAFIM [63]) ainsi que l'algorithme FP-Growth sur Spark. En effet, nous avons utilisé le Framework Spark lors de l'implémentation de notre approche. Pour informations, ils existent plusieurs implémentations d'Apriori sous Hadoop qui ont les des performances équivalentes à MRApriori.

Dans nos expériences, de nombreux jeux de données de référence ont été utilisés. Toutes les expériences ont été exécutées quatre fois et nous avons considéré la moyenne comme résultat final. Notre algorithme ainsi que YAFIM sont développés sous Spark 2.2.0.

Toutes les expériences ont été menées sur un cluster de deux nœuds, chacun ayant huit cœurs et 32 Go de RAM. On a utilisé la version 3.9.0 d'Alpine comme OS et la version 1.9.0 de JAVA.

5.2 ÉCHANTILLON DE DONNÉES

Les expériences ont été menées avec trois grands échantillons de données présentant des caractéristiques différentes. Le premier échantillon Extract0809 correspond à une extraction de données collectées dans le cadre du projet E-Care. Il contient 105 transactions, dont 900 items. Le deuxième et le troisième échantillon correspondent à des données générées par le générateur de données artificielles d'IBM. Les détails de ces échantillons sont présentés dans le tableau ci-dessous :

Échantillon de données	***Nombre d'items***	***Nombre de Transactions***
Extract0809	870	100 000
T25I10D10K	16 470	88 163
T10I4D100K	41 270	9,90 002

Tableau 5-1 Détails des échantillons de données utilisés

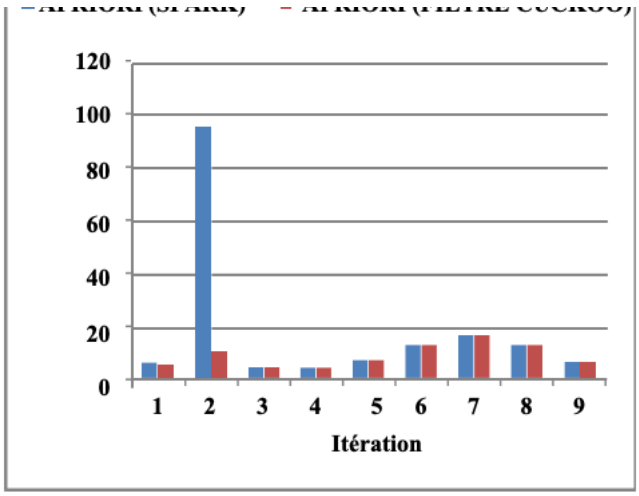
5.3 ANALYSES DES PERFORMANCES

Dans ce sous-chapitre, nous présentons les résultats de notre analyse par rapport aux performances des algorithmes étudiés. Nous avons comparé les performances et la rapidité de l'implémentation classique d'Apriori sur Spark, de FP-Growth sur Spark ainsi que celles de notre implémentation. Chacune des implémentations récupère les données à partir des fichiers HDFS.

Les résultats de cette analyse sont présentés comme suit :

La figure 5.1 illustre la rapidité de notre approche par rapport à la version classique d'Apriori sur Spark. En effet, on peut constater que pour l'échantillon T25I10D10K, notre algorithme est 9 fois plus rapide que la version conventionnelle d'Apriori sur Spark.

Les figures 5.2 et 5.3 présentent respectivement les résultats de l'évaluation comparatifs des deux approches par rapport aux échantillons Extract0809 et T10I4D100K. Nous en pouvons constater que notre approche est 3 fois plus performante que l'implémentation classique d'Apriori sur Spark.



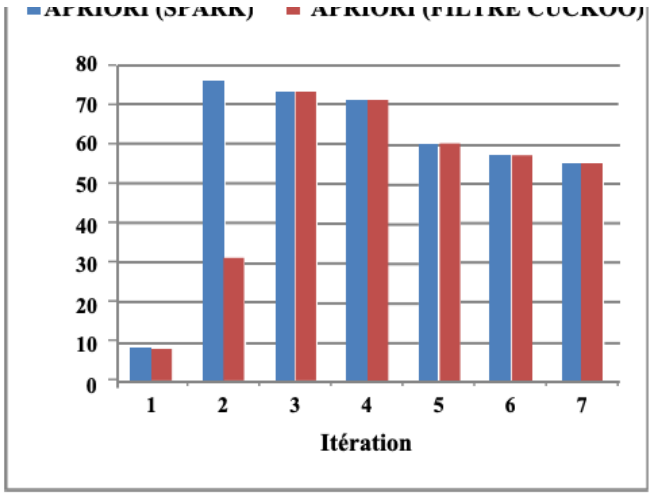


Figure 5.2 : Echantillon Extract0809 avec un min sup=0.60 %

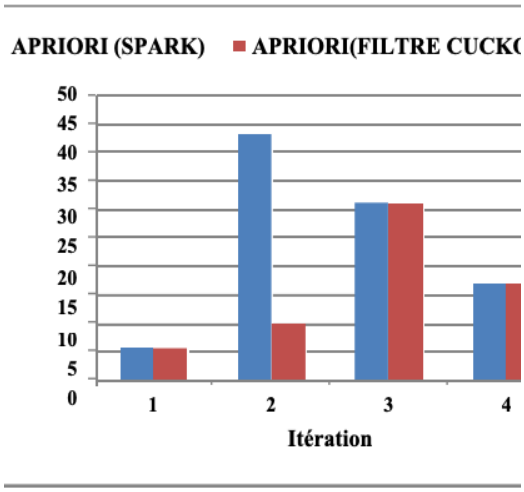


Figure 5.3 : Échantillon T10I4D100K avec un min sup=1.00 %

Le tableau 5.2 présente les résultats de l'évaluation comparative élargie que l'on a menée. Les échantillons de tests utilisés sont T10I4D100K et T25I10D10K. Nous avons effectué les tests à deux reprises pour chaque approche et chaque échantillon selon la valeur du support minimum.

Échantillon	Minimum Support	FP-Growt (Spark 2.2. Temps (se	Apriori (Spark 2.2.0) Temps (sec)	CUCKOO, Spark 2.2.0) Temps (sec)

Tableau 5-2 : Détails des échantillons de données utilisés

Nous pouvons ainsi constater que dans le cas des échantillons T10I4D100K et T25I10D10K, notre approche est jusqu'à 5 fois plus rapide que les autres approches testées à savoir la version conventionnelle d'Apriori et FP-Growth.

Les résultats obtenus ne font que confirmer les conclusions que l'on a faites précédemment.

5.4 CONCLUSION

Dans ce chapitre nous avons proposé une définition de la notion d'évolution des motifs, nous avons également présenté notre approche d'extraction des motifs fréquents basée sur Apriori.

En effet, nous avons mis en place un nouvel algorithme plus rapide et plus efficace permettant l'extraction des motifs fréquents à partir de grands volumes de données.

L'algorithme Apriori conventionnel est consommateur de temps et d'espace lors de la deuxième itération. Autrement dit, si l'on a 10^4 singletons fréquents, l'ensemble des candidats généré lors de la deuxième itération sera équivalent à 10^8. Il est ainsi quasi impossible de vérifier l'occurrence d'un ensemble de candidats aussi important à chaque transaction.

L'approche que l'on a proposée est conçu autour de deux choix techniques à savoir Spark et les structures de hachage Cuckoo. Cela nous a permis de réduire le nombre total des calculs effectués lors de la deuxième itération, ce qui implique une amélioration des performances importante notamment dans le cas où l'on traite des gros volumes de données. En outre, l'utilisation de Spark nous permet de profiter de la puissance d'un environnement de calcul hautement évolutif et parallèle.

Pour démontrer l'efficacité de notre approche, nous avons mené des comparaisons théoriques et empiriques de notre implémentation avec une des implémentations déjà existantes telles Apriori, et FP-Growth. Les résultats obtenus nous ont permis d'illustrer l'efficacité et l'efficience de notre approche. Cette supériorité est traduite par le fait que notre approche est jusqu'à 9 fois plus rapide que les approches standard développées sous Spark.

6 CONCLUSION ET PERSPECTIVES

6.1 CONCLUSION GENERALE

Nous avons abordé tout au long de cette thèse la problématique d'extraction des motifs fréquents ayant la particularité d'évoluer dans le temps. L'objectif applicatif étant la conception d'un programme permettant de concevoir des chemins cliniques ad hoc à partir des données des patients.

Le projet de thèse s'inscrit dans le cadre de la plateforme E-Care. C'est une plateforme dont le but est de suivre des patients afin de détecter de manière précoce toute décompensation, en prenant en charge les patients dans toutes leurs démarches de soin.

Notre objectif était scindé en deux grands axes majeurs.

Premièrement, nous devions proposer une solution qui s'intègre facilement avec l'architecture distribuée de la plateforme E-Care. En gros nous devions proposer une nouvelle fonctionnalité tout en préservant l'expérience client déjà assurée par les autres composantes de la plateforme. Cela a constitué un élément majeur lors du choix des solutions techniques autour desquelles nous devrions concevoir notre approche.

Deuxièmement, nous devrions proposer une définition de la notion d'évolution des motifs ainsi qu'une approche permettant l'extraction de ces derniers.

Pour y parvenir, nous avons effectué un tour d'horizon des différents algorithmes et approches existants sous forme d'un état de l'art. Cela nous a permis de mieux cerner le contexte dans lequel se placeront nos travaux. Dans ce sens, nous avons mené des analyses quantitatives et qualitatives des techniques d'extraction des motifs existantes dans la littérature afin d'explorer les avantages et les limites de chacune d'elles. Cette partie de notre travail de recherche nous a aidés à rassembler un ensemble d'informations pertinentes qui ont grandement influencé nos choix lors de la conception de notre approche.

En effet, l'approche que l'on a proposée s'articule autour de trois éléments essentiels qui sont l'algorithme d'extraction Apriori, le Framework Spark ainsi que les structures de hachage Cuckoo. La combinaison de ces derniers nous a permis de concevoir une solution performante permettant l'extraction des motifs évolutifs.

Nous avons réussi à démontrer l'efficacité de notre approche en la comparant aux approches existantes par rapport à la complexité temporelle et spatiale

Les résultats obtenus ont été confirmés par la suite par le biais de l'étude empirique que l'on a menée. Nous avons utilisé plusieurs types de jeux de données (données réelles collectées par la plateforme E-Care ainsi que des données fictives que l'on a générées à l'aide du générateur de données d'IBM).

Nous avons ainsi montré que notre approche peut être jusqu'à neuf fois plus performante que les méthodes d'extractions standard dans le cas où l'on traite des gros volumes de données.

6.2 PERSPECTIVES

Nous sommes actuellement en phase de tests d'intégration. En effet, nous sommes en train de tester l'intégration de la nouvelle fonctionnalité que l'on a développée lors de notre projet de recherche dans l'écosystème E-Care. Le but de cette étape est de vérifier les aspects fonctionnels de la nouvelle composante ainsi que les éventuels effets de bords résultants de l'intégration de cette dernière.

À la fin de cette phase, nous allons mettre à disposition cette nouvelle version de la plateforme E-Care aux experts médicaux afin de la valider d'un point de vue fonctionnel. En d'autres termes, les experts médicaux auront pour rôle de valider la cohérence des chemins cliniques proposés par notre solution.

La collaboration avec les experts médicaux va également nous permettre de mieux configurer la solution. Cette collaboration va nous permettre dans un premier temps de définir le support minimum que l'on va utiliser lors de notre quête d'exploration des motifs évolutifs ainsi que l'intervalle de temps au bout duquel on entamera la détection des changements des motifs fréquents.

Autrement dit, elle va nous permettre de définir l'occurrence minimale des items à extraire ainsi que l'ordonnancement de la couche Batch (voir chapitre 4.4).

L'ultime perspective par rapport à notre projet de recherche est d'explorer l'implémentation de notre approche avec le langage de programmation Scala qui est considérée comme étant plus rapide et plus performant au sein du Framework Spark dont le noyau est développé en Scala.

BIBLIOGRAPHIE

[1] MapReduce : https://www.ibm.com/analytics/hadoop/mapreduce. 2018

[2] Apache Hadoop. Open-source software for reliable, scalable, distributed computing Apache Hadoop. 2016. http://hadoop.apache.org/docs/r2.7.2/.

[3] Apache spark : https://spark.apache.org.2018

[4] Mooney, C. H. et J. F. Roddick. 2013, «Sequential pattern mining – approaches and algorithms»,ACM Comput. Surv., vol. 45, no 2, doi :10.1145/2431211.2431218, p. 19 :1–19 :39, ISSN 0360-0300. URL http://doi.acm.org/10.1145/2431211.2431218.

[5] Agrawal, R. et R. Srikant. 1995, «Mining sequential patterns», dans Data Engineering, 1995. Proceedings of the Eleventh International Conference on, IEEE, p. 3–14.

[6] Zaki, M. J. 2000, «Sequence mining in categorical domains : Incorporating constraints», dans Proceedings of the Ninth International Conference on Information and Knowledge Management,CIKM ’00, ACM, New York, NY, USA, ISBN 1-58113-320-0, p. 422–429, doi :10.1145/354756.354849. URL http://doi.acm.org/10.1145/354756.354849.

[7] Pei, J., J. Han, B. Mortazavi-asl, H. Pinto, Q. Chen, U. Dayal et M. chun Hsu. 2001, «Prefixspan :Mining sequential patterns efficiently by prefix-projected pattern growth», p. 215–224.

[8] Yan et al (2005) Summarizing itemset patterns: a profile-based approach. In: Proceeding of the 2005 ACM SIGKDD international conference on knowledge discovery in databases (KDD’05), Chicago, IL, pp 314–323

[9] NANNI M. & RIGOTTI C. (2007). Extracting trees of quantitative serial episodes. In Knowledge Discovery in Inductive Databases 5th Int. Workshop KDID’06, Revised Selected and Invited Papers, p. 170–188 : Springer-Verlag LNCS 4747.

[10] Gomariz et al. (2013) An efficient algorithm for mining frequent closed sequences. In J. PEI, V. S. TSENG, L. CAO, H. MOTODA & G. XU, Eds., Proc. of the Pacific-Asia Conf.

[11] Mannila et Collab, Discovering generalized episodes using minimal occurrences.», dans KDD, vol. 96, p. 146–151.

[12] Gan, M. et H. Dai. 2010, «A study on the accuracy of frequency measures and its impact on knowledge discovery in single sequences», dans Data Mining Workshops (ICDMW), 2010 IEEE International Conference on, p. 859–866, doi :10.1109/ICDMW.2010.83.

[13] «Sequential pattern mining – approaches and algorithms», ACM Comput. Surv., vol. 45, no 2, doi :10.1145/2431211.2431218, p. 19 :1–19 :39, ISSN 0360-0300. URL http://doi.acm.org/10.1145/2431211.2431218.

[14] Agrawal, R. et R. Srikant. 1995, «Mining sequential patterns», dans Data Engineering, 1995.Proceedings of the Eleventh International Conference on, IEEE, p. 3–14.

[15] Han, J. et J. Pei. 2000, «Mining frequent patterns by pattern-growth : methodology and implications», ACM SIGKDD explorations newsletter, vol. 2, no 2, p. 14–20.

[16] Yang, J., W. Wang, P. S. Yu et J. Han. 2002, «Mining long sequential patterns in a noisy environment», dans Proceedings of the 2002 ACM SIGMOD international conference on Management of data, ACM, p. 406–417.

[17] Srikant, R. et R. Agrawal. 1996b, Mining sequential patterns : Generalizations and performance improvements, Springer.

[18] Masseglia, F., F. Cathala et P. Poncelet. 1998, «The psp approach for mining sequential patterns», dans Principles of Data Mining and Knowledge Discovery, Lecture Notes in Computer Science, vol. 1510, édité par J. Zytkow et M. Quafafou, Springer Berlin Heidelberg, ISBN 978-3-540-65068-3, p. 176–184, doi :10.1007/BFb0094818. URL http://dx.doi.org/10.1007/BFb0094818.

[19] Garofalakis, M. N., R. Rastogi et K. Shim. 1999, «Spirit : Sequential pattern mining with regular expression constraints», dans VLDB, vol. 99, p. 7–10.

[20] Zhang, M., B. Kao, C.-L. Yip et D. Cheung. 2001, «A gsp-based efficient algorithm for mining frequent sequences», dans Proc. of IC-AI, p. 497–503.

[21] Zaki, M. 2001, «Spade: An efficient algorithm for mining frequent sequences», Machine Learning, vol. 42, no 1-2, doi :10.1023/A:1007652502315, p. 31–60, ISSN 0885-6125. URL http://dx.doi.org/10.1023/A%3A1007652502315.

[22] Ayres, J., J. Flannick, J. Gehrke et T. Yiu. 2002, «Sequential pattern mining using a bitmap representation», dans Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '02, ACM, New York, NY, USA, ISBN 1-58113-567-X, p. 429–435, doi :10.1145/775047.775109. URL http://doi.acm.org/10.1145/775047.775109.

[23] Yang, Z. et M. Kitsuregawa. 2005, «Lapin-spam : An improved algorithm for mining sequential pattern», dans Data Engineering Workshops, 2005. 21st International Conference on, p. 1222–1222, doi :10.1109/ICDE.2005.235.

[24] J. Han, H. Pei and Y. Yin. Mining Frequent Patterns without Candidate Generation. In Proc. Conf. on the Management of Data (SIGMOD'00, Dallas, TX), ACM Press, New York, NY, USA 2000.

[25] Han, J. et J. Pei. 2000, «Mining frequent patterns by pattern-growth : methodology and implications», ACM SIGKDD explorations newsletter, vol. 2, no 2, p. 14–20.

[26] Pei, J., J. Han, B. Mortazavi-asl, H. Pinto, Q. Chen, U. Dayal et M. chun Hsu. 2001, «Prefixspan :Mining sequential patterns efficiently by prefix-projected pattern growth», p. 215–224.

[27] X. Yan, J. Han, and R. Afshar. Clospan: Mining closed sequential patterns in large datasets, 2003.

[28] F. Masseglia, P. Poncelet, and M. Teisseire. Incremental mining of sequential patterns in large databases. Data Knowl. Eng., 46(1):97–121, 2003.

[29] S. Parthasarathy, M. Zaki, M. Ogihara, and S. Dwarkadas. Incremental and interactive sequence mining. In Proc. of the 8th Int. Conf. on Information and Knowledge Management (CIKM'99), Nov 199

[30] Misra J, Gries D (1982) Finding repeated elements. Science of Computer Programming 2:143–152

[31] G.S.Manku, and R.Motwani. “Approximate frequency counts over data streams”. In Proceedings of the 28th International Conference on Very Large Data Bases (VLDB), pp. 346-357, August 2002.

[32] M. Charikar, K. Chen, M. Farach-Colton, Finding frequent items in data streams, Theoretical Computer Science, 312(1), p.3–13, 2004.

[33] G. Cormode, S. Muthukrishnan, An improved data stream summary: the count-min sketch and its applications, Journal of Algorithms, 55(1), p.58–75, 2005.

[34] Li H, Wang Y, Zhang D, Zhang M, Chang EY. Pfp: parallel fp-growth for query recommendation. In: Proceedings of the 2008 ACM conference on recommender systems. RecSys ’08. New York: ACM; 2008. p. 107–14. https ://doi.org/10.1145/14540 08.14540 27.

[35] Load Balancing, https://sysreseau.net/load-balancing-repartition-de-charge/. 2018

[36] Zhou L, Zhong Z, Chang J, Li J, Huang JZ, Feng S. Balanced parallel fp-growth with mapreduce. In: 2010 IEEE youth conference on information, computing and telecommunications. 2010. p. 243–6. https ://doi.org/10.1109/YCICT.2010.57130 90.

[37] Yang XY, Liu Z, Fu Y. Mapreduce as a programming model for association rules algorithm on hadoop. In: Proceedings of the 3rd international conference on information sciences and interaction sciences. 2010. p. 99–102. https ://doi.org/10.1109/ICICI S.2010.55347 18.

[38] Li L, Zhang M. The strategy of mining association rule based on cloud computing. In: Proceeding of international conference on business computing and global informatization. 2011. p. 475–8. https ://doi.org/10.1109/BCGIn.2011.125.

[39] Yu H, Wen J, Wang H, Jun L. An improved apriori algorithm based on the boolean matrix and hadoop. Procedia Eng.2011;15:1827–31.

[40] Li. A parallel algorithm of association rules based on cloud computing. In: Proceedings of 8th international conference on communications and networking in China

(CHINACOM). 2013. p. 415–9. https ://doi.org/10.1109/China Com.2013.66946 32.

[41] Lin X. Mr-apriori: association rules algorithm based on mapreduce. In: Proceedings of IEEE 5th international conference on software engineering and service science. 2014. p. 141–4. https ://doi.org/10.1109/ICSES S.2014.69335 31.

[42] Li: parallel fp-growth for query recommendation. In: Proceedings of the 2008 ACM conference on recommender systems. RecSys ’08. New York: ACM; 2008. p. 107–14. https ://doi.org/10.1145/14540 08.14540 27.

[43] Yahya O, Hegazy O, Ezat E. An efficient implementation of A-Priori algorithm based on Hadoop-Mapreduce model.Int J Rev Comput. 2012;12.

[44] Riondato M, DeBrabant JA, Fonseca R, Upfal E. Parma: a parallel randomized algorithm for approximate association rules mining in mapreduce. In: Proceedings of the 21st ACM international conference on information and knowledge management. CIKM ’12. New York: ACM; 2012. p. 85–94. https ://doi.org/10.1145/23967 61.23967 76.

[45] Kovács F, Illés J. Frequent itemset mining on hadoop. In: 2013 IEEE 9th international conference on computational cybernetics (ICCC). 2013. p. 241–5. https://doi.org/10.1109/ICCCy b.2013.66175 96.

[46] Oruganti S, Ding Q, Tabrizi N. Exploring hadoop as a platform for distributed association rule mining. In: FUTURE COMPUTING 2013-the fifth international conference on future computational technologies and applications. 2013.p. 62–7.

[47] Yong W, Zhe Z, Fang W. A parallel algorithm of association rules based on cloud computing. In: Proceedings of 8th international conference on communications and networking in China (CHINACOM). 2013. p. 415–9. https ://doi.org/10.1109/China Com.2013.66946 32.

[48] Moens S, Aksehirli E, Goethals B. Frequent itemset mining for big data. In: Proceedings of IEEE international conference on big data. 2013. p. 111–8. https ://doi.org/10.1109/BigDa ta.2013.66917 42.

[49] Farzanyar Z, Cercone N. Efficient mining of frequent itemsets in social network data based on mapreduce framework.In: Proceedings of the 2013 IEEE/ACM international

conference on advances in social networks analysis and mining. ASONAM ’13. New York: ACM; 2013. p. 1183–8. https ://doi.org/10.1145/24925 17.25003 01.

[50] Barkhordari M, Niamanesh M. Scadibino: an effective mapreduce-based association rule mining method. In:Proceedings of the sixteenth international conference on electronic commerce. ICEC ’14. New York: ACM; 2014. p.1–118. https ://doi.org/10.1145/26178 48.26178 53.

[51] Singh S, Garg R, Mishra P. Performance analysis of apriori algorithm with different data structures on hadoop cluster.2015. arXiv preprint arXiv :1511.07017 .

[52] Zhang F, Liu M, Gui F, Shen W, Shami A, Ma Y. A distributed frequent itemset mining algorithm using spark for bigdata analytics. Clust Comput. 2015;18(4):1493–501.

[53] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In SIGMOD ’92 : Proceedings of the 1992 ACM SIG- MOD international conference on Management of data, pages 81–90, New York, NY, USA, 1992. ACM.

[54] Tsai, C.-Y. et Y.-C. Shieh. 2009, «A change detection method for sequential patterns», Decision Support Systems, vol. 46, no 2, p. 501–511.

[55] Böttcher, M., F. Höppner et M. Spiliopoulou. 2008, «On exploiting the power of time in data mining», SIGKDD Explor. Newsl., vol. 10, no 2, doi :10.1145/1540276.1540278, p. 3–11, ISSN 1931-0145. URL http://doi.acm.org/10.1145/1540276.1540278.

[56] Lambda Architecture :http://blog.xebia.fr/2014/05/07/devoxxfr-lambda-architecture-choose-tools-for-real-timebig-data/. 2018

[57] Holden Karau, Rachel Warren: High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark, p. 17-35, 2016.

[58] Apache Kafka. http://kafka.apache.org/. 2018

[59] Apache Flume. https://flume.apache.org/.2017

[60] ZeroMq http://zeromq.org .2018

[61] Kinesis https://aws.amazon.com/fr/kinesis/.2019

[62] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In Proc. VLDB,pages 487–499, 1994.

[63] Hongjian Qiu, Rong Gu, Chunfeng Yuan and Yihua Huang. YAFIM: A Parallel Frequent Itemset MiningAlgorithm with Spark. In2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops,2014.

[64] Li N., Zeng L., He Q. & Shi Z. Parallel Implementation of Apriori Algorithm Based on MapReduce. In Proc. of the13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD '12), Kyoto,IEEE: 236 – 241, 2012.

TABLE DES FIGURES

Figure 2.1 : Une séquence et sa représentation sous forme de table 12
Figure 2.2 : Une séquence et sa représentation sous forme de table 14
Figure 2.3 : Exemple d'implémentation de l'algorithme GSP 16
Figure 2.6 : Pseudo implémentation de l'algorithme SPADE 18
Figure 2.7 : Système de management des flots de données 27
Figure 2.8 : Une base de données DB et un incrément de base db contenant de nouvelles transactions 31
Figure 2.9 : Une base de données DB et un incrément de base db contenant de nouvelles transactions 32
Figure 2.10 : La bordure négative considérée par ISM après l'exécution du SPAD sur la base de données du tableau 2.6 avant la mise à jour 34
Figure 2.11 : Exemple de matrice utilisée par l'algorithme Count-min Sketch 37
Figure 3.1 : Trois occurrences d'un motif (A, B, B, C) 46
Figure 3.2 : Différentes évolutions possibles d'un motif et de ses occurrences d'un intervalle de temps à un autre. 49
Figure 3.3 : Deux partitions possibles sur des données identiques 50
Figure 3.4 : Évolution de la fréquence d'un motif sur une tripartition 51
Figure 3.5 : Évolution de la fréquence d'un motif sur une bipartition 52
Figure 3.6 : Les composants de l'architecture Lambda [56] 54
Figure 3.7 : La couche Batch [56] 55
Figure 3.8 : Vue temps réel= fonction (données récentes) 56
Figure 3.9 : Processus d'extraction des motifs évolutifs 58
Figure 3.10 : L'écosystème Spark [57] 60
Figure 3.11 : Architecture du Spark Runtime [57] 62
Figure 3.12 : Spark streaming [57] 64
Figure 4.1 diffusion du fichier d'entrée sur les nœuds du cluster 73
Figure 4.2 : Phase 1 de notre approche (a). L'algorithme utilisé lors de la phase (b). Exemple d'exécution de la phase 1 (c) où T : $\{T_1,T_2,\ldots\}$ est un ensemble de transaction 75
Figure 4.3 : Phase 1 de notre approche (a). L'algorithme utilisé lors de la phase (b). Exemple d'exécution de la phase 1 (c) où T : $\{T_1,T_2,\ldots\}$ est un ensemble de transaction 83
Figure 4.4 : L'algorithme utilisé lors de la phase 3 (a). Exemple d'exécution de la phase 3 (b) où T : {T1,T2,…} est un ensemble de transaction 85
Figure 5.1: Échantillon T25I10D10K avec un min sup=0.10 % 89
Figure 5.2 : Échantillon Extract0809 avec un min sup=0.60 % 89
Figure 5.3 : Échantillon T10I4D100K avec un min sup=1.00 % 90

LISTE DES TABLES

Tableau 2-1 : Base de séquences S ayant un min supp = 2 20
Tableau 2-2 : Tableau comparatif des différents algorithmes d'extraction des motifs séquentiels 26
Tableau 2-3 : Ensemble des batch B01, B12 et B23 29
Tableau 2-4 : Motifs fréquents dans l'intervalle [a_i, b_i] avec un min sup de 50 % 30
Tableau 2-5 : DBspade, une base de données ainsi que sa mise à jour 33
Tableau 3-1 : Cadre d'évolution de chaque type d'évolution 48
Tableau 3-2 : Rôles des différents composants Spark 63
Tableau 5-1 Détails des échantillons de données utilisés 88
Tableau 5-2 : Détails des échantillons de données utilisés 90

ANNEXES

Algorithmes	*Support %*	*Runtime (ms)*	*Mémoire utilisée (mb)*	*Nb de séquence fréquente*	*Taille du fichier*
Apriori	1	1726	80,09	83	3.5 mb
	5	132	90.27	2	
	10	174	16.78	2	
	15	151	21.33	2	
	20	150	25.62	2	
	25	205	11.84	2	
	30	149	16.83	2	
GSP	1	35 841	165,21	81	
	5	0	120,74	0	
	10	0	115,5	0	
	15	0	109,17	0	
	20	0	121,24	0	
	25	0	108,11	0	
	30	0	107,62	0	
SPADE	1	KO	Java heap space	0	
	5	KO	Java heap space	0	

	10	KO	Java heap space	0	
	15	KO	Java heap space	0	
	20	KO	Java heap space	0	
	25	KO	Java heap space	0	
	30	KO	Java heap space	0	
Prefix Span	1	194	51.32	81	
	5	153	51.32	0	
	10	153	51.32	0	
	15	153	51.32	0	
	20	153	51.32	0	
	25	153	51.32	0	
	30	153	51.32	0	
Clospan	1	780	98.65	81	
	5	45	135.37	0	
	10	66	97	0	
	15	36	122.55	0	
	20	39	137.91	0	
	25	217	68.74	0	
	30	44	184.81	0	
Lapin	1	KO	Java heap space	0	

	5	500	40.47	0	
	10	516	74.34	0	
	15	542	107.84	0	
	20	518	141	0	
	25	509	176	0	
	30	516	141	0	
SPAM	1	2223	208.97	81	
	5	0	0	4	
	10	0	0	0	
	15	0	0	0	
	20	0	0	0	
	25	0	0	0	
	30	0	0	0	
CM_SPAM	1	551	46.23	81	
	5	388	0	4	
	10	424	0	0	
	15	417	0	0	
	20	395	0	0	
	25	441	0	0	
	30	461	0	0	

CM-SPADE	1	KO	Java heap space	0	
	5	KO	Java heap space	0	
	10	KO	Java heap space	0	
	15	KO	Java heap space	0	
	20	KO	Java heap space	0	
	25	KO	Java heap space	0	
	30	KO	Java heap space	0	
FPGrowth	1	433	86	164	
	5	486	35	2	
	10	286	52	2	
	15	253	70.85	2	
	20	264	71.79	2	
	25	283	36.92	2	
	30	259	55	2	

Tableau A 1 : Détails du Benchmark réalisé pour comparer les performances des algorithmes (Apriori, GSP, SPADE, Prefix Span, Clospan, Lapin,SPAM, .etc) lors du traitment d'un fichier de 3.5 mb

Algorithmes	*Support %*	*Runtime (ms)*	*Mémoire utilisée (mb)*	*Nb de séquence fréquente*	*Taille du fichier*
Apriori	1	19 192	98,53	79	14 mb
	5	861	35.96	6	
	10	720	45.39	2	
	15	721	43.59	2	
	20	731	41.7	2	
	25	782	36.75	2	
	30	738	40.54	2	
GSP	1	KO	Java heap space	0	
	5	KO	Java heap space	0	
	10	KO	Java heap space	0	
	15	KO	Java heap space	0	
	20	KO	Java heap space	0	
	25	KO	Java heap space	0	
	30	KO	Java heap space	0	
SPADE	1	KO	Java heap space	0	
	5	KO	Java heap space	0	
	10	KO	Java heap space	0	
	15	KO	Java heap space	0	
	20	KO	Java heap space	0	
	25	KO	Java heap space	0	
	30	KO	Java heap space	0	
Prefix Span	1	1049	138.15	77	
	5	846	153.85	4	
	10	946	170.89	0	
	15	747	170.89	0	
	20	855	170.9	0	
	25	804	170.91	0	
	30	792	170.89	0	
Clospan	1	KO	Java heap space	0	
	5	KO	Java heap space	0	

	10	KO	Java heap space	0	
	15	KO	Java heap space	0	
	20	KO	Java heap space	0	
	25	KO	Java heap space	0	
	30	KO	Java heap space	0	
Lapin	1	KO	Java heap space	0	
	5	KO	Java heap space	0	
	10	KO	Java heap space	0	
	15	KO	Java heap space	0	
	20	KO	Java heap space	0	
	25	KO	Java heap space	0	
	30	KO	Java heap space	0	
SPAM	1	25486	149.37	77	
	5	1278	119.55	4	
	10	1060	0	0	
	15	1067	0	0	
	20	1239	0	0	
	25	1096	0	0	
	30	1042	0	0	
CM- SPAM	1	2553	140.76	77	
	5	1699	150.11	4	
	10	1484	0	0	
	15	1478	0	0	
	20	1499	0	0	
	25	1625	0	0	
	30	1768	0	0	
CM- SPADE	1	KO	Java heap space	0	
	5	KO	Java heap space	0	
	10	KO	Java heap space	0	
	15	KO	Java heap space	0	
	20	KO	Java heap space	0	
	25	KO	Java heap space	0	

	30	KO	Java heap space	0	
FPGrowth	1	1666	66	156	
	5	1223	67	10	
	10	1183	73	2	
	15	1205	75	2	
	20	1216	80	2	
	25	1270	81	2	
	30	1210	83	2	

Tableau A 1 : Détails du Benchmark réalisé pour comparer les performances des algorithmes (Apriori, GSP, SPADE, Prefix Span, Clospan, Lapin,SPAM, .etc) lors du traitment d'un fichier de 14 mb

RESUME

La révolution numérique influence fortement le domaine de la santé, bouleversant dès à présent la recherche médicale tout autant que l'organisation des soins.

Notre projet de recherche s'inscrit dans le contexte de la réhabilitation améliorée qui a été introduite en chirurgie colorectale en 1997 par le chirurgien danois Henrik Kehlet. Il s'agit d'un programme pluridisciplinaire et inter métiers ayant pour but de réduire au maximum l'impact du stress chirurgical sur l'homéostasie physiologique du patient.

L'application d'un processus de réhabilitation améliorée offre des bénéfices sur les suites opératoires et permet de réduire jusqu'à plus de 50 % des complications postopératoires avec une amélioration de la qualité de vie des patients.

Les travaux de cette thèse portent sur la conception de chemins cliniques ad hoc, c'est-à-dire le protocole et la médication les mieux adaptés afin que le patient retrouve au plus vite ses capacités. On s'intéresse plus particulièrement à l'exploitation de données relatives au suivi des patients, dans une configuration Big Data, afin d'extraire des motifs permettant de modéliser des chemins cliniques personnalisés. On se place donc dans le cadre de la fouille de données temporelles, en s'intéressant à la fouille de motifs dits évolutifs.

Dans ce sens, nous proposons à travers cette thèse une nouvelle approche d'extraction des items fréquents. Le challenge par rapport à notre approche est d'exploiter des techniques d'extraction traditionnelles dans un environnement distribué.

Mots-clés : **Fouille de données, Big Data, Motifs temporels évolutifs.**

Université Bourgogne Franche-Comté
32, avenue de l'Observatoire
25000 Besançon