Modélisation et développement d'un observatoire générique pour la collecte et l'analyse de données massives

Annabelle Gillet

▶ **To cite this version:**

Annabelle Gillet. Modélisation et développement d'un observatoire générique pour la collecte et l'analyse de données massives. Autre [cs.OH]. Université Bourgogne Franche-Comté, 2021. Français. NNT : 2021UBFCK076 . tel-04085276

HAL Id: tel-04085276

https://theses.hal.science/tel-04085276

Submitted on 28 Apr 2023

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.

THÈSE DE DOCTORAT

DE L'ÉTABLISSEMENT UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

PRÉPARÉE À L'UNIVERSITÉ DE BOURGOGNE

École doctorale n° 37 Sciences Pour l'Ingénieur et Microtechnique

Doctorat d'Informatique

Par

Annabelle Gillet

Modélisation et développement d'un observatoire générique pour la collecte et l'analyse de données massives

Thèse présentée et soutenue à Dijon le 2 décembre 2021, devant le jury composé de :

M. Jérôme Darmont	Professeur à l'Université Lyon 2	Président
M. Bernd Amann	Professeur à Sorbonne Université	Rapporteur
M. Ladjel Bellatreche	Professeur à l'École Nationale Supérieure de Mécanique et d'Aérotechnique de Poitiers	Rapporteur
Mme Ioana Manolescu	Directrice de recherche à Inria Saclay–Île-de-France	Examinatrice
Mme Nadine Cullot	Professeure à l'Université de Bourgogne	Directrice de thèse
M. Éric Leclercq	Professeur à l'Université de Bourgogne	Codirecteur de thèse

Remerciements

*****

Je tiens tout d'abord à remercier les membres du jury qui ont accepté d'évaluer mon travail de thèse :

- Monsieur Bernd Amann, professeur à Sorbonne Université ;
- Monsieur Ladjel Bellatreche, professeur à l'École Nationale Supérieure de Mécanique et d'Aérotechnique de Poitiers ;
- Monsieur Jérôme Darmont, professeur à l'Université Lyon 2 ;
- Madame Ioana Manolescu, directrice de recherche à Inria Saclay–Île-de-France.

C'est un honneur de présenter mes travaux devant ce jury.

Je remercie Nadine Cullot, professeure à l'Université de Bourgogne et directrice de thèse, et tout particulièrement Éric Leclercq, professeur à l'Université de Bourgogne et co-directeur de thèse, pour m'avoir donné l'opportunité de réaliser ma thèse dans de bonnes conditions, mais également pour l'accompagnement, le soutien, et les échanges et débats scientifiques passionnés.

Je remercie également Alexis Guyot, futur doctorant à l'Université de Bourgogne qui a tout ce qu'il faut pour continuer sur la voie académique, pour les échanges qualitatifs, mais aussi les autres doctorants du Laboratoire d'Informatique de Bourgogne pour l'environnement de travail.

Le projet Cocktail m'ayant permis de développer et de mettre en pratique plusieurs de mes travaux, je remercie ses membres pour cette opportunité.

Je n'oublie pas non plus ma mère et ma sœur, qui m'ont aidée à être là où j'en suis aujourd'hui.

Table des matières

*****

1 Introduction **1**
- 1.1 Les données massives : caractéristiques et utilisation . . . 2
- 1.2 Les architectures dédiées aux données massives . . . 4
- 1.3 Les outils d'analyse . . . 5
- 1.4 Le projet Cocktail : un contexte applicatif . . . 7
- 1.5 Contributions et plan de la thèse . . . 8

2 Lambda+ Architecture et théorie des catégories : les propriétés au cœur des architectures **11**
- 2.1 Introduction . . . 12
  - 2.1.1 Les styles et patrons d'architectures . . . 12
  - 2.1.2 Les architectures dédiées au traitement des données massives . . . 13
  - 2.1.3 Le besoin de formaliser les architectures . . . 14
  - 2.1.4 Contribution . . . 14
- 2.2 État de l'art : la Lambda Architecture, l'avènement des systèmes de *stream processing* et la formalisation d'architectures . . . 15
  - 2.2.1 La Lambda Architecture . . . 15
  - 2.2.2 L'avènement des systèmes de *stream processing* . . . 18
  - 2.2.3 La formalisation d'architectures . . . 21
  - 2.2.4 Synthèse . . . 23
- 2.3 La Lambda+ Architecture . . . 23
  - 2.3.1 La définition du patron . . . 23
  - 2.3.2 L'architecture Hydre : une implémentation du patron . . . 27
- 2.4 Théorie des catégories pour la vérification des propriétés des architectures . . . 29
  - 2.4.1 Les concepts de base . . . 29
  - 2.4.2 Les concepts avancés . . . 31
  - 2.4.3 Les principes de la vérification . . . 33
  - 2.4.4 La démonstration de la vérification des propriétés d'une architecture . . . 36
  - 2.4.5 La vérification des propriétés de la Lambda Architecture . . . 40
  - 2.4.6 La vérification des propriétés de la Lambda+ Architecture . . . 42
- 2.5 Conclusion et perspectives . . . 45

3 Tensor Data Model : un modèle de données tensoriel pour les polystores 49
3.1 Introduction . . . 50
3.1.1 L'essor des *polystores* . . . 51
3.1.2 Les tenseurs comme outils d'analyse . . . 52
3.1.3 L'importance de la sûreté des programmes d'analyse . . . 53
3.1.4 Contribution . . . 53
3.2 État de l'art : modèles pivots, tenseurs et librairies de tenseurs . . . 54
3.2.1 Les modèles pivots pour les données hétérogènes . . . 54
3.2.2 L'objet mathématique tenseur . . . 56
3.2.3 Les librairies de tenseurs existantes . . . 61
3.2.4 Synthèse . . . 63
3.3 Tensor Data Model et opérateurs . . . 63
3.3.1 Le schéma . . . 64
3.3.2 Les opérateurs de TDM . . . 66
3.3.3 Les correspondances entre TDM et les autres modèles de représentation des données 74
3.4 Typage sûr et inférence de schéma . . . 75
3.4.1 Les mécanismes utilisés . . . 75
3.4.2 Vers un langage de requêtes fonctionnel . . . 78
3.5 Mise en œuvre d'optimisations . . . 81
3.5.1 Les optimisations de la librairie générale . . . 81
3.5.2 Les optimisations de la décomposition CANDECOMP/PARAFAC . . . 82
3.6 Conclusion . . . 87

4 Implémentation de l'architecture Hydre et du modèle TDM 91
4.1 Introduction : sûreté, optimisation et reproductibilité . . . 92
4.2 L'architecture Hydre : une implémentation du patron Lambda+ Architecture . . . 94
4.2.1 Les spécifications de l'architecture . . . 94
4.2.2 L'architecture physique . . . 99
4.3 Implémentation de TDM . . . 100
4.3.1 Les *implicits* . . . 100
4.3.2 La sûreté du typage . . . 101
4.3.3 L'inférence de schéma . . . 102
4.3.4 L'utilisation de TDM . . . 103
4.4 Conclusion . . . 104

5 Un observatoire des données sociales : méthodes et techniques de collecte et d'analyse 109
5.1 Introduction . . . 110
5.1.1 Le besoin d'un observatoire des données sociales . . . 110
5.1.2 La démarche . . . 111
5.1.3 Les cas d'étude . . . 112
5.2 De la collecte à la constitution des corpus et leur mise à disposition . . . 113
5.2.1 La collecte des données . . . 113
5.2.2 La constitution des corpus d'étude . . . 113
5.2.3 La mise à disposition des corpus . . . 114

5.3 Techniques d'analyses . . . 114
5.3.1 Les analyses macroscopiques . . . 116
5.3.2 Les analyses mésoscopiques . . . 118
5.3.3 Les analyses microscopiques . . . 121
5.4 Conclusion . . . 131

6 Conclusion et perspectives 135
6.1 Contributions . . . 136
6.1.1 La Lambda+ Architecture et la formalisation d'architectures basée sur la théorie des catégories . . . 136
6.1.2 Le *Tensor Data Model* . . . 137
6.1.3 L'implémentation des contributions . . . 138
6.1.4 La mise en place de l'observatoire . . . 138
6.2 Perspectives . . . 138
6.2.1 La Lambda+ Architecture et la formalisation d'architectures basée sur la théorie des catégories . . . 139
6.2.2 Le *Tensor Data Model* . . . 139
6.2.3 Synthèse . . . 140

Liste générale des publications 141

Bibliographie 143

Index des termes 155

Index des auteurs 157

Table des figures

*****

1.1 Les tâches les plus chronophages et les moins appréciées des *data scientists* 3

2.1 Les composants de la Lambda Architecture 15
2.2 Aperçu de la Kappa Architecture 20
2.3 Aperçu de la Lambda+ Architecture 24
2.4 L'architecture Hydre 27
2.5 Illustration des foncteurs 31
2.6 Illustration des préordres 32
2.7 Un ***power set*** représentant l'ensemble $\{a, b, c\}$ 32
2.8 Produit de catégories et sa simplification schématique 33
2.9 Une architecture simple avec deux composants a et b 34
2.10 Déduction d'une propriété complexe en utilisant un produit de catégories 35
2.11 Démonstration de la vérification : déclaration des éléments connus 36
2.12 Démonstration de la vérification : déduction des valeurs des propriétés pour une composition de composants 40
2.13 Formalisation de la Lambda Architecture 40
2.14 Formalisation de la Lambda+ Architecture 43

3.1 Illustration des tenseurs de différents modes 57
3.2 Matricisation d'un tenseur 58
3.3 L'architecture de TDM 64
3.4 Correspondance entre le modèle relationnel et TDM 74
3.5 Étapes permettant de vérifier si un type `T` est dans une `HList L` à l'aide d'*implicits* 77
3.6 Exemple du retrait de tous les éléments de type `T` dans une `HList L` à l'aide des *implicits* et des *path dependent types* 77
3.7 Temps d'exécution pour des opérations équivalentes avec TDM et avec Spark 82
3.8 *Mapping* des blocs pour une multiplication entre deux `BlockMatrix` 84
3.9 Temps d'exécution de la décomposition CANDECOMP/PARAFAC 87

4.1 L'architecture Hydre détaillée 95
4.2 Collecte de tweets et articulation avec les topics Kafka 96

4.3 Interface de *monitoring* des machines de collecte . . . 98
4.4 Architecture physique utilisée pour le déploiement d'Hydre . . . 99

5.1 *Workflow* général d'une analyse . . . 111
5.2 Détail des étapes de la création d'un corpus d'étude . . . 114
5.3 Modèle relationnel du stockage des tweets . . . 115
5.4 Modèle graphe du stockage des tweets . . . 116
5.5 Modèle séries temporelles du stockage des tweets . . . 117
5.6 Top-k des hashtags, leur évolution et visualisation d'un tweet populaire . . . 118
5.7 Série temporelle d'un hashtag . . . 119
5.8 Série temporelle de la publication des hashtags dans le corpus concernant la vaccination contre le COVID . . . 119
5.9 Méta-schéma pour stocker les résultats d'analyse . . . 120
5.10 Centralité des utilisateurs dans le corpus concernant la vaccination contre le COVID . . . 120
5.11 Extrait des communautés de hashtags dans le corpus concernant la vaccination contre le COVID . . . 121
5.12 Extrait des méthodes *elbow* et silhouette pour déterminer le nombre de *clusters* pour l'algorithme *k-means* . . . 124
5.13 Matrice d'antagonisme . . . 126
5.14 Décomposition d'un tenseur de taille $60 \times 60 \times 60$ avec trois *clusters* . . . 127
5.15 Décomposition de rang 2 sur un tenseur déflaté de taille $60 \times 60 \times 60$ avec deux *clusters* . . . 128
5.16 Communautés détectées dans le jeu de données de l'école primaire . . . 129
5.17 Extrait des attaques détectées dans le jeu de données DARPA1998 . . . 130

Liste des tableaux

*****

2.1 Déclaration des morphismes (les morphismes identité sont omis) 37
2.2 Déclaration de l'effet des foncteurs sur les composants individuels 37
2.3 Déclaration de l'effet des foncteurs sur les objets du produit de catégories 38
2.4 Déclaration des morphismes des catégories de la Lambda Architecture (les morphismes identité sont omis) 41
2.5 Déclaration de l'effet des foncteurs sur les composants individuels de la Lambda Architecture 41
2.6 Déclaration des morphismes des catégories de la Lambda+ Architecture (les morphismes identité sont omis) 44
2.7 Déclaration de l'effet des foncteurs sur les composants individuels de la Lambda+ Architecture 45
2.8 Déclaration de l'effet des foncteurs sur les objets du produit de catégories de la Lambda+ Architecture 46

3.1 Symboles et opérateurs utilisés 57
3.2 Exemple de l'application de l'opérateur de projection 67
3.3 Exemple de l'application de l'opérateur de sélection 68
3.4 Exemple de l'application de l'opérateur de restriction 68
3.5 Exemple de l'application de l'opérateur d'union 69
3.6 Exemple de l'application de l'opérateur d'intersection 70
3.7 Exemple de l'application de l'opérateur de jointure naturelle 71
3.8 Exemple de l'application de l'opérateur de différence 72

5.1 Communautés significatives découvertes par l'algorithme de Louvain et leur caractérisation par les hashtags 125
5.2 Probabilité d'apparition de chaque *cluster* avec une décomposition de rang 1 128

Chapitre

1

Introduction

*****

1.1 Les données massives : caractéristiques et utilisation 2
1.2 Les architectures dédiées aux données massives 4
1.3 Les outils d'analyse 5
1.4 Le projet Cocktail : un contexte applicatif 7
1.5 Contributions et plan de la thèse 8

Les données massives (ou *Big Data*) sont présentes dans toutes les organisations. Les outils de production et de collecte de données se sont multipliés dans tous les domaines, allant du bâtiment à la santé en passant par la gestion des métropoles avec les *smart cities*. Il ne s'agit plus aujourd'hui de données sur les activités de l'entreprise dont la sémantique est bien connue et dont les schémas sont bien définis, mais de données hétérogènes présentant de grandes variabilités syntaxiques, du fait des formats ou modèles de représentation, et sémantiques, du fait des usages multiples. Les données d'entreprise bien identifiées sont interconnectées à d'autres types de données, comme celles issues des réseaux sociaux pour des besoins marketing par exemple. Le mouvement *open-data* initié par les états contribue également à la diversité, au volume et à l'interconnexion des données.

Bien que l'accès aux données massives ait été démocratisé, leur usage est encore principalement réservé à des organisations qui peuvent disposer d'une équipe d'analystes métier, de *data engineers*, de *data scientists* et de développeurs. Le rôle des *data scientists* et *data engineers* est crucial, puisqu'ils doivent traduire les besoins métiers en analyses adaptées, sélectionner et extraire les données, s'assurer de leur qualité, déterminer les algorithmes à appliquer en fonction des besoins identifiés mais aussi en fonction du type et des caractéristiques des données [1], puis exécuter ces algorithmes avec les données mises en forme pour les alimenter. Toutes ces tâches doivent idéalement s'effectuer en intégrant les capacités de passage à l'échelle, de reproductibilité et de réutilisation pour aboutir à des *workflows* d'analyse robustes. Ceci est parfois rendu compliqué par les outils utilisés qui peuvent privilégier la rapidité de développement à sa pérennité. Par ailleurs, certaines données sont produites en continu et ne peuvent pas être stockées dans leur intégralité.

1. Comme les données contenant des relations qui suivent une loi de puissance.

Elles nécessitent alors d'avoir recours à d'autres paradigmes tels que le *stream processing*. Ce paradigme est fortement orienté vers le temps réel, et vise à produire des résultats d'analyse en continu, en fonction du flux de données source. Au niveau de la prise de décision, cela permet de produire des indicateurs répondant à des besoins bien identifiés, et ainsi d'alerter lorsque ces indicateurs prennent une valeur nécessitant une intervention. Ces différentes exigences constituent des verrous technologiques empêchant les entreprises ne disposant pas d'équipe dédiée de pouvoir collecter, stocker et analyser plus facilement les masses de données.

Cette thèse s'inscrit dans ce cadre général et s'intéresse aux traitements réalisés sur les données massives, que ce soit leur collecte, leur stockage ou leur analyse. Ces trois aspects sont dépendants les uns des autres, et peuvent donc avoir un impact, en fonction de la qualité de leur réalisation, sur les systèmes construits et les décisions qui sont prises à partir de résultats d'analyses de données. Dans les sections suivantes, après avoir présenté les données massives et leurs caractéristiques, nous nous intéressons à deux éléments qui serviront de base pour définir les problématiques de recherche traitées dans cette thèse : les architectures logicielles et les outils d'analyse.

1.1 Les données massives : caractéristiques et utilisation

Les données massives sont convoitées du fait de la quantité d'informations qu'elles contiennent et qui peuvent être extraites afin de produire de la connaissance. Toutefois, les données massives sont difficiles à manipuler, en raison de leurs caractéristiques souvent qualifiées des **5V**, qui correspondent à :

- **Volume** : les données massives ont un volume important, généralement de plusieurs Téraoctets et pouvant aller jusqu'à plusieurs Pétaoctets. Le volume entraîne des problématiques pour **stocker** et **traiter** ou **analyser** les données, et nécessite des capacités de disques et parfois de RAM élevées ;
- **Vélocité** : en plus de la caractéristique de volume, celle de vélocité indique que les données massives sont **créées rapidement et en continu**. Cela force les systèmes à être suffisamment performants pour avoir le temps de traiter les données avant qu'elles ne s'accumulent, et éventuellement ne stocker que celles qui sont essentielles ou les réduire à des résumés ;
- **Variété** : il existe une grande variété dans le **format** des données massives, qui peut manquer de **structure**. Ce format n'est pas forcément statique. Il est nécessaire d'avoir une **flexibilité** permettant de s'adapter à cette variété pour pouvoir exploiter au mieux les données massives. Les données peuvent également avoir plusieurs centaines d'attributs descriptifs pour une même entité, et de ce fait le regroupement de plusieurs milliers d'entités ou de mesures induit une complexité descriptive qui va au delà du simple volume ;
- **Véracité** : les données n'ont pas toutes le même **niveau de qualité**. Il est important de connaître leur source ainsi que son degré de fiabilité. Comme certaines sources de données peuvent être utilisées conjointement, l'intégration d'une source peu fiable peut impacter l'ensemble du système, et diminuer la **qualité** des analyses produites, ce qui peut mener à des interprétations erronées ;
- **Valeur** : l'attrait des données massives provient de la valeur que ceux qui les exploitent cherchent à obtenir. À cause des caractéristiques précédemment citées, la valeur est **difficile à obtenir**, mais représente un **apport significatif** en termes de prévision, d'anticipation, d'aide à la décision, etc. C'est pourquoi il est important de travailler avec des données susceptibles de contenir la valeur recherchée. La valeur est souvent la plus élevée lorsqu'elle est obtenue au plus près du moment de création des données.

Popularisés par le cabinet Gartner [105], les 3V initiaux étaient le volume, la vélocité et la variété, qui sont considérés comme des caractéristiques techniques. Les 5V font aujourd'hui office de référence, mais certains autres V font parfois surface pour tenter de caractériser plus finement les spécificités des données massives, tels que la **variabilité**, qui prend en compte le fait que les données peuvent évoluer, que ce soit au niveau de leur format ou de leur contenu, la **visualisation**, qui implique de correctement afficher les données et les résultats d'analyses en sélectionnant une représentation graphique adaptée pour obtenir une présentation synthétique et compréhensible, ou la **vertu**, qui fait écho aux problématiques de confidentialité.

De nombreux domaines peuvent tirer profit des données massives, tels que le marketing, la politique, la gestion de crise environnementale ou sanitaire, etc. On peut citer de nombreux travaux utilisant les données massives. Munshi et al. [129] s'intéressent à la publicité en ligne, plus précisément au système d'enchères auxquelles les annonceurs participent pour afficher leur publicité. Ceux-ci doivent être capables d'estimer si la publicité qu'ils souhaitent placer vaut l'investissement, et fournir une réponse en un temps minimal pour éviter que les performances du site hébergeur de la publicité ne soient impactées. Gaumont et al. [54] ont construit le politoscope lors des élections présidentielles françaises de 2017, afin d'étudier les discours politiques sur Twitter. Qadir et al. [147] recensent les différentes données et techniques d'analyse qui peuvent être utilisées lors d'une crise environnementale, comme un tremblement de terre ou un tsunami.

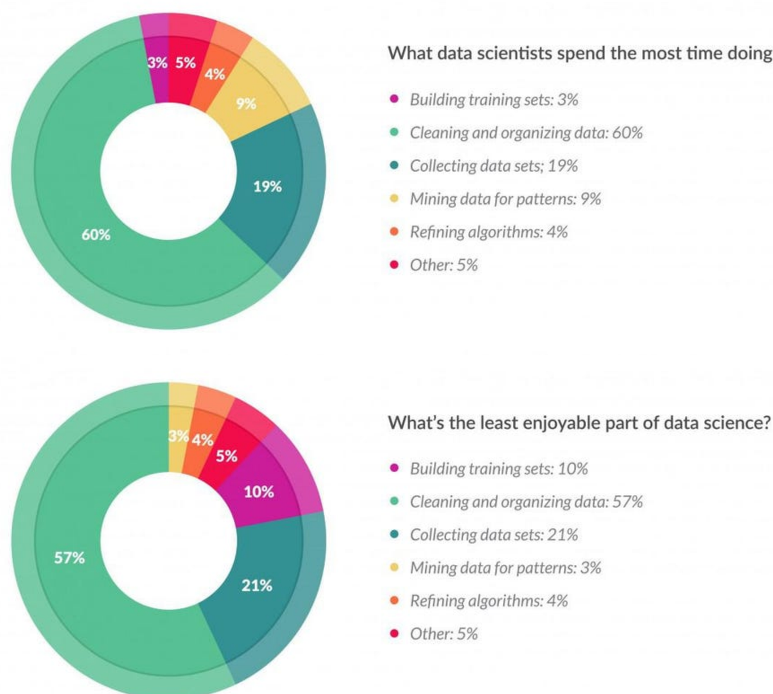


FIGURE 1.1 – Les tâches les plus chronophages et les moins appréciées des *data scientists* [2]

2. `https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says/?sh=689b42826f63`

Avec l'essor des données massives, de nombreux systèmes et techniques d'analyse pour les traiter et les exploiter ont vu le jour. Cependant, une part importante des analyses consiste à préparer et à nettoyer les données. Cette tâche est même citée comme la plus chronophage parmi celles des *data scientists*, tout en étant aussi la moins appréciée (figure 1.1). Le problème de nettoyage peut être abordé d'une manière différente : du fait du besoin d'extraire de la valeur des données, et sous l'impulsion des techniques de *machine learning*, le cœur de l'exploitation des données massives s'est en fait décalé des données vers les analyses, négligeant ainsi une vision globale qui va de la sélection des données à l'interprétation des résultats. À la suite de ce décalage, les outils développés visent donc en priorité les analyses, plutôt que la manipulation, la transformation ou la modélisation des données. Le format des données est imposé par les analyses, et entraîne parfois une diminution de l'expressivité du modèle de base. Par exemple, les algorithmes de détection de communautés travaillent principalement sur des graphes, et très peu laissent la possibilité de rajouter des informations pourtant essentielles, telles que la temporalité qui peut être associée à la création d'un lien. On retrouve ce problème également au niveau de la conception des architectures logicielles. Les caractéristiques évidentes des données massives, que sont le volume et la vélocité, font que les architectures doivent d'abord être capables de les supporter avant même de pouvoir prendre en considération les autres caractéristiques, qui souvent sont considérées comme annexes, et la responsabilité de leur prise en charge est alors reportée en fin de chaîne, lorsque les données servent aux analyses.

Afin de traiter les données massives, cette thèse se concentre sur deux problématiques fortement liées, tout en cherchant à conserver les données au centre des préoccupations pour avoir une vision globale des enjeux de l'analyse des données massives :

— les **architectures logicielles**, qui doivent être capables de supporter les caractéristiques des données massives, tout en garantissant certaines propriétés pour lesquelles il doit être possible de vérifier leur conservation lorsque les composants d'une architecture sont associés entre eux ;

— les **outils d'analyse**, qui doivent être capables de prendre en compte et d'exploiter la diversité des modèles de données, de détecter des erreurs et d'aider l'utilisateur à manipuler les données. Ils doivent aussi produire des résultats qui soient interprétables, et ce sans ambiguïté.

1.2 Les architectures dédiées aux données massives

Les architectures logicielles jouent un rôle essentiel dans la création des données, puisque ce sont elles qui régissent le cheminement des données, les informations qui sont ajoutées, extraites ou mises en correspondance, les propriétés appliquées lors des transformations de données (par exemple, si un traitement est exact ou approximatif), le stockage, etc. Les données produites par une architecture peuvent parfois être rendues publiques, en fournissant un jeu de données que d'autres organisations peuvent utiliser. Bien que peu souvent évoquées dans les processus d'analyses, les architectures ont pourtant une part de responsabilité élevée dans la qualité des données. Par exemple, dans le cas d'un site marchand, si certains champs des fiches produits peuvent être laissés vides même s'ils sont essentiels, cela se traduira par un travail supplémentaire lors de la phase de préparation des données.

Le décalage d'attention des données vers les analyses se retrouve aussi dans les architectures proposées ces dernières années, principalement en ce qui concerne le stockage des données. Les *data warehouses* rigides ont été remplacés par les *data lakes* stockant les données dans leur format brut, et souvent sans modification [84]. Le but des *data lakes* est de s'affranchir du schéma fixe des *data warehouses*, en remplaçant

le *schema on write* par un *schema on read*, afin de laisser le soin à l'analyste de créer lui-même un schéma correspondant à ses besoins. Ce gain de flexibilité est toutefois accompagné d'un coût : le *schema on write* permettait d'avoir un schéma adapté aux données, sur lequel tous les utilisateurs pouvaient s'appuyer, mais le *schema on read* force à répéter l'étape de création de schéma à chaque utilisation des données. L'apparente simplicité des *data lakes* ne fait que décaler les problématiques de création de schéma à l'utilisation plutôt qu'à l'ingestion des données dans le système.

Cette tendance architecturale d'osciller entre le *one size fits all* et *the right tool for the right job* n'est pas nouvelle [161]. Lin [110] argumente que ce cycle se répète sans fin, tel un pendule, qui pencherait vers *the right tool for the right job*, puis, lorsque la complexité de cette solution dépasserait le bénéfice de sa mise en place, entamerait son mouvement inverse vers le *one size fits all*, avant de changer à nouveau de sens lorsque la simplification de ce type de solution ne permet pas de satisfaire correctement les besoins liés à l'architecture.

La conception d'architectures entraîne inévitablement des compromis [151]. Les besoins menant à l'architecture servent à définir quelles propriétés sont à privilégier. Par exemple, si on souhaite augmenter la sécurité, des opérations de vérification et de cryptage des données seront réalisées, mais cela se fera au détriment des performances puisque chaque opération nécessitera un temps de traitement plus long. Les architectures dédiées au traitement des données massives n'échappent pas à cette règle, d'autant plus que les caractéristiques des 5V de ce type de données forcent à conserver des propriétés indispensables, telles qu'une performance suffisante pour supporter au moins le volume et la vélocité.

Les éléments exposés précédemment m'ont permis d'identifier deux problématiques principales qui semblent essentielles et qui concernent les architectures :

- être capable de connaître et surtout **déduire les propriétés supportées par une architecture**, tout en prenant en considération son cycle de vie, qui inclut des évolutions et parfois des fusions avec d'autres architectures. C'est un exercice difficile sans formalisation adaptée, puisque face à une architecture imposante il peut être compliqué de prendre en considération toutes les compositions que les interactions entre les composants représentent ;
- identifier un **patron d'architecture adapté aux données massives**, permettant à la fois d'extraire de la valeur en temps réel des données, tout en laissant suffisamment de flexibilité pour explorer les données et faire face à leur variété ainsi qu'à leur variabilité.

1.3 Les outils d'analyse

La variété et la variabilité des données massives font qu'elles sont souvent stockées sous forme semi- ou non-structurée. Il est alors nécessaire de réaliser une étape d'étude des données, afin de comprendre leur contenu et comment les manipuler et les utiliser. De nombreux outils sont disponibles pour faciliter cette tâche, souvent basés sur les *dataframes*, qui sont des abstractions sous forme de tables pouvant représenter de nombreux modèles d'origine variée, tels que CSV, JSON, relationnel ou graphe par exemple. On retrouve les *dataframes* dans divers outils tels que Pandas [125] ou MODIN [142]. Ces outils sont couramment utilisés pour réaliser des analyses exploratoires (appelées EDA pour *Exploratory Data Analysis*), pour développer des prototypes ou pour tester des algorithmes.

Une fois que les données sont suffisamment connues pour avoir une idée des analyses qu'il est possible de mener, la phase d'analyse en elle-même peut débuter. Pour être pérenne, la réalisation de cette phase

nécessite plus de sûreté que la phase d'exploration. En effet, il faut pouvoir identifier les attributs sur lesquels les analyses sont appliquées, ainsi qu'être capable d'interpréter les résultats. Une erreur de manipulation peut fausser les résultats obtenus et mener à une interprétation erronée. Cela a été le cas par exemple lors de l'inversion de deux colonnes d'un jeu de données[3] : les résultats obtenus avaient permis de produire plusieurs articles acceptés, et d'autres chercheurs avaient basé leur travail sur ces résultats, avant que l'erreur puisse enfin être identifiée plusieurs années plus tard en entraînant le retrait de ces articles.

La phase d'analyse a donc besoin d'outils plus sûrs que ceux utilisés lors de la phase d'exploration, afin de prévenir des erreurs d'exécution et d'interprétation. Or, c'est rarement le cas. La plupart des outils sont en fait basés sur ceux de la phase exploratoire et n'ont donc pas de propriétés fortes concernant la prévention des erreurs. Petersohn et al. [142] relatent par exemple qu'il existe plus de 200 opérateurs pour manipuler les *dataframes* dans Pandas et 70 dans R avec des redondances. Ils évoquent aussi de grandes différences de performance pour des opérateurs équivalents. La réutilisation des outils exploratoires peut s'expliquer par une volonté d'accélérer le développement des analyses, en profitant des découvertes fraîchement acquises lors de l'exploration pour pouvoir les utiliser directement dans les analyses. Si cela peut sembler bénéfique à court terme, cela l'est beaucoup moins à moyen et long termes lorsque les analyses ont besoin d'être modifiées ou réutilisées.

Il ne s'agit pas d'un problème nouveau, puisque le génie logiciel a déjà dû y faire face. En effet, le but étant d'obtenir un logiciel sûr produisant les résultats escomptés, de nombreuses méthodes, bonnes pratiques et outils augmentant la qualité et la facilité de maintenance du logiciel ont été développés. On peut ainsi retrouver les *design patterns* [29], fortement présents dans les langages orientés objets et permettant d'avoir des structures récurrentes, ou les conventions de nommage des éléments et les conventions de formatage de code, qui permettent de standardiser le code des logiciels et donc de faciliter sa compréhension, y compris par ceux qui n'en sont pas auteurs. Le typage statique est également important puisqu'il permet de garantir qu'une variable est bien du type attendu à tout instant, ce qui élimine de nombreux risques de dysfonctionnement [128].

Le *one size fits all* des architectures logicielles se retrouve également du côté des outils d'analyse avec le *in-database analysis*, poussé par certaines grandes entreprises telles qu'ORACLE ou SAP. Cette approche consiste à réaliser les analyses directement dans le SGBD. Toutefois, elle souffre d'un grand manque de flexibilité, tant du point de vue des familles d'outils d'analyse que du support du matériel. En effet, cette approche ne permet pas d'intégrer facilement les nouveaux algorithmes de *machine learning* ou autre, ni de tirer profit d'architectures matérielles spécifiques comme les GPU pour le calcul matriciel. C'est donc une approche qui peut être viable lorsque les données et les analyses à réaliser sont restreintes, mais qui s'applique difficilement dans le cas de données massives. La plus grande contrainte de cette approche réside cependant dans la limitation des modèles de données supportés, et qui sont liés au format initial du SGBD. Le modèle utilisé pour construire un outil d'analyse, qu'il soit directement intégré dans le SGBD ou non, est pourtant une caractéristique essentielle de ces outils. En effet, dans l'idéal ils doivent être suffisamment flexibles pour supporter les formats variés des sources de données dans le but d'éviter les transformations préalables des données (par exemple avec un *Extract Transform Load*, ou ETL), garantir une expressivité afin d'éviter la perte d'informations lorsque les données sont transformées de leur format d'origine vers ce modèle et permettre le développement d'algorithmes d'analyse diversifiés.

3. `https://people.ligo-wa.caltech.edu/~michael.landry/calibration/S5/getsignright.pdf`

On peut donc voir plusieurs problématiques émerger de la conception et de l'utilisation des outils d'analyse :

— prendre en compte la **sûreté et la qualité des *workflows* d'analyses**, afin de guider l'utilisateur et de réduire les erreurs potentielles qu'elles soient techniques ou fonctionnelles ;
— placer les **données au cœur des analyses**, dans le but de faciliter et d'avoir une plus grande confiance dans l'interprétation des résultats ;
— **considérer les nombreuses sources et formats de données existants**, afin de limiter l'utilisation de structures ou de modèles intermédiaires ad-hoc et de transformations de type ETL pour rendre compatibles les données avec les outils d'analyse.

1.4 Le projet Cocktail : un contexte applicatif

Le projet Cocktail [4] est un projet pluridisciplinaire qui a débuté en 2019. Il regroupe des chercheurs en science des données, en sciences de la communication, en sciences du langage et en sciences de l'alimentation [5]. Son but final est de créer un observatoire de Twitter [6], afin d'étudier les discours concernant les domaines de l'alimentaire et de la santé, et d'identifier les tendances, les signaux faibles et les singularités en temps-réel.

Ce projet rassemble une vingtaine de chercheurs. Il est financé principalement pour la partie recherche académique par ISITE-BFC [7]. Le projet bénéfice aussi d'un financement BPI France/FEDER incluant la métropole pour sa partie de transfert de technologie vers deux Entreprises des Services du Numérique (ESN) régionales : Atol CD et Webdrone. Le projet global est labelisé depuis 2018 par le pôle de compétitivité Vitagora [8] pour le développement de l'agro-alimentaire dans la région Bourgogne Franche-Comté.

L'objectif du projet est, à partir de données collectées de Twitter, d'offrir aux entreprises, aux distributeurs et institutions des secteurs agro-alimentaire et santé un outil permettant d'anticiper leurs prises de décision en matière de politique commerciale, industrielle et/ou de stratégie de communication. Pour les chercheurs en sciences sociales et de la communication, il s'agit d'un outil pour comprendre la forme des discours et leur diffusion dans le réseau social Twitter. Les chercheurs en sciences sociales ont défini l'anticipation comme le fil conducteur :

— anticiper pour désamorcer, identifier les risques d'impact des prises de paroles des communautés et des consommateurs sur une image, une marque, un produit, une activité, etc. ;
— anticiper pour innover, identifier les signaux faibles, les singularités qui font émerger des nouvelles pratiques alimentaires, des nouvelles tendances culturelles, des détournements de produits, des nouvelles communautés d'utilisateurs ou de consommateurs, etc.

4. https://projet-cocktail.fr/
5. Les laboratoires porteurs du projet sont : TIL (cenTre InterLangues, https://til.u-bourgogne.fr/), CIMEOS (Communications, Médiations, Organisations, Savoirs, https://cimeos.u-bourgogne.fr/), CSGA INRAE (Centre des Sciences du Goût et de l'Alimentation, https://www2.dijon.inrae.fr/csga/) et LIB (Laboratoire d'Informatique de Bourgogne, https://lib.u-bourgogne.fr/)
6. Ce projet fait suite à des collaborations antérieures établies depuis 2014 entre l'équipe sciences des données et des chercheurs en sciences sociales. La thèse de Ian Basaille a permis d'identifier les grandes fonctionnalités de l'observatoire à partir des plateformes existantes de l'état de l'art [21, p.109]
7. https://www.ubfc.fr/isite-bfc/
8. https://www.vitagora.com/

Le projet se divise en six *workpackages*, de l'identification des fonctionnalités à la production d'un prototype et de son industrialisation, en passant par la conception des scénarios d'analyse. Trois de ces *workpackages* ont un lien direct avec ma thèse : les spécifications fonctionnelles et techniques de l'observatoire, la conception des scénarios d'analyse et le prototypage de l'observatoire. J'ai pu participer à la rédaction des livrables concernant ces *workpackages*, et ai également coordonné la rédaction de l'un d'eux.

Ce projet a été un cadre applicatif idéal pour la thèse, et de ce fait les contributions qui en découlent ont servi au projet pour le faire avancer sur presque toutes ses facettes. Plusieurs cas d'étude ont été identifiés, et ils ont permis de révéler les lacunes des méthodes d'analyse standards liées aux problématiques relatées dans les sections précédentes. Le projet a donc servi à développer et tester des contributions permettant de combler ces lacunes, mais aussi de mener de bout en bout diverses analyses afin de vérifier leur applicabilité.

1.5 Contributions et plan de la thèse

Cette thèse apporte deux contributions majeures qui visent à répondre aux problématiques exposées dans les sections précédentes.

La première contribution concerne les architectures logicielles. Elle propose un **patron d'architecture** intégrant l'ingestion et le traitement en temps réel des données, ainsi que leur entreposage, dans le but de produire des indicateurs macroscopiques tout en offrant des fonctionnalités permettant de réaliser des analyses exploratoires. Une **formalisation basée sur la théorie des catégories** fait également partie de cette contribution, dont le but est de vérifier la conservation des propriétés dans une architecture logicielle intégrant de nombreux composants.

La deuxième contribution concerne les outils d'analyse. Elle propose un **modèle associant l'objet mathématique tenseur et un schéma de donnés**, muni d'opérateurs analytiques et de manipulation de données. Placer les données au cœur des analyses est une préoccupation majeure, qui implique de renforcer la sûreté lors de l'utilisation du modèle. Ce modèle peut être qualifié de modèle pivot, et se positionne en tant que surchouche aux *polystores*, afin d'exploiter la variété des modèles de données contenus dans ces derniers. Ces deux contributions ont des fondations théoriques bien établies (théorie des catégories et algèbre tensorielle), qui permettent d'assurer leur qualité pour intégrer ces composants dans une plateforme de traitement des données massives.

Une attention particulière a été portée aux éléments techniques mis en œuvre dans l'implémentation de ces contributions. Le patron d'architecture a servi de modèle pour l'architecture Hydre, qui est utilisée dans le projet Cocktail pour collecter, stocker et analyser les tweets, et qui utilise les technologies les plus adaptées pour chacun de ses composants (Akka, Kafka, Kafka Streams, Hadoop, Spark). Plusieurs cas d'étude ont pu être menés sur les corpus de tweets obtenus avec l'architecture Hydre. Le modèle tensoriel a été implanté en Scala, et est disponible sous forme de librairie. Le typage statique des expressions et l'inférence de schéma sont deux mécanismes reposant sur les *implicits* de Scala (ils permettent d'agir sur le comportement du compilateur), dans le but de faciliter l'utilisation des opérateurs du modèle tout en évitant les erreurs techniques et fonctionnelles.

Cette thèse est construite de la manière suivante. Le chapitre 2 est centré sur les architectures. Il décrit le patron Lambda+ Architecture, qui remet à jour la Lambda Architecture pour étendre ses cas d'utilisation et la rendre plus adaptée aux données massives, tout en se libérant de ses défauts principaux. Ce chapitre propose également une utilisation de la théorie des catégories pour étudier la conservation des propriétés

dans les compositions de composants d'une architecture. Le chapitre 3 s'intéresse aux outils d'analyse, et propose TDM (*Tensor Data Model*), un modèle de données s'appuyant sur les tenseurs. TDM leur ajoute un schéma de données, dont ils manquent cruellement, ainsi que des opérateurs de manipulation de données. Il propose aussi un typage fort, permettant d'intégrer une sûreté des expressions et une inférence du schéma lors de l'exécution d'opérateurs. Les opérateurs d'analyse, tels que les décompositions tensorielles, sont optimisés pour pouvoir être exécutés à large échelle en un temps raisonnable. Le chapitre 4 met l'accent sur la technique, notamment en ce qui concerne les implémentations en lien avec les chapitres 2 et 3 : l'architecture Hydre et TDM. Il met en avant l'importance des aspects techniques, et ce même au cœur de la recherche. Le chapitre 5 relate de la mise en place d'un observatoire d'analyse de données, en se servant du projet Cocktail comme exemple pratique. Les différentes méthodes utilisées, depuis la collecte jusqu'à l'analyse des données, constituent des applications concrètes des éléments détaillés dans les chapitres précédents. Enfin, le chapitre 6 conclut cette thèse, et présente des perspectives de travail se plaçant à la suite des contributions développées dans les chapitres précédents.

Chapitre

2

Lambda+ Architecture et théorie des catégories : les propriétés au cœur des architectures

*****

2.1 Introduction 12
2.1.1 Les styles et patrons d'architectures 12
2.1.2 Les architectures dédiées au traitement des données massives 13
2.1.3 Le besoin de formaliser les architectures 14
2.1.4 Contribution 14
2.2 État de l'art : la Lambda Architecture, l'avènement des systèmes de *stream processing* et la formalisation d'architectures 15
2.2.1 La Lambda Architecture 15
2.2.2 L'avènement des systèmes de *stream processing* 18
2.2.3 La formalisation d'architectures 21
2.2.4 Synthèse 23
2.3 La Lambda+ Architecture 23
2.3.1 La définition du patron 23
2.3.2 L'architecture Hydre : une implémentation du patron 27
2.4 Théorie des catégories pour la vérification des propriétés des architectures 29
2.4.1 Les concepts de base 29
2.4.2 Les concepts avancés 31
2.4.3 Les principes de la vérification 33
2.4.4 La démonstration de la vérification des propriétés d'une architecture 36
2.4.5 La vérification des propriétés de la Lambda Architecture 40
2.4.6 La vérification des propriétés de la Lambda+ Architecture 42
2.5 Conclusion et perspectives 45

La conception d'architectures logicielles est une tâche complexe qui doit prendre en considération plusieurs paramètres, tels que les fonctionnalités attendues, les propriétés essentielles, ou encore les technologies adaptées à ces fonctionnalités et propriétés. Obtenir toutes les propriétés idéales dans une architecture est bien souvent utopiste, puisque la conception d'architectures nécessite de faire des compromis pour conserver les propriétés les plus importantes dans un contexte donné, au détriment des propriétés dispensables. Les styles et patrons d'architectures sont utilisés pour guider cette conception en représentant des architectures standards ayant des caractéristiques bien définies.

Le but de ce chapitre est double, il vise à présenter :

— un nouveau patron d'architecture : la Lambda+ Architecture. Ce dernier s'inspire de la Lambda Architecture (voir section 2.2.1), et l'adapte pour lui fournir une flexibilité plus importante nécessaire au traitement des données massives ;
— une formalisation basée sur la théorie des catégories, qui permet d'étudier la conservation des propriétés dans les compositions de composants d'une architecture.

Le chapitre est construit de la manière suivante. La section 1 introduit les styles, les patrons, ainsi que les architectures dédiées au traitement des données massives et le besoin de formalisation des architectures. La section 2 constitue un état de l'art, présentant tout d'abord la Lambda Architecture ainsi qu'une synthèse des articles qui relatent de son utilisation, puis l'évolution des systèmes de *stream processing* auxquels la Lambda est fortement liée, et enfin des travaux relatifs à la formalisation d'architectures. La section 3 est consacrée à la description du patron que je propose, la Lambda+ Architecture, ainsi qu'à son implémentation dans le cadre du projet Cocktail. La section 4 détaille un cadre théorique de formalisation d'architectures basé sur la théorie des catégories, permettant d'étudier la conservation ou la perte des propriétés dans les architectures. La section 5 conclut ce chapitre et identifie des pistes pour enrichir le cadre théorique de la formalisation.

2.1 Introduction

La définition et le développement d'architectures logicielles nécessitent de connaître les propriétés que le système d'information doit supporter (par exemple la capacité de passage à l'échelle) et de définir les composants ainsi que leurs interactions [104]. Il est également nécessaire de garder une cohérence entre les spécifications initiales et les mises à jour ou évolutions afin d'éviter de transformer l'architecture en *Big Ball of Mud* [51], qui rend extrêmement complexe toute modification. Pour éviter cette situation, les styles et les patrons d'architecture [151] permettent de guider la définition d'une architecture. Les architectures dédiées au traitement des données massives ne font pas exception, et requièrent même une attention particulière pour les définir puisque les caractéristiques des données massives sont exacerbées. Cela soulève une nécessité de formaliser les architectures, afin d'avoir une définition la plus objective possible pour pouvoir ensuite implémenter l'architecture en suivant précisément les spécifications.

2.1.1 Les styles et patrons d'architectures

Les **styles** sont des spécifications à niveau de granularité élevé, guidant les interactions entre les composants [2]. Chaque style apporte certaines propriétés, tout en imposant des compromis sur d'autres propriétés. Il n'y a pas un style meilleur que les autres, mais un style particulier peut être mieux adapté qu'un autre à une situation donnée. Certains styles sont bien connus, comme par exemple l'architecture en couches (*layered*

architecture) [122] qui est composée d'une succession de couches, souvent d'une couche de persistance à une couche de présentation, et dans laquelle les communications se font entre couches voisines. C'est un style d'architecture simple avec un coût de mise en place faible, mais en contrepartie ses capacités d'évolution et de passage à l'échelle sont limitées. Un autre style d'architecture rendu populaire par l'essor du *cloud computing* est l'architecture micro-services [130]. Ce style isole chaque service (par exemple dans un conteneur ou une machine virtuelle), et lui fournit des ressources propres, comme des SGBD, inaccessibles directement pour les autres services. Les propriétés apportées par ce style sont opposées à celles de l'architecture en couche : les capacités d'évolution et de passage à l'échelle sont élevées, au prix d'une orchestration complexe. Un dernier exemple de style d'architecture est l'architecture orientée évènements (*event driven architecture*) [37]. Elle possède plusieurs composants découplés les uns des autres, qui réagissent lorsqu'ils reçoivent des évènements afin de les traiter. Au niveau des propriétés, cette architecture partage des similarités avec l'architecture micro-services, tout en offrant de meilleures performances et capacités de passage à l'échelle, mais elle est également difficile à tester du fait de la nature dynamique du flux d'évènements.

Les **patrons** sont la définition d'une abstraction spécifique d'un style d'architecture, visant à obtenir des propriétés essentielles [69]. Ils aident à identifier quel ensemble de composants et d'interactions pourrait convenir à un contexte défini, tout en laissant suffisamment de liberté pour adapter l'implémentation aux spécificités de la situation de création de l'architecture. Un patron peut être plus ou moins détaillé, et spécifier certaines contraintes à respecter lors de sa mise en œuvre. Par exemple, le patron du tableau noir [41] (*blackboard pattern*) propose de mettre les données à disposition des processus dans un répertoire commun, afin de pouvoir accéder aux données et pouvoir les utiliser pour des calculs, en mettant ensuite à disposition dans ce même répertoire les résultats obtenus. Ce patron suit un style d'architecture centrée sur les données (*data-centric architecture* [164]). Le *blackboard pattern* simule les interactions d'experts travaillant conjointement à la résolution d'un problème sur un tableau commun. Cela permet d'obtenir des propriétés d'adaptabilité et de réutilisation : les processus ne sont pas directement dépendants d'un autre processus, ils utilisent uniquement un répertoire commun de données. Cela permet de produire des systèmes découplés et robustes, puisqu'une panne d'un processus n'impactera pas les autres. Un autre patron est le Modèle-Vue-Contrôleur [45], dans lequel le contrôleur reçoit les demandes d'utilisateurs, accède aux données et les traite avec le modèle, puis génère le résultat grâce à la vue. En s'en tenant à cette description, ce patron suit le style de la *layered architecture*, mais il est à noter que certaines représentations incluent des variations comme des communications directes entre la vue et le modèle. Là aussi, les propriétés visées sont la réutilisation et le découplage : le modèle est responsable de l'accès aux données, et la vue l'interroge, ce qui permet de modifier la partie stockage en impactant uniquement le modèle. Plusieurs vues peuvent également se servir d'un même modèle. La Lambda Architecture [123] est aussi un patron d'architecture, bien connu dans le contexte du traitement de données massives, qui propose une forte tolérance aux pannes, des traitements en temps réel dans une partie de l'architecture, et les mêmes traitements pour générer un résultat correct sur le long terme dans une autre partie. La Lambda Architecture est détaillée dans la section 2.2.1.

2.1.2 Les architectures dédiées au traitement des données massives

Lorsque l'on souhaite concevoir des architectures dédiées au traitement des données massives, il est nécessaire qu'elles supportent des propriétés qui sont en relation avec les caractéristiques des données massives. Le volume et la vélocité sont des caractéristiques impactantes pour une architecture, puisque les différents composants doivent être capables de les prendre en charge, ce qui les amène à prendre en considération des propriétés telles que la capacité de passage à l'échelle ou l'élasticité si la charge n'est

pas constante au cours du temps. D'autres caractéristiques des données massives sont également à prendre en compte, comme leur variabilité, qui implique d'y répondre en dotant l'architecture d'une flexibilité lui permettant de s'adapter rapidement à des cas d'usages qui n'auraient pas été prévus initialement.

En plus de devoir prendre en considération les caractéristiques des données massives, de nombreuses situations requièrent d'obtenir des résultats en continu et en temps réel, comme cela peut être le cas pour la détection de fraudes bancaires ou pour réagir à des alertes remontées via des capteurs. Cela nécessite de pouvoir adapter les traitements et les algorithmes utilisés directement sur des flux de données, en se servant souvent des systèmes de *stream processing*, qui ont leur propre paradigme de programmation.

La plupart des articles de recherche relatant d'architectures dédiées au traitement des données massives présentent des architectures spécifiques, ancrées dans leur contexte d'utilisation, et avec un ensemble de technologies prédéfinies [170, 89, 129, 108, 75]. Cela mène à une forte *connascence* entre l'architecture, les outils et les propriétés. La *connascence* est un terme introduit par Page-Jones [136], elle mesure ou traduit la complexité induite par les relations de dépendance entre composants. Plusieurs types de *connascences* statiques ou dynamiques ont été identifiés pour évaluer les architectures. La *connascence* statique est un lien direct entre les composants au niveau du code (par exemple, le partage d'une classe commune), tandis que la *connascence* dynamique révèle un lien lors de l'exécution des composants (par exemple, l'échange de messages ou l'appel à un service). La *connascence* limite fortement la réutilisation des architectures, ce qui réduit leur apport.

2.1.3 Le besoin de formaliser les architectures

Les architectures logicielles sont des entités complexes. Elles ont une dynamique propre, qui intègre la conception, l'implémentation, mais aussi la maintenance et l'évolution de l'architecture. Les composants des architectures peuvent avoir un niveau de granularité plus ou moins fin, et peuvent intégrer eux-mêmes plusieurs sous-composants communiquant entre eux, dont l'assemblage peut être considéré comme une architecture. Dans ce cas, il est nécessaire de pouvoir représenter les différents niveaux de granularité de l'architecture, tout en gardant une cohérence globale.

Les besoins peuvent également évoluer au cours du temps. Du point de vue de l'architecture, cela signifie une évolution, une modification ou un ajout de composants, voire même une fusion avec une partie ou une autre architecture entière. Comme ces évolutions peuvent intervenir longtemps après la conception de l'architecture initiale, comme c'est le cas par exemple des systèmes bancaires, leur spécification peut être réalisée par une équipe différente de celle de départ. Les représentations schématiques proches de l'UML ne sont alors pas suffisantes pour garantir une interprétation commune et intemporelle. Il existe un besoin de formalisation important dans le monde des architectures [30, 80], et cette formalisation doit pouvoir retranscrire les effets d'une composition de composants, leurs impacts sur les propriétés et anticiper les conséquences d'une évolution.

2.1.4 Contribution

Dans ce chapitre, je propose un patron d'architecture, la Lambda+ Architecture, qui permet de tirer profit de la dualité entre les analyses exploratoires et les analyses en temps réel des données massives. Ce patron est une amélioration de la Lambda Architecture, qui est fortement dépendante de son contexte de création, et qui présente quelques lacunes qui peuvent désormais être corrigées. Je détaille une implémentation du patron Lambda+ Architecture à travers Hydre, une plateforme servant à collecter, stocker et analyser les

données issues de Twitter. Je propose également un cadre pour la formalisation d'architectures s'appuyant sur la théorie des catégories, permettant d'étudier l'évolution des propriétés dans des ensembles de composants, et de naviguer entre différents niveaux d'abstraction. La théorie des catégories place les transformations au centre de sa définition, avec les morphismes et les foncteurs, ainsi que les compositions. C'est une approche prometteuse pour répondre aux besoins identifiés précédemment et pour apporter une meilleure compréhension du système construit. Sa représentation sous forme de diagrammes est une aide visuelle pour comprendre la formalisation, et son application à la programmation fonctionnelle lui donne une proximité avec le domaine de l'ingénierie logicielle.

2.2 État de l'art : la Lambda Architecture, l'avènement des systèmes de *stream processing* et la formalisation d'architectures

La Lambda Architecture est un patron d'architecture logicielle bien connu pour traiter des données massives. Toutefois, c'est un patron qui présente certaines faiblesses, mises en évidence par les évolutions technologiques. Dans cette section, nous présentons la Lambda comme elle a été conçue, puis l'effet des évolutions technologiques, principalement des systèmes de *stream processing*, sur ce patron. La dernière partie est consacrée aux outils formels pensés pour la conception d'architectures, dans le but d'identifier une formalisation capable d'étudier la conservation des propriétés lors de la construction d'une architecture, notamment lors de la composition de composants.

2.2.1 La Lambda Architecture

Les propriétés d'exactitude des traitements, de faible latence et de tolérance aux pannes ont toujours été une préoccupation majeure lors de la spécification d'architectures. Lampson a esquissé quelques principes [104] pour obtenir ces propriétés, qui sont toujours d'actualité et que l'on peut retrouver, entre autres, dans la Lambda Architecture définie en 2011 par Nathan Marz [123, 124].

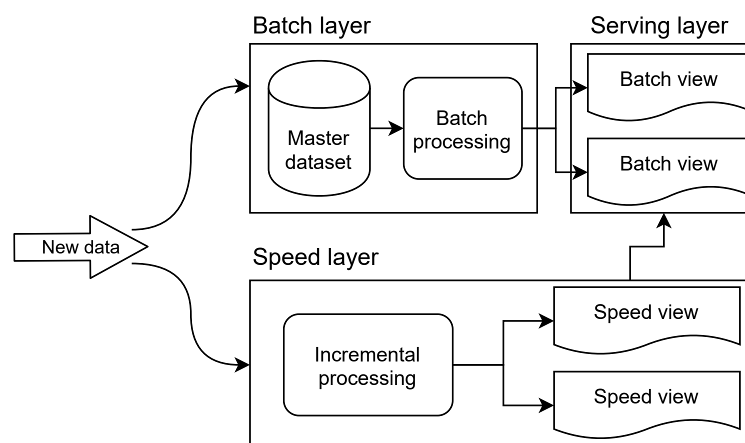


FIGURE 2.1 – Les composants de la Lambda Architecture

L'objectif de la Lambda est simple, et consiste à calculer des requêtes prédéfinies avec une faible latence, tout en assurant l'exactitude des traitements (figure 2.1), en s'appuyant sur trois couches (ou *layers*), la couche *speed*, la couche *batch* et la couche *serving*, telles que définies par Marz :

— **la couche *speed*** réalise les traitements de manière incrémentale afin de produire les **vues *speed***, en se servant des systèmes de *stream processing* pour assurer une **faible latence**. Toutefois, ces systèmes étant réputés peu fiables, la couche *speed* ne peut pas garantir l'exactitude des traitements ;

— **la couche *batch*** possède deux rôles :

1. stocker les données brutes dans le ***master dataset***, l'élément de l'architecture qui ajoute la propriété de **résistance aux pannes** ;
2. recalculer périodiquement les résultats obtenus avec la couche *speed* pour obtenir les **vues *batch***, mais cette fois à l'aide d'un traitement *batch* réalisé directement sur les données du *master dataset*. Cette duplication de traitements apporte la propriété d'**exactitude des résultats** qui manque à la couche *speed* ;

— **la couche *serving*** se charge d'agréger les vues *batch* et *speed* afin de fournir un résultat complet lorsqu'un utilisateur en fait la demande. Cette couche possède également un rôle d'indexation et de présentation des données, permettant de fournir les résultats de manière efficace.

Le but initial de Marz avec la Lambda était de réussir à battre le *CAP theorem* [27]. Le *CAP theorem* établit que seules deux des trois propriétés *consistency*, *availability* et *partition tolerance* peuvent être obtenues simultanément, bien que cela soit sujet à débat [160, 92]. Pour battre le *CAP theorem*, Marz part du principe que si les données étaient immuables et que les requêtes étaient calculées sur l'ensemble des données, alors le problème du *CAP theorem* disparaît entièrement : les données existent ou n'existent pas, mais ne peuvent pas être dans un état inconsistant. Toutefois, le lien de la Lambda avec le *CAP theorem* n'est pas aussi clair que Marz le laisse entendre [110]. Il est en pratique impossible de calculer les requêtes sur l'ensemble des données, il faut donc réaliser des traitements incrémentaux au quotidien, y compris dans la couche *batch*, et mettre à jour les résultats précédents. Cela n'entraîne donc aucune avancée par rapport au *CAP theorem*. La consistance du CAP est même encore plus difficile à obtenir que dans un système distribué classique, puisque les traitements dans la couche *speed* et dans la couche *batch*, qui sont basés sur différents paradigmes, sont développés pour produire des résultats identiques, ce qui ne peut pas être garanti ne serait-ce que face à l'erreur humaine. Le lien de la Lambda Architecture avec le *CAP theorem* s'est donc estompé, et n'est que rarement évoqué à présent.

Les avantages de la Lambda Architecture sont une forte tolérance aux pannes d'origine aussi bien humaine que machine, la garantie d'un résultat exact avec la couche *batch* et une faible latence avec la couche *speed*. La Lambda Architecture a servi de patron pour de nombreuses architectures dédiées au traitement des données massives, telles que RADStack [170], une plateforme *open source* produisant des analyses interactives, qui utilise les briques logicielles Apache Kafka pour l'échange de messages, Samza pour le *stream processing*, Hadoop pour des traitements *map reduce* créant des segments de données et Druid en tant que *serving layer* pour les insertions en temps réel et différé, et pour des analyses exploratoires. Les calculs en temps réel servent à guider les analyses exploratoires.

Munshi et al. [129] présentent une implémentation de la Lambda Architecture appliquée aux traitements des données issues des réseaux électriques. La résistance aux pannes, le temps de réponse rapide, le passage à l'échelle et la flexibilité de ce type d'architecture sont les paramètres qui ont motivé leur choix d'utiliser la Lambda Architecture. Hadoop HDFS est utilisé pour implanter un *data lake* et traiter de la diversité des

données (capteurs, images, vidéos). La couche d'interrogation, séparée de la couche d'analyse, intègre les technologies SparkSQL, Hive et Impala. Ces deux couches sont ajoutées afin de fournir plus de flexibilité aux requêtes des utilisateurs par rapport aux vues uniquement pré-calculées de la Lambda Architecture.

Lee et al. [108] spécifient une Lambda Architecture pour construire un système de recommandation de restaurants. Apache Mesos leur sert à abstraire les composants matériels de la plateforme, faciliter son déploiement et sa mise à l'échelle. Les technologies utilisées sont Kafka pour la couche *speed* et les traitements temps réel, Hadoop HDFS pour le stockage des données brutes, et Spark pour la couche *batch*. Cette implémentation s'éloigne du patron de la Lambda, puisqu'elle propose des traitements différents dans les couches *batch* et *speed* : la couche *batch* s'occupe d'exécuter des algorithmes lourds de *machine learning*, tandis que la couche *speed* se sert de ces résultats pour proposer les recommandations aux utilisateurs.

LinkedIn a développé le système Pinot [75]. Il est conçu pour traiter des requêtes OLAP avec une latence faible. Il utilise la Lambda Architecture comme patron, avec Apache Helix et Zookeeper, afin de fournir aux utilisateurs de LinkedIn des fonctionnalités analytiques en temps réel, comme par exemple la liste des personnes qui ont consulté leur profil. Pour obtenir de bonnes performances, seul un sous-ensemble d'opérateurs SQL est supporté, excluant les jointures et les requêtes imbriquées. Ils adaptent eux aussi le patron de la Lambda, puisque les vues *bacth* sont en réalité les anciennes vues *speed* qui sont calculées entièrement, c'est-à-dire lorsqu'elles représentent un jour ou une heure en fonction de la granularité choisie. Les nouvelles données sont ensuite traitées dans une nouvelle vue *speed*.

La Lambda Architecture a donc été la source d'inspiration de nombreuses implémentations, et ce dans des domaines variés, tels que les réseaux électriques, les systèmes de recommandation ou encore les réseaux sociaux.

Toutefois, la Lambda a aussi été beaucoup critiquée, en partie à cause de la duplication des traitements dans les couches *speed* et *batch*, basées sur des paradigmes différents, qui entraîne une complexité non négligeable [110]. Des *frameworks* tels que Summingbird [26] ont tenté de réduire cette complexité en unifiant les traitements *batch* et ceux réalisés en *stream processing* sous un modèle commun.

La Lambda manque aussi de flexibilité, et les cas d'utilisation alternatifs, tels que les analyses exploratoires bien souvent indispensables dans un contexte de données massives, nécessitent une modification du patron pour être intégrés, comme constaté à travers les implémentations précédemment citées. De plus, la couche *speed* seule, basée sur le *stream processing*, ne peut pas répondre à tous les besoins. En prenant l'exemple d'un algorithme pour résoudre le problème du sac à dos, s'approcher de la solution optimale nécessitera d'avoir accès à tous les éléments avant de prendre une décision : une implémentation via *stream processing* devra prendre une décision pour les éléments de manière individuelle, ou au mieux pour un groupe d'éléments sur une fenêtre de temps restreinte, et ne pourra donc pas être proche de la solution optimale [116].

La définition de la Lambda est aussi assez légère. Cela se ressent principalement pour la couche *serving* qui doit agréger les données : aucune précision n'est apportée pour le cas de données arrivant dans le désordre, alors que c'est un cas fréquent, par exemple avec des données provenant de capteurs qui peuvent passer hors-ligne. Cela pose pourtant des problématiques complexes, comme savoir si la couche *speed* doit traiter les données arrivant après que la vue *batch* correspondante ait été calculée. La manque de définition de la couche *serving* induit des différences d'interprétation. Certaines versions incluent les vues *speed*, tandis que d'autres versions laissent ces vues dans la couche *speed* et la couche *serving* est chargée d'aller les récupérer.

Ce manque de précision dans la définition s'étend au style de l'architecture. Les composants sont appelés "couches", alors que les architectures en couches sont un empilement de plusieurs composants, qui ne peuvent communiquer qu'avec les composants directement au-dessus ou en-dessous d'eux [151]. Cette nomenclature peut donc être considérée comme approximative. Le style d'architecture *event-driven* aurait été beaucoup plus adapté que le style *layered architecture*.

Bien que la résistance aux pannes soit une caractéristique avérée de la Lambda, le temps réel et l'exactitude des traitements sont deux caractéristiques qui sont plus à nuancer. En effet, la *batch layer* est garante des résultats exacts, et la *speed layer* est en charge du temps réel. Individuellement, ces couches respectent les caractéristiques qui leur sont affectées, cependant en regardant l'architecture dans son ensemble, la Lambda est capable soit de produire des résultats corrects, mais périodiquement, soit de calculer les résultats en temps réel, mais de manière approximative. Pour comprendre cette position, il faut s'intéresser à l'évolution des systèmes de *stream processing*.

2.2.2 L'avènement des systèmes de *stream processing*

Les systèmes de *stream processing* ont commencé à se développer à la même période que la conception de la Lambda Architecture. Leur paradigme est différent de celui des systèmes *batch*, et peut être vu comme le traitement d'une collection infinie de données. L'ouvrage d'Akidau et al. [8] détaille les différentes notions liées à ce paradigme, dont les principales sont résumées dans les paragraphes suivants.

La **latence** est un élément fondamental lorsque les données sont traitées en flux. Elle représente le temps nécessaire pour traiter un message de manière individuelle. Plus la valeur est faible, meilleure est la latence. Cette notion est à mettre en perspective avec celle de **débit** (*throughput*), qui, de son côté, représente le nombre de messages qui peut être traité en un temps donné. Plus la valeur est élevée, meilleur est le débit. Ce sont des notions complémentaires, qui s'améliorent souvent au détriment l'une de l'autre. Un système de *stream processing* basé sur le *micro-batch*, c'est-à-dire qui traite les messages en lots de taille réduite, aura une latence et un débit élevés.

La **temporalité des messages** est une autre caractéristique importante. On peut distinguer trois principales temporalités :

1. l'**instant** (*timestamp*) **de l'évènement**, qui indique le moment auquel l'évènement réel s'est produit ;
2. l'**instant d'émission du message**, qui indique le moment auquel le message correspondant à l'évènement a été émis ;
3. l'**instant de traitement du message**, qui indique le moment auquel le message a été traité.

Ces trois temporalités peuvent être perçues comme le cycle de vie d'un message dans un système de *stream processing*. Plus ces temporalités sont proches les unes des autres, plus le résultat obtenu sera proche de l'évènement initial. Cependant, il arrive que ces valeurs aient un décalage important, par exemple lorsque l'on traite des données issues de capteurs, si un capteur est déconnecté du réseau, il émettra les messages correspondant aux évènements qui se sont produits pendant cette période seulement lorsqu'il pourra se reconnecter. Cela induit une possibilité de traiter des messages qui arrivent tardivement et dans le désordre, ce qui peut parfois impacter le système s'ils ne sont pas pris en considération, par exemple pour émettre une alerte si un seuil est dépassé.

La **garantie de traitement des messages** est une caractéristique essentielle pour obtenir la propriété d'exactitude des traitements en se servant du paradigme de *stream processing*. Trois niveaux de garantie sont identifiés :

— ***at-most-once*** : les messages sont traités au plus une fois, et peuvent ne pas être traités ;
— ***at-least-once*** : les messages sont traités au moins une fois, et peuvent être traités plusieurs fois ;
— ***exactly-once*** (aussi appelé ***effectively-once***) : les messages sont traités au plus et au moins une fois.

Si la garantie *exactly-once* semble idéale puisqu'elle permet d'obtenir la propriété d'exactitude des traitements au sein même des systèmes de *stream processing*, elle reste toutefois utopique. En pratique, cet effet de ne traiter qu'une et une seule fois les messages n'est visible qu'en tant que résultat final du processus. Le système de *stream processing* en lui-même peut rencontrer des problèmes techniques qui nécessitent de traiter à nouveau un ou plusieurs messages, mais sa gestion interne rend ce mécanisme invisible sur le résultat. En prenant cette caractéristique en compte, le terme *effectively-once* est plus approprié que le terme *exactly-once* pour dénommer ce niveau de garantie. Toutefois, ce fonctionnement entraîne une contrainte forte lors de l'utilisation des systèmes de *stream processing* : **le traitement des messages ne doit pas avoir d'effet de bord qui ne soit pas idempotent**, c'est-à-dire que le résultat ne doit pas changer quelque soit le nombre d'exécutions de l'effet de bord.

Au fil des années, plusieurs systèmes de *stream processing* ont émergé, avec leurs caractéristiques propres. Une synthèse des principaux systèmes et de leurs caractéristiques est présentée dans les paragraphes suivants.

Le premier système ayant attiré l'intérêt de l'industrie est **Apache Storm**, créé par Marz en 2011 [165]. Il résulte de la constatation que, pour produire un système comportant un composant ayant la propriété temps réel, il fallait plus de temps pour créer les *workers* et les *queues* tout en s'assurant du bon fonctionnement de leur interactions que pour implémenter la logique métier. La priorité de Storm était de fournir un système ayant une faible latence, et ce au prix d'autres propriétés, comme celle de l'exactitude des traitements. En 2015, Twitter a fait évoluer Apache Storm en **Apache Heron** [103]. Leurs besoins ayant grandi au delà de la capacité d'Apache Storm, ils ont donc décidé de développer un système leur correspondant, en améliorant les performances, la capacité de passage à l'échelle ainsi que la facilité de gestion du système.

Apache Spark s'est dans un premier temps fait connaître en 2009 grâce à son système *batch*, qui, contrairement au *map-reduce* de Hadoop, permet de réaliser les traitements directement en mémoire plutôt que de devoir passer par le stockage sur disque, ce qui lui a valu un gain de performances important. En 2012, **Spark Streaming** [173] voit le jour, et utilise ce système de traitement *batch* éprouvé afin d'appréhender le problème des traitements en temps réel à l'aide de micro-*batch*. Contrairement à Storm, ce mécanisme permet à Spark Streaming de supporter la propriété d'exactitude des traitements. On peut toutefois reprocher aux premières versions de Spark Streaming de ne pouvoir supporter que des flux de messages ordonnés, en ne prenant pas en considération la différence entre l'instant de l'évènement et l'instant du traitement du message.

MillWheel [6], depuis sa création par Google en 2008, a suivi les besoins de ses utilisateurs pour guider son évolution. Son approche initiale était proche de celle de Storm, focalisée sur la faible latence au détriment de l'exactitude des traitements. Par la suite, MillWheel a intégré cette propriété d'exactitude, tout en prenant en considération les flux de données désordonnés. Une problématique importante en découle : celle de savoir à quel moment la majorité des messages pour une période donnée ont été traités, ce qui permet de connaître le moment pertinent pour émettre une alerte, par exemple lors d'une hausse ou d'une baisse de fréquentation. En 2015, Google s'appuie sur son expérience avec MillWheel pour rendre disponible **Cloud Dataflow** [7], un système de *stream processing* orienté *cloud* et proposant une interface unifiée pour le *batch* et le *stream processing*.

Kafka Streams [100] est apparu en 2016, et se différencie des autres systèmes de *stream processing* par un déploiement plus léger. Effectivement, Kafka Streams se sert de Kafka, un *middleware* orienté messages à hautes performances, pour gérer les problématiques de résistance aux pannes et de répartition des messages lorsque cela est nécessaire. Cette légèreté permet de déployer les applications en leur indiquant uniquement la localisation du serveur Kafka, la répartition de charge se fera via les *brokers* Kafka, et ce même si l'application est déployée sur plusieurs machines. Cela est également avantageux puisque les mécanismes supportant les garanties de traitement des messages sont alors partagés avec Kafka, ce qui permet d'avoir une meilleure maîtrise du système complet d'acheminement des messages. Toutefois, cela induit quelques limites en termes de performances, puisque la communication entre *workers* peut nécessiter l'écriture et la lecture de messages depuis un topic Kafka en cas de ré-ordonnancement des messages. Kafka Streams a été précédé par **Apache Samza** [93] en 2015, qui a un fonctionnement similaire tout en étant moins léger à déployer puisque dépendant du gestionnaire de ressources YARN.

Apache Flink [32], d'abord créé sous le nom de StratoSphere [11] en 2009, a connu un essor important à partir de 2015. En plus d'une expressivité importante, Flink est reconnu pour son efficacité, et reste à la pointe de l'innovation en termes de *stream processing*. Son mécanisme de *snapshot* lui a permis de mettre en place une robuste résistance aux pannes tout en assurant une exactitude des traitements. Ce mécanisme a d'ailleurs été repris par la suite par d'autres systèmes de *stream processing* tels que Storm. Flink continue son développement, avec notamment un principe de *savepoint* qui étend les *snapshots* afin de pouvoir reprendre le *workflow* à un point donné dans son exécution passée, et un principe de *streaming SQL*, que Kafka rejoint avec ksql, une surcouche de Kafka Streams permettant de l'utiliser avec des requêtes proches du SQL.

Apache Beam [22] est un peu à l'écart des systèmes de *stream processing*, puisqu'il cherche à se positionner comme une API, qui serait l'équivalent du SQL pour les SGBD. Pour cela, différents systèmes de *stream processing* peuvent être utilisés comme *runners*, liés à Beam via un SDK. Les utilisateurs peuvent se servir du DSL (*Domain Specific Language*) pour implémenter leurs *workflows*, sans être dépendants du système sous-jacent utilisé.

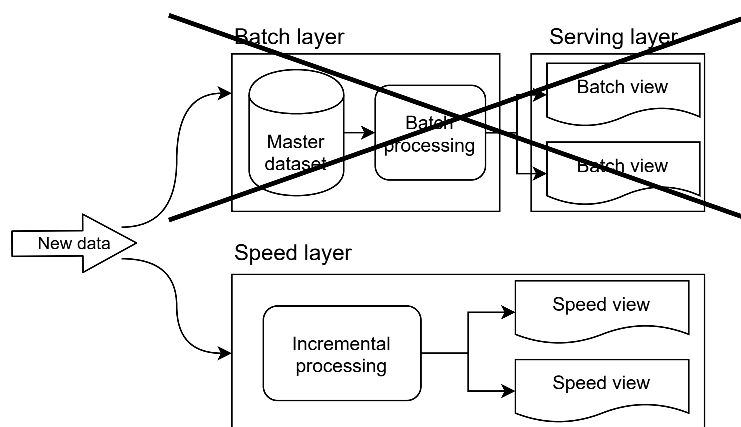


FIGURE 2.2 – Aperçu de la Kappa Architecture

Lorsque la Lambda Architecture a été conçue, peu de systèmes de *stream processing* étaient dispo-

nibles, et ils se concentraient souvent sur les performances au détriment de l'exactitude des traitements. Ils possédaient uniquement les garanties de traitement *at-most-once* et/ou *at-least-once*. Dans ce contexte, **la Lambda peut être vue comme un moyen de compenser les faiblesses d'une technologie naissante, plutôt que comme un patron d'architecture qui en tire réellement profit**. En partant de ce constat, Krepps propose de remplacer la Lambda Architecture par la Kappa Architecture [99]. Son principe est simple : en intégrant la garantie de traitement *effectively-once*, la *speed layer* est capable d'assurer à la fois la propriété de temps réel et la propriété d'exactitude des traitements. Il est donc possible de garder uniquement la *speed layer* pour remplir les fonctions de la Lambda (figure 2.2). Cependant, en supprimant la *batch layer*, le *master dataset* est également supprimé, et la propriété de résistance aux pannes avec lui, qui est pourtant un des avantages majeurs de la Lambda Architecture. La Kappa ne permet pas non plus d'intégrer des cas d'utilisations plus variés comparé à la Lambda Architecture, tels que les analyses exploratoires.

2.2.3 La formalisation d'architectures

Le besoin d'un cadre théorique et d'une formalisation adaptée pour concevoir des architectures logicielles a gagné en importance ces dernières années [30, 80]. L'élaboration, la spécification et l'implémentation des architectures sont des tâches complexes, qui nécessitent de pouvoir prouver la conservation des propriétés voulues tout au long des étapes de la création, puis de l'évolution de l'architecture. Établir un cadre formel dans ce domaine requiert à la fois des compétences théoriques et pratiques, afin de pouvoir proposer un modèle correspondant aux attentes tout en intégrant les imperfections du monde réel. Deux articles récents [82, 157] étudient différents systèmes pour les comparer. Ils montrent que les systèmes dédiés à l'analytique ne sont pas suffisamment définis formellement pour pouvoir clairement prouver qu'ils supportent les propriétés qu'ils mettent en avant. Cette constatation est appuyée par la multitude de technologies disponibles, supportant chacune des propriétés différentes, mais étant composées au sein d'une même architecture. Il devient difficile de prédire ou d'établir que le système construit supportera ces propriétés globalement. C'est le cas par exemple pour les technologies de stockage de données ou de traitement de flux.

Plusieurs approches ont cherché à formaliser les architectures, en prenant des bases différentes. Les *Architecture Description Languages* (ADL) [38] ont une place importante dans la formalisation d'architecture. Les ADL *boxes and lines* ainsi que les ADL basés sur UML [31] ne permettent pas de vérifier facilement les propriétés, à cause de leur faible niveau de formalisation. De ce fait ils ont recours à des outils annexes, tels que les automates à états finis pour simuler des diagrammes d'état transition, et doivent par conséquent établir des passerelles plus ou moins simples vers ces outils [68]. Nous nous concentrons sur les ADL ayant des bases théoriques bien établies.

Abowd et al. [2] proposent un *framework* formel en Z permettant de décrire les styles d'architecture. Ils argumentent que les diagrammes ne sont pas suffisants pour imposer une unique interprétation d'une architecture, et qu'ils peuvent donc mener à des incompréhensions. Leur approche consiste à définir une sémantique plus précise pour définir les styles, afin de pouvoir affecter un style à une implémentation sans que l'interprétation de l'architecture ne soit porteuse d'ambiguïté. Une telle sémantique permet d'analyser techniquement les architectures, mais aussi de comparer les styles entre eux.

Malkis et Marmsoler [121, 122] travaillent également sur la formalisation des styles d'architectures. Ils s'appuient sur la théorie des ensembles et la logique du premier ordre pour construire un modèle avec des ports et des services, utilisés pour représenter les interactions entre les composants. Ils ont appliqué leur modèle à deux styles d'architecture : l'architecture en couches et l'architecture orientée services. À partir

de leur formalisation, ils analysent mathématiquement l'architecture afin de prouver le support de certaines propriétés, telles que l'indépendance sémantique des couches de haut niveau par rapport aux couches de plus bas niveaux en ce qui concerne le style d'architecture en couches.

Le Métayer [107] propose un modèle se basant sur la théorie des graphes. Les nœuds sont les entités des architectures (en fonction du niveau d'abstraction, un client, un serveur, ou un objet), et les liens sont les communications entre ces entités. Les styles sont définis en tant que grammaires de graphes, ce qui permet ensuite de vérifier qu'une implémentation d'architecture correspond à une grammaire donnée pour savoir si elle respecte le style associé. Toutefois, les capacités de modélisation sont limitées car les liens ne représentent que des communications, et il n'y a pas de mécanisme permettant d'intégrer les propriétés supportées par l'architecture.

Mabrok [117] a proposé une approche différente, et se sert de la théorie des catégories pour formaliser les propriétés et les attributs des architectures. Il utilise Ologs, une spécialisation particulière de la théorie des catégories dont le but est de représenter l'étude ontologique d'un sujet. C'est une approche intéressante, mais qui effleure seulement la conception d'une architecture, puisqu'il est uniquement possible de représenter les propriétés fonctionnelles et non la structure technique de l'architecture. Il ne tire pas profit de toute la puissance de la théorie des catégories, qui propose pourtant des mécanismes formels pertinents.

Les ADL existants se basent sur la théorie des ensembles, des graphes, et utilisent la logique du premier ordre ou d'ordre supérieur pour vérifier les propriétés et la consistance des architectures. Cependant, les architectures ont différents niveaux d'abstraction, et les interactions entre les composants ont autant d'importance que les composants eux-mêmes. Ces interactions forment des compositions de composants, représentant alors un cheminement de données à travers l'architecture. Les compositions peuvent être difficiles à représenter si la base formelle utilisée ne propose pas de mécanisme permettant de les prendre en compte.

La théorie des catégories [48] est une approche prometteuse, puisqu'elle place les notions de morphismes et de compositions parmi les concepts essentiels de ses fondations. Elle permet également d'utiliser les foncteurs, pour passer d'une catégorie à une autre, simplifiant ainsi le passage d'un niveau d'abstraction à un autre. La théorie des catégories a été utilisée dans de nombreux domaines aussi bien théoriques qu'appliqués.

Parmi les domaines proches de notre thématique, Diskin et al. [46] l'utilisent dans le domaine de l'ingénierie guidée par les modèles, pour représenter les modèles et les méta-modèles par des catégories. Ils se servent des catégories de Kleisli et de la notion de monade pour établir des relations entre modèles et définir les bases d'un langage de requête. Dans [163], les auteurs proposent une démarche similaire pour la gestion des lignes de produits logiciels (*Software Line Product*), ils s'intéressent plus particulièrement aux aspects dynamiques et au contrôle des systèmes. Dans [166], les auteurs proposent une formalisation d'un modèle de données s'appuyant sur XML et des opérateurs algébriques. Dans le domaine des bases de données elle est utilisée pour représenter des schémas, lier des schémas dans les bases de données distribuées et guider l'intégration des données. Kokar et al. [94] l'ont utilisée pour définir un cadre formel à la fusion de données. Dans le même ordre d'idées, Majkic [120] met en avant que la construction d'outils robustes pour traiter les données massives et leurs spécificités n'est possible que dans un cadre théorique et algébrique clair (similaire à ce qui a été réalisé pour le modèle relationnel). Pour exprimer des correspondances complexes entre des données issues de plusieurs sources, il propose une algébrisation d'un sous-ensemble de la logique de second ordre au moyen de la théorie des catégories.

2.2.4 Synthèse

Les besoins liés au traitement des données massives nécessitent de construire des architectures pouvant passer à l'échelle, supporter des calculs d'indicateurs en temps réel et offrir suffisamment de flexibilité pour réaliser des analyses exploratoires. La Lambda Architecture a cherché à proposer une solution, mais le patron présente de nombreuses déficiences comme en témoignent les adaptations que les projets de l'état de l'art s'y référant ont dû mettre en place. Le gain de maturité des systèmes de *stream processing* a diminué l'utilité de la Lambda, qui peut être désormais remplacée par la Kappa Architecture, plus simple à mettre en place. Cependant cette dernière n'apporte pas un progrès significatif puisqu'elle n'étend pas les cas d'utilisation limités de la Lambda et n'incorpore donc pas les analyses exploratoires essentielles.

De plus, lors de la conception d'architectures il est essentiel de pouvoir connaître les propriétés supportées lors des traitements réalisés, et ce quelle que soit la composition de composants responsable d'un traitement donné. Toutefois, les outils de formalisation actuels d'architectures ne sont pas satisfaisants : d'un côté les méthodes proches de l'UML sont trop subjectives pour être utilisées sans ambiguïté, de l'autre les méthodes ayant des bases formelles plus solides ne peuvent bien souvent pas représenter toutes les subtilités d'une architecture ni mettre l'accent sur les compositions induites par les interactions entre les composants, et les évolutions et fusions des architectures.

2.3 La Lambda+ Architecture

La forte imbrication de la Lambda Architecture dans son contexte de création la mène à son obsolescence. Comme l'a montré la Kappa Architecture, la duplication des processus dans les couches *batch* et *speed* n'est plus nécessaire pour obtenir des résultats corrects en temps réel, puisque les systèmes de *stream processing* supportent directement la garantie de traitement *effectively-once*. De ce fait la complexité induite par la Lambda n'est plus justifiée. Les idées de la Lambda Architecture peuvent donc être réutilisées et modifiées pour fournir un patron plus flexible et adapté au traitement des données massives.

La Lambda+ Architecture est pensée pour s'inscrire dans un contexte de traitement des données massives, en proposant deux fonctionnalités principales complémentaires :

- le stockage des données orienté vers une utilisation prévue pour des analyses exploratoires ;
- le calcul en temps réel d'indicateurs prédéfinis directement sur le flux de données, afin d'obtenir des informations sur des besoins bien identifiés.

La dualité entre les analyses exploratoires et les calculs en temps réel sur le flux de données doit être prise en compte pour traiter les données massives. En effet, la combinaison du volume et de la variété rend difficile l'extraction de la valeur cachée dans les données. L'extraction de cette valeur en temps réel ne peut se faire que lorsqu'un cas d'utilisation ainsi qu'une méthode d'extraction ont été trouvés et validés, ce qui n'est possible qu'à travers des analyses exploratoires.

2.3.1 La définition du patron

Nous proposons de définir la Lambda+ Architecture. Tout d'abord, comme indiqué dans la section 2.2.1, le style en couches adopté par la nomenclature des différentes parties de la Lambda Architecture ne correspond pas au style de l'architecture. La Lambda+ n'est donc pas constituée de couches, mais de composants interagissant de manière asynchrone à l'aide d'un *middleware* orienté messages. La Lambda+ applique le style *event-driven architecture*, qui met en avant les propriétés de performance, de passage à l'échelle et de

facilité d'évolution. Ce style privilégie les communications asynchrones entre les composants, en organisant les flux de messages à l'aide de *queues* et de topics disponibles avec les *middlewares* orientés messages. Chaque composant traite les messages, et envoie éventuellement son résultat dans une autre *queue* afin qu'il soit traité de manière incrémentale par d'autres composants. Les compromis de ce style d'architecture sont une certaine complexité à la mettre en œuvre, ainsi qu'une difficulté à tester l'ensemble de l'architecture, à cause de la nature dynamique du système d'échange de messages entre les composants. La figure 2.3 représente les cinq composants de la Lambda+ Architecture.

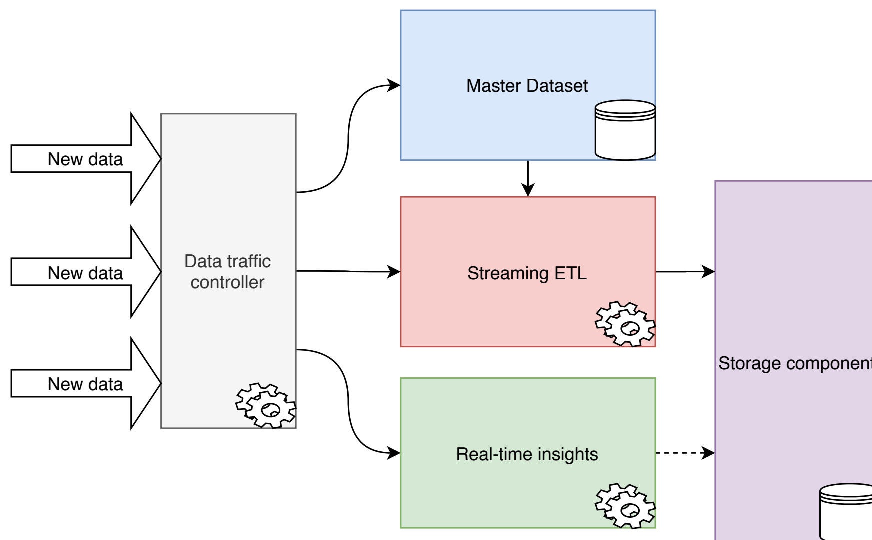


FIGURE 2.3 – Aperçu de la Lambda+ Architecture

2.3.1.1 Le composant *data traffic controller*

Les nouvelles données sont réceptionnées par le *data traffic controller*. Les sources de données peuvent être de natures variées, comme une extraction depuis un SGBD existant ou une connexion à une API externe. Cette diversité nécessite une structuration du flux de données.

Le composant *data traffic controller* se charge d'organiser les flux de données, et choisit si certains flux doivent être regroupés ou au contraire être considérés de manière individuelle. Il peut également réaliser des transformations simples sur les flux, comme une suppression de doublons ou l'ajout d'un *timestamp* dans les données.

Le style d'architecture *filter and pipe* correspond à la philosophie du composant. Des transformations simples sont réalisées, avant d'envoyer les données dans un système de communication permettant aux autres composants d'y avoir accès. Un *middleware* orienté messages est la solution la plus évidente dans ce genre de situation, grâce à ses capacités de découplement entre les applications productrices et consommatrices de messages. Cela permet d'obtenir une grande indépendance entre les composants, ce qui contribue à supporter la propriété de résistance aux pannes.

2.3.1.2 Le composant *streaming ETL*

Grâce à l'évolution des systèmes de *stream processing*, les traitements peuvent produire un résultat exact directement en temps réel. De ce fait, comme il n'est plus nécessaire de dupliquer les traitements en *streaming* et en *batch*, la couche *batch* peut être remplacée par un composant plus adapté à un contexte Big Data : le composant de *streaming ETL*.

Les ETL (*Extract-Transform-Load*) sont ancrés dans le paysage analytique, et utilisés principalement en combinaison avec un *data warehouse*. Toutefois, le besoin de travailler sur des données plus fraîches s'est grandement développé ces dernières années, et le caractère périodique des ETL n'est pas adapté à cette évolution. Plusieurs travaux de recherche se concentrent sur l'exécution d'ETL en temps réel, afin de rendre possible les analyses à faible latence [169, 49, 127].

Dans ce contexte, le *streaming* ETL est utilisé afin de traiter les données en continu, contrairement au comportement historique en *batch* des ETL. Les systèmes de *stream processing* démontrent leur utilité dans ce type d'utilisation. Avec le *streaming* ETL, il est possible de réaliser des analyses nécessitant des données récentes afin de maximiser la pertinence de la valeur extraite.

Le rôle du composant *streaming ETL* est d'alimenter le composant *storage*, en transformant les données au besoin. Il fonctionne en deux temps :

- en temps réel lorsqu'il réceptionne les données du *data traffic controller* ;
- en temps différé lors d'une extraction des données du *master dataset*, ce qui peut se produire par exemple lors d'un changement de schéma dans le composant *storage*.

En fonction du volume et de la vélocité des données traitées, des contraintes supplémentaires peuvent apparaître. Par exemple, la plupart des SGBD montrent de meilleures performances à l'insertion lorsque cela se fait par lots. Même si la propriété de temps réel du composant *streaming ETL* s'oppose aux traitements *batch*, le composant *storage* est principalement dédié aux analyses exploratoires, ce qui permet d'augmenter le débit au détriment de la latence sans impacter négativement l'architecture, la partie critique en temps réel se situant dans le composant *real-time insigths*.

2.3.1.3 Le composant *master dataset*

Son utilité est la même que dans la Lambda Architecture. Il conserve l'ensemble des données brutes, ce qui permet de les traiter à nouveau si le composant de *streaming ETL* subit une évolution impactante, ou si une panne nécessite de traiter une nouvelle fois les données.

Pour éviter la complexité bien souvent reprochée à la Lambda Architecture induite par la duplication des traitements dans les couches *batch* et *speed*, si les données doivent être traitées à nouveau à partir du *master dataset*, elles sont simplement extraites afin d'être envoyées au composant *streaming ETL*. Ainsi, les évolutions et la maintenance ne s'appliquent que sur un seul composant, et l'inconvénient principal de la Lambda Architecture est contourné.

Le *master dataset* est essentiel pour permettre à l'architecture de supporter la propriété de résistance aux pannes. Si un problème survient, il permet de reprendre les données d'origine pour le corriger. Il est donc important que le *master dataset* puisse absorber le flux de données entrantes, afin d'être constamment à jour.

2.3.1.4 Le composant *real-time insights*

C'est le composant de l'architecture dédié aux calculs en temps réel, que ce soit pour des requêtes prédéfinies ou pour l'exécution d'algorithmes. La faible latence ainsi que l'exactitude des résultats sont la priorité de ce composant, mais les avancées des systèmes de *stream processing* permettent de supporter ces propriétés. Les calculs réalisés peuvent être simples, comme des agrégations consistant à calculer une somme, ou plus complexes, comme de la détection d'anomalies sur des séries temporelles [4].

Les traitements effectués dans ce composant doivent être identifiés et définis en amont. Ils servent à extraire des indicateurs pertinents concernant les données récupérées. Pour découvrir les traitements permettant d'extraire de la valeur, le composant *storage* est utilisé pour d'abord fouiller dans les données de manière exploratoire. La connaissance acquise peut ensuite être utilisée pour mettre au point les traitements du composant *real-time insights*.

Les résultats des traitements effectués peuvent être conservés dans le composant *storage*, mais comme précisé dans la section 2.2.2, cela doit être fait de manière idempotente : le traitement en doublon d'un message dans le cadre de la garantie de traitement *effectively-once* doit toujours produire le même résultat qu'un traitement unique.

2.3.1.5 Le composant *storage*

Le composant *storage* est alimenté par le composant *streaming ETL*, et éventuellement par le composant *real-time insights*. Son rôle est de mettre les données à disposition des utilisateurs, dans un modèle favorisant les analyses exploratoires. L'implémentation de ce composant peut se faire différemment en fonction des besoins conduisant à l'architecture, par exemple avec un *data warehouse*, un *polystore*, ou un *data lake*.

Les *data warehouses* sont apparus bien avant l'avènement des données massives, et sont une technologie mature et largement adoptée par les entreprises [76]. Ils sont souvent utilisés pour agréger les données provenant de différentes sources disponibles dans une organisation, en les extrayant à l'aide d'ETL, puis en les nettoyant et en les mettant en forme, et enfin en réalisant des analyses décisionnelles, plus difficiles à exécuter sur un système transactionnel. Dans un *data warehouse*, les données possèdent plusieurs caractéristiques :

- elles concernent un sujet particulier, afin de faciliter les requêtes d'informatique décisionnelle ;
- elles sont intégrées pour éviter les inconsistances induites lors de l'utilisation de sources multiples ;
- elles possèdent une dimension temporelle, afin d'exécuter des requêtes pour des périodes différentes ;
- elle sont non-volatiles, elles ne peuvent donc pas être mises à jour ou supprimées.

Les *data warehouses* sont utilisés pour modéliser les données concernant un sujet particulier en utilisant un schéma fixé. Cela permet aux analystes de se référer à ce schéma pour extraire de la valeur de la masse de données, mais cela a aussi l'inconvénient d'induire un manque de flexibilité. Face à l'augmentation du volume et de la variété des données, les *data warehouses* peuvent se montrer trop restrictifs lorsque les entreprises veulent bénéficier pleinement de l'ensemble des données qu'elles accumulent [43].

Les *polystores* sont des systèmes intégrant divers SGBD, systèmes de stockage, ainsi que plusieurs langages de manipulation de données ou des langages de programmation se basant sur différents paradigmes [52]. Ce sont des systèmes permettant à la fois de profiter des optimisations de cas d'utilisation particuliers (comme la recherche de chemins dans un SGBD orienté graphes), et de profiter de la parallélisation de requêtes dans différents systèmes de stockage, en tirant partie de la spécificité de chacun [96, 12].

Les *data lakes* sont moins bien définis que les *data warehouses* ou que les *polystores* [47]. Ils sont

souvent utilisés lorsque des données sont collectées mais que leur cas d'utilisation n'est pas clairement identifié. Les *data lakes* apportent une solution au manque de flexibilité des *data warehouses*. En effet, les données sont rassemblées sans schéma commun, souvent dans une forme semi ou non-structurée. Toutefois, cette flexibilité s'accompagne d'un coût, et on retrouve une grande hétérogénéité dans les données d'un *data lake*, à la fois dans leur source et leur format, mais aussi dans leur contenu et dans leur fiabilité. Si cette caractéristique n'est pas correctement prise en compte dans l'organisation du *data lake*, ce dernier peut se transformer en *data swamp*, enlisant alors les analyses par manque de visibilité dans les jeux de données. Les travaux concernant les *data lakes* se concentrent donc sur la recherche de mécanismes permettant d'éviter la formation d'un *data swamp*, et sur l'aide à la localisation de données pouvant servir à une analyse. Les approches les plus populaires consistent à intégrer des métadonnées dans le *data lake* [155, 154], ou à partitionner le *data lake* en fonction de divers critères (comme regrouper les mêmes formats, les mêmes sujets, etc.) en *data ponds* [77]. L'organisation en *data ponds* a quelques points communs avec les *polystores*, puisque les *data ponds* regroupent les données en fonction de leurs caractéristiques, ce qui peut inclure des caractéristiques techniques comme le format des données, qui requiert l'utilisation de plusieurs SGBD différents.

2.3.2 L'architecture Hydre : une implémentation du patron

J'ai mis en œuvre le patron Lambda+ pour développer une architecture utilisée dans le projet Cocktail, présenté dans la section 1.4. Les détails techniques de l'architecture sont approfondis dans le chapitre 4.

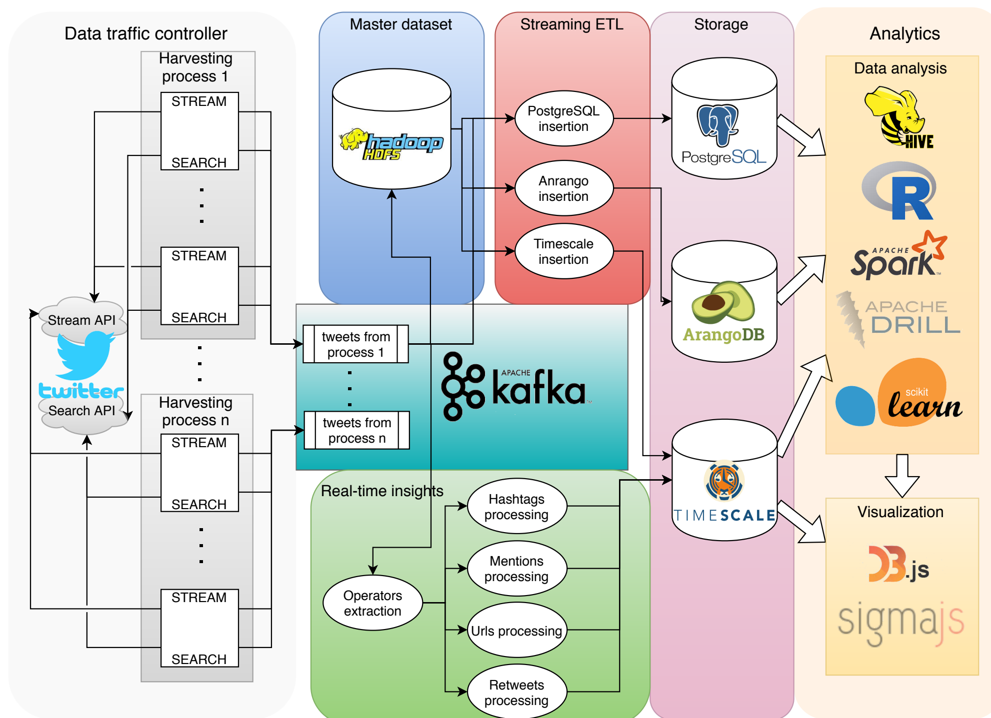


FIGURE 2.4 – L'architecture Hydre

Afin de collecter, stocker et analyser les données issues de Twitter, l'architecture Hydre a été spécifiée et développée (figure 2.4). Elle est en production depuis avril 2019, et permet de calculer des indicateurs en temps réel sur les collectes, ainsi que de préparer les données pour des analyses exploratoires. Au niveau matériel, elle utilise un *cluster* Hadoop de 20 nœuds, un *cluster* Kafka de 5 nœuds et 4 autres serveurs servant à héberger un *polystore* et exécuter les analyses exploratoires, avec des *notebooks* Jupyter ou des applications en Spark/Scala. Les composants principaux de la Lambda+ Architecture sont implémentés de la manière suivante.

Le ***data traffic controller*** regroupe plusieurs machines virtuelles de collecte, utilisant les API *Stream* (en temps réel) et *Search* (sur un historique de sept jours) de Twitter, en filtrant les tweets collectés à l'aide de différents critères comme des comptes ou des hashtags. Les tweets sont dans un format JSON, et les machines de collecte ne rajoutent que peu d'informations aux tweets bruts, par exemple le *timestamp* du moment de la collecte. Ces machines de collecte suivent le modèle d'acteurs en étant implémentées avec Akka, et envoient les tweets dans des topics Kafka. Kafka est le composant central de l'architecture, puisqu'il permet une communication faiblement couplée entre les composants.

Le ***master dataset*** est constitué d'un entrepôt Hadoop HDFS. Les tweets bruts sont stockés dans des fichiers, un tweet étant écrit sur une ligne dans son format JSON. Ces fichiers peuvent ensuite être lus pour envoyer les tweets dans un topic Kafka, afin de traiter à nouveau les données anciennes. Les données brutes représentent actuellement environ 13To.

Le composant de ***streaming ETL*** insère les données issues des topics Kafka dans le composant ***storage*** en *micro-batchs*. Les données d'entrée sont envoyées par le *data traffic controller* ou par le *master dataset*. Le composant *storage* étant implémenté avec un *polystore*, des transformations sont réalisées sur les données afin de les faire correspondre aux schémas des différents SGBD utilisés : un SGBD relationnel (PostgreSQL), un SGBD graphe (ArangoDB) et un SGBD séries temporelles (TimescaleDB). Ces bases de données peuvent ensuite être utilisées pour réaliser des analyses exploratoires.

En parallèle du remplissage du *polystore*, le composant ***real-time insights*** sert à extraire et agréger des indicateurs sur les collectes de tweets, comme par exemple les hashtags ou utilisateurs populaires. Les résultats sont stockés dans TimescaleDB, afin de pré-calculer les requêtes permettant de mettre à disposition directement le résultat de séries temporelles construites à partir de tous les éléments des tweets (hashtags, comptes, etc.), représentant ainsi plusieurs milliers de séries temporelles mises à jour en temps réel. Bien que cette insertion soit un effet de bord du traitement sur le flux, celui-ci est idempotent. En effet, seul le résultat calculé par le système de *stream processing* est enregistré et remplace le précédent s'il en existe déjà un, ce qui fait qu'il est en accord avec l'état du système de *stream processing*.

Un *benchmark* de l'architecture avait été réalisé lors des débuts de sa conception [57], qui confirmait sa capacité à absorber le flux moyen de Twitter s'élevant à 6 000 tweets/s. Ces expériences ne sont pas relatées dans cette thèse, puisque l'architecture a depuis été déplacée sur un ensemble de serveurs plus adaptés à son déploiement, et l'expertise acquise a également permis de produire des composants plus performants, ce qui rend le résultat des expériences obsolète et en-deçà de la capacité acutelle de l'architecture Hydre.

Les données des collectes sont mises à disposition des chercheurs et des analystes du projets Cocktail à travers des interfaces exposées par un serveur d'application, ainsi que par des *notebooks* Jupyter pré-configurés. Des analyses exploratoires plus complexes sont réalisées lors de l'identification de questions de recherche, qui nécessitent de créer un corpus de tweets centré autour du sujet ciblé, puis d'appliquer différents algorithmes (communautés, centralités, etc.) afin d'éclairer plusieurs aspects des données. Les résultats obtenus sont ensuite interprétés en collaboration avec les chercheurs en sciences humaines et

sociales, et apportent des éléments de réponse aux questions initiales. Ces éléments sont expliqués plus en détail dans le chapitre 5.

2.4 Théorie des catégories pour la vérification des propriétés des architectures

Lors de la conception d'architectures logicielles, il est fréquent de manipuler plusieurs architectures plus petites, mais ayant chacune ses propres propriétés. Considérons deux exemples d'illustration de ce phénomène. Le premier, lorsque les architectures logicielles évoluent et se développent. Elles peuvent alors combiner plusieurs parties d'architectures différentes réalisées séparément. Le second, lors de la conception d'architectures à large échelle, complexes et bien souvent distribuées, qui combinent diverses parties d'architectures ayant des caractéristiques propres, et parfois des styles variés. Ces raisons motivent l'utilisation d'une formalisation qui puisse à la fois décrire sans ambiguïté et contrôler ces compositions.

La théorie des catégories [48] est une approche prometteuse pour répondre à ces besoins. En effet, en se concentrant sur les relations (les morphismes) et les compositions, elle propose de puissants mécanismes qui peuvent être appliqués à la formalisation d'architectures logicielles. De plus, elle permet de passer d'un modèle à un autre et de naviguer entre les niveaux d'abstraction.

D'autre part, d'un point de vue formel, la théorie des catégories est une branche des mathématiques, comparable à la logique, qui étudie les structures et les objets mathématiques ainsi que leurs relations. Il s'agit d'une théorie très générale qui irrigue toutes les branches des mathématiques et qui, au même titre que la logique mathématique, en constitue un des piliers. De ce fait, mon approche n'est pas exclusive des méthodes formelles qui utilisent des théories moins générales, car son universalité rend la définition de ponts entre théories plus aisée.

La théorie des catégories est donc utilisée pour exprimer des problèmes variés provenant de différents domaines scientifiques, tels que les mathématiques, la physique ou l'informatique [158, 50]. En l'utilisant pour formaliser les architectures, nous nous concentrons sur l'étude de la conservation des propriétés dans des compositions de composants, en s'appuyant sur le comportement des foncteurs associés aux préordres. Grâce à cette formalisation, il est possible de déduire :

- la valeur d'une propriété complexe en connaissant les valeurs des propriétés simples dont elle dépend ;
- la valeur d'une propriété pour une composition de composants en connaissant la valeur prise individuellement par chaque composant de la composition.

2.4.1 Les concepts de base

Née en 1940 [48], la théorie des catégories se base sur deux éléments principaux : les catégories et les foncteurs.

Définition 2.4.1 : Catégorie

Une **catégorie** C contient quatre éléments fondamentaux :

1. $Ob(C)$, une collection d'objets ;
2. pour chaque paire $x, y \in Ob(C)$, un ensemble $Hom_C(x, y)$ contenant les **morphismes** allant de x vers y, c'est-à-dire un moyen d'obtenir un objet y (le codomaine) depuis un objet x (le domaine). Un morphisme f de x vers y est noté $f : x \rightarrow y$;
3. pour chaque $x \in Ob(C)$, un morphisme particulier id_x : le morphisme identité de x, noté $id_x : x \rightarrow x$;
4. pour chaque triplet $x, y, z \in Ob(C)$, une **composition** $\circ : Hom_C(y, z) \times Hom_C(x, y) \rightarrow Hom_C(x, z)$. Pour deux morphismes $f : x \rightarrow y$ et $g : y \rightarrow z$, la composition est notée $g \circ f : x \rightarrow z$.

Et deux lois :

1. $f \circ id_x = f$ et $id_y \circ f = f$, avec f un morphisme $f : x \rightarrow y$ et $x, y \in Ob(C)$;
2. $(h \circ g) \circ f = h \circ (g \circ f) \in Hom_C(w, z)$, avec $f : w \rightarrow x$, $g : x \rightarrow y$ et $h : y \rightarrow z$, et $w, x, y, z \in Ob(C)$.

Dans les diagrammes de la suite de ce chapitre, les catégories sont représentées par des boîtes. Les foncteurs sont représentés par des flèches épaisses et leurs effets sur les objets par des flèches en pointillées entre les catégories. Pour simplifier les diagrammes, les morphismes identités ne sont pas représentés bien que toujours existants.

Définition 2.4.2 : Foncteur

Un **foncteur** F est un morphisme entre deux catégories, il sert à passer d'une catégorie C à une catégorie C'. Il est noté $F : C \rightarrow C'$, et agit :

1. sur les objets : pour tout objet $x \in Ob(C)$, on obtient un objet $F(x) \in Ob(C')$;
2. sur les morphismes : pour chaque paire $x, y \in Ob(C)$, on obtient $F : Hom_C(x, y) \rightarrow Hom_{C'}(F(x), F(y))$.

Un foncteur doit respecter deux lois pour être valide :

1. la préservation des identités : $\forall x \in Ob(C)$, $F(id_x) = id_{F(x)}$;
2. la préservation des compositions : $F(h \circ g) = F(h) \circ F(g)$, pour tout triplet $x, y, z \in Ob(C)$ avec les morphismes $g : x \rightarrow y$, $h : y \rightarrow z$.

L'effet d'un foncteur sur les morphismes impose des contraintes fortes sur la validité du foncteur, sur lesquelles il est possible de se baser pour déduire plusieurs propriétés. En effet, pour respecter $F : Hom_C(x, y) \rightarrow Hom_{C'}(F(x), F(y))$, **le domaine et le codomaine d'un morphisme transformé par un foncteur doivent être les mêmes objets que ceux de la catégorie source sur laquelle le foncteur est appliqué**. La figure 2.5 illustre plusieurs cas de foncteurs transformant une catégorie $C1$ en une catégorie $C2$. Dans la première figure de la partie a, on a $F(a1) = a2$ et $F(b1) = c2$ pour les objets, et $F(f1) = g2 \circ f2$ pour le morphisme $f1$. Le foncteur de la deuxième figure est plutôt direct, on retrouve $F(a1) = b2$ et $F(b1) = c2$

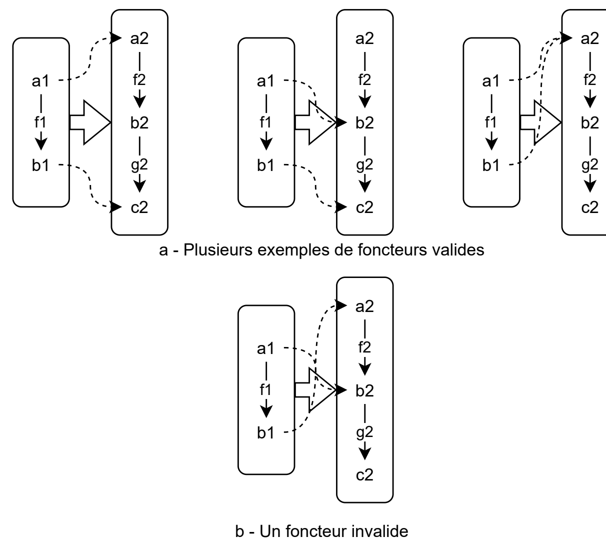


Figure 2.5 – Illustration des foncteurs

pour les objets, et $F(f1) = g2$ pour le morphisme. Sur la troisième figure, on a $F(a1) = a2$ et $F(b1) = a2$ pour les objets et, pour valider la définition du foncteur, $F(f1) = id_{a2}$ pour le morphisme. La partie b de la figure montre un foncteur non valide : s'il agit sur les objets avec $F(a1) = b2$ et $F(b1) = a2$, il n'existe alors pas de morphisme dans $C2$ entre $b2$ et $a2$ qui permette de valider le foncteur.

2.4.2 Les concepts avancés

Quelques définitions de concepts avancés de la théorie des catégories nécessaires à la formalisation d'architectures sont présentées dans cette section. Il s'agit des préordres, des *power sets* et des produits de catégories.

> **Définition 2.4.3 : Préordre**
>
> Un **préordre** est un type de catégorie particulier, dans lequel entre chaque paire $x, y \in Ob(C)$ il ne peut exister qu'au maximum un unique morphisme $f : x \to y$. S'il existe $f : x \to y$ et $g : x \to y$, alors $f = g$.

Les préordres apportent une forme de hiérarchie dans une catégorie. En effet, entre chaque paire $x, y \in \mathrm{Ob}(C)$, il ne peut exister qu'un seul morphisme, en comptant également les compositions. Sur la figure 2.6, on remarque que la catégorie de la partie a est un préordre, puisqu'on a un seul moyen de passer d'un objet à un autre. Dans la partie b, on voit que les préodres permettent d'exclure les morphismes créant des boucles, puisque cela multiplie les moyens de passer d'un objet à un autre. Dans cet exemple, pour passer de l'objet a à l'objet b, on peut utiliser le morphisme f ou la composition $g_2 \circ g_1 \circ f$. Il est tout de même possible d'inclure des morphismes induisant plusieurs chemins pour aller d'un objet à un autre, comme dans la partie c, en précisant une équation stipulant que ces chemins sont égaux. Dans la suite de

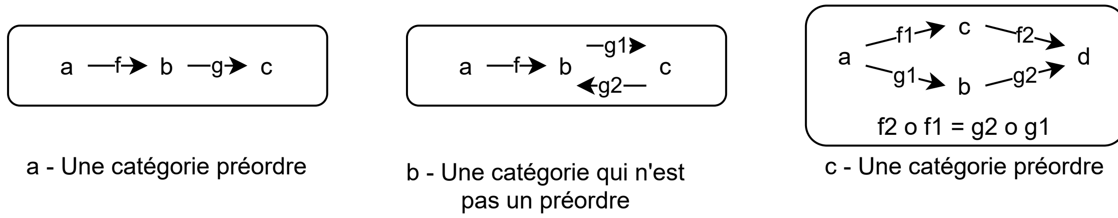


FIGURE 2.6 – Illustration des préordres

la formalisation, lorsque le cas de chemins parallèles similaires à ceux de la partie c se présentera, nous omettrons d'écrire l'équation.

Définition 2.4.4 : Power set

Un ***power set*** est un ensemble qui contient tous les sous-ensembles d'un ensemble donné. Avec la théorie des catégories, un *power set* peut être représenté en formalisant chaque sous-ensemble par un objet, et en liant ces sous-ensembles avec des morphismes uniquement lorsqu'un sous-ensemble X est inclus entièrement dans un sous-ensemble Y. Dans ce cas, une catégorie représentant un *power set* est un préordre.

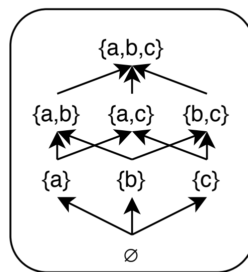


FIGURE 2.7 – Un ***power set*** représentant l'ensemble $\{a, b, c\}$

Un exemple de *power set* pour l'ensemble $\{a, b, c\}$ est donné dans la figure 2.7. En étant un préodre, un *power set* donne une relation d'ordre qui permet d'inclure les éléments un à un dans les sous-ensembles, et d'obtenir l'ensemble final étape par étape.

Définition 2.4.5 : Produit de catégories

Un **produit de catégories** est une opération permettant de produire une nouvelle catégorie à partir de deux catégories existantes $C1$ et $C2$. Les objets de cette nouvelle catégorie sont toutes les paires possibles (x, y) avec $x \in Ob(C1)$ et $y \in Ob(C2)$. Les morphismes $(x, y) \rightarrow (x', y')$ sont les paires (f, g) telles que $f : x \rightarrow x' \in Hom_{C1}(x, x')$ et $g : y \rightarrow y' \in Hom_{C2}(y, y')$.

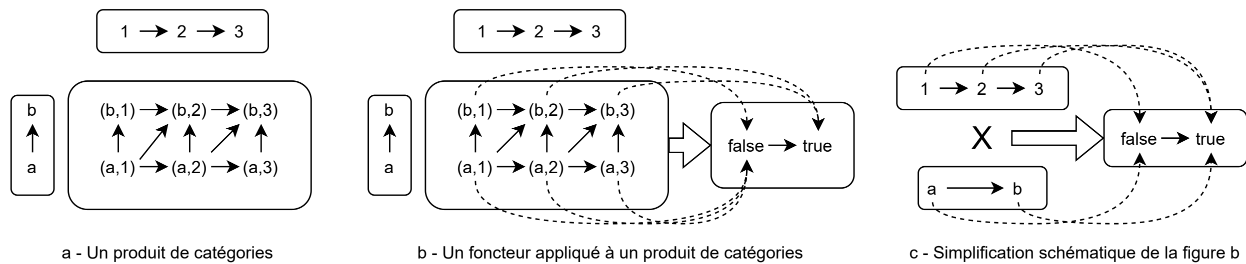


FIGURE 2.8 – Produit de catégories et sa simplification schématique

La figure 2.8 représente le fonctionnement du produit de catégories. La partie a montre la formation des objets et des morphismes dans la nouvelle catégorie créée à partir du produit entre la catégorie du haut et la catégorie de gauche. Toutes les paires possibles contenant un objet d'une catégorie et un deuxième objet de l'autre catégorie sont formées. Des morphismes sont créés si un morphisme existe entre les objets de la paire domaine et de la paire codomaine dans les catégories d'origine des objets. La partie b montre comment un foncteur peut être appliqué sur le résultat d'un produit de catégories, qui suit les mêmes contraintes qu'un foncteur appliqué entre deux catégories standards, et la partie c montre la simplification schématique de la partie b que nous utiliserons dans la suite de ce manuscrit par souci de lisibilité lorsque ce sera suffisant pour refléter l'effet du foncteur. Cette simplification peut être utilisée lorsqu'un foncteur est appliqué sur le résultat d'un produit de catégories entre deux préordres, et que la catégorie codomaine du foncteur est également un préordre. Elle permet de déduire l'objet dans lequel le foncteur transformera une des paires du produit de catégories, uniquement en connaissant l'objet cible pour chaque objet de la paire : ce sera l'objet atteint le plus bas dans la hiérarchie du préordre cible du foncteur. Par exemple si l'objet $(a, 3)$ avait été transformé en l'objet $true$, la simplification schématique n'aurait pas été applicable.

2.4.3 Les principes de la vérification

Je propose une formalisation d'architectures s'appuyant sur la théorie des catégories, détaillée ci-après. Trois catégories principales et deux foncteurs sont définis :

Définition 2.4.6 : La catégorie *Components*

Dans cette catégorie, les objets sont les composants de l'architecture, sans morphismes pour les relier, sauf les morphismes identités.

Définition 2.4.7 : La catégorie *Architecture*

Dans cette catégorie, les objets sont les composants. Ils sont reliés par des morphismes représentant les communications entre eux. L'existence d'un foncteur $f : x \longrightarrow y$ indique que le composant x envoie des données au composant y.

Définition 2.4.8 : La catégorie $ComponentsPS$

Dans cette catégorie, les objets sont les éléments du *power set* des composants.

La catégorie $ComponentsPS$ sera utilisée pour relier les composants aux propriétés, et étudier comment elles évoluent en fonction des ensembles de composants considérés.

Définition 2.4.9 : Le foncteur CA

Le foncteur CA est défini entre les catégories $Components$ et $Architecture$ ($CA : Components \rightarrow Architecture$). Il lie les composants considérés individuellement à leur agencement dans l'architecture.

Ainsi, ce foncteur sert à intégrer les composants pris individuellement dans l'architecture et à définir leurs liens en termes d'échange de données.

Définition 2.4.10 : Le foncteur $CCPS$

Le foncteur $CCPS$ est défini entre les catégories $Components$ et $ComponentsPS$ ($CCPS : Components \rightarrow ComponentsPS$). Il lie les composants considérés individuellement aux éléments du *power set* constitués uniquement d'un seul composant.

De cette manière, une fois que le foncteur $CCPS$ est spécifié, il est possible d'étudier le comportement des propriétés induites par la composition pour un ensemble de composants.

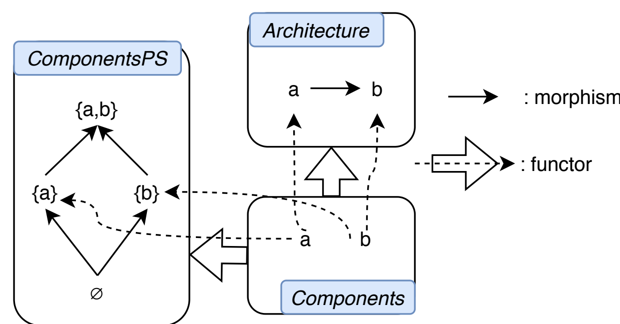


FIGURE 2.9 – Une architecture simple avec deux composants a et b

La figure 2.9 présente la formalisation d'une architecture simple, contenant deux composants a et b, représentés dans la catégorie $Components$. La catégorie $Architecture$ permet de préciser que a envoie des données à b. Chaque composant est envoyé vers la catégorie $ComponentsPS$, plus précisément dans l'ensemble contenant uniquement ce composant. Les morphismes partant des ensembles réduits à un seul élément servent à former les compositions en ajoutant un élément à la fois.

Définition 2.4.11 : Formalisation d'une propriété

Une propriété est représentée par une catégorie préordre, dans laquelle les objets sont les différentes valeurs de la propriété et les morphismes vont de la valeur la plus satisfaisante à la moins satisfaisante. Le symbole $\top$ est utilisé pour les cas où un composant n'est pas concerné par une propriété, afin d'éviter les impacts lors de la composition de composants. $\top$ ne possède qu'un seul morphisme allant vers la valeur la plus satisfaisante de la propriété.

Une propriété peut être simple (en contenant seulement les valeurs $true$ ou $false$), multivaluée, ou plus complexe, c'est-à-dire dépendante d'une combinaison d'autres propriétés. Les propriétés complexes sont formées en associant deux à deux les catégories de propriétés dont elles dépendent à l'aide d'un produit de catégories, puis en reliant le résultat de ce produit à la propriété suivante à l'aide d'un foncteur. La complexité des propriétés peut être identifiée en attribuant un niveau à chaque propriété : celles reliées directement aux composants de l'architecture étant de niveau 1, celles dépendant d'un produit de catégories sont d'un niveau plus élevé. La catégorie $ComponentsPS$ est reliée à toutes les catégories de propriétés de niveau 1 à l'aide de foncteurs.

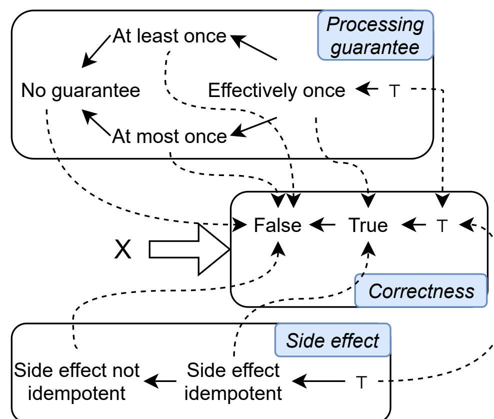


FIGURE 2.10 – Déduction d'une propriété complexe en utilisant un produit de catégories

La figure 2.10 permet de visualiser comment la propriété d'exactitude des résultats (représentée par la catégorie $Correctness$) est formée à partir de la propriété de garantie de traitement du *stream processing* (catégorie $ProcessingGuarantee$), ainsi que des effets de bord éventuels lors de l'exécution des traitements (catégorie $SideEffect$). Afin de simplifier la figure, le résultat du produit de catégories entre les deux propriétés de niveau 1 est représenté par le symbole X, comme illustré dans la figure 2.8. On peut voir que l'exactitude des résultats est garantie seulement lorsque le niveau de garantie est à *effectively-once* et qu'il n'y a pas d'effets de bord, ou alors qu'ils sont idempotents (voir le tableau 2.3 pour une description formelle de l'effet du foncteur sur le produit de catégories).

En utilisant la théorie des catégories, il est possible d'extraire des connaissances de haut niveau à partir de cette formalisation :

— selon la seconde propriété des foncteurs (c'est-à-dire $F : Hom_C(x, y) \rightarrow Hom_{C'}(F(x), F(y))$), si un composant a une valeur de propriété plus faible que d'autres composants et qu'il est ajouté à ces composants, le nouvel ensemble de composants prendra la valeur de propriété la plus faible de celles prises par chaque composant individuellement pour cette propriété. **C'est une déduction essentielle dans la formalisation, puisqu'elle permet de mettre en valeur quelle composition de composants est responsable de la perte d'une propriété dans une architecture** ;

— le produit de deux catégories de niveau 1 permet d'obtenir automatiquement la valeur d'une propriété de niveau 2 en fonction de l'effet du foncteur définit entre le produit de catégories et la propriété de niveau 2.

2.4.4 La démonstration de la vérification des propriétés d'une architecture

Cette formalisation rend possible l'étude de l'évolution des propriétés en fonction de la composition de l'architecture, au moyen de la définition même des foncteurs, des *power sets* ainsi que des produits de catégories. La définition des foncteurs impose des contraintes fortes qui permettent de structurer et de conditionner la préservation des propriétés en fonction des composants faisant partie de l'architecture. Cette section est dédiée à l'utilisation de ces différents éléments dans une formalisation qui permet de démontrer la préservation ou la perte des propriétés d'une architecture.

2.4.4.1 Étape 1 : déclaration des éléments connus

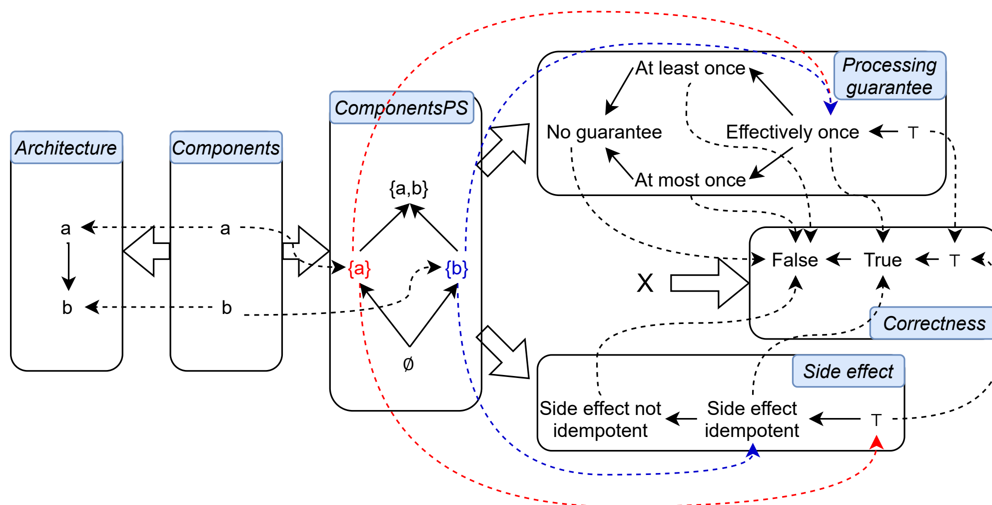


Figure 2.11 – Démonstration de la vérification : déclaration des éléments connus

La première étape consiste à déclarer tous les éléments connus de l'architecture. Du point de vue de la formalisation, cela signifie définir :

— les morphismes à l'intérieur des catégories ;

— les effets du foncteur *CCPS* de la catégorie *Components* vers la catégorie *ComponentsPS* et du foncteur *CA* de la catégorie *Components* vers la catégorie *Architecture* ;

- l'effet des foncteurs entre les produits de catégories représentant des propriétés de niveau 1 et les catégories représentant une propriété de niveau supérieur ;
- les effets des foncteurs entre la catégorie $ComponentsPS$ et les catégories des propriétés.

Cependant, les effets des foncteurs entre les composants et les propriétés sont uniquement définis pour les composants individuels (autrement dit les ensembles du *power set* constitués d'un seul élément). La représentation schématique de cette étape est illustrée dans la figure 2.11.

D'un point de vue plus formel, les morphismes des catégories $ComponentsPS$, $ProcessingGuarantee$, $SideEffect$ et $Correctness$ sont décrits dans le tableau 2.1, et l'effet des foncteurs sur les composants individuels entre les catégories $ComponentsPS$ et $ProcessingGuarantee$, et entre $ComponentsPS$ et $SideEffect$ sont recensés dans le tableau 2.2.

Morphismes de la catégorie $ComponentsPS$ (CPS)
les morphismes avec ∅ comme domaine sont omis
$a-ab:\{a\}\rightarrow\{a,b\}$
$b-ab:\{b\}\rightarrow\{a,b\}$
Morphismes de la catégorie $ProcessingGuarantee$ (PG)
$\top-eo:\top\rightarrow EffectivelyOnce$
$eo-alo:EffectivelyOnce\rightarrow AtLeastOnce$
$eo-amo:EffectvelyOnce\rightarrow AtMostOnce$
$amo-ng:AtMostOnce\rightarrow NoGuarantee$
$alo-ng:AtLeastOnce\rightarrow NoGuarantee$
Morphismes de la catégorie $SideEffect$ (SE)
$\top-sei:\top\rightarrow SideEffectIdempotent$
$sei-seni:SideEffectIdempotent\rightarrow SideEffectNotIdempotent$
Morphismes de la catégorie $Correctness$ (C)
$\top-t:\top\rightarrow True$
$t-f:True\rightarrow False$

Table 2.1 – Déclaration des morphismes (les morphismes identité sont omis)

Foncteur $CPS-PG$ de la catégorie $ComponentsPS$ à la catégorie $ProcessingGuarantee$
$\{a\}\rightarrow EffectivelyOnce$
$\{b\}\rightarrow EffectivelyOnce$
Foncteur $CPS-SE$ de la catégorie $ComponentsPS$ à la catégorie $SideEffect$
$\{a\}\rightarrow\top$
$\{b\}\rightarrow SideEffectIdempotent$

Table 2.2 – Déclaration de l'effet des foncteurs sur les composants individuels

Concernant les propriétés complexes, un dernier tableau (tableau 2.3[1]) détaille l'effet du foncteur entre le produit de catégories $ProcessingGuarantee\times SideEffect$ et la catégorie $Correctness$.

1. Il est à noter que l'effet $\top\times SideEffectNotIdempotent\rightarrow False$ n'est pas correct dans le cas où un processus *batch* aurait des effets de bord non idempotents. En effet, dans une telle situation, les effets de bord n'influent pas sur l'exactitude du traitement *batch*. Toutefois, dans le but de ne pas complexifier l'exemple, ce détail n'est pas pris en compte et on considère que l'on se place dans un contexte de *stream processing* uniquement. Une modélisation plus complète de ce phénomène est représentée dans la section 2.4.6.

Foncteur $PG \times SE - C$ du produit de catégories $ProcessingGuarantee \times SideEffect$ à la catégorie $Correctness$
$\top \times \top \rightarrow \top$
$\top \times SideEffectIdempotent \rightarrow True$
$\top \times SideEffectNotIdempotent \rightarrow False$
$EffectivelyOnce \times \top \rightarrow True$
$EffectivelyOnce \times SideEffectIdempotent \rightarrow True$
$EffectivelyOnce \times SideEffectNotIdempotent \rightarrow False$
$AtLeastOnce \times \top \rightarrow False$
$AtLeastOnce \times SideEffectIdempotent \rightarrow False$
$AtLeastOnce \times SideEffectNotIdempotent \rightarrow False$
$AtMostOnce \times \top \rightarrow False$
$AtMostOnce \times SideEffectIdempotent \rightarrow False$
$AtMostOnce \times SideEffectNotIdempotent \rightarrow False$
$NoGuarantee \times \top \rightarrow False$
$NoGuarantee \times SideEffectIdempotent \rightarrow False$
$NoGuarantee \times SideEffectNotIdempotent \rightarrow False$

TABLE 2.3 – Déclaration de l'effet des foncteurs sur les objets du produit de catégories

2.4.4.2 Étape 2 : déduction des valeurs de propriétés pour les compositions de composants

À partir de ces éléments déclarés, il est ensuite possible de se servir de plusieurs mécanismes de déduction pour connaître la valeur pour chaque propriété que prennent les compositions de composants, ainsi que la valeur des propriétés complexes à partir des propriétés simples dont elles dépendent.

Pour déduire la valeur d'une propriété pour une composition de composants, il faut utiliser la contrainte des foncteurs sur les morphismes ($F : Hom_C(x, y) \rightarrow Hom_{C'}(F(x), F(y))$) stipulée dans la définition 2.4.2. En appliquant simultanément cette contrainte sur les morphismes ayant la composition de composants comme codomaine, on peut alors réduire les objets solutions de la catégorie destination (celle de la propriété) dans lesquels l'objet de la composition de composants peut être transformé par le foncteur.

Par exemple, si l'on cherche à déduire la valeur de la propriété $SideEffect$ pour la composition $\{a, b\}$, il faut chercher à trouver comment les deux morphismes ayant $\{a, b\}$ en codomaine vont être transformés par le foncteur en même temps :

$$CPS - SE : \underline{Hom_{CPS}(\{a\}, \{a, b\})} \rightarrow Hom_{SE}(\underline{CPS - SE(\{a\})}, CPS - SE(\{a, b\}))$$
$$CPS - SE : \underline{Hom_{CPS}(\{b\}, \{a, b\})} \rightarrow Hom_{SE}(\underline{CPS - SE(\{b\})}, CPS - SE(\{a, b\}))$$

Les éléments soulignés sont ceux qui sont déjà connus. En se référant au tableau 2.2, on sait que $CPS - SE(\{a\}) : \{a\} \rightarrow \top$ et que $CPS - SE(\{b\}) : \{b\} \rightarrow SideEffectIdempotent$. On sait également que les deux morphismes vont avoir le même codomaine. Parmi les morphismes de la catégorie $SideEffect$, nous cherchons donc un morphisme ayant pour domaine $\top$ et un morphisme ayant pour domaine $SideEffectIdempotent$, tout en sachant que leur codomaine sera identique. Pour résoudre cette transformation, les morphismes $\top - sei$ et id_{sei} (morphisme identité de l'objet $SideEffectIdempotent$) correspondent à ces contraintes. Comme leur codomaine est $SideEffectIdempotent$, on peut en déduire que :

$$CPS - SE(\{a, b\}) : \{a, b\} \rightarrow SideEffectIdempotent$$

Autrement dit, la valeur de la propriété $SideEffect$ pour la composition $\{a, b\}$ est $SideEffectIdempotent$.

Remarque 2.4.1

Les solutions ne sont pas forcément uniques. Par exemple, pour trouver la valeur de la propriété $SideEffect$ pour l'objet $\{a, b\}$, nous aurions pu nous servir de la composition (qui est un morphisme) $sei - seni \circ \top - sei$ et du morphisme $sei - seni$. Dans ce cas, leur codomaine est $SideEffectNotIdempotent$. Toutefois, cette non-unicité des solutions ne permet en aucun cas d'attribuer une propriété plus élevée que celle réellement supportée. En conséquent, nous conserverons la propriété correspondant au codomaine le plus favorable au sens de la hiérarchie du préordre de la propriété, afin de n'intégrer la perte de la propriété seulement lorsqu'elle est réellement constatée.

On peut utiliser le même mécanisme pour déduire la valeur de la composition de composants pour la propriété $ProcessingGuarantee$. Cette fois, nous cherchons à résoudre les transformations de morphismes suivantes :

$$CPS - PG : \underline{Hom_{CPS}(\{a\}, \{a, b\})} \rightarrow Hom_{PG}(CPS - PG(\{a\}), CPS - PG(\{a, b\}))$$
$$CPS - PG : \underline{Hom_{CPS}(\{b\}, \{a, b\})} \rightarrow Hom_{PG}(\underline{CPS - PG(\{b\})}, CPS - PG(\{a, b\}))$$

On sait que $CPS-PG(\{a\}) : \{a\} \rightarrow EffectivelyOnce$ et que $CPS-PG(\{b\}) : \{b\} \rightarrow EffectivelyOnce$. À partir des morphismes de la catégorie $ProcessingGuarantee$, on peut en déduire que l'on peut utiliser le morpisme identité id_{eo} de l'objet $EffectivelyOnce$ pour résoudre les transformations, et donc que :

$$CPS - PG(\{a, b\}) : \{a, b\} \rightarrow EffectivelyOnce$$

La valeur de la propriété $ProcessingGuarantee$ pour la composition $\{a, b\}$ est donc $EffectivelyOnce$.

Il est également possible de définir la valeur d'une propriété complexe en connaissant les valeurs des propriétés simples dont elle dépend. Ainsi, à partir du tableau 2.3, on peut déduire que le composant a, comme il a la valeur $EffectivelyOnce$ dans la catégorie $ProcessingGuarantee$ et la valeur $\top$ dans la catégorie $SideEffect$, prendra la valeur $True$ dans la catégorie $Correctness$, puisqu'il est déclaré que $PG \times SE - C(EffectivelyOnce \times \top) : EffectivelyOnce \times \top \rightarrow True$. De la même manière, comme $PG \times SE - C(EffectivelyOnce \times SideEffectIdempotent) : EffectivelyOnce \times SideEffectIdempotent \rightarrow True$, on peut en déduire que le composant b et la composition $\{a, b\}$ prennent également la valeur $True$ dans la catégorie $Correctness$, et donc que l'architecture supporte l'exactitude des traitements. La figure 2.12 résume schématiquement ces mécanismes de déduction.

Remarque 2.4.2

En s'appuyant sur la définition des foncteurs, et plus particulièrement sur $F : Hom_C(x, y) \rightarrow Hom_{C'}(F(x), F(y))$, on peut déduire la valeur d'une propriété pour un ensemble de composants : **la perte d'une propriété par un composant ne peut pas être compensée s'il est intégré dans un ensemble de composants**. De plus, en ce qui concerne les propriétés complexes, tous les foncteurs transformant un produit de catégories en une propriété de niveau strictement supérieur à 1 sont préalablement définis, et permettent de déterminer la valeur que prendra un ensemble de composants pour cette propriété uniquement en connaissant la valeur qu'ils prennent pour les propriétés de niveau 1.

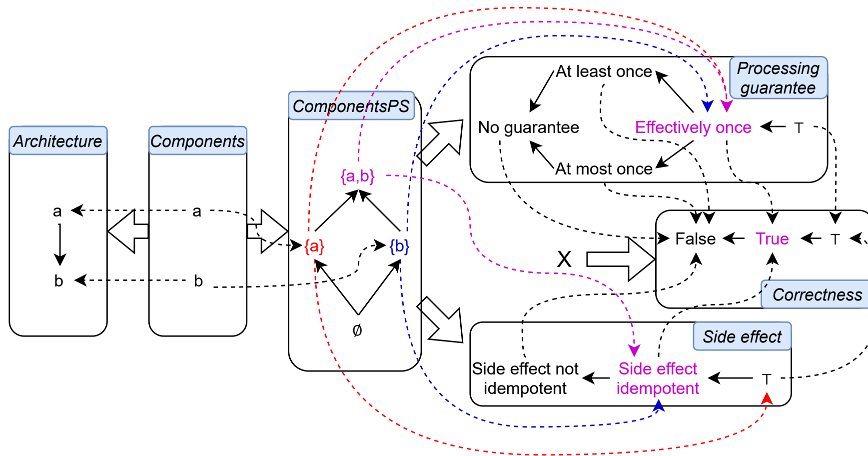


FIGURE 2.12 – Démonstration de la vérification : déduction des valeurs des propriétés pour une composition de composants

2.4.5 La vérification des propriétés de la Lambda Architecture

La formalisation nous permet d'étudier l'évolution des propriétés de la Lambda Architecture. Pour simplifier les schémas, la catégorie *ComponentsPS* ne contient que les composants individuels ainsi que les compositions qui représentent un sous-ensemble possible de l'architecture, et seuls les liens des composants vers les propriétés qui ne mènent pas à $\top$ sont présents.

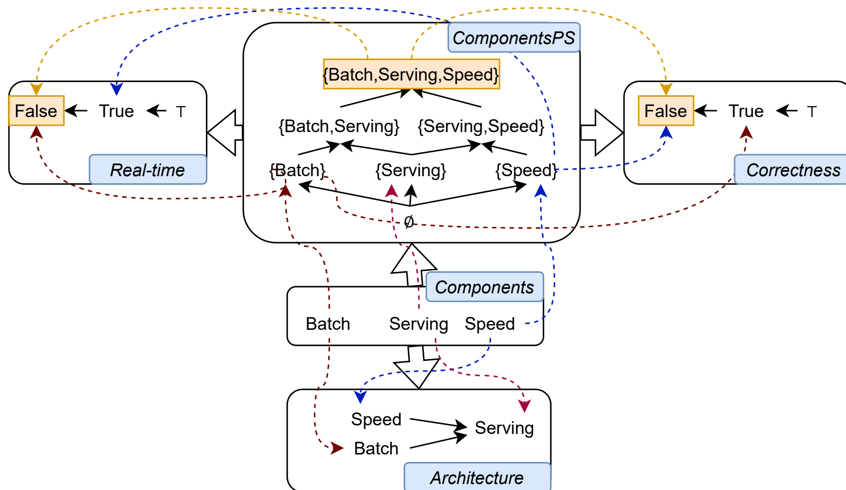


FIGURE 2.13 – Formalisation de la Lambda Architecture

La figure 2.13 montre la formalisation de la Lambda Architecture en suivant la spécification de sa création. La couche *serving* est neutre par rapport aux propriétés d'exactitude des traitements et de temps réel, étant

donné qu'elle met seulement à disposition les résultats qui ont été calculés dans les autres couches. La couche *speed* supporte la propriété de temps réel, mais pas celle d'exactitude des traitements. La couche *batch* a un comportement inverse, en supportant la propriété d'exactitude des traitements mais pas celle de temps réel. Avec la formalisation, il est possible de prouver que l'architecture globale ne supporte pas les propriétés d'exactitude des traitements et de temps réel simultanément. Pour cela, nous utilisons le même procédé que dans la démonstration.

Nous définissons tout d'abord les morphismes (tableau 2.4) et l'effet des foncteurs sur les composants individuels (tableau 2.5).

Morphismes de la catégorie $ComponentsPS$ **(**CPS**)**
les morphismes avec Ø comme domaine sont omis
$B-BSe : \{Batch\} \rightarrow \{Batch, Serving\}$
$Se-BSe : \{Serving\} \rightarrow \{Batch, Serving\}$
$Sp-SeSp : \{Speed\} \rightarrow \{Serving, Speed\}$
$Se-SeSp : \{Serving\} \rightarrow \{Serving, Speed\}$
$BSe-BSeSp : \{Batch, Serving\} \rightarrow \{Batch, Serving, Speed\}$
$SeSp-BSeSp : \{Serving, Speed\} \rightarrow \{Batch, Serving, Speed\}$
Morphismes de la catégorie $RealTime$ **(**RT**)**
$\top - t : \top \rightarrow True$
$t-f : True \rightarrow False$
Morphismes de la catégorie $Correctness$ **(**C**)**
$\top - t : \top \rightarrow True$
$t-f : True \rightarrow False$

Table 2.4 – Déclaration des morphismes des catégories de la Lambda Architecture (les morphismes identité sont omis)

Foncteur $CPS-RT$ **de la catégorie** $ComponentsPS$ **à la catégorie** $RealTime$
$\{Batch\} \rightarrow False$
$\{Serving\} \rightarrow \top$
$\{Speed\} \rightarrow True$
Foncteur $CPS-C$ **de la catégorie** $ComponentsPS$ **à la catégorie** $Correctness$
$\{Batch\} \rightarrow True$
$\{Serving\} \rightarrow \top$
$\{Speed\} \rightarrow False$

Table 2.5 – Déclaration de l'effet des foncteurs sur les composants individuels de la Lambda Architecture

À partir de ces éléments et des mécanismes démontrés dans la section 2.4.4, ils est possible de connaître la valeur que prennent les compositions de composants pour chaque propriété.

Pour déduire la valeur que va prendre la composition $\{Batch, Serving, Speed\}$ pour la propriété $Correctness$, il faut résoudre plusieurs transformations de morphismes effectuées par le foncteur $CPS-C$ [2] :

2. Nous utilisons ici directement des compositions, qui sont aussi des morphismes. En effet, comme la catégorie $ComponentsPS$ est un préordre, $Hom_{CPS}(\{a\},\{a,b,c\}) = Hom_{CPS}(\{a,b\},\{a,b,c\}) \circ Hom_{CPS}(\{a\},\{a,b\}) = Hom_{CPS}(\{a,c\},\{a,b,c\}) \circ Hom_{CPS}(\{a\},\{a,c\})$. Comme les foncteurs préservent les compositions, la formalisation est aussi applicable dans ce cas.

$$CPS-C : \underline{Hom_{CPS}(\{Batch\},\{Batch,Serving,Speed\})} \rightarrow Hom_{C}(\underline{CPS-C(\{Batch\})},CPS-C(\{Batch,Serving,Speed\}))$$
$$CPS-C : \underline{Hom_{CPS}(\{Serving\},\{Batch,Serving,Speed\})} \rightarrow Hom_{C}(\underline{CPS-C(\{Serving\})},CPS-C(\{Batch,Serving,Speed\}))$$
$$CPS-C : \underline{Hom_{CPS}(\{Speed\},\{Batch,Serving,Speed\})} \rightarrow Hom_{C}(\underline{CPS-C(\{Speed\})},CPS-C(\{Batch,Serving,Speed\}))$$

On sait que $CPS-C(\{Batch\}) : \{Batch\} \rightarrow True$, $CPS-C(\{Serving\}) : \{Serving\} \rightarrow \top$ et que $CPS-C(\{Speed\}) : \{Speed\} \rightarrow False$. Avec ces informations, on en déduit que les morphismes ci-dessus sont transformés respectivement en $t-f$, en composition $t-f \circ \top - t$ et en morphisme identité id_f sur l'objet $False$. Donc, pour préserver le codomaine commun :

$$CPS-C(\{Batch,Serving,Speed\}) : \{Batch,Serving,Speed\} \rightarrow False$$

Pour déduire la valeur que va prendre la composition $\{Batch,Serving,Speed\}$ pour la propriété $RealTime$, on utilise le même mécanisme en s'intéressant cette fois au foncteur $CPS-RT$:

$$CPS-RT : \underline{Hom_{CPS}(\{Batch\},\{Batch,Serving,Speed\})} \rightarrow Hom_{RT}(\underline{CPS-RT(\{Batch\})},CPS-RT(\{Batch,Serving,Speed\}))$$
$$CPS-RT : \underline{Hom_{CPS}(\{Serving\},\{Batch,Serving,Speed\})} \rightarrow Hom_{RT}(\underline{CPS-RT(\{Serving\})},CPS-RT(\{Batch,Serving,Speed\}))$$
$$CPS-RT : \underline{Hom_{CPS}(\{Speed\},\{Batch,Serving,Speed\})} \rightarrow Hom_{RT}(\underline{CPS-RT(\{Speed\})},CPS-RT(\{Batch,Serving,Speed\}))$$

On sait que $CPS-RT(\{Batch\}) : \{Batch\} \rightarrow False$, $CPS-RT(\{Serving\}) : \{Serving\} \rightarrow \top$ et que $CPS-RT(\{Speed\}) : \{Speed\} \rightarrow True$. Avec ces informations, on en déduit que les morphismes ci-dessus sont transformés respectivement en en morphisme identité id_f sur l'objet $False$, en composition $t-f \circ \top - t$ et en morphisme $t-f$. Donc, pour préserver le codomaine commun :

$$CPS-RT(\{Batch,Serving,Speed\}) : \{Batch,Serving,Speed\} \rightarrow False$$

En utilisant la théorie des catégories, on peut prouver que la Lambda Architecture ne peut pas supporter à la fois la propriété d'exactitude des traitements et celle de temps réel en fusionnant les vues *batch* et *speed* dans la couche *serving*.

2.4.6 La vérification des propriétés de la Lambda+ Architecture

Dans la Lambda+ Architecture, il y a deux compositions de composants qui nous intéressent : celle avec tous les composants sauf le *master dataset*, dans le cas d'un fonctionnement normal de l'architecture, et celle avec tous les composants y compris le *master dataset*, lorsqu'une reprise des données est nécessaire. Ce sont donc ces deux compositions qui seront étudiées dans cette section. Les propriétés étudiées sont celles de temps réel (catégorie $RealTime$), de résistance aux pannes (catégorie $FaultTolerance$) et d'exactitude des traitements (catégorie $OverallCorrectness$). Cette dernière propriété dépend des propriétés d'exactitude des traitements dans le cadre d'un traitement en *batch* (catégorie $BatchCorrectness$) et dans le cadre d'un traitement en *stream processing* (catégorie $StreamingCorrectness$), qui elle-même dépend des propriétés de garantie des traitements (catégorie $ProcessingGuarantee$) et des effets de bord (catégorie $StreamingSideEffect$).

La figure 2.14 montre la synthèse de la formalisation de la Lambda+ Architecture, le tableau 2.6 recense les morphismes dans les différentes catégories, et le tableau 2.7 les foncteurs. La tableau 2.3 peut être utilisé pour voir le résultat de l'application du foncteur sur le produit de catégories $ProccessingGuarantee \times StreamingSideEffect$, et le tableau 2.8 pour voir le résultat de l'application du foncteur sur le produit de catégories $BatchCorrectness \times StreamingCorrectness$. On utilise la notation Dtc = *Data traffic controller*, Se = *Streaming ETL*, Rti = *Real-time insights*, Md = *Master dataset* et S = *Storage*.

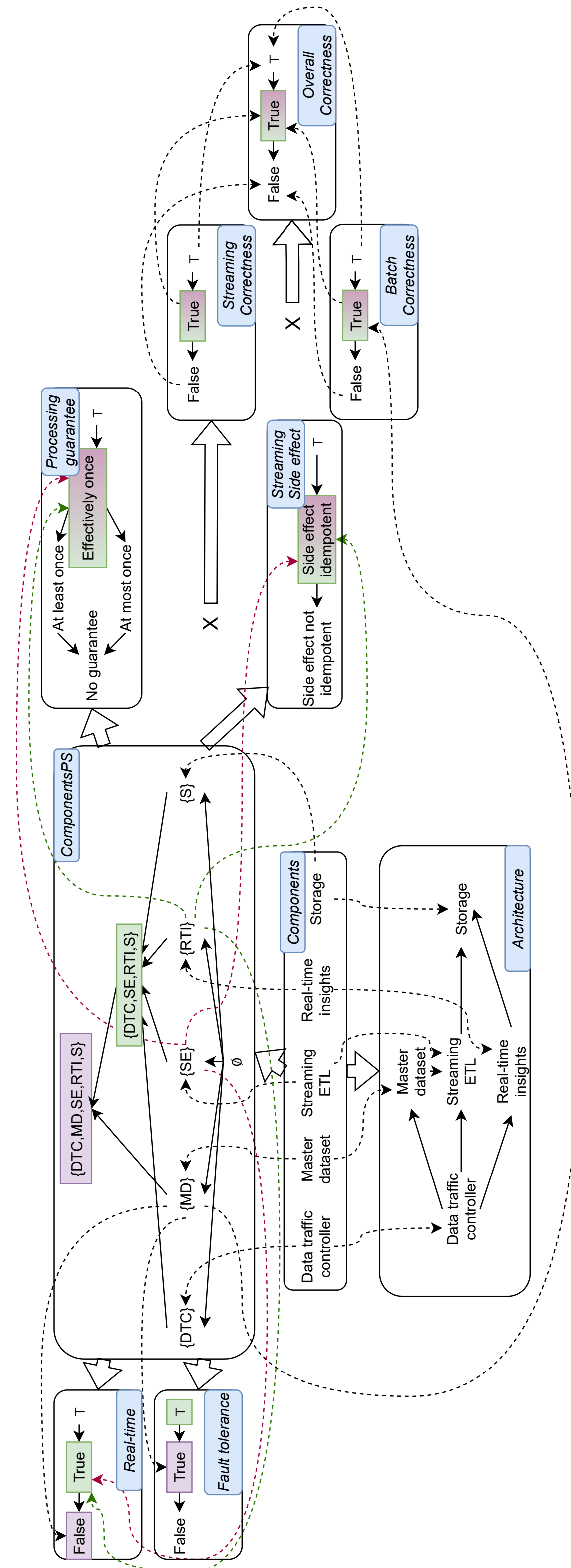


Figure 2.14 – Formalisation de la Lambda+ Architecture

Morphismes de la catégorie $ComponentsPS$ **(**CPS**)**
seules les compositions nous intéressant sont indiquées
$Dtc - DtcSeRtiS : \{DTC\} \rightarrow \{DTC, SE, RTI, S\}$
$Se - DtcSeRtiS : \{SE\} \rightarrow \{DTC, SE, RTI, S\}$
$Rti - DtcSeRtiS : \{RTI\} \rightarrow \{DTC, SE, RTI, S\}$
$S - DtcSeRtiS : \{S\} \rightarrow \{DTC, SE, RTI, S\}$
$Dtc - DtcMdSeRtiS : \{DTC\} \rightarrow \{DTC, MD, SE, RTI, S\}$
$Md - DtcMdSeRtiS : \{MD\} \rightarrow \{DTC, MD, SE, RTI, S\}$
$Se - DtcMdSeRtiS : \{Se\} \rightarrow \{DTC, MD, SE, RTI, S\}$
$Rti - DtcMdSeRtiS : \{Rti\} \rightarrow \{DTC, MD, SE, RTI, S\}$
$S - DtcMdSeRtiS : \{S\} \rightarrow \{DTC, MD, SE, RTI, S\}$
Morphismes de la catégorie $RealTime$ **(**RT**)**
$\top - t : \top \rightarrow True$
$t - f : True \rightarrow False$
Morphismes de la catégorie $FaultTorelance$ **(**FT**)**
$\top - t : \top \rightarrow True$
$t - f : True \rightarrow False$
Morphismes de la catégorie $BatchCorrectness$ **(**BC**)**
$\top - t : \top \rightarrow True$
$t - f : True \rightarrow False$
Morphismes de la catégorie $StreamingCorrectness$ **(**SC**)**
$\top - t : \top \rightarrow True$
$t - f : True \rightarrow False$
Morphismes de la catégorie $ProcessingGuarantee$ **(**PG**)**
$\top - eo : \top \rightarrow EffectivelyOnce$
$eo - alo : EffectivelyOnce \rightarrow AtLeastOnce$
$eo - amo : EffectvelyOnce \rightarrow AtMostOnce$
$amo - ng : AtMostOnce \rightarrow NoGuarantee$
$alo - ng : AtLeastOnce \rightarrow NoGuarantee$
Morphismes de la catégorie $StreamingSideEffect$ **(**SSE**)**
$\top - sei : \top \rightarrow SideEffectIdempotent$
$sei - seni : SideEffectIdempotent \rightarrow SideEffectNotIdempotent$

Table 2.6 – Déclaration des morphismes des catégories de la Lambda+ Architecture (les morphismes identité sont omis)

Avec ces spécifications et en se servant des mécanismes de déduction présentés précédemment, on peut déduire que la Lambda+ Architecture supporte l'exactitude des traitements en toutes circonstances. La propriété de temps réel est supportée tant qu'une extraction des données n'est pas effectuée depuis le *master dataset*. Mais c'est un cas exceptionnel, utilisé uniquement lorsqu'une défaillance s'est produite ou que les besoins ont évolué et que les données doivent être traitées à nouveau. Dans ce cas, on perd la propriété de temps réel, mais on active celle de résistance aux pannes. La propriété de résistance aux pannes peut être interprétée de la manière suivante : aucun composant n'entraîne la perte de cette propriété, mais seule la reprise sur panne du *master dataset* permet de l'activer.

Foncteur $CPS-RT$ **de la catégorie** $ComponentsPS$ **à la catégorie** $RealTime$
$\{DTC\} \rightarrow True$
$\{MD\} \rightarrow False$
$\{SE\} \rightarrow True$
$\{RTI\} \rightarrow True$
$\{S\} \rightarrow \top$
Foncteur $CPS-FT$ **de la catégorie** $ComponentsPS$ **à la catégorie** $FaultTolerance$
$\{DTC\} \rightarrow \top$
$\{MD\} \rightarrow True$
$\{SE\} \rightarrow \top$
$\{RTI\} \rightarrow \top$
$\{S\} \rightarrow \top$
Foncteur $CPS-BC$ **de la catégorie** $ComponentsPS$ **à la catégorie** $BatchCorrectness$
$\{DTC\} \rightarrow \top$
$\{MD\} \rightarrow True$
$\{SE\} \rightarrow \top$
$\{RTI\} \rightarrow \top$
$\{S\} \rightarrow \top$
Foncteur $CPS-PG$ **de la catégorie** $ComponentsPS$ **à la catégorie** $ProcessingGuarantee$
$\{DTC\} \rightarrow EffectivelyOnce$
$\{MD\} \rightarrow \top$
$\{SE\} \rightarrow EffectivelyOnce$
$\{RTI\} \rightarrow EffectivelyOnce$
$\{S\} \rightarrow \top$
Foncteur $CPS-SE$ **de la catégorie** $ComponentsPS$ **à la catégorie** $SideEffect$
$\{DTC\} \rightarrow SideEffectIdempotent$
$\{MD\} \rightarrow \top$
$\{SE\} \rightarrow SideEffectIdempotent$
$\{RTI\} \rightarrow SideEffectIdempotent$
$\{S\} \rightarrow \top$

TABLE 2.7 – Déclaration de l'effet des foncteurs sur les composants individuels de la Lambda+ Architecture

2.5 Conclusion et perspectives

Dans ce chapitre, nous avons montré que la Lambda Architecture présente plusieurs limites. Tout d'abord, les propriétés de **temps réel** et d'**exactitude des traitements** ne peuvent pas être obtenues simultanément. Ensuite, sa **complexité est élevée** du fait de la duplication des traitements dans les couches *speed* et *batch*, qui ont chacune leur propre paradigme. Ses **cas d'utilisations sont également limités**, puisque seuls des indicateurs prédéfinis peuvent être calculés, ce qui exclut les analyses exploratoires. Enfin, sa **définition manque de précision**, notamment au niveau de la couche *serving* lors de la fusion des vues *batch* et *speed*.

La Lambda Architecture a été conçue pour compenser les faiblesses d'une technologie naissante. De nos jours, elle peut être remplacée par la Kappa Architecture, qui simplifie la Lambda en conservant uniquement la couche *speed*, mais au coût de la perte de la propriété de résistance aux pannes, et sans pour autant permettre d'élargir les cas d'utilisation.

Afin de donner un nouveau souffle à la Lambda Architecture, j'ai proposé le patron Lambda+ Architecture,

Foncteur $BC \times SC - OC$ **du produit de catégories** $BatchCorrectness \times StreamingCorrectness$ **à la catégorie** $OverallCorrectness$
$\top \times \top \rightarrow \top$
$\top \times True \rightarrow True$
$\top \times False \rightarrow False$
$True \times \top \rightarrow True$
$True \times True \rightarrow True$
$False \times False \rightarrow False$
$False \times \top \rightarrow False$
$False \times True \rightarrow False$
$False \times False \rightarrow False$

Table 2.8 – Déclaration de l'effet des foncteurs sur les objets du produit de catégories de la Lambda+ Architecture

qui supporte les propriétés promises par la Lambda tout en élargissant ses cas d'utilisation. La Lambda+ met en avant la dualité entre les analyses exploratoires et les indicateurs en temps réel permettant de tirer profit des données massives. Elle prend aussi en considération les progrès des systèmes de *stream processing* afin de garantir la propriété d'exactitude des traitements dans tous ses composants, et évite la complexité de la Lambda induite par la duplication des traitements.

La formalisation des architectures étant essentielle pour étudier la conservation des propriétés dans des compositions de composants qui peuvent être complexes, j'ai également proposé une technique de formalisation s'appuyant sur la théorie des catégories.

Plusieurs perspectives se dessinent pour cette formalisation, en plus de l'appliquer à d'autres patrons et implémentations afin de vérifier sa généralisation. La catégorie $ComponentsPS$ peut être simplifiée en conservant uniquement les sous-ensembles représentant une composition de composants valide de l'architecture. Par exemple, dans la Lambda Architecture les couches *batch* et *speed* ne peuvent pas former une composition sans la couche *serving*. Une piste pour réaliser cela serait d'appliquer un foncteur entre les catégories $Architecture$ et $ComponentsPS$.

L'instauration d'un moyen de navigation entre différents niveaux de granularité de l'architecture, et également entre une partie de l'architecture plus détaillée et l'architecture complète, rendrait la formalisation encore plus complète. Cela pourrait être réalisé en représentant chaque niveau de granularité avec la formalisation actuelle, puis en utilisant des foncteurs allant de la représentation la plus spécifique à la plus générale. De cette manière, les propriétés des foncteurs permettraient de transmettre les déductions d'un niveau de granularité à un autre.

Ajouter un mécanisme vérifiant si une implémentation suit un certain style ou patron rendrait la classification des architectures plus formelle. L'utilisation d'un foncteur plein (c'est-à-dire un foncteur surjectif) est une hypothèse prometteuse, qui pourrait laisser suffisamment de liberté aux implémentations (par exemple, en découpant un composant d'un style en deux sous-composants), tout en vérifiant si l'implémentation suit l'esprit du style ou du patron.

Publications relatives au chapitre

*****

Concernant les publications réalisées en relation avec ce chapitre, les publications [A2] et [A3] sont des colloques sans actes, qui ont résulté en un résumé suivi d'une présentation, la publication [A5] sera également présentée à BDA 2021 au mois d'octobre. Une action de transfert de technologie avec une des entreprises partenaires du projet Cocktail est prévue pour qu'ils puissent déployer l'architecture Hydre.

[A1] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **Lambda Architecture pour une analyse à haute performance des données des réseaux sociaux**. In : *INFormatique des ORganisations et Systèmes d'Information et de Décision (INFORSID)*. 2019, pp. 1–16.

[A2] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **Une plateforme haute performance pour l'exploitation des données massives**. In : *DataBFC2 (`https://databfc2.sciencesconf.org/`)*. 2019.

[A3] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **Lambda+ Architecture — Un patron d'architecture haute performance pour le traitement des Big Data**. In : *Journées Calcul et Données (JCAD, `https://jcad2020.sciencesconf.org/`)*. 2020.

[A4] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **Évolution et formalisation de la Lambda Architecture pour des analyses à hautes performances — Application aux données de Twitter**. In : *Revue ouverte d'ingénierie des systèmes d'information (ISI)* Numéro 1 (2021), pp. 1–26.

[A5] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **Lambda+, the renewal of the Lambda Architecture : Category Theory to the rescue**. In : *33rd Conference on Advanced Information Systems Engineering (CAiSE)*. Springer LNCS, 2021, pp. 1–15.

Chapitre

3

Tensor Data Model : un modèle de données tensoriel pour les polystores

*****

3.1 Introduction 50
3.1.1 L'essor des *polystores* 51
3.1.2 Les tenseurs comme outils d'analyse 52
3.1.3 L'importance de la sûreté des programmes d'analyse 53
3.1.4 Contribution 53
3.2 État de l'art : modèles pivots, tenseurs et librairies de tenseurs 54
3.2.1 Les modèles pivots pour les données hétérogènes 54
3.2.2 L'objet mathématique tenseur 56
3.2.3 Les librairies de tenseurs existantes 61
3.2.4 Synthèse 63
3.3 Tensor Data Model et opérateurs 63
3.3.1 Le schéma 64
3.3.2 Les opérateurs de TDM 66
3.3.3 Les correspondances entre TDM et les autres modèles de représentation des données 74
3.4 Typage sûr et inférence de schéma 75
3.4.1 Les mécanismes utilisés 75
3.4.2 Vers un langage de requêtes fonctionnel 78
3.5 Mise en œuvre d'optimisations 81
3.5.1 Les optimisations de la librairie générale 81
3.5.2 Les optimisations de la décomposition CANDECOMP/PARAFAC 82
3.6 Conclusion 87

L'analyse des données massives ne se limite pas à l'exécution d'algorithmes sur un jeu de données. Les données concernées doivent d'abord être manipulées, c'est-à-dire sélectionnées, mises en forme, normalisées, etc., afin qu'elles soient adaptées aux analyses à effectuer. C'est une étape délicate, dont dépend directement la qualité des analyses, aussi bien d'un point de vue technique, avec des erreurs d'exécution qui peuvent interrompre des *workflows* au bout de plusieurs heures de traitement, mais aussi d'un point de vue fonctionnel, en attribuant une signification erronée aux données (initialement ou après une manipulation), qui peut conduire à de mauvaises interprétations des résultats ne correspondant alors pas à ce qui est attendu. Les outils d'analyse doivent prendre ces aspects en considération, à la fois pour faciliter la manipulation des données, mais également pour prévenir un maximum d'erreurs aussi bien techniques que fonctionnelles, dans le but de laisser l'utilisateur se concentrer sur les analyses en elles-mêmes.

Ce chapitre présente TDM (Tensor Data Model), un modèle basé sur les tenseurs répondant à ces attentes et intégrant nativement de puissantes techniques d'analyse avec les décompositions tensorielles. TDM se positionne comme une surcouche adaptée aux *polystores*. En plus de cela, son implémentation bénéficie d'un typage sûr et d'un mécanisme d'inférence de schéma, ainsi que d'une optimisation poussée lui permettant de surpasser les librairies de l'état de l'art.

Le plan du chapitre est le suivant. La section 1 identifie les avantages des *polystores* dans un contexte de stockage et d'analyse de données, les capacités des tenseurs en tant que modèles pivots et outils analytiques, ainsi que les besoins de sûreté lors de la manipulation de données et l'exécution d'algorithmes d'analyse. La section 2 est un état de l'art divisé en trois parties : une sur les modèles pivots utilisés pour des systèmes de stockages hétérogènes, une deuxième sur l'objet mathématique tenseur et ses opérateurs les plus courants, et une dernière sur les librairies qui proposent une implémentation des tenseurs. La section 3 présente le modèle TDM ainsi que ses différents opérateurs de manipulation de données ou de décompositions tensorielles, et montre comment TDM peut jouer le rôle de modèle pivot par rapport aux modèles courants. La section 4 introduit les mécanismes utilisés pour obtenir un typage sûr et une inférence de schéma, et ce même lorsque les opérateurs induisent une modification du schéma. La section 5 détaille les techniques d'optimisations générales mises en œuvre au niveau de la librairie TDM, mais aussi d'autres techniques plus spécifiques pour améliorer grandement les performances de la décomposition CANDECOMP/PARAFAC. La section 6 conclut ce chapitre et identifie plusieurs perspectives de recherche.

3.1 Introduction

Les données massives, et plus particulièrement leur volume et leur variété, ont entraîné une diversification des besoins de stockage, qui se sont alors éloignés du classique et rigide *data warehouse*. L'approche *polystore* est un moyen de gérer l'hétérogénéité des les modèles de stockage, mais entraîne toutefois des difficultés lors des analyses, puisqu'il faut être capable d'alimenter les algorithmes en puisant les données dans différents systèmes. Les tenseurs sont des candidats idéaux pour servir de modèles pivots entre les modèles de ces systèmes variés. Cependant, leur définition mathématique n'est pas adaptée aux données, et les librairies existantes manquent de sûreté notamment pour éviter deux sortes de problèmes : des erreurs de typage pendant l'exécution et des ambiguïtés sur les expressions composées d'opérateurs de transformation de données. Ces dernières auront une influence majeure lors de l'interprétation des résultats et des prises de décision qui en découlent.

3.1.1 L'essor des *polystores*

La variété et le volume des données massives ont changé les besoins liés au stockage des données. Plutôt que d'avoir un unique Système de Gestion de Base de Données (SGBD) pour conserver l'ensemble des données, une multitude de systèmes de stockage spécialisés ont émergé, tels que les bases de données NewSQL, NoSQL, orientées colonnes, graphes, etc., chacun répondant à un cas d'utilisation plus ou moins spécifique. Les analyses se sont elles aussi diversifiées, et nécessitent désormais plusieurs types d'algorithmes s'appuyant sur différentes fondations théoriques, comme par exemple l'algèbre linéaire, les statistiques, la théorie des graphes. Les algorithmes sont implémentés avec différents paradigmes de calcul tels que *map-reduce*, GPU, concurrent ou parallèle, et sont utilisés en tant qu'opérateurs sur les données dans des *workflows* d'analyse. Avec l'adoption de modèles de données variés combinés aux *frameworks* d'analyse de données et aux nouvelles techniques telles que le *machine learning*, il est possible d'extraire plus d'informations des données et d'avoir une meilleure compréhension des phénomènes étudiés. Toutefois, les *workflows* de traitement de données deviennent complexes, et l'hétérogénéité n'est plus seulement observée au niveau des données, mais également au niveau des analyses [3].

Les processus *Extract Transform Load* (ETL), les *data warehouses* et le *in-database analytics* ne sont pas suffisants pour supporter les *workflows* complexes d'analyses :

- les processus ETL sont des tâches coûteuses et chronophages. Transformer de multiples jeux de données en un modèle de données unique peut impacter négativement les performances et réduire l'expressivité du modèle de données originel. Les processus ETL ne permettent pas non plus de traiter des données en temps réel et induisent un problème de fraîcheur des données ;
- les *data warehouses* se sont souvent vus reprocher leur manque de flexibilité. En effet, le schéma décidé à la création n'est pas facilement modifiable, et le modèle imposé ne permet pas de représenter toutes les subtilités présentes dans les données sources ;
- l'approche *in-database analytics*, qui consiste à intégrer directement dans le SGBD les outils d'analyse [109], ne peut pas prendre en compte facilement les nouveaux algorithmes, puisqu'ils nécessitent un développement spécifique pour chaque modèle de SGBD afin de faire correspondre le modèle de données à la structure de données attendue par l'algorithme [112, 113].

L'article bien connu de M. Stonebraker [161], "*one size does not fit all*", explique que le potentiel des données est mieux exploité avec une architecture *polystore*. Un *polystore* fait référence à un système qui intègre des SGBD et systèmes de stockage hétérogènes, et qui utilise des langages de manipulation de données ou de programmation variés [52]. L'utilisation d'un *polystore* permet non seulement d'organiser les données selon des cas d'utilisation particuliers (par exemple les SGBD graphes supportent bien les données liées ainsi que les requêtes sur les chemins dans les graphes), mais aussi de réaliser les traitements de données en parallèle sur plusieurs systèmes de stockage en fonction des spécificités de chacun de ces systèmes dans le *polystore* [96, 12].

Les recherches sur les *polystores* essaient de dépasser les limitations des outils traditionnels. Elles sont donc concentrées sur :

- les systèmes de *streaming* ETL [49, 169], qui bénéficient déjà d'une réduction de leur complexité grâce à la variété des modèles de données plus proches des cas d'usage. Le *streaming* ETL a pour objectif de se rapprocher d'une intégration des données en temps réel ;
- les langages de requêtes multi-SGBD [52], visant à proposer une interface unifiée facilitant l'interrogation des SGBD disponibles, et de ce fait à simplifier leur utilisation ;
- l'unification des modèles [53], proposant de charger les données sous un modèle commun lors de leur

exploitation, afin de faciliter leur manipulation lorsque plusieurs sources de données différentes sont utilisées ;
— le traitement en parallèle des requêtes et analyses [96], cherchant à optimiser l'exécution des traitement réalisés sur les *polystores* en profitant des caractéristiques particulières de ces derniers (par exemple l'efficacité d'un SGBD par rapport à un autre pour traiter un certain type de requête).

L'unification des modèles possède l'avantage de charger les données dans un modèle commun suffisamment flexible pour ensuite exécuter des analyses, facilitant de ce fait le développement des algorithmes d'analyse. Elle se positionne comme une surcouche vis-à-vis des *polystores*, qui peuvent de cette manière bénéficier des autres axes de recherche. Proposer un modèle pivot capable de généraliser les formats de données contenus dans un *polystore* et faciliter les transformations de données pour alimenter les différents algorithmes est donc un enjeu majeur [112, 83, 78].

3.1.2 Les tenseurs comme outils d'analyse

Les tenseurs, qui sont des objets mathématiques multi-dimensionnels abstraits et puissants, sont des candidats idéaux pour jouer le rôle de modèles pivots, grâce à leur capacité naturelle à représenter divers modèles de données et les relations complexes. En effet, on peut voir un tenseur comme une généralisation du modèle clé-valeur à n clés, comme le résultat d'une requête GROUP BY ou la projection d'un cube OLAP, ou encore comme la matrice d'adjacence d'un graphe, qui peut contenir plusieurs informations supplémentaires, telles que la temporalité des liens.

De plus, les tenseurs jouent un rôle important dans de nombreux *frameworks* et librairies d'analyse de données développés ces dernières années. Ils sont utilisés dans plusieurs catégories d'analyse de données [162], comme en *deep learning* pour traiter les données multi-dimensionnelles ou en *data mining* pour étudier les relations latentes grâce aux décompositions tensorielles [138, 95]. Grâce à cela, les tenseurs couvrent de nombreuses applications dans divers domaines, par exemple pour étudier le changement climatique [171] ou en recherche médicale avec les analyses d'images de l'activité cérébrale [10].

Toutefois, la définition mathématique pure des tenseurs est relativement abstraite, éloignée des besoins des utilisateurs. L'utilisation uniquement d'index numériques, pour représenter les dimensions mais aussi les éléments contenus dans les dimensions, ne permet pas d'identifier facilement les données réelles auxquelles ils font référence. Ils sont donc déconnectés des modèles, des schémas et des sources de données, et de ce fait échouent à tirer profit de l'expressivité des modèles et de leurs opérateurs de manipulation de données. Il devient alors nécessaire d'utiliser des structures intermédiaires afin de réaliser les manipulations complexes, dans le but de pouvoir ensuite appliquer les opérateurs tensoriels tels que les décompositions. Ce constat est d'autant plus vrai lorsque des sources de données multiples sont utilisées, et renforce le besoin d'adapter les tenseurs pour améliorer leur positionnement en tant que modèles pivots.

Leur implémentation dans les outils d'analyse ne suit pas les standards de programmation et entraîne de mauvaises habitudes, d'ordinaire exclues des bonnes pratiques du développement logiciel, réduisant les évolutions, la réutilisation, le travail collaboratif, etc. Dans les *frameworks* tels que Tensorflow, Theano ou NumPy, les tenseurs sont définis comme des tableaux multi-dimensionnels : les utilisateurs attribuent une signification implicite aux index numériques des dimensions, et au mieux commentent leur code pour se rappeler de la signification des constructions [153]. Ces mauvaises habitudes peuvent aussi concerner le nom ou l'ordre des dimensions après qu'une transformation ait été appliquée au tenseur. Cela peut mener à des erreurs difficilement identifiables car le calcul est techniquement correct mais fonctionnellement erroné.

3.1.3 L'importance de la sûreté des programmes d'analyse

Un modèle de données pivot qui permet de structurer les données d'un *polystore* en vue de leur analyse est un élément essentiel, comme le souligne Arun Kumar dans une présentation intitulée *The New DBfication of ML/AI : Saving Them from Themselves* lors de la conférence ICDE 2021 [1]. Il montre également que pour démocratiser l'utilisation des algorithmes de *machine learning* et d'IA, il est nécessaire de développer de nouveaux paradigmes et outils pour améliorer l'efficacité des systèmes et réduire les artefacts produits par les analystes.

De multiple *frameworks* d'analyse de données se basant sur divers paradigmes de programmation sont disponibles, tels que TensorFlow [1], PyTorch [140], Theano [150], TensorLy [98], NumPy [168] ou Spark [172] (avec SparkSQL, GraphX, MLlib). Ces *frameworks* aident à construire des *workflows* d'analyse de données en utilisant des techniques venant du *machine learning*, du *deep learning*, des réseaux complexes, etc. Toutefois, ils se concentrent souvent exclusivement sur l'application des algorithmes, et non sur la manipulation ou la transformation des données, même si c'est une tâche chronophage et sujette aux erreurs. Chacun de ces outils ne supporte pas une ou plusieurs des propriétés importantes jugées habituellement essentielles, comme les opérateurs à typage sûr qui peuvent garantir à la compilation qu'une composition d'opérations est valide (c'est une propriété qui est invariablement perdue avec le typage dynamique, menant à des erreurs durant l'exécution [153]), ou l'inférence de schéma permettant de déduire automatiquement le type du résultat.

En l'absence de ces propriétés, deux catégories principales d'erreurs peuvent survenir :

— les erreurs de type, lorsque les opérateurs ne sont pas appliqués sur les attributs du bon type ;
— les erreurs fonctionnelles, lorsque les opérateurs sont appliqués sur les attributs du bon type, mais pas sur ceux qui représentent l'information visée.

La deuxième catégorie d'erreurs est non seulement la plus difficile à détecter, mais aussi la plus dangereuse. En effet, l'erreur passe inaperçue, les traitements peuvent techniquement s'exécuter, mais le résultat produit sera incorrect. L'exemple de l'inversion de colonnes donné dans l'introduction générale est dû à ce type d'erreur, et entraîne des conséquences non négligeables.

Ces constatations renforcent le besoin de vérifier l'exactitude des *workflows* d'analyse [34], comme cela peut se voir dans l'évolution des structures de données de Spark : au départ les `RDD` représentaient des données non structurées, puis les `DataFrame` ont amené un format en colonnes, et enfin les `Dataset` rajoutent une couche typée aux `DataFrame`. De ce fait, les programmes d'analyse ont besoin d'un moyen de prévenir ce genre d'erreur, sans nécessiter une intervention de l'utilisateur ni ajouter de la complexité, afin que les *data scientists* puissent se concentrer davantage sur le cœur des analyses plutôt que sur le contrôle des résultats à chaque étape du *workflow*.

3.1.4 Contribution

Dans ce chapitre, je propose d'étendre la notion de tenseur avec TDM, pour en faire un modèle pivot, suffisamment flexible pour représenter des modèles de données variés, tels que le relationnel, les graphes ou les séries temporelles. En s'appuyant sur ses capacités de modèle pivot, TDM se positionne en tant que surcouche aux *polystores*, et ses opérateurs tensoriels en font un outil d'analyse idéal. Le modèle est accompagné d'opérateurs de manipulation de données, facilitant de ce fait l'utilisation des tenseurs par rapport au modèle mathématique d'origine. TDM a été implémenté sous la forme d'une librairie en libre

1. `http://cseweb.ucsd.edu/~arunkk/downloads/2021ICDERisingStarAwardTalkArunKumar.pdf`

accès [2], qui est centrée sur les données. Développée en Scala et Spark, la librairie propose des mécanismes de typage sûr et d'inférence de schéma, mis en place grâce aux fonctionnalités avancées de Scala. De plus, l'optimisation des différentes opérations disponibles dans la librairie permet de l'utiliser à large échelle, et le temps d'exécution de la décomposition CANDECOMP/PARAFAC est réduit de plusieurs ordres de magnitude par rapport aux librairies de l'état de l'art travaillant également à large échelle.

3.2 État de l'art : modèles pivots, tenseurs et librairies de tenseurs

Dans la suite de cette section, les modèles pivots qui ont été proposés pour les *polystores* sont étudiés, ensuite l'objet mathématique tenseur est introduit pour enfin passer en revue et classer les librairies qui proposent de supporter les tenseurs.

3.2.1 Les modèles pivots pour les données hétérogènes

Face à la variété de formats accompagnant les données massives, de nouveaux besoins ont émergé, incluant la prise en considération de cette variété, dans le but de traiter, manipuler, exploiter ou encore analyser les données sous-jacentes aux formats. On peut distinguer deux courants majeurs cherchant à répondre à ce besoin : l'approche langage et l'approche modèle. La première consiste à se servir d'un langage pour accéder à divers SGBD hétérogènes et à récupérer leurs données, tandis que la deuxième propose d'utiliser un modèle suffisamment flexible pour pouvoir intégrer la majorité des modèles et pouvoir ensuite s'en servir pour interagir avec les données.

L'approche langage s'apparente bien souvent aux *polystores*, et a pour objectif de proposer une interface commune pour gérer les SGBD sous-jacents aux *polystores*. Les travaux de recherche s'intéressent à faciliter la parallélisation des traitements lorsque les données de plusieurs SGBD doivent être rassemblées, à trouver des opérateurs pouvant tirer profit de la répartition des données dans différentes bases ou encore à obtenir une transparence à la localisation qui permet de choisir le SGBD donnant les meilleures performances en fonction des données et de la requête à exécuter.

LeanXcale [96] est basé sur le langage CloudMdsQL [97], qui permet d'interroger des *polystores* tout en laissant la liberté d'écrire des requêtes natives sur les SGBD concernés. Il propose de tirer profit du parallélisme inhérent à l'utilisation simultanée de plusieurs SGBD, mais également d'intégrer des optimisations qui améliorent les performances globales des requêtes, telles que le *bind join* qui consiste à réaliser une jointure entre un jeu de données de taille réduite et un autre jeu de données plus imposant, qui n'est pas chargé directement mais qui est filtré au préalable en fonction du jeu de données de taille réduite pour éviter de récupérer des données inutilisées par la suite.

BigDAWG [52] est un *polystore* cherchant à tirer profit de la diversité des formats de données, tout en évitant d'avoir à les charger dans un SGBD commun qui risque de réduire la performance et l'efficacité du requêtage. BigDAWG vise également la transparence à la localisation. Le système ne subit qu'un très léger *overhead* en comparaison de l'exécution des requêtes natives, et rejoint LeanXcale en profitant d'optimisations qui sont accessibles uniquement en utilisant simultanément plusieurs SGBD.

ESTOCADA [13] met en avant la transparence à la localisation et la sélection du SGBD permettant d'obtenir les meilleures performances lorsque les données recherchées sont répliquées dans plusieurs bases.

2. `https://github.com/AnnabelleGillet/TDM`

Il repose sur un système de vues matérialisées qui permet d'accéder efficacement aux données, et supporte de nombreux formats, comprenant les classiques relationnel et JSON, mais aussi graphe et matrices. ESTOCADA propose de s'utiliser par dessus un *polystore* existant tel que BigDAWG, ce qui permet de choisir le plus adapté en fonction des besoins.

Le *polystore* AWESOME (*Analytical Workbench for Exploration of SOcial MEdia*) [44] se concentre plus particulièrement sur les données sociales et leur variété. Il supporte les modèles relationnel, semi-structuré, graphe et texte, et met en avant l'analytique grâce à ADIL, le langage associé au *polystore* AWESOME qui permet de spécifier le placement des données mais également des résultats des traitements précédents afin d'en faciliter la réutilisation. Cela permet de tirer profit du résultat des analyses même si le format n'est pas identique aux données d'origine, et de s'en servir ensuite pour produire de nouvelles analyses directement à partir de ces résultats.

Maccioni et al. [118] utilisent le *query augmentation* pour contextualiser davantage les résultats de requêtes exécutées sur un *polystore* en allant chercher dans les différentes bases les informations liées à celles de la requête. De cette manière, l'intégration des données ainsi que la découverte d'informations sont facilitées. Ce système repose sur un index A^{+}, qui prend la forme d'un graphe, et qui est étendu par transitivité lorsqu'une nouvelle relation est insérée.

Hybrid.poly [144] propose une architecture utilisant une extension du langage SQL visant à rajouter des capacités de requêtes analytiques, comme par exemple en intégrant des algorithmes de *machine learning*. L'optimiseur de requêtes est capable de traiter des données hybrides en prenant aussi bien en compte des données relationnelles que multimédias, en passant par le XML et le JSON. Le but des auteurs est de faciliter à la fois l'utilisation des sources de données hétérogènes, mais aussi le développement des algorithmes d'analyse.

L'approche modèle vise à utiliser une structure intermédiaire permettant de représenter simplement la plupart des modèles existants, de manière à ensuite utiliser divers algorithmes d'analyse ou méthodes de manipulation directement sur ce modèle. On retrouve dans cette approche différents modèles tels que les tableaux multi-dimensionnels ou les *dataframes*.

Kuang et al. [102] décrivent un modèle de tenseur unifié pour représenter des données non structurées, semi-structurées et structurées, et proposent un opérateur d'extension de tenseur pour prendre en compte divers types de données sous la forme de sous-tenseurs fusionnés dans un tenseur unifié. Un opérateur de *Incremental High Order Singular Value Decomposition* (IHOSVD) est également présenté afin d'étendre la *High Order Singular Value Decomposition* (HOSVD) aux flux de données.

TensorDB [87, 85] est un système qui a pour objectif de supporter les tenseurs dans le modèle relationnel ; il s'agit d'une approche *in-database analytics*. TensorDB s'appuie sur SciDB comme système de stockage et propose des opérateurs de manipulation de données en mémoire, mais tire aussi profit du stockage sur disque pour calculer les décompositions, évitant ainsi le dépassement de mémoire causé par les explosions de données des structures intermédiaires volumineuses. Toutefois, cela ne peut se faire qu'au prix d'une augmentation importante du temps d'exécution. Un langage algébrique TRM (*Tensor Relationnal Model*) définit des opérateurs de décomposition et de jointure [86].

Lara [73, 74], fusion de *Linear Algebra* (LA) et *Relational Algebra* (RA), propose un modèle logique et une algèbre pour les *associative tables*, qui reposent sur la notion de tableau associatif, avec un ensemble d'opérations permettant d'unifier différents modèles de données tels que relationnel, tableau et clé-valeur. Les auteurs montrent comment utiliser Lara en tant que *middleware*, leur approche étant orientée vers la traduction et l'optimisation des opérateurs. Lara s'accompagne d'une implémentation basée sur Apache

Accumulo, qui présente de meilleures performances qu'une exécution standard avec *map-reduce* pour une multiplication de matrices ayant jusqu'à 10^6 éléments non-nuls.

Les travaux de Hutchinson, Kepner et al. [74, 78] définissent un modèle et des opérateurs pour les tableaux associatifs afin d'unifier l'algèbre relationnelle, les opérations sur les tableaux (ou autrement dit les matrices) et les algèbres clé-valeur. Les modèles LaraDB et D4M reposent essentiellement sur la notion de relation binaire. L'universalité des relations binaires permet de représenter presque tous les modèles. Cependant, les opérations de reconstruction de relations complexes sont coûteuses et ces modèles ne bénéficient pas du potentiel d'analyse des opérations de décomposition tensorielle.

TARS (*Typed ARray Schema*) [115] est un modèle basé sur les tableaux multi-dimensionnels, qui vise à représenter les données scientifiques, pour lesquelles le modèle classique relationnel est bien souvent un frein à leur format plus courant de tableaux. TARS intègre un système d'annotation sémantique à l'aide de rôles, dont leur association résulte en types qui contextualisent les données manipulées. TARS est accompagné d'une implémentation, SAVIME [114], écrite en C++.

Les *dataframes* [42, 126, 16] sont devenus incontournables au cours de ces dernières années. Ils sont disponibles dans de nombreux langages et *frameworks*, tels que R, Python et Spark. Ils adoptent un format colonnes, qui reste suffisamment flexible pour héberger toutes sortes de données tout en permettant d'ajouter une signification aux données en se servant de ces colonnes pour représenter les attributs. De part leur popularité, de nombreux algorithmes d'analyse ont été développés à partir de ce modèle, ce qui augmente leur versatilité.

L'approche langage et l'approche modèle n'abordent pas le problème de la même manière :

- l'approche langage vise à utiliser les SGBD déjà existants et à créer une surcouche permettant de les utiliser, dans le meilleur des cas en intégrant une transparence à la localisation afin de limiter l'impact de la complexité que la manipulation de différents SGBD pourrait entraîner du côté de l'utilisateur ;
- l'approche modèle de son côté cherche plutôt à trouver un modèle suffisamment flexible pour intégrer les données quelle que soit leur source, pour pouvoir ensuite les manipuler, les transformer et les analyser dans ce modèle.

Malgré ces différences, le but de ces approches reste commun, et consiste à simplifier l'utilisation et l'exploitation des sources de données hétérogènes sans pour autant perdre en expressivité par rapport au format d'origine, tout en facilitant les analyses et le développement de nouveaux algorithmes. L'approche modèle a toutefois l'avantage d'intégrer nativement les caractéristiques et fonctionnalités du modèle utilisé comme base, comme c'est le cas pour les tenseurs et leurs décompositions.

3.2.2 L'objet mathématique tenseur

Les tenseurs sont des objets mathématiques abstraits, qui peuvent être considérés selon différents points de vue tels que les applications multi-linéaires ou la généralisation des matrices à plus de deux dimensions. Nous définissons un tenseur comme un élément de l'ensemble des fonctions du produit de N ensembles $I_j, j = 1, \ldots, N$ vers $\mathbb{R}$: $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, avec N le nombre de dimensions du tenseur (aussi appelé ordre ou mode). La figure 3.1 illustre les tenseurs ayant un nombre différent de dimensions et le tableau 3.1 contient les notations utilisées dans cette thèse.

Les tenseurs sont dotés de plusieurs opérations, pour manipuler ou réarranger les données qu'ils contiennent, associer deux tenseurs avec une opération binaire ou les factoriser. Les travaux de Cichocki [36]

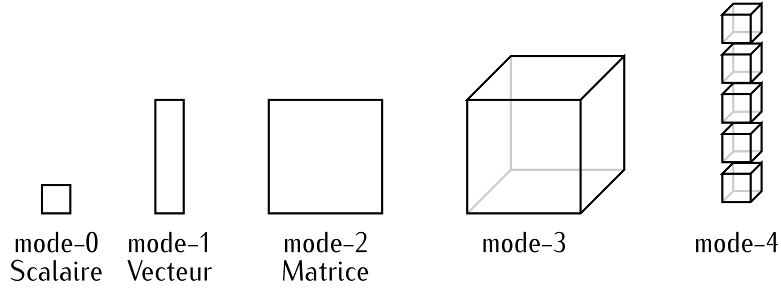


FIGURE 3.1 – Illustration des tenseurs de différents modes

Symbole	Définition	Symbole	Définition
$\mathcal{X}$	Un tenseur	$\otimes$	Produit de Kronecker
$\mathbf{M}$	Une matrice	$\circledast$	Produit de Hadamard
$\mathbf{v}$	Un vecteur colonne	$\oslash$	Division de Hadamard
a	Un scalaire	$\odot$	Produit de Khatri-Rao
$\mathcal{X}_{(n)}$	Matricisation d'un tenseur $\mathcal{X}$ sur le mode-n	$\circ$	Produit tensoriel (*outer product*)
$\times_n$	Produit mode-n	$\dagger$	Pseudo inverse

TABLE 3.1 – Symboles et opérateurs utilisés

et de Kolda [95] sont des références pour une explication complète des tenseurs et de leurs opérateurs. Dans la suite de cette section, les éléments nécessaires au développement du Tensor Data Model sont présentés.

3.2.2.1 Fibres et tranches

Une fibre est un fragment mono-dimensionnel d'un tenseur analogue à la notion de vecteur et aux lignes ou colonnes d'une matrice. Ainsi, pour un tenseur d'ordre 3 les fibres peuvent être des colonnes, des lignes ou des tubes notés respectivement : $\mathcal{X}_{:jk}$, $\mathcal{X}_{i:k}$ et $\mathcal{X}_{ij:}$. Une tranche de tenseur est un fragment d'ordre 2. Par exemple, pour un tenseur d'ordre 3, il est possible de définir des tranches horizontales, latérales et frontales notées respectivement : $\mathcal{X}_{i::}$, $\mathcal{X}_{:j:}$ et $\mathcal{X}_{::k}$. Que ce soit pour les fibres ou les tranches, il est courant de les nommer en fonction de leur dimension d'origine (mode-1, mode-2, mode-3, etc.).

3.2.2.2 Produit tensoriel (*outer product*)

Le produit tensoriel entre deux tenseurs consiste à former toutes les combinaisons possibles des produits entre tous les éléments. Un produit tensoriel entre deux vecteurs formera une matrice, tandis qu'un produit tensoriel entre deux matrices formera un tenseur d'ordre 4. D'une manière générale, le produit tensoriel entre un tenseur $\mathcal{Y} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ et un tenseur $\mathcal{X} \in \mathbb{R}^{J_1 \times J_2 \times \cdots \times J_M}$ noté $\mathcal{Y} \circ \mathcal{X}$ forme un tenseur $\mathcal{Z} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N \times J_1 \times J_2 \times \cdots \times J_M}$ dont les éléments sont égaux à :

$$\mathcal{Z}_{i_1 \times i_2 \times \cdots \times i_N \times j_1 \times j_2 \times \cdots \times j_M} = \mathcal{Y}_{i_1 \times i_2 \times \cdots \times i_N} \mathcal{X}_{j_1 \times j_2 \times \cdots \times j_M}$$

3.2.2.3 Produit mode-n

Le produit mode-n permet de multiplier une matrice ou un vecteur avec un tenseur. Dans le cas d'une matrice, le tenseur garde le même nombre de dimensions, mais la taille de la dimension n change en fonction de celle de la matrice. Pour un tenseur $\mathcal{X} \in \mathbb{R}^{J_1 \times J_2 \times \cdots \times J_n \times \cdots \times J_N}$ et une matrice $\mathbf{M} \in \mathbb{R}^{I_n \times J_n}$, le produit mode-$n$ résultera en un tenseur $\mathcal{Y} \in \mathbb{R}^{J_1 \times J_2 \times \cdots \times I_n \times \cdots \times J_N}$ dont les éléments sont égaux à :

$$\mathcal{Y}_{j_1,\ldots,i_n,\ldots,j_N} = \sum_{j_n=1}^{J_n} \mathcal{X}_{j_1,\ldots,i_n,\ldots,j_N} \mathbf{M}_{i_n,j_n}$$

Le produit mode-n entre un tenseur $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times I_n \times I_{n+1} \times \cdots \times I_N}$ et un vecteur $\mathbf{v} \in \mathbb{R}^{I_n}$ résulte en un tenseur $\mathcal{Y} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times I_{n+1} \times \cdots \times I_N}$ dont les éléments sont égaux à :

$$\mathcal{Y}_{i_1,\ldots,i_n,\ldots,i_N} = \sum_{i_n=1}^{I_n} \mathcal{X}_{i_1,\ldots,i_n,\ldots,j_N} \mathbf{v}_{i_n}$$

3.2.2.4 Matricisation

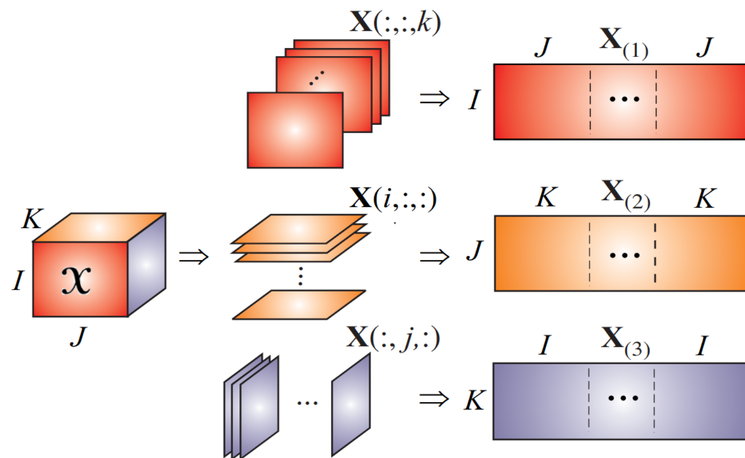


FIGURE 3.2 – Matricisation d'un tenseur (extrait de [36])

La matricisation sur le mode-n d'un tenseur $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ noté $\mathcal{X}_{(n)}$ produit une matrice $\mathbf{M} \in \mathbb{R}^{I_n \times \prod_{j \neq n} I_j}$ (voir figure 3.2). Une fois un tenseur matricisé, il est alors possible de lui appliquer des opérations matricielles.

3.2.2.5 Produit/division de Hadamard

Le produit de Hadamard de deux matrices ayant la même taille (le même nombre de colonnes et le même nombre de lignes) noté $\mathbf{A} \circledast \mathbf{B}$ est le produit terme à terme des matrices.

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} = \begin{bmatrix} 1 \times 5 & 3 \times 7 \\ 2 \times 6 & 4 \times 8 \end{bmatrix} = \begin{bmatrix} 5 & 21 \\ 12 & 32 \end{bmatrix}$$

3.2.2.6 Produit de Kronecker

Le produit de Kronecker entre une matrice $\mathbf{A} \in \mathbb{R}^{I \times J}$ et une matrice $\mathbf{B} \in \mathbb{R}^{K \times L}$ noté $\mathbf{A} \otimes \mathbf{B}$ résulte en une matrice $\mathbf{C} \in \mathbb{R}^{(IK) \times (JL)}$, dans laquelle tous les éléments de la matrice $\mathbf{A}$ sont multipliés par la matrice $\mathbf{B}$.

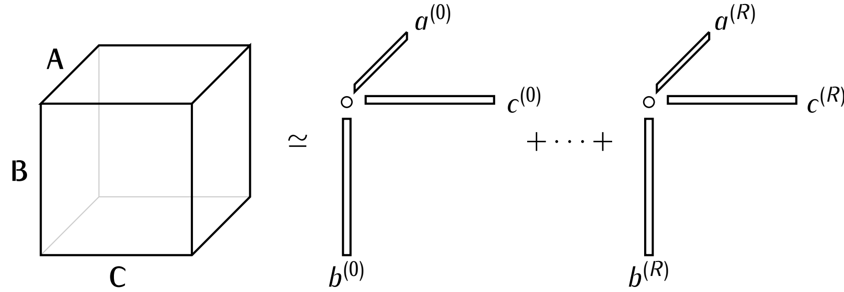
$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} = \begin{bmatrix} 1 \times 5 & 1 \times 7 & 3 \times 5 & 3 \times 7 \\ 1 \times 6 & 1 \times 8 & 3 \times 6 & 3 \times 8 \\ 2 \times 5 & 2 \times 7 & 4 \times 5 & 4 \times 7 \\ 2 \times 6 & 2 \times 8 & 4 \times 6 & 4 \times 8 \end{bmatrix} = \begin{bmatrix} 5 & 7 & 15 & 21 \\ 6 & 8 & 18 & 24 \\ 10 & 14 & 20 & 28 \\ 12 & 16 & 24 & 32 \end{bmatrix}$$

3.2.2.7 Produit de Khatri-Rao

Le produit de Khatri-Rao entre deux matrices ayant le même nombre de colonnes $\mathbf{A} \in \mathbb{R}^{I \times J}$ et $\mathbf{B} \in \mathbb{R}^{K \times J}$ noté $\mathbf{A} \odot \mathbf{B}$ est un produit de Kronecker colonne à colonne. Il produit une matrice $\mathbf{C} \in \mathbb{R}^{(IK) \times J}$.

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} = \begin{bmatrix} 1 \times 5 & 3 \times 7 \\ 1 \times 6 & 3 \times 8 \\ 2 \times 5 & 4 \times 7 \\ 2 \times 6 & 4 \times 8 \end{bmatrix} = \begin{bmatrix} 5 & 21 \\ 6 & 24 \\ 10 & 28 \\ 12 & 32 \end{bmatrix}$$

3.2.2.8 Décomposition CANDECOMP/PARAFAC



La décomposition CANDECOMP/PARAFAC (décomposition CP) permet de factoriser un tenseur en un ensemble de vecteurs [70, 148], qui est plus facilement exploitable que le tenseur en lui-même. À partir d'un tenseur $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ d'ordre N et d'un rang $R \in \mathbb{N}$, la décomposition CP factorise le tenseur $\boldsymbol{\mathcal{X}}$ en N matrices de facteurs avec colonnes normalisées $\mathbf{A}^{(i)} \in \mathbb{R}^{I_i \times R}$ pour $i = 1, \ldots, N$ et leur coefficient de normalisation $\lambda \in \mathbb{R}^R$ de la manière suivante :

$$\boldsymbol{\mathcal{X}} \simeq [\![\lambda, \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}]\!] = \sum_{r=1}^{R} \lambda_r a_r^{(1)} \circ a_r^{(2)} \circ \cdots \circ a_r^{(N)}$$

avec $a_r^{(i)}$ la colonne r de $\mathbf{A}^{(i)}$.

Plusieurs méthodes existent pour calculer la décomposition CP [156], qui peuvent être regroupées en deux catégories majeures : l'*Alternating Least Squares* (ALS) [70] ou la descente de gradient [119] moins gourmande en mémoire mais moins précise.

Algorithme 1 Décomposition CANDECOMP/PARAFAC — calculée avec la méthode *Alternating Least Squares*

Require: Tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and target rank R
1: Initialize $A^{(1)}, \ldots, A^{(N)}$, with $A^{(n)} \in \mathbb{R}^{I_n \times R}$
2: **repeat**
3: **for** $n = 1, \ldots, N$ **do**
4: $V \leftarrow A^{(1)T}A^{(1)} \circledast \ldots \circledast A^{(n-1)T}A^{(n-1)} \circledast A^{(n+1)T}A^{(n+1)} \circledast \ldots \circledast A^{(N)T}A^{(N)}$
5: $A^{(n)} \leftarrow \mathcal{X}_{(n)}(A^{(N)} \odot \cdots \odot A^{(n+1)} \odot A^{(n-1)} \odot \cdots \odot A^{(1)})V^{\dagger}$
6: normalize columns of $A^{(n)}$
7: $\lambda \leftarrow$ norms of $A^{(n)}$
8: **end for**
9: **until** $<$ convergence $>$

L'algorithme 1 présente les étapes générales pour calculer l'ALS. Son fonctionnement consiste à fixer toutes les matrices de facteurs sauf une, pour laquelle la solution optimale est recherchée, puis à remplacer la matrice de facteurs en question par la solution optimale, et recommencer la même procédure avec la matrice de facteurs suivante. Ces étapes sont réitérées jusqu'à ce qu'un critère de convergence satisfaisant soit atteint. La multiplication du tenseur matricisé de taille $\mathbb{R}^{I_n \times J}$, où $J = \Pi_{j \neq n} I_j$, avec le produit de Khatri-Rao de taille $\mathbb{R}^{J \times R}$ (ou *Matricized Tensor Times Khatri-Rao Product* abrégé en MTTKRP, ligne 5 de l'algorithme 1) est toutefois très gourmand en ressources, et nécessite beaucoup de mémoire pour stocker le résultat intermédiaire du produit de Khatri-Rao qui n'est utilisé que dans cette opération [143].

3.2.2.9 Exemples d'applications

Les tenseurs, et plus particulièrement leurs décompositions, sont utilisés dans des domaines variés. L'analyse des signaux du cerveau est un de ces domaines [39]. L'aspect multi-dimensionnel des tenseurs permet de modéliser les signaux en fonction de leur position dans le cerveau, mais aussi en fonction de différents paramètres, tels que le type de stimulation utilisé pour obtenir le signal. Les décompositions sont alors utilisées pour regrouper les patients ayant des signaux semblables, et ainsi aider au diagnostique médical.

Les décompositions tensorielles sont aussi utilisées pour réaliser du *clustering* non-supervisé. Gauvin et al. s'en sont servis pour regrouper des élèves en groupes en fonction de leurs interactions entre eux au cours de la journée [55]. Deux types de *clusters* sont détectés : les classes d'élèves et les regroupements qui se font lors des pauses. Araujo et al. appliquent une décomposition sur des échanges de mails avec le *Lawrence Berkeley National Laboratory* [15]. Ils parviennent à identifier des groupes de travail, uniquement à partir des adresses IP sources et destinations ainsi que du moment de l'envoi.

Les anomalies sont également bien détectées en se servant des décompositions tensorielles. Jeon et al. appliquent une décomposition sur un jeu de données comportant des appels entre deux individus [79]. Ils arrivent à faire émerger les télévendeurs, qui appellent beaucoup de personnes mais ne se font jamais appeler. Ces applications montrent le potentiel d'utilisation de la décomposition CANDECOMP/PARAFAC et la diversité de modélisation des données en entrée.

3.2.3 Les librairies de tenseurs existantes

Plusieurs librairies de tenseurs ont été proposées. Elles peuvent être classées en trois catégories : les librairies mathématiques, les librairies se servant des tenseurs en tant que moyen pour manipuler les données ou pour les passer d'une tâche à une autre, et les librairies spécialisées, qui n'implémentent que peu d'opérateurs tensoriels. Seule une partie des librairies existantes est présentée ici, Psarras et al. [146] ont récemment réalisé un recensement plus complet.

Les librairies mathématiques sont souvent développées en Matlab ou en R, et visent principalement à rendre disponible les opérateurs tensoriels, sans toutefois accorder une grande importance aux données en elles-mêmes. Ces librairies ont souvent des performances limitées lorsque la taille du tenseur devient importante.

rTensor [3] fournit les opérateurs standards pour manipuler des tenseurs en R, y compris les décompositions tensorielles, mais il ne supporte pas les tenseurs creux. Même s'il offre des performances correctes sur les petits tenseurs, il est vite dépassé lorsque la taille des dimensions augmente ou que l'ordre du tenseur est élevé.

Tensor Algebra Compiler (TACO) [90] possède des opérateurs optimisés en C++. Il couvre une large partie des opérateurs, de la multiplication entre un vecteur et une matrice jusqu'au plus complexe MTTKRP. C'est un librairie qui se sert de différents formats compressés pour optimiser les opérations, tels que le *Compressed Sparse Row* (CSR) ou le *Compressed Sparse Fiber* (CSF).

High-Performance Tensor Transpose [159] est une librairie en C++ permettant seulement de réaliser des transpositions sur des tenseurs, il lui manque donc de nombreux opérateurs utiles. En décomposant les tenseurs en "micro-kernels", c'est à dire en fragments d'un tenseur, et en exécutant la transposition sur chacun de ces fragments avant de reconstituer le tenseur final, des opportunités se créent pour paralléliser les traitements et appliquer des optimisations.

TensorToolbox [4] est une librairie Matlab réputée, née des travaux de Kolda. Elle propose de nombreux opérateurs tensoriels qui incluent les décompositions Tucker et CANDECOMP/PARAFAC, et propose aussi de nombreux formats de tenseurs denses et creux dont l'utilisateur peut se servir pour optimiser ses structures. Elle n'est cependant pas prévue pour les calculs à large échelle.

La deuxième catégorie de librairies tensorielles est celle permettant de manipuler et de passer les données d'une tâche à une autre. Cette catégorie est bien souvent utilisée dans les librairies orientées *machine learning*. Bien que ce soit la catégorie la plus connue, elle ne reflète pas réellement les capacités des tenseurs, et peut malheureusement entraîner de mauvaises compréhensions de l'objet mathématique tenseur.

TensorFlow [1] est une célèbre librairie Python, pensée pour exécuter différents modèles de *machine learning* et *deep learning*. Sa notoriété lui a permis de développer un écosystème allant de la création et de l'entraînement des modèles à la visualisation. Toutefois, c'est une librairie qui dispose de peu d'opérateurs tensoriels, et qui se sert des tenseurs uniquement pour passer les données d'une tâche à une autre. Elle se concentre sur le développement classique des méthodes de *machine learning*.

PyTorch [140] est également une librairie Python, développée par Facebook. Elle offre des fonctionnalités proches de celles de TensorFlow, et possède également les mêmes limites. PyTorch possède toutefois un éco-

3. `http://jamesyili.github.io/rTensor/`
4. `https://www.tensortoolbox.org/`

système moins complet que TensorFlow, mais offre une flexibilité plus grande en laissant plus de possibilités à l'utilisateur pour intervenir et modifier le *workflow* de calculs.

NumPy [168] est une autre librairie Python, qui s'affilie plus aux tableaux multi-dimensionnels qu'aux tenseurs. Cette librairie propose des opérateurs de manipulation de base pour ces tableaux. Elle sert plus de base pour d'autres librairies proposant des mécanismes de plus haut niveau, comme TensorFlow et PyTorch par exemple.

TensorLy [98], une librairie Python, permet de passer d'une librairie *backend* à une autre, en s'appuyant entre autres sur TensorFlow et PyTorch. Elle est cependant plus proche des librairies mathématiques que ces dernières, puisqu'elle dispose de plusieurs opérateurs tensoriels dont elles sont dépourvues, tels que les décompositions. C'est une librairie plutôt polyvalente, se rapprochant de la définition et des fonctionnalités des tenseurs.

La dernière catégorie est celle des librairies spécialisées, qui se servent des tenseurs uniquement pour implémenter un opérateur ou un algorithme spécifique. Certaines de ces librairies peuvent aussi s'exécuter à large échelle, souvent de manière distribuée. Nous nous concentrons sur celles implémentant les décompositions tensorielles.

HaTen2 [79] est une implémentation Hadoop des décompositions Tucker et CANDECOMP/PARAFAC se servant du paradigme *map-reduce*. Elle a ensuite été améliorée avec BigTensor [139]. Bien que ses performances soient aujourd'hui contestables, ce fut l'une des premières librairies à proposer les décompositions tensorielles sur des tenseurs volumineux. Toutefois, Hadoop est un choix assez limité pour charger les données puis exploiter les résultats, puisque le chargement doit s'effectuer depuis des fichiers stockés sur HDFS, et leur lecture nécessite une étape supplémentaire pour être exploités.

SamBaTen [63] propose de calculer la décomposition CANDECOMP/PARAFAC de manière incrémentale, afin de prendre en charge les tenseurs évolutifs. Il a tout d'abord été développé en Matlab, avant de proposer une version Spark pour traiter des tenseurs plus volumineux. Cependant, cette dernière est limitée puisqu'elle prend seulement en charge les tenseurs d'ordre 3, et la décomposition initiale, qui doit être calculée avant de pouvoir être incrémentée, représente un temps d'exécution long.

Gudibanda et al. [61] ont développé une librairie basée sur Spark permettant de calculer la décomposition CANDECOMP/PARAFAC sur des tenseurs couplés. Les tenseurs couplés sont des tenseurs ayant au moins une dimension en commun. Cette décomposition prend en compte ce point commun, et consiste à chercher des *patterns* en croisant les données des tenseurs disponibles. Cette décomposition est utile par exemple en ayant un tenseur représentant les achats faits par des utilisateurs, et un autre tenseur représentant les caractéristiques de ces utilisateurs.

ParCube [137] est une implémentation Matlab de la décomposition CANDECOMP/PARAFAC. Elle permet de paralléliser les opérations, mais ne peut pas être distribuée sur plusieurs machines. Bien qu'elle présente de bons résultats par rapport aux librairies classiques, la taille des tenseurs qu'elle peut prendre en charge reste limitée, tout comme le nombre de dimensions, puisqu'elle ne peut exécuter la décomposition que sur des tenseurs d'ordre 3 ou 4.

CSTF [23] se base sur Spark pour proposer une décomposition CANDECOMP/PARAFAC distribuée. Cependant, une vieille version de Spark est utilisée (version 1.6), et le *framework* a subi des changements majeurs depuis, qui incluent une modification du mécanisme qui était utilisé pour implémenter l'algorithme proposé. C'est une librairie qui peut donc être considérée comme obsolète.

Pour conclure, l'état de l'art des librairies de tenseurs montre que chaque catégorie possède ses propres limites :

— les librairies mathématiques sont souvent prévues pour des petits tenseurs, et prennent peu en compte les données qui peuvent être insérées dans les tenseurs ainsi que l'interprétation des résultats obtenus après application des opérateurs ;
— les librairies permettant de manipuler et de passer les données d'une tâche à une autre n'ont qu'une vision limitée des tenseurs, et n'incorporent pas toutes les fonctionnalités qu'ils peuvent proposer. Ce sont des librairies qui se servent plus de tableaux multi-dimensionnels que de tenseurs, bien qu'elles s'en soient approprié le nom ;
— les librairies spécialisées n'implémentent qu'un petit nombre d'opérateurs, et sont souvent à l'état de prototypes, qui peuvent difficilement être utilisés dans un environnement d'analyse standard.

On peut aussi distinguer des limites communes à l'ensemble de ces catégories. En effet, peu d'efforts sont consacrés à rendre les librairies de tenseurs utilisables en gardant les données au cœur des opérations. Les tenseurs doivent être manipulés à travers leurs indices, que ce soit pour les dimensions ou pour les valeurs des dimensions, ce qui augmente considérablement le risque d'erreurs.

Les librairies spécialisées, et quelquefois celles servant à manipuler les données, se préoccupent parfois d'exécuter des opérateurs à large échelle, mais cela reste encore marginal. Pourtant, de par leur nombre non limité de dimensions, les tenseurs peuvent représenter bien plus d'éléments que les matrices. En considérant la taille croissante des jeux de données, il est donc important de prendre en compte cette caractéristique dans le développement de librairies.

3.2.4 Synthèse

Avec l'étude de l'état de l'art, on constate que, parmi les approches modèle visant à prendre en considération la variété des formats de données, celles étant basées sur les tableaux multi-dimensionnels ou les tenseurs se démarquent. En effet, modéliser les données en intégrant cette multi-dimensionnalité permet de représenter la plupart des données existantes, y compris les données complexes et scientifiques.

Les tenseurs, contrairement aux tableaux, possèdent en plus des opérateurs très intéressants pour les analyses : les décompositions tensorielles. Ces dernières peuvent être vues comme la généralisation de la SVD à n dimensions, dépassant ainsi la limite des deux dimensions des matrices, ce qui est utile pour l'étude des relations complexes.

Plusieurs librairies proposent d'implémenter les tenseurs, et se divisent en trois catégories : les librairies mathématiques, les librairies de manipulation de données et les librairies spécialisées. Cependant, peu de librairies mettent les données au centre de leurs préoccupations, bien que cela soit essentiel pour comprendre et interpréter les manipulations de données et les analyses. Un besoin de développer un modèle basé sur les tenseurs émerge, afin de faciliter l'intégration des données et leur manipulation.

3.3 Tensor Data Model et opérateurs

Les tenseurs étant des objets mathématiques polyvalents et puissants, ils sont tout indiqués pour servir de modèles pivots et réaliser des analyses avancées sur les données. Toutefois, sous leur forme mathématique accompagnée de leur indexation fonctionnant uniquement avec des entiers, ils ne sont pas adaptés pour manipuler des données. Dans cette section, nous proposons le Tensor Data Model (TDM), qui permet de

rajouter un schéma de données ainsi que des opérateurs de manipulation pour rendre les tenseurs plus à même de gérer des données. L'architecture générale de TDM est donnée dans la figure 3.3.

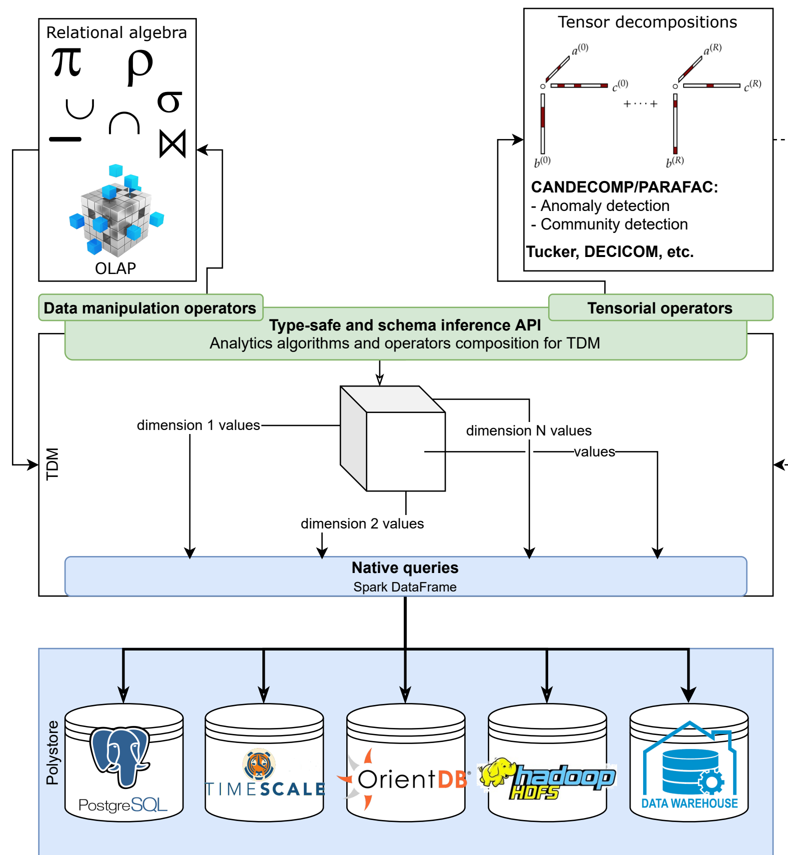


FIGURE 3.3 – L'architecture de TDM

3.3.1 Le schéma

Afin de donner une signification aux dimensions qui puisse être plus évoluée que celle obtenue uniquement avec des entiers simples, les capacités des tenseurs doivent être étendues pour représenter des valeurs plus variées, telles que des chaînes de caractères ou des décimaux. Pour cela, des tableaux associatifs sont utilisés.

Définition 3.3.1 : Tableau associatif

Un **tableau associatif** est une application A qui associe des clés à une valeur, telle que $A : K_1 \times \cdots \times K_N \rightarrow \mathbb{V}$ où $K_i, i = 1, .., N$ sont des ensembles de clés et $\mathbb{V}$ est l'ensemble des valeurs.

Les tableaux associatifs servent assez classiquement pour représenter les tenseurs creux. Pour cela, en considérant un tenseur d'ordre N, il suffit d'associer la combinaison des indices des dimensions à la valeur correspondante, à l'aide d'un tableau associatif $A_{dv} : K_1 \times \cdots \times K_N \rightarrow \mathbb{V}$.

Afin de rajouter un schéma de données aux tenseurs, il faut principalement agir au niveau des dimensions. Pour cela, les tableaux associatifs ont besoin de porter une information supplémentaire : celle du type de la valeur significative de la dimension. Cette information est contenue dans des tableaux associatifs typés nommés.

Définition 3.3.2 : Tableau associatif typé nommé

Un **tableau associatif typé nommé** est un triplet (Nom, A, T) où Nom est une chaîne de caractères unique, A est le tableau associatif et T la signature de type du tableau associatif, c'est-à-dire $K \rightarrow \mathbb{N}$. Le schéma du tableau associatif typé nommé est $Nom : K$.

Ainsi, chaque dimension est représentée par un tableau associatif typé nommé permettant de passer de la valeur significative à l'entier servant à indexer les valeurs du tenseur. Dom_{Nom} est le domaine des valeurs prises par les clés de A, c'est-à-dire un sous-ensemble de K. Par exemple, le schéma du tableau associatif $User$ est $User : String$. Il désigne la dimension d'un tenseur qui représente les noms d'utilisateurs en leur associant une valeur unique dans $\mathbb{N}$, Dom_{User} est le sous-ensemble des noms des utilisateurs.

À partir des tableaux associatifs typés nommés des dimensions, il est possible d'étendre la définition aux tenseurs pour obtenir des tenseurs typés nommés, et intégrer la notion de schéma de données au cœur des tenseurs.

Définition 3.3.3 : Tenseur typé nommé

Un **tenseur typé nommé** est un quadruplet (Nom, D, V, T) où Nom est le nom du tenseur, D est une liste de tableaux associatifs typés nommés (un par dimension), V est un tableau associatif qui stocke les valeurs du tenseur et T est le type du tenseur, c'est-à-dire le type de ses valeurs. Le schéma d'un tenseur typé nommé est de la forme $Nom(S) : T$ où S est la liste des schémas des dimensions, c'est-à-dire des tableaux associatifs de D.

$Dom_{\mathcal{X}}$ est le sous-ensemble contenant les valeurs d'un tenseur $\mathcal{X}$. Le tableau associatif qui stocke ces valeurs est de la forme $k_{d_1} \times \cdots \times k_{d_N} \rightarrow v$, avec $k_{d_1} \in Dom_{d_1}, \ldots, k_{d_N} \in Dom_{d_N}$ et $v \in Dom_{\mathcal{X}}$. Les valeurs nulles du tenseur ne sont pas représentées directement, et elles n'ont pas d'entrée dans le tableau associatif des valeurs. Donc, $K_{\mathcal{X}} \subseteq Dom_{d_1} \times \cdots \times Dom_{d_N}$, avec $K_{\mathcal{X}}$ représentant l'ensemble des clés du tableau associatif, c'est-à-dire un ensemble de n-uplets.

Par exemple, si le tenseur $\mathcal{UHT}$ représente le nombre de fois où un hashtag a été publié par un utilisateur durant une tranche horaire, son schéma sera :

$$\mathcal{UHT}(U : String, H : String, T : Integer) : Integer$$

et le tableau associatif des valeurs du tenseur prendra la forme $u \times h \times t \rightarrow v$ avec $u \in Dom_U, h \in Dom_H, t \in Dom_T$ et $v \in Dom_{UHT}$.

Remarque 3.3.1

Avec TDM, l'ordre des dimensions n'a pas d'importance. Un tenseur $\mathcal{UHT}(U : String, H : String, T : Integer) : Integer$ et un tenseur $\mathcal{UTH}(U : String, T : Integer, H : String) : Integer$ ont un schéma équivalent.

3.3.2 Les opérateurs de TDM

Maintenant que la notion de schéma est ajoutée aux tenseurs, des opérateurs de manipulation de données sont définis, proches de ceux de l'algèbre relationnelle. Ils permettent de mettre en forme les tenseurs ou de les associer pour ensuite se servir des algorithmes analytiques sans avoir à utiliser un ou plusieurs modèles intermédiaires.

Cet ensemble d'opérateurs s'applique sur des tenseurs typés nommés à deux niveaux différents : au niveau des tableaux associatifs, c'est-à-dire du schéma, et au niveau des valeurs des tenseurs. Nous nous concentrons sur le sous-ensemble d'opérateurs suivants : projection, sélection, restriction, union, intersection, jointure naturelle, différence et certains opérateurs analytiques tels que la décomposition de tenseurs. Ces derniers peuvent produire plusieurs tenseurs, le plus souvent d'ordre inférieur.

Les définitions proposées dans la suite de cette section sont inspirées du formalisme de Kanellakis [81] utilisé pour définir le modèle relationnel. $\alpha(\boldsymbol{\mathcal{X}})$ désigne le schéma formel du tenseur typé nommé $\boldsymbol{\mathcal{X}}$. À un niveau plus détaillé, α_T représente le type du tenseur et α_D la liste des dimensions. Lorsque $\alpha(\boldsymbol{\mathcal{X}}_1) = \alpha(\boldsymbol{\mathcal{X}}_2)$, alors $\alpha_T(\boldsymbol{\mathcal{X}}_1) = \alpha_T(\boldsymbol{\mathcal{X}}_2)$ et $\alpha_D(\boldsymbol{\mathcal{X}}_1) = \alpha_D(\boldsymbol{\mathcal{X}}_2)$. Les définitions sont construites de la manière suivante :

- la clause (1) décrit le comportement de l'opérateur sur le schéma α, c'est-à-dire la spécification du schéma résultat ;
- la clause (2) donne la sémantique de l'opérateur sur les valeurs, et donc sur le contenu du tableau associatif V stockant les valeurs du tenseur.

Remarque 3.3.2

Concernant les clés $K_{\boldsymbol{\mathcal{X}}_1}$ d'un tenseur $\boldsymbol{\mathcal{X}}_1$, on considère qu'elles sont égales aux clés $K_{\boldsymbol{\mathcal{X}}_2}$ d'un tenseur $\boldsymbol{\mathcal{X}}_2$ seulement si les dimensions et les valeurs des dimensions sont les mêmes.
Les clés individuelles sont désignées par $k_{\boldsymbol{\mathcal{X}}_1} \in K_{\boldsymbol{\mathcal{X}}_1}$ et $k_{\boldsymbol{\mathcal{X}}_2} \in K_{\boldsymbol{\mathcal{X}}_2}$, tandis que $k_{\boldsymbol{\mathcal{X}}_1 \cap \boldsymbol{\mathcal{X}}_2}$ forme une clé individuelle uniquement avec les dimensions en commun dans les deux tenseurs, et $k_{\boldsymbol{\mathcal{X}}_1 \cup \boldsymbol{\mathcal{X}}_2}$ forme une clé individuelle avec toutes les dimensions des deux tenseurs sans duplication.
Par exemple, si on a $\boldsymbol{\mathcal{X}}_1(d1 : Integer, d2 : String) : Integer$ et $\boldsymbol{\mathcal{X}}_2(d1 : Integer, d3 : Integer) : Integer$, alors $k_{\boldsymbol{\mathcal{X}}_1 \cap \boldsymbol{\mathcal{X}}_2}$ représente $k_{\{d_1\}}$ et $k_{\boldsymbol{\mathcal{X}}_1 \cup \boldsymbol{\mathcal{X}}_2}$ représente $k_{\{d_1, d_2, d_3\}}$, $k_{\{d_i\}}$ étant la partie de la clé correspondant à la dimension d_i.

3.3.2.1 Opérateurs de manipulation de données

L'opérateur de projection permet de supprimer une dimension du tenseur, en se concentrant sur une seule valeur de cette dimension. L'opérateur de projection peut être réalisé en utilisant le produit mode-n entre un vecteur booléen qui contient 1 pour la valeur du mode (dimension) que l'on veut sélectionner : $\boldsymbol{\mathcal{X}} \times_n \mathbf{b}$.

Définition 3.3.4 : Projection

La **projection** d'un tenseur typé nommé $\boldsymbol{\mathcal{X}}$ d'ordre N, notée $\pi_{[expr]}\boldsymbol{\mathcal{X}}$, où $expr$ est un couple (d_i, c) avec $d_i \in \alpha_D$ la dimension qui sera supprimée et $c \in Dom_{d_i}$ une constante représentant la valeur de la dimension en question à conserver, réduit les dimensions du tenseur en conservant les valeurs associées aux autres dimensions.

(1) $\alpha_T(\pi_{[expr]}\boldsymbol{\mathcal{X}}) = \alpha_T(\boldsymbol{\mathcal{X}})$ et $\alpha_D(\pi_{[expr]}\boldsymbol{\mathcal{X}}) = \alpha_D(\boldsymbol{\mathcal{X}}) - \{d_i\}$

(2) $\pi_{[expr]}\boldsymbol{\mathcal{X}} = \{k \to v$, avec $k = k_{\boldsymbol{\mathcal{X}}} - k_{\boldsymbol{\mathcal{X}}\{d_i\}}, k_{\boldsymbol{\mathcal{X}}} \in K_{\boldsymbol{\mathcal{X}}}, v \in Dom_{\boldsymbol{\mathcal{X}}}, k_{\boldsymbol{\mathcal{X}}} \to v \in V_{\boldsymbol{\mathcal{X}}}$, tel que $k_{\boldsymbol{\mathcal{X}}\{d_i\}} = c\}$

Pour illustrer l'opérateur de projection, reprenons le tenseur $\boldsymbol{\mathcal{UHT}}$ d'ordre 3 (voir p.65). En se servant d'une projection, nous souhaitons obtenir le nombre de hashtags publiés par tranche horaire par l'utilisateur $u1$ avec $\pi_{[U,'u1']}\boldsymbol{\mathcal{UHT}}$. Le résultat est un tenseur d'ordre 2 (tableau 3.2), de schéma $\boldsymbol{\mathcal{HT}}_{u1}(H : String, T : Integer) : Integer$ et dont l'ensemble des valeurs est égal à $\boldsymbol{\mathcal{UHT}} \times_1 \mathbf{b}$ avec $b_i = 1$ si $i = A_U(u1)$ et $b_i = 0$ si $i \neq A_U(u1)$, A_U étant le tableau associatif de la dimension U. Le produit mode-1 est utilisé car U est la première dimension dans la représentation physique du tenseur $\boldsymbol{\mathcal{UHT}}$.

User	Hashtag	Time	Value
u1	h1	1	5
u1	h2	2	2
u2	h1	3	11
u2	h2	4	3

→

Hashtag	Time	Value
h1	1	5
h2	2	2

TABLE 3.2 – Exemple de l'application de l'opérateur de projection sur le tenseur $\boldsymbol{\mathcal{UHT}}$: $\pi_{[U,'u1']}\boldsymbol{\mathcal{UHT}}$

L'opérateur de sélection permet de filtrer les valeurs du tenseur en conservant uniquement celles qui satisfont l'expression souhaitée.

Définition 3.3.5 : Sélection sur les valeurs d'un tenseur

L'opérateur de **sélection** noté $\sigma_{[expr]}\boldsymbol{\mathcal{X}}$ crée un nouveau tenseur de même schéma et de population égale à l'ensemble des valeurs du tenseur qui satisfont $expr$ [a].

(1) $\alpha(\sigma_{[expr]}\boldsymbol{\mathcal{X}}) = \alpha(\boldsymbol{\mathcal{X}})$

(2) $\sigma_{[expr]}\boldsymbol{\mathcal{X}} = \{k \to v$, tel que $expr(v)$, avec $k \in K_{\boldsymbol{\mathcal{X}}}, v \in Dom_{\boldsymbol{\mathcal{X}}}, k \to v \in V_{\boldsymbol{\mathcal{X}}}\}$

a. $expr$ est une expression logique pour comparer les valeurs de $\boldsymbol{\mathcal{X}}$ à des constantes. Sa forme est la suivante :
expr : := <condition> | <condition> <opérateur logique> <condition> | ¬ <condition> | (<condition>)
Les opérateurs logiques sont $\{\wedge, \vee\}$
<condition> : := valeurs de $\boldsymbol{\mathcal{X}}$ (implicite) < opérateur de comparaison > constante
Les opérateurs de comparaison sont $\{<, \leq, =, \neq, \geq, >\}$.

Par exemple, l'expression $\sigma_{[>10]}\boldsymbol{\mathcal{UHT}}$ sélectionne les utilisateurs, les hashtags et les tranches de temps durant lesquelles un utilisateur a employé au moins 10 fois un hashtag (tableau 3.3).

User	Hashtag	Time	Value
u1	h1	1	5
u1	h2	2	2
u2	h1	3	11
u2	h2	4	3

→

User	Hashtag	Time	Value
u2	h1	3	11

TABLE 3.3 – Exemple de l'application de l'opérateur de sélection sur le tenseur $\mathcal{UHT}$: $\sigma_{[>10]}\mathcal{UHT}$

Une autre manière de filtrer les données d'un tenseur est de le faire par rapport aux valeurs des dimensions. Pour cela, nous définissons l'opérateur de restriction.

Définition 3.3.6 : Restriction sur les valeurs des dimensions

L'opérateur de **restriction** noté $\rho_{[expr]}\mathcal{X}$ crée un tenseur de même schéma et de population égale aux valeurs correspondant aux clés des dimensions sélectionnées par $expr$ [a]. L'opérateur de restriction ne modifie pas le schéma du tenseur $\mathcal{X}$ mais il peut réduire la taille des dimensions.

(1) $\alpha(\rho_{[expr]}\mathcal{X}) = \alpha(\mathcal{X})$

(2) $\rho_{[expr]}\mathcal{X} = \{k \rightarrow v$, tel que $expr(k)$, avec $k \in K_{\mathcal{X}}, v \in Dom_{\mathcal{X}}, k \rightarrow v \in V_{\mathcal{X}}\}$

a. $expr$ permet de comparer les clés des dimensions à des constantes. Sa forme est la même que pour l'opérateur σ sauf pour <condition> : := nom d'une dimension < opérateur de comparaison > constante.

Par exemple, si l'on souhait conserver uniquement le hashtag h1 et l'utilisateur u1, on utilisera la restriction avec $\rho_{[U='u1' \wedge H='ht1']}\mathcal{UHT}$ (tableau 3.4).

User	Hashtag	Time	Value
u1	h1	1	5
u1	h2	2	2
u2	h1	3	11
u2	h2	4	3

→

User	Hashtag	Time	Value
u1	h1	1	5

TABLE 3.4 – Exemple de l'application de l'opérateur de restriction sur le tenseur $\mathcal{UHT}$: $\rho_{[U='u1' \wedge H='ht1']}\mathcal{UHT}$

Certaines manipulations de données requièrent des opérateurs binaires. C'est le cas de l'union, qui sert à combiner deux tenseurs de même schéma afin d'obtenir la fusion de leurs clés et valeurs associées dans un nouveau tenseur.

Définition 3.3.7 : Union

L'opérateur d'**union** noté $\cup_f$ appliqué à deux tenseurs typés nommés $\mathcal{X}_1$ et $\mathcal{X}_2$ de même schéma, pour lequel $f(T_{\mathcal{X}_1}, T_{\mathcal{X}_2}) : T$ [a], produit un tenseur $\mathcal{X}_3$ de type T ayant les mêmes dimensions que $\mathcal{X}_1$ et $\mathcal{X}_2$, dont l'ensemble des clés des dimensions est l'union des ensembles des clés des dimensions des opérandes : $K_{\mathcal{X}_3} = K_{\mathcal{X}_1} \cup K_{\mathcal{X}_2}$. Les valeurs de $\mathcal{X}_3$ sont les valeurs provenant de $\mathcal{X}_1$ et $\mathcal{X}_2$, excepté pour les clés en commun pour lesquelles la fonction f est appliquée.

(1) $\alpha_T(\mathcal{X}_1 \cup_f \mathcal{X}_2) = T$ et $\alpha_D(\mathcal{X}_1 \cup_f \mathcal{X}_2) = \alpha_D(\mathcal{X}_1) = \alpha_D(\mathcal{X}_2)$

(2) $\mathcal{X}_1 \cup_f \mathcal{X}_2 = \{k \rightarrow v, \text{ avec } k \in K_{\mathcal{X}_1} \cup K_{\mathcal{X}_2}$

$$v = \begin{cases} v_1, & \text{si } \exists k \rightarrow v_1 \text{ avec } k \rightarrow v_1 \in V_{\mathcal{X}_1}, k \notin K_{\mathcal{X}_2} \\ v_2, & \text{si } \exists k \rightarrow v_2 \text{ avec } k \rightarrow v_2 \in V_{\mathcal{X}_2}, k \notin K_{\mathcal{X}_1} \\ f(v_1, v_2), & \text{si } \exists k \rightarrow v_1 \text{ et } \exists k \rightarrow v_2 \text{ avec } k \rightarrow v_1 \in V_{\mathcal{X}_1} \text{ et } k \rightarrow v_2 \in V_{\mathcal{X}_2} \end{cases}$$

$\}$

a. Par exemple, $f(x_1, x_2)$ peut représenter $min(x_1, x_2)$, $x_1 + x_2$, etc.

L'union peut être utilisée par exemple lorsque les données sont situées dans deux bases de données différentes et que l'on souhaite les regrouper, ou alors en reprenant l'exemple du tenseur $\mathcal{UHT}$, les utilisateurs qui auraient utilisé des hashtags sur deux réseaux sociaux différents. Dans ce cas, les valeurs sont regroupées dans un même tenseur, comme illustré dans le tableau 3.5.

$\mathcal{UHT}_1$				$\mathcal{UHT}_2$			
User	Hashtag	Time	Value	User	Hashtag	Time	Value
u1	h1	1	5	u2	h2	4	8
u1	h2	2	2	u3	h1	1	1
u2	h1	3	11	u1	h3	3	2
u2	h2	4	3	u3	h2	5	5

↓

User	Hashtag	Time	Value
u1	h1	1	5
u1	h2	2	2
u2	h1	3	11
u2	h2	4	11
u3	h1	1	1
u1	h3	3	2
u3	h2	5	5

Table 3.5 – Exemple de l'application de l'opérateur d'union sur les tenseurs $\mathcal{UHT}_1$ et $\mathcal{UHT}_2$: $\mathcal{UHT}_1 \cup_f \mathcal{UHT}_2$, avec $f(x_{\mathcal{UHT}_1}, x_{\mathcal{UHT}_2}) = x_{\mathcal{UHT}_1} + x_{\mathcal{UHT}_2}$

Un autre opérateur binaire proche de l'union est l'intersection. Il permet lui aussi de fusionner deux tenseurs, mais cette fois en conservant uniquement les clés présentes dans les deux tenseurs.

Définition 3.3.8 : Intersection

L'opérateur d'**intersection** noté $\cap_f$ appliqué à deux tenseurs typés nommés $\mathcal{X}_1$ et $\mathcal{X}_2$ de même schéma, pour lequel $f(T_{\mathcal{X}_1}, T_{\mathcal{X}_2}) : T$, produit un tenseur $\mathcal{X}_3$ de type T ayant les mêmes dimensions que $\mathcal{X}_1$ et $\mathcal{X}_2$, dont l'ensemble des clés des dimensions est l'intersection des ensembles des clés des dimensions des opérandes : $K_{\mathcal{X}_3} = K_{\mathcal{X}_1} \cap K_{\mathcal{X}_2}$. Les valeurs de $\mathcal{X}_3$ sont les valeurs provenant de $\mathcal{X}_1$ et $\mathcal{X}_2$ sur lesquelles la fonction f est appliquée.

(1) $\alpha_T(\mathcal{X}_1 \cap_f \mathcal{X}_2) = T$ et $\alpha_D(\mathcal{X}_1 \cap_f \mathcal{X}_2) = \alpha_D(\mathcal{X}_1) = \alpha_D(\mathcal{X}_2)$

(2) $\mathcal{X}_1 \cap_f \mathcal{X}_2 = \{k \rightarrow v$, avec $k \in K_{\mathcal{X}_1} \cap K_{\mathcal{X}_2}$
$v = f(v_1, v_2)$, si $\exists k \rightarrow v_1$ et $\exists k \rightarrow v_2$ avec $k \rightarrow v_1 \in V_{\mathcal{X}_1}$ et $k \rightarrow v_2 \in V_{\mathcal{X}_2}$
$\}$

On peut se servir de l'opérateur d'intersection pour ne conserver les valeurs que si elles sont présentes dans les deux tenseurs. Si on reprend l'exemple précédent des utilisateurs ayant émis des hashtags sur deux réseaux sociaux différents, on peut vouloir étudier les utilisateurs uniquement s'ils sont actifs sur les deux réseaux. Le tableau 3.6 montre un exemple d'application de l'intersection.

$\mathcal{UHT}_1$				$\mathcal{UHT}_2$			
User	Hashtag	Time	Value	User	Hashtag	Time	Value
u1	h1	1	5	u2	h2	4	8
u1	h2	2	2	u3	h1	1	1
u2	h1	3	11	u1	h3	3	2
u2	h2	4	3	u3	h2	5	5

↓

User	Hashtag	Time	Value
u2	h2	4	11

TABLE 3.6 – Exemple de l'application de l'opérateur d'intersection sur les tenseurs $\mathcal{UHT}_1$ et $\mathcal{UHT}_2$: $\mathcal{UHT}_1 \cap_f \mathcal{UHT}_2$, avec $f(x_{\mathcal{UHT}_1}, x_{\mathcal{UHT}_2}) = x_{\mathcal{UHT}_1} + x_{\mathcal{UHT}_2}$

La jointure naturelle est un opérateur binaire travaillant sur des tenseurs de schémas différents, à condition qu'ils aient au moins une dimension en commun. Le schéma du tenseur résultat est alors l'union des schémas des tenseurs sur lesquels la jointure naturelle est appliquée.

Définition 3.3.9 : Jointure naturelle

La **jointure naturelle** entre deux tenseurs typés nommés $\mathcal{X}_1$ et $\mathcal{X}_2$ qui possèdent au moins une dimension en commun est notée $\bowtie_f$, pour laquelle $f(T_{\mathcal{X}_1}, T_{\mathcal{X}_2}) : T$, et produit un tenseur typé nommé $\mathcal{X}_3$ de type T dont les dimensions sont l'union des dimensions des deux tenseurs. Les clés retenues sur la ou les dimension(s) en commun sont celles qui sont identiques. Les valeurs de $\mathcal{X}_3$ sont celles provenant de $\mathcal{X}_1$ et $\mathcal{X}_2$ sur lesquelles la fonction f est appliquée.

(1) $\alpha_T(\mathcal{X}_1 \bowtie_f \mathcal{X}_2) = T$ et $\alpha_D(\mathcal{X}_1 \bowtie_f \mathcal{X}_2) = \alpha_D(\mathcal{X}_1) \cup \alpha_D(\mathcal{X}_2)$

(2) $\mathcal{X}_1 \bowtie_f \mathcal{X}_2 = \{k \rightarrow v$, avec $k = k_{\mathcal{X}_1} \cup k_{\mathcal{X}_2}, k_{\mathcal{X}_1} \in K_{\mathcal{X}_1}$ et $k_{\mathcal{X}_2} \in K_{\mathcal{X}_2}$
tels que $k_{\mathcal{X}_1 \cap \mathcal{X}_2} \subseteq k_{\mathcal{X}_1}, k_{\mathcal{X}_1 \cap \mathcal{X}_2} \subseteq k_{\mathcal{X}_2}$ et $k_{\mathcal{X}_1 \cap \mathcal{X}_2} \subseteq k$
$v = f(v_1, v_2)$, avec $k_{\mathcal{X}_1} \rightarrow v_1 \in V_{\mathcal{X}_1}$ et $k_{\mathcal{X}_2} \rightarrow v_2 \in V_{\mathcal{X}_2}$
$\}$

La jointure naturelle peut être utilisée pour ajouter des dimensions aux tenseurs, et donc leur donner plus de contexte. Le tableau 3.7 présente un exemple d'une jointure naturelle, qui permet de compléter le tenseur $\mathcal{UHT}$ avec une dimension contenant les adresses email des utilisateurs.

$\mathcal{UHT}$				$\mathcal{UE}$		
User	Hashtag	Time	Value	User	Email	Value
u1	h1	1	5	u1	u1@domain.com	1
u1	h2	2	2	u2	u2@domain.com	1
u2	h1	3	11	u2	u2@private.com	1
u2	h2	4	3			

↓

User	Hashtag	Time	Email	Value
u1	h1	1	u1@domain.com	5
u1	h2	2	u1@domain.com	2
u2	h1	3	u2@domain.com	11
u2	h1	3	u2@private.com	11
u2	h2	4	u2@domain.com	3
u2	h2	4	u2@private.com	3

Table 3.7 – Exemple de l'application de l'opérateur de jointure naturelle sur les tenseurs $\mathcal{UHT}$ et $\mathcal{UE}$: $\mathcal{UHT} \bowtie_f \mathcal{UE}$, avec $f(x_{\mathcal{UHT}}, x_{\mathcal{UE}}) = x_{\mathcal{UHT}}$

La différence est un opérateur binaire permettant de supprimer les valeurs du premier tenseur si la combinaison de clés indexant une valeur existe également dans le second tenseur.

Définition 3.3.10 : Différence

L'opérateur de **différence** noté $-$, appliqué à deux tenseurs typés nommés $\mathcal{X}_1$ et $\mathcal{X}_2$ de même schéma, produit un tenseur $\mathcal{X}_3$ de schéma identique et dont les ensembles des clés des dimensions sont issus de la différence entre l'ensemble des clés du tableau associatif des valeurs de $\mathcal{X}_1$ et l'ensemble des clés du tableau associatif des valeurs de $\mathcal{X}_2$. Les valeurs de $\mathcal{X}_3$ sont les valeurs provenant de $\mathcal{X}_1$ pour lesquelles la clé correspondante n'existe pas dans $\mathcal{X}_2$.

(1) $\alpha(\mathcal{X}_1 - \mathcal{X}_2) = \alpha(\mathcal{X}_1) = \alpha(\mathcal{X}_2)$

(2) $\mathcal{X}_1 - \mathcal{X}_2 = \{k \rightarrow v, \text{ avec } k \in K_{\mathcal{X}_1}, k \notin K_{\mathcal{X}_2}, v \in Dom_{\mathcal{X}_1}, k \rightarrow v \in V_{\mathcal{X}_1}\}$

L'opérateur de différence peut agir comme un filtre. Par exemple, contrairement à l'exemple de l'intersection, on peut vouloir étudier les utilisateurs qui utilisent des hashtags sur un réseau social mais pas sur un autre. Ce comportement est illustré dans le tableau 3.8.

$\mathcal{UHT}_1$				$\mathcal{UHT}_2$			
User	Hashtag	Time	Value	User	Hashtag	Time	Value
u1	h1	1	5	u2	h2	4	8
u1	h2	2	2	u3	h1	1	1
u2	h1	3	11	u1	h3	3	2
u2	h2	4	3	u3	h2	5	5

↓

User	Hashtag	Time	Value
u1	h1	1	5
u1	h2	2	2
u2	h1	3	11

Table 3.8 – Exemple de l'application de l'opérateur de différence sur les tenseurs $\mathcal{UHT}_1$ et $\mathcal{UHT}_2$: $\mathcal{UHT}_1 - \mathcal{UHT}_2$

3.3.2.2 Décompositions tensorielles

Les capacités analytiques des tenseurs étant importantes, grâce notamment aux décompositions, nous les intégrons donc dans TDM. Pour le moment, seule la décomposition CANDECOMP/PARAFAC est disponible dans TDM, accompagnée d'un opérateur d'extraction et de reconstruction afin de pouvoir exploiter le résultat de la décomposition.

Définition 3.3.11 : Décomposition CANDECOMP/PARAFAC

La **décomposition CANDECOMP/PARAFAC** notée CP_R appliquée à un tenseur $\mathcal{X}$ typé nommé d'ordre N produit une liste de N tenseurs d'ordre 2, avec R le rang de la décomposition.

(1) $\alpha(CP_R(\mathcal{X})) = (\mathcal{CP}_{Nom_i}(Nom_i : T_i, rank : Integer) : Float)$, où $(Nom_i : T_i) \in D_{\mathcal{X}}$ et $i = 1, \ldots, N$, c'est-à-dire le schéma de chaque tenseur résultat

(2) les valeurs de chaque tenseur résultat $\mathcal{CP}_{Nom_i}$ sont données par :
$\mathcal{CP}_{Nom_i} = \{k_{\mathcal{CP}_{Nom_i}} \rightarrow v_{\mathcal{CP}_{Nom_i}}$, avec $k_{\mathcal{CP}_{Nom_i}} = k_{\mathcal{X}\{d_i\}} \times r, k_{\mathcal{X}\{d_i\}} \in K_{\mathcal{X}\{d_i\}}$,
$r \in [1, R], v_{\mathcal{CP}_{Nom_i}} = \mathrm{A}^{(i)}_{A_{\mathcal{X}_{d_i}}(k_{\mathcal{X}\{d_i\}}), r}$
et $\mathrm{A}^{(i)} \in [\![\mathrm{A}^{(1)}, \ldots, \mathrm{A}^{(N)}]\!]$ sont les matrices de facteurs $\}$

Pour exploiter le résultat de la décomposition CANDECOMP/PARAFAC, nous définissons un opérateur pour extraire un des tenseurs résultat d'ordre 2. De cette manière, il est possible d'étudier les valeurs obtenues pour une dimension spécifique.

Définition 3.3.12 : Extraction

L'opérateur d'**extraction** noté E_{d_n} appliqué à une liste L contenant M tenseurs typés nommés $\mathcal{CP}_{Nom_n}$ résultant de l'application de l'opérateur CP_R sur un tenseur $\mathcal{X}$, avec $n \in [1, M]$, produit un tenseur typé nommé.

(1) $\alpha(E_{d_n}(L)) = \alpha(\mathcal{CP}_{Nom_n})$, c'est-à-dire le schéma du tenseur $\mathcal{CP}_{Nom_n}$, avec $d_n \in D_{\mathcal{CP}_{Nom_{d_n}}}$ et $Nom_n = Nom_{d_n}$

(2) les valeurs sont celles du tenseur $\mathcal{CP}_{Nom_n}$

Un opérateur permettant de reconstruire un tenseur proche de celui sur lequel a été appliqué l'opérateur CP_R est également proposé.

Définition 3.3.13 : Reconstruction

L'opérateur de **reconstruction** noté Re appliqué à une liste L contenant M tenseurs typés nommés $\mathcal{CP}_{Nom_i}, i = 1, \ldots, M$ obtenus avec l'opérateur CP_R appliqué sur un tenseur $\mathcal{X}$ produit un tenseur $\mathcal{Y}$ typé nommé d'ordre M.

(1) $\alpha(Re(L)) = \alpha(\mathcal{Y}) = \mathcal{Y}(S) : Float$ avec S une liste de schémas de dimensions, pour chaque $s_i \in S$, $s_i = Nom_i : T_i$, obtenu à partir de $\mathcal{CP}_{Nom_i}(Nom_i : T_i, rank : Integer) : Float$, $\mathcal{CP}_{Nom_i} \in L$ pour $i = 1, \ldots, M$

(2) $Re(L) = \{k_{\mathcal{X}} \rightarrow v$, avec $k_{\mathcal{X}} \in K_{\mathcal{X}}$, $v = \sum_{r=1}^{R} a^{(1)}_{A_{\mathcal{X}_{d_1}}(k_{\mathcal{X}\{d_1\}})} \circ \cdots \circ a^{(M)}_{A_{\mathcal{X}_{d_M}}(k_{\mathcal{X}\{d_M\}})}$
et $a^{(i)}_j$ la colonne j de $\mathrm{A}^{(i)}, \mathrm{A}^{(i)} \in [\![\mathrm{A}^{(1)}, \ldots, \mathrm{A}^{(N)}]\!]$ sont les matrices de facteurs $\}$

3.3.3 Les correspondances entre TDM et les autres modèles de représentation des données

Un des objectifs d'utiliser les tenseurs comme base pour TDM est de pouvoir développer un modèle pivot. Dans la suite de cette section, nous exposons les manières de représenter différents modèles avec TDM.

3.3.3.1 Modèle clé-valeur

La plupart des systèmes de stockage clé-valeur stockent les données sous forme de couples $(clé, valeur)$ dans une table de hachage distribuée [17]. Le schéma d'un tenseur typé ainsi que ses valeurs étant décrits par des tableaux associatifs, il existe une correspondance directe entre ce modèle clé-valeur et le modèle TDM : la clé est représentée par une dimension tandis que la valeur est contenue dans les valeurs du tenseur. Lorsque les valeurs associées aux clés ne sont pas numériques, il est possible de les représenter dans un tenseur booléen, avec une dimension représentant la clé, une autre la valeur, et un 1 en tant que valeur du tenseur si ce couple $(clé, valeur)$ existe.

Avec TDM, il est même possible de pousser le modèle $(clé, valeur)$ encore plus loin, puisqu'on peut représenter un modèle à clés multiples. Chaque clé est alors représentée par une dimension, indexant les valeurs du tenseur.

3.3.3.2 Modèle relationnel

En ce qui concerne le modèle relationnel, il est possible de considérer deux cas pour le représenter dans un tenseur. Le premier cas est le résultat d'une requête de type GROUP BY. Dans ce cas son intégration dans un tenseur est directe : les attributs sur lesquels le regroupement est effectué sont chacun envoyé dans une dimension, tandis que les valeurs numériques de l'agrégation (somme, produit, nombre d'occurrences, etc.) sont représentées par les valeurs du tenseur. Ce cas de figure est celui qui permet de bénéficier le plus des capacités analytiques du tenseur, et peut se rapprocher de la construction d'un cube OLAP.

Le deuxième cas qui peut se présenter est lorsque l'on souhaite représenter une table, par exemple de schéma $R(\underline{A : String, B : String}, C : Integer, D : Integer)$ dans lequel $\underline{A, B}$ représente la clé primaire composée de deux attributs. On peut alors imaginer une correspondance avec deux tenseurs $\mathcal{C}(A : String, B : String) : Integer$ et $\mathcal{D}(A : String, B : String) : Integer$, portant respectivement les valeurs des attributs C et D (figure 3.4).

A	B	C	D
a	b	25	4
a	b′	22	1
a′	b″	5	2

(a)

$\mathcal{C}$	B	b	b′	b″
A		1	2	3
a	1	25	22	
a′	2			5

(b)

$\mathcal{D}$	B	b	b′	b″
A		1	2	3
a	1	4	1	
a′	2			2

(c)

FIGURE 3.4 – Correspondance entre le modèle relationnel et TDM

Il est à noter que si un des attributs ne faisant pas partie de la clé primaire n'a pas un format numérique, il est possible, de la même manière que pour le modèle clé-valeur, de le représenter dans un tenseur booléen, dont la valeur de l'attribut en question est envoyée dans une dimension supplémentaire et les valeurs du tenseurs sont fixées à 1 lorsque la relation existe. En poussant cette logique, il devient même possible de

représenter une table n'ayant pas de clé primaire : chaque attribut est une dimension, et les valeurs du tenseur sont le nombre d'occurrences de cette combinaison de valeurs d'attributs.

3.3.3.3 Modèle orienté colonnes

Cette dernière vision du modèle relationnel peut aussi servir dans le cas d'un modèle orienté colonne. On considère un modèle orienté colonne comme un modèle semi-structuré à schéma fixe, dont le nombre et le type des colonnes sont invariables, comme pour les fichiers CSV et les *dataframes* [142]. Ce type de modèle est techniquement proche du relationnel, bien que moins restrictif. De ce fait, les mêmes méthodes de modélisation que celles du modèle relationnel peuvent être employées.

3.3.3.4 Modèle graphe

Un graphe simple désigné par $G = (V, E)$, où V est l'ensemble des sommets et $E \subset V \times V$ l'ensemble des arêtes ou des arcs, peut être représenté par sa matrice d'adjacence ou d'incidence et donc par un tenseur d'ordre 2 qui peut prendre en compte l'orientation et les poids des arcs.

Il est également possible de représenter des graphes plus détaillés possédant des attributs sur les liens, comme un attribut temporel ou des labels apportant des informations supplémentaires. Par exemple, dans un graphe représentant les cooccurrences d'achats de produits, avec les sommets représentant les produits et les liens les achats simultanés, on peut vouloir représenter, en plus de la quantité pour le poids des liens, le moment de l'achat mais également le magasin dans lequel il a eu lieu.

3.4 Typage sûr et inférence de schéma

En plus de ses opérateurs de manipulation de données et de décompositions tensorielles, TDM possède un typage fort ainsi que des capacités d'inférence de schéma. Cela permet de réduire le risque d'erreurs aussi bien techniques que fonctionnelles, assurant de cette manière une sécurité renforcée lors de l'utilisation des opérateurs mais également lors de l'interprétation des résultats. Pour cela, la librairie développée [5] utilise le langage Scala, qui propose des mécanismes de contrôle de type avancés permettant d'intégrer cette sûreté lors de la phase de compilation. Des détails techniques complémentaires sur la mise en œuvre de ces mécanismes sont donnés dans le chapitre 4.

3.4.1 Les mécanismes utilisés

Afin d'obtenir les propriétés de typage fort et d'inférence de schéma dans TDM, il est nécessaire d'utiliser plusieurs mécanismes : les *phantom types*, la librairie shapeless [65] et les *implicits*. Ils sont détaillés dans la suite de cette section.

Les ***phantom types*** sont des types qui ne peuvent jamais être instanciés. Ils sont utilisés pour appliquer des contraintes directement sur les types, sans avoir à créer un nouvel objet pour cela. Leur utilisation aide à obtenir un typage sûr, en permettant au compilateur de vérifier des contraintes plus précises concernant les types.

5. `https://github.com/AnnabelleGillet/TDM`

Dans la librairie TDM, les dimensions d'un tenseur sont définies en tant que *phantom types*, qui étendent `TensorDimension[T]` avec un type paramétré `T` :

```
object User extends TensorDimension[String]
object Hashtag extends TensorDimension[String]
object Time extends TensorDimension[Long]
```

De cette manière, les dimensions supportent plusieurs propriétés :
- chaque dimension peut avoir un nom significatif, tout en étant d'un type simple (tel que `String` ou `Long`) ;
- chaque dimension est facilement identifiable, car elle est référencée à l'aide d'un type plutôt qu'un nom (par exemple une chaîne de caractère ou une instance d'objet) ;
- les dimensions des tenseurs peuvent être contrôlées plus finement, en acceptant une seule fois un *phantom type*, mais plusieurs fois un type simple. Par exemple, pour un tenseur représentant des coordonnées, deux *phantom types* sont utilisés : `Longitude` et `Latitude`, qui étendent tous les deux `TensorDimension[Double]` ;
- plusieurs tenseurs peuvent partager un même *phantom type*, et l'utiliser comme une contrainte pour appliquer les opérateurs, renforçant de cette manière le typage sûr au niveau du schéma en s'appuyant sur les *implicits* (voir ci-dessous).

Avec la **librairie shapeless** [6], des structures permettant de manipuler des objets au niveau de leur type sont disponibles. C'est le cas de la `HList` (*Heterogeneous List*), qui permet de construire une liste contenant des objets de différents types, tout en gardant le détail du type de chaque objet sans avoir à utiliser un supertype commun pour travailler avec la liste.

Dans la librairie TDM, une `HList` de *phantom types* (par exemple `User::Hashtag::Time::HNil`) est utilisée pour représenter le schéma d'un tenseur, et son type est constitué d'un type paramétré. Cette technique d'implémentation permet d'utiliser des outils et des méthodes pour interagir directement avec le schéma du tenseur en s'appuyant sur les *implicits*.

En Scala, les ***implicits*** [132, 133] permettent de déléguer une partie de la logique du code au compilateur. C'est le mécanisme essentiel pour assurer les vérifications au moment de la compilation. Quand une fonction définit un paramètre comme *implicit*, si l'utilisateur ne fournit pas explicitement le paramètre, le compilateur va vérifier dans son *scope* courant s'il est possible de trouver une valeur ayant le type correspondant au paramètre attendu afin de pouvoir utiliser la fonction (ce mécanisme est discuté plus en détail dans le chapitre 4).

Cette fonctionnalité peut être poussée afin de s'en servir pour vérifier des contraintes avancées, ou pour inférer le type résultat en fonction des paramètres d'entrée. Si on veut vérifier qu'un type `T` est dans une `HList L`, il est possible d'utiliser les *implicits* de manière récursive (figure 3.5). Nous vérifions tout d'abord si la tête de la liste `L` est de type `T`. Si c'est le cas, l'*implicit* compile car une correspondance a été trouvée dans son *scope*. Si la tête de la liste n'est pas de type `T`, l'*implicit* doit résoudre lui-même le même *implicit* cette fois appelé sur le type `T` et la queue de `L`. Si aucune correspondance n'a été trouvée lorsque la fin de `L` est atteinte, la chaîne entière d'*implicits* ne compile pas, et de ce fait invalide la fonction utilisant l'*implicit*

6. https://github.com/milessabin/shapeless

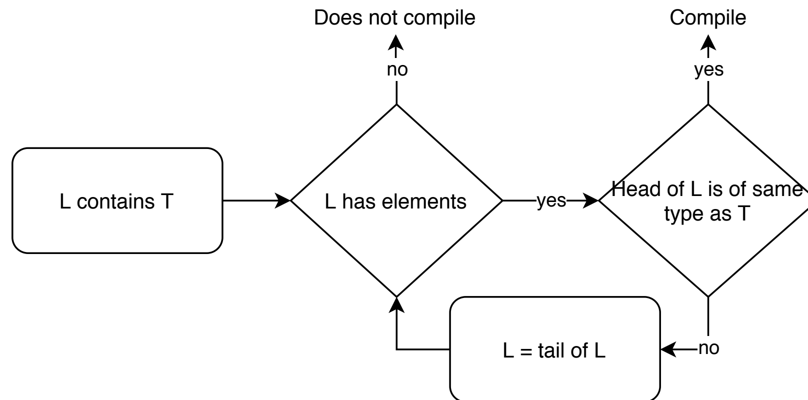


Figure 3.5 – Étapes permettant de vérifier si un type T est dans une HList L à l'aide d'*implicits*

à la racine de l'appel. En composant ces mécanismes de contraintes, il est possible d'en construire de plus complexes, et de **les utiliser pour assurer la validité des opérateurs en fonction du schéma des tenseurs**.

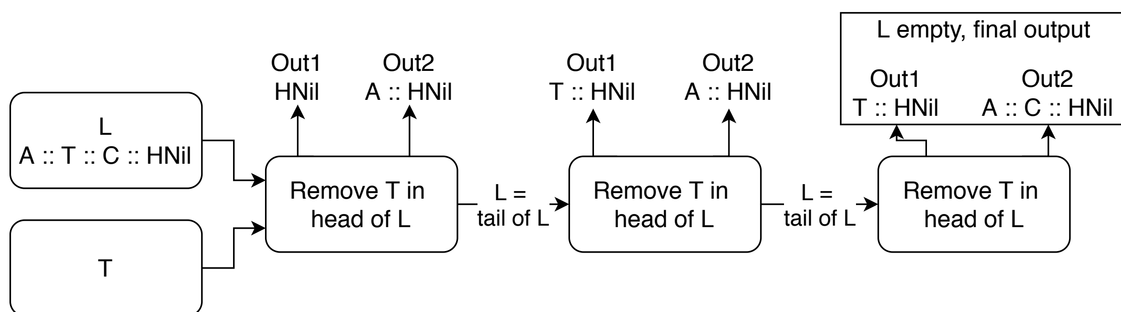


Figure 3.6 – Exemple du retrait de tous les éléments de type T dans une HList L à l'aide des *implicits* et des *path dependent types*

Les *implicits* peuvent aussi être utilisés pour **inférer automatiquement le schéma** du tenseur résultat lorsqu'un opérateur modifiant son schéma est utilisé (par exemple la projection ou la jointure naturelle). Pour cela, les *implicits* sont combinés avec les *path dependent types* [14], qui permettent d'obtenir un type en fonction d'un ou plusieurs types en entrée. L'utilisation des *implicits* est alors la même que pour la vérification de contraintes, sauf que le type résultat est ajouté au paramètre *implicit*, et lorsque le compilateur essaie de résoudre l'*implicit*, le type résultat est construit. Un exemple de ce fonctionnement est de retirer tous les éléments correspondant à un type donné dans une HList (figure 3.6). À partir d'une HList L et d'un type T, il est possible de construire deux HList résultats :
— Out1, qui contient tous les éléments de L étant de type T ;
— Out2, qui contient tous les éléments de L n'étant pas de type T.

Cette suppression peut être utilisée pour appliquer l'opérateur de projection sur un tenseur en retirant la dimension sur laquelle on souhaite se concentrer. Le schéma du tenseur résultat dépend du schéma du tenseur sur lequel est appliqué l'opérateur ainsi que de la dimension sur laquelle on souhaite réaliser la projection.

3.4.2 Vers un langage de requêtes fonctionnel

La librairie TDM est basée sur Spark et utilise la libraire shapeless. Le schéma, les opérateurs de manipulation de données et les décompositions tensorielles sont ajoutés par dessus un `DataFame` de Spark, afin d'obtenir une implémentation sûre de TDM. Les détails de shapeless sont cachés à l'utilisateur, ce qui permet de détecter les erreurs lors de la compilation tout en étant *user-friendly*.

Dans TDM, un tenseur est construit en trois étapes :

1. les dimensions sont définies ;
2. les dimensions sont ajoutées au tenseur, et elles peuvent être réutilisées dans d'autres tenseurs ;
3. les valeurs sont obtenues à partir d'un `DataFrame` de Spark ou ajoutées manuellement.

Par exemple, pour construire le tenseur $\mathcal{UHT}$, ses trois dimensions sont créées de la manière suivante :

```
1 object User extends TensorDimension[String]
2 object Hashtag extends TensorDimension[String]
3 object Time extends TensorDimension[Long]
```

Il est à noter que les dimensions définies de cette manière, en plus d'être des *phantom types* et de contribuer à supporter les propriétés énoncées dans la section 3.4.1, sont aussi utilisées pour aider les utilisateurs à produire des valeurs pour une dimension et pour exprimer des conditions servant aux opérateurs de manipulation de données. Les dimensions sont ensuite utilisées avec un objet `TensorBuilder`, à partir duquel le type du tenseur est défini :

```
1 val tensorUHT = TensorBuilder[Long]
2    .addDimension(User)
3    .addDimension(Hashtag)
4    .addDimension(Time)
5    .build()
6 tensorUHT.addValue(User.value("u1"), Hashtag.value("ht1"), Time.value(1))(1)
```

Alternativement, un tenseur peut aussi être créé en récupérant ses valeurs directement à partir d'un `DataFrame` de Spark. Dans ce cas, l'utilisateur doit fournir :

1. le `DataFrame` à partir duquel extraire les valeurs ;
2. la correspondance entre chaque dimension et le nom de sa colonne dans le `DataFrame` ;
3. le nom de la colonne correspondant aux valeurs du tenseur dans le `DataFrame`.

```
1 val df: DataFrame = ...
2
3 val tensorUHT = TensorBuilder[Long](df)
4    .addDimension(User, "user")
5    .addDimension(Hashtag, "hashtag")
6    .addDimension(Time, "time")
7    .build("value")
```

Ce mécanisme permet de tirer profit de toutes les sources de données que le `DataFrame` de Spark peut utiliser nativement, c'est-à-dire les connecteurs JDBC, les différents formats de fichiers, etc. Cela renforce la position de TDM en tant que modèle pivot, puisqu'une multitude de connecteurs ont été développés pour Spark.

Les opérateurs de TDM ajoutent des capacités de manipulation de données aux tenseurs et infèrent automatiquement le schéma du tenseur résultat, même lorsque l'opérateur induit une transformation de schéma (par exemple la jointure naturelle). De plus, les opérateurs nécessitant une dimension en paramètre, comme la projection, ne compilent que si la dimension fournie en paramètre est bien incluse dans le tenseur sur lequel l'opérateur est appliqué.

Par exemple, utiliser une projection sur le tenseur $\mathcal{UHT}$ préalablement défini, pour la dimension `User` et la valeur $u1$, va produire un nouveau tenseur $\mathcal{HT}$ peuplé avec les valeurs correspondant à la dimension et à la valeur spécifiées :

```
val tensorHT = tensorUHT.projection(User)("u1")
```

Avec les capacités de nommage de Scala, il est également possible d'utiliser l'opérateur de projection de la manière suivante :

```
val tensorHT = π(tensorUHT)(User)("u1")
```

L'accès aux valeurs peut être réalisé de la manière suivante :

```
tensorHT(Hashtag.value("ht1"), Time.value(1)) // Some(1)
tensorHT(Hashtag.value("ht2"), Time.value(2)) // None
```

D'autres opérateurs associent deux tenseurs de différentes manières. Par exemple, l'union produit un nouveau tenseur avec toutes les clés et valeurs associées des deux tenseurs fusionnés, tandis que l'intersection produit un nouveau tenseur contenant uniquement les clés en commun. Les deux opérateurs prennent une fonction en paramètre qui détermine l'opération à réaliser quand les clés sont les mêmes dans les deux tenseurs. La différence est aussi un de ces opérateurs binaires, qui conserve uniquement les valeurs du premier tenseur pour lesquelles la clé n'existe pas dans le deuxième tenseur.

```
val t1 = tensor1.union(tensor2)((v1,v2) => max(v1,v2))
val t2 = tensor1.intersection(tensor2)((v1,v2) => v1 + v2)
val t3 = tensor1.difference(tensor2)
```

La jointure naturelle permet de fusionner le schéma de deux tenseurs qui ont au moins une dimension en commun, et elle applique également une fonction pour former les nouvelles valeurs.

```
val t3 = tensor1.naturalJoin(tensor3)((v1, v2) => v1)
```

La décomposition CANDECOMP/PARAFAC (nommée canonicalPolyadicDecomposition dans la librairie TDM) peut être exécutée directement sur un tenseur, en fournissant uniquement en paramètre le nombre de rangs que l'on souhaite obtenir. Il est alors possible d'extraire du résultat les tenseurs à deux dimensions (la dimension sur laquelle est réalisée l'extraction et une dimension `Rank`) afin d'en analyser les résultats.

```
val kruskalUHT = tensorUHT.canonicalPolyadicDecomposition(2) // De type
    KruskalTensor[User::Hashtag::Time::HNil]
val userTensor = kruskalUHT.extract(User)
```

Les opérateurs de TDM sont implémentés avec de fortes contraintes sur les types. Ils peuvent être groupés en trois catégories :

— les opérateurs qui fonctionnent au niveau des valeurs du tenseur, comme la sélection, dans ce cas le paramètre utilisé doit correspondre au type du tenseur ;
— les opérateurs qui fonctionnent sur les dimensions, comme la projection ou la restriction. La dimension fournie en paramètre doit faire partie du schéma du tenseur ;
— les opérateurs binaires, comme l'union ou la jointure naturelle. Les schémas des deux tenseurs doivent correspondre aux attentes de l'opérateur : pour l'union les deux schémas doivent être identiques tandis que pour la jointure naturelle les tenseurs doivent avoir au moins une dimension en commun.

La garantie de sûreté du type est obtenue lors de la compilation grâce à la librairie shapeless et aux *implicits* de Scala, comme expliqué dans la section 3.4.1. Une erreur de compilation avertit l'utilisateur si une inconsistance est détectée. L'utilisation des *implicits* permet aussi de créer des messages d'erreur de compilation personnalisés, ajoutant ainsi la capacité d'afficher des messages adaptés au contexte et aux tenseurs qui permettent de guider l'utilisateur pour résoudre son erreur. L'exemple suivant montre des inconsistances détectées lors de la compilation[7] :

```
tensorHT.addValue(Hashtag.value(1), Time.value(2))(2) // Wrong type of dimension's
    value
tensorHT.addValue(Hashtag.value("ht2"))(2) // Wrong number of dimensions
tensorHT.projection(User)("u2") // Dimension not in tensor
tensorHT.union(tensorUHT) // Different schemas of tensors
```

L'inférence de schéma, tout aussi essentielle que la sûreté des types, repose sur les mécanismes expliqués dans la section 3.4.1. De cette manière, le schéma du tenseur résultat est inféré lors de la compilation, ce qui permet de continuer à appliquer les contraintes sur les opérateurs sans que l'utilisateur ait à définir lui-même chaque schéma résultat comme c'est souvent le cas dans les libraires, y compris en Scala, lorsqu'un opérateur induit un changement de schéma. Par exemple, les `Dataset` et `DataFrame` de Spark requièrent une intervention explicite de l'utilisateur dans ce cas.

Les mécanismes de ces deux propriétés doivent pouvoir être combinés pour travailler non seulement sur le type paramétré du tenseur, mais également au niveau des dimensions du tenseur, ce qui complexifie grandement la tâche puisque le nombre des dimensions est variable et que leur type peut être différent. Le support de ces propriétés est donc un apport majeur pour vérifier les erreurs aussi bien techniques que fonctionnelles, tout en gardant les données et leur réelle signification au cœur des analyses.

7. Voir aussi `https://github.com/AnnabelleGillet/TDM/tree/master/src/test/scala/tdm/core` pour plus d'exemples d'inconsistances détectées à la compilation

3.5 Mise en œuvre d'optimisations

En plus de proposer une utilisation sûre et orientée données des tenseurs, la librairie TDM a été développée de manière optimisée, afin de pouvoir manipuler des tenseurs volumineux sans pour autant souffrir de temps d'exécution longs lors des calculs intensifs. Ces optimisations peuvent se décomposer en deux catégories : les optimisations de la librairie en général, et les optimisations visant spécifiquement la décomposition CANDECOMP/PARAFAC, qui présente des défis techniques.

3.5.1 Les optimisations de la librairie générale

Dans TDM, les données sont conservées dans un `DataFrame` de Spark à n colonnes, dans lequel $n-1$ colonnes représentent les valeurs des dimensions, et la dernière colonne représente la valeur du tenseur associée à la combinaison des valeurs des dimensions. De cette manière, il est possible de profiter directement des optimisations et de l'efficacité de Spark.

Tout d'abord, cela permet de bénéficier des opérateurs robustes et bien définis de Spark, sur lesquels ceux de TDM peuvent s'appuyer afin de les associer d'une certaine manière et obtenir le résultat escompté des opérateurs de TDM. Comme décrit dans la section 3.4, une couche de sûreté du typage est ajoutée dans la signature des opérateurs de TDM, ainsi qu'un mécanisme d'inférence de schéma du résultat, ce qui permet de limiter les erreurs techniques et fonctionnelles pouvant survenir en utilisant Spark nativement. Par exemple, une jointure de deux `Dataset` de Spark (leur structure de données la plus fortement typée) nécessite de fournir explicitement le type du résultat, alors que ce n'est pas le cas dans TDM.

En plus de cet avantage, les optimisations de Spark sont également conservées lors de l'exécution des opérateurs de TDM. Spark fonctionne avec un mécanisme d'**évaluation tardive**, ce qui veut dire que le résultat d'une opération n'est pas forcément calculé au moment de l'appel de l'opération, mais plutôt au moment où le résultat a besoin d'être utilisé. Ce mécanisme permet d'appliquer des optimisations lorsque plusieurs opérateurs sont appelés avant que le résultat ne soit utilisé. Spark est aussi perçu comme le successeur du *map-reduce* d'Hadoop, grâce à ses **calculs distribués qu'il peut réaliser directement dans la RAM**. Cela permet de fragmenter les opérations en tâches, qui sont réparties sur différents cœurs et/ou différentes machines afin de fortement paralléliser les traitements.

Les opérateurs de TDM sont donc construits en s'appuyant sur ceux de Spark, mais sans interférer avec l'évaluation tardive ni diminuer les performances. Pour prouver cela, nous exécutons la même combinaison d'opérateurs, avec TDM et avec Spark seul, puis nous comparons les temps d'exécution. Le jeu de données utilisé est un corpus de tweets collectés dans le but de réaliser des analyses sur les élections présidentielles françaises de 2017, qui contient 50 millions de tweets. Ces expériences sont réalisées sur un serveur Dell PowerEdge R740 (Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz, 20 cœurs, 256Go RAM).

Pour vérifier qu'il n'y a pas de différence d'exécution entre Spark et TDM, nous enchaînons les opérations suivantes avec les deux librairies :

1. un tenseur $\mathcal{U}$ d'ordre 1 est construit avec la dimension représentant les utilisateurs, et les valeurs du tenseur sont le nombre de tweets publiés par l'utilisateur correspondant ;
2. un tenseur $\mathcal{UHT}$ d'ordre 3 est construit afin de représenter le nombre de chaque hashtag publié par les utilisateurs sur des fenêtres de temps d'une heure ;

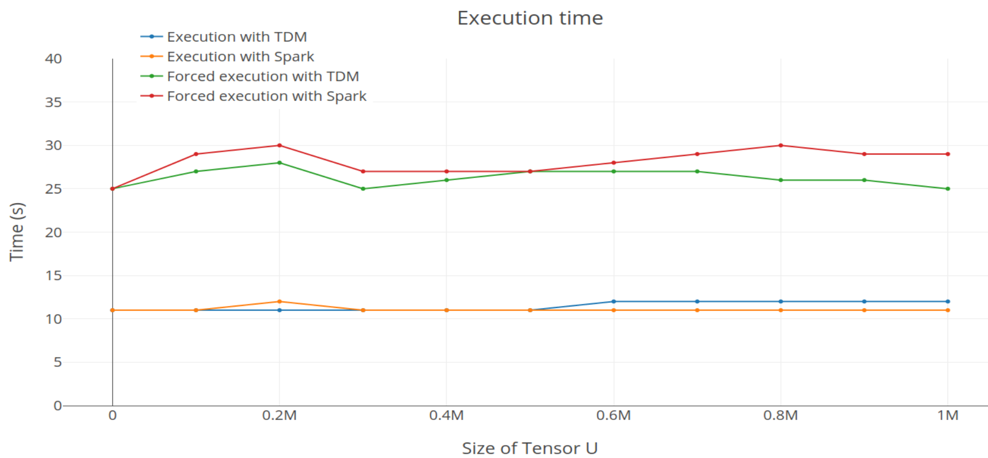


FIGURE 3.7 – Temps d'exécution pour des opérations équivalentes avec TDM et avec Spark

3. une sélection sur le tenseur $\mathcal{U}$ est réalisée pour conserver uniquement les utilisateurs qui ont publié au moins 100 tweets ;
4. une jointure naturelle est réalisée entre le tenseur $\mathcal{U}$ et le tenseur $\mathcal{UHT}$, afin de conserver uniquement les entrées de $\mathcal{UHT}$ pour les utilisateurs actifs.

Lors de cette expérience, la taille du tenseur $\mathcal{U}$ varie de 0 à 1 million d'éléments par pas de 100 000. Nous réalisons deux scénarios :

— en forçant l'exécution à chaque opération (avec Spark cela peut se faire par exemple avec un `count` sur le `DataFrame`, puisque celui-ci doit alors exécuter les opérations précédant le `count` pour pouvoir afficher le résultat) ;

— en forçant l'exécution uniquement à la fin de toutes les étapes de l'expérience.

Un jeu de données anonymisé ainsi que le code source de l'expérience sont disponibles sur Github [8]. Comme montré dans la figure 3.7, les opérateurs de TDM n'entraînent pas d'augmentation du temps d'exécution, et les capacités d'évaluation tardive de Spark sont bien conservées (bas de la figure 3.7).

3.5.2 Les optimisations de la décomposition CANDECOMP/PARAFAC

Optimiser l'algorithme ALS de la décomposition CANDECOMP/PARAFAC induit plusieurs défis techniques, qui gagnent en importance proportionnellement à la taille des données. De ce fait, pour permettre son exécution sur des données volumineuses, plusieurs problèmes doivent être résolus.

Tout d'abord, l'**explosion des données du MTTKRP** (ligne 5 de l'algorithme 1) est un sérieux verrou pour obtenir de bonnes performances, qui peut également mener à un dépassement de RAM, empêchant de ce fait de travailler avec des tenseurs volumineux, même s'ils sont creux. En effet, la matrice résultat du produit de Khatri-Rao a $J \times R$ éléments non-nuls, avec $J = \prod_{j \neq n} I_j$, pour un tenseur de taille $\mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$. Je propose de réordonner les opérations de cette étape de l'algorithme, dans le but d'éviter l'explosion de données et de pouvoir améliorer significativement le temps d'exécution (voir l'algorithme 3).

8. `https://github.com/AnnabelleGillet/TDM-experiments/tree/master/SparkComparison`

Les opérations principales de l'algorithme ALS (les mises à jour des matrices de facteurs), **ne sont pas parallélisables en elles-mêmes** (lignes 4 et 5 de l'algorithme 1). Dans une telle situation, il est préférable de penser à d'autres méthodes pour tirer profit du parallélisme, qui pourraient être appliquées sur des opérations plus fines. Par exemple, paralléliser les multiplications de matrices est une optimisation qui peut être appliquée dans de nombreuses situations. Cela permet aussi de faciliter la réutilisation de telles optimisations, sans attendre des caractéristiques particulières de l'algorithme sur lequel elles sont appliquées (voir la section 3.5.2.2).

La **nature des structures de données utilisées dans la décomposition CANDECOMP/PARAFAC est hétérogène** : les tenseurs sont souvent creux, tandis que les matrices de facteurs sont denses. Leurs besoins pour être efficacement implémentés divergent, donc plutôt que de conserver globalement une unique structure de données creuse qui convient la plupart du temps aux tenseurs, les particularités de chaque structure devraient être exploitées pour améliorer l'exécution entière (voir la section 3.5.2.1). À notre connaissance, cette stratégie n'a pas été explorée par d'autres travaux.

Enfin, le **critère d'arrêt** peut aussi être un frein. Dans les implémentations distribuées de l'algorithme ALS de la décomposition CANDECOMP/PARAFAC, les solutions principales utilisées pour arrêter l'algorithme sont soit d'attendre un nombre fixe d'itérations, soit de calculer la norme de Frobenius sur la différence entre le tenseur d'origine et celui reconstruit à partir des matrices de facteurs. La première solution manque cruellement de précision, et la deuxième est coûteuse en temps de calcul puisqu'elle implique le produit tensoriel (*outer product*) entre toutes les matrices de facteurs. Cependant, une autre option est disponible pour vérifier la convergence, et consiste à mesurer les similarités entre les matrices de facteurs de deux itérations successives, comme évoqué dans [36, 95]. C'est une solution très efficace à large échelle, puisqu'elle allie précision et calculs peu coûteux (voir section 3.5.2.3).

Pour surmonter ces défis, j'utilise trois principes d'optimisations pour développer une décomposition efficace : les optimisations à gros grain, les optimisations fines, et les calculs incrémentaux. Les optimisations à gros grains reposent sur les structures de données spécifiques et sur les capacités de Spark à distribuer efficacement les opérations. Les optimisations fines sont appliquées sur le MTTKRP pour éviter les structures de données intermédiaires volumineuses et les calculs coûteux. Pour cela j'ai étendu les opérations de matrices de Spark pour supporter celles nécessaires au calcul de la décomposition CANDECOMP/PARAFAC. Les calculs incrémentaux sont utilisés pour ne pas avoir à calculer le produit de Hadamard complet à chaque itération. En plus de ces optimisations, j'ai choisi un critère de convergence adapté, qui est efficace pour travailler avec des données massives. De ce fait, la décomposition CANDECOMP/PARAFAC de la librairie TDM est capable de traiter des tenseurs ayant des millions d'éléments non-nuls sur un serveur moyen de gamme, et des tenseurs de taille moyenne ou réduite sur un ordinateur de bureau standard, tout cela en un temps raisonnable.

3.5.2.1 Les structures de données : matrices distribuées et passant à l'échelle

Une **structure de données simple mais efficace pour les matrices creuses** est le COO (*COOrdinate storage*) [5, 59]. Les `CoordinateMatrix`, disponibles dans le *package* `mllib` de Spark [25], implémentent le COO, et ne stockent que les coordonnées et la valeur de chaque élément existant dans un `RDD` (*Resilient Distributed Datasets*). **C'est un format bien adapté pour traiter les matrices creuses.**

Une autre structure utile est celle des `BlockMatrix`. C'est une structure composée de plusieurs blocs contenant chacun un fragment de la matrice globale. Les opérations peuvent être parallélisées en les exécutant

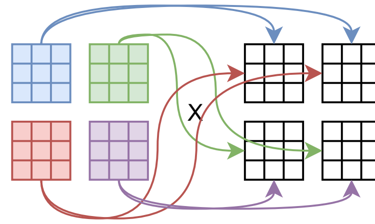


FIGURE 3.8 – *Mapping* des blocs pour une multiplication entre deux `BlockMatrix`

sur chaque fragment de matrice. Pour les opérations binaires telles que la multiplication, seuls les blocs de chaque `BlockMatrix` qui vont être associés sont réunis, et le résultat est ensuite agrégé si nécessaire (figure 3.8). **C'est une structure efficace pour les matrices denses**, qui permet de fragmenter et distribuer les opérations pour traiter tous les blocs.

Malheureusement, seules quelques opérations basiques sont disponibles pour les `BlockMatrix`, telles que la multiplication ou l'addition. Les plus complexes, comme le produit de Hadamard ou de Khatri-Rao, sont manquantes. J'ai donc étendu les `BlockMatrix` de Spark pour supporter ces opérations avancées, tout en gardant la logique d'optimisation de la multiplication. J'ai aussi ajouté de nouvelles opérations, qui impliquent également des `CoordinateMatrix` en plus des `BlockMatrix` afin de pouvoir exploiter les particularités de chacune de ces structures dans le but d'optimiser le MTTKRP.

3.5.2.2 L'association de trois principes d'optimisation

Les tenseurs sont généralement fortement creux. Dans la décomposition CANDECOMP/PARAFAC, ils apparaissent seulement sous leur forme matricisée, ils sont donc naturellement manipulés en tant que `CoordinateMatrix` dans l'implémentation. Inversement, les matrices de facteurs **A** sont denses, car elles portent les informations concernant chaque valeur de chaque dimension. Elles bénéficient donc des capacités de l'extension des `BlockMatrix` que j'ai développée. En utilisant la structure de données la plus adaptée à chaque partie de l'algorithme, cela permet de profiter d'optimisations qui peuvent accélérer l'exécution de l'algorithme entier.

Algorithme 2 Décomposition CANDECOMP/PARAFAC version ALS optimisée

Require: Tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and target rank R
1: Initialize $\mathsf{A}^{(1)}, \ldots, \mathsf{A}^{(N)}$, with $\mathsf{A}^{(n)} \in \mathbb{R}^{I_n \times R}$
2: $\mathsf{V} \leftarrow \mathsf{A}^{(1)T}\mathsf{A}^{(1)} \circledast \ldots \circledast \mathsf{A}^{(N)T}\mathsf{A}^{(N)}$
3: **repeat**
4:     **for** $n = 1, \ldots, N$ **do**
5:         $\mathsf{V} \leftarrow \mathsf{V} \oslash \mathsf{A}^{(n)T}\mathsf{A}^{(n)}$
6:         $\mathsf{A}^{(n)} \leftarrow MTTKRP(\boldsymbol{\mathcal{X}}_{(n)}, (\mathsf{A}^{(N)}, \ldots, \mathsf{A}^{(n+1)}, \mathsf{A}^{(n-1)}, \ldots, \mathsf{A}^{(1)}))\mathsf{V}^{\dagger}$
7:         $\mathsf{V} \leftarrow \mathsf{V} \circledast \mathsf{A}^{(n)T}\mathsf{A}^{(n)}$
8:         normalize columns of $\mathsf{A}^{(n)}$
9:         $\lambda \leftarrow$ norms of $\mathsf{A}^{(n)}$
10:     **end for**
11: **until** $<$ convergence $>$

En plus d'utiliser et d'améliorer les matrices proposées par Spark en fonction des spécificités des données, j'ai également introduit des optimisations fines et des calculs incrémentaux dans l'algorithme pour éviter les opérations coûteuses en temps et en mémoire. Ces améliorations sont synthétisées dans l'algorithme 2 et expliquées ci-dessous.

Tout d'abord, pour éviter de calculer **V** complètement à chaque itération pour chaque dimension, je propose de réaliser cette étape de manière incrémentale. Avant de commencer l'itération, le produit de Hadamard est calculé pour toutes les matrices de facteurs **A** (ligne 2 de l'algorithme 2). Au début de chaque itération, une division de Hadamard est réalisée entre **V** et $\mathbf{A}^{(n)T}\mathbf{A}^{(n)}$, donnant ainsi le résultat attendu à cette étape (ligne 5 de l'algorithme 2). À la fin de l'itération, le produit de Hadamard entre la nouvelle matrice de facteurs $\mathbf{A}^{(n)T}\mathbf{A}^{(n)}$ et **V** prépare **V** pour la prochaine itération (ligne 7 de l'algorithme 2).

Algorithme 3 Détails du MTTKRP de l'algorithme 2

Require: The index of the factor matrix n, the matricized tensor $\boldsymbol{\mathcal{X}}_{(n)} \in \mathbb{R}^{I_n \times J}$ with $J = \Pi_{j \neq n} I_j$ and $\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(n-1)}, \mathbf{A}^{(n+1)}, \ldots, \mathbf{A}^{(N)}$, with $\mathbf{A}^{(i)} \in \mathbb{R}^{I_i \times R}$

1: Initialize $\mathbf{A}^{(n)}$ at 0, with $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}$
2: **for each** (x, y, v) in $\boldsymbol{\mathcal{X}}_{(n)}$ with x, y the coordinates and v the value **do**
3:   **for** $r = 1, \ldots, R$ **do**
4:     $value \leftarrow v$
5:     **for each** $\mathbf{A}^{(i)}$ with $i \neq n$ **do**
6:       $c \leftarrow$ extract $\mathbf{A}^{(i)}$ coordinate from y
7:       $value \leftarrow value \times \mathbf{A}^{(i)}(c, r)$
8:     **end for**
9:     $\mathbf{A}^{(n)}(x, r) \leftarrow \mathbf{A}^{(n)}(x, r) + value$
10:   **end for**
11: **end for**

La partie concernant le MTTKRP (ligne 6 de l'algorithme 2) est sensible aux améliorations. En se concentrant sur le résultat plutôt que sur l'opération, le MTTKRP peut facilement être réordonné. Par exemple, si l'on multiplie un tenseur d'ordre 3 matricisé sur la dimension 1 avec le résultat de $\mathbf{A}^{(3)} \odot \mathbf{A}^{(2)}$, on peut remarquer que dans le résultat les indices des dimensions du tenseur correspondent directement à ceux des matrices $\mathbf{A}^{(3)}$ et $\mathbf{A}^{(2)}$. Ce comportement est représenté ci-dessous — avec un raccourci de notation $B_i = \mathbf{A}^{(2)}(i, 1)$ et $C_i = \mathbf{A}^{(3)}(i, 1)$ — dans un exemple simplifié avec seulement un rang :

$$\begin{bmatrix} a_1b_1c_1 & a_1b_2c_1 & a_1b_1c_2 & a_1b_2c_2 \\ a_2b_1c_1 & a_2b_2c_1 & a_2b_1c_2 & a_2b_2c_2 \end{bmatrix} \times \begin{bmatrix} B_1C_1 \\ B_2C_1 \\ B_1C_2 \\ B_2C_2 \end{bmatrix} =$$

$$\begin{bmatrix} a_1b_1c_1.B_1C_1 + a_1b_2c_1.B_2C_1 + a_1b_1c_2.B_1C_2 + a_1b_2c_2.B_2C_2 \\ a_2b_1c_1.B_1C_1 + a_2b_2c_1.B_2C_1 + a_2b_1c_2.B_1C_2 + a_2b_2c_2.B_2C_2 \end{bmatrix}$$

En conséquence, plutôt que de calculer le produit de Khatri-Rao complet puis de calculer la multiplication avec le tenseur matricisé, j'applique une optimisation fine qui bénéficie du *mapping* des indices, et qui anticipe la construction de la matrice finale. Pour chaque entrée de la `CoordinateMatrix` représentant le tenseur

matricisé, la correspondance avec chaque matrice de facteurs **A** est utilisée pour trouver quels éléments utiliser, et de cette manière calculer la matrice résultat (algorithme 3).

3.5.2.3 Critère d'arrêt

Pour évaluer la convergence de l'algorithme et vérifier à quel moment il peut être arrêté, la majorité des implémentations de la décomposition CANDECOMP/PARAFAC utilisent la norme de Frobenius sur la différence entre le tenseur d'origine et le tenseur reconstruit à partir des matrices de facteurs. Toutefois, pour un tenseur volumineux, la reconstruction est coûteuse, davantage même que le MTTKRP naïf. Attendre un nombre d'itérations prédéterminé n'est pas non plus très efficace pour éviter d'itérer après qu'une solution satisfaisante ait été trouvée.

D'autres critères d'arrêt, tels que l'évaluation de la différence entre les matrices de facteurs d'une itération à l'autre [36, 95], sont donc bien plus intéressants car ils travaillent sur des volumes de données beaucoup plus petits pour un même tenseur de départ. À cette fin, j'utilise le *Factor Match Score* (FMS) [35] pour mesurer la différence entre les matrices de facteurs de l'itération courante ($[\![\lambda, \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}]\!]$) et celles de l'itération précédente ($[\![\hat{\lambda}, \hat{\mathbf{A}}^{(1)}, \hat{\mathbf{A}}^{(2)}, \ldots, \hat{\mathbf{A}}^{(N)}]\!]$). Le FMS est défini comme suit :

$$FMS = \frac{1}{R}\sum_{r=1}^{R}\left(1 - \frac{\xi - \hat{\xi}}{max(\xi, \hat{\xi})}\right)\prod_{n=1}^{N}\frac{a_r^{(n)T}.\hat{a}_r^{(n)}}{a_r^{(n)}.\hat{a}_r^{(n)}}$$

avec $\xi = \lambda_r \prod_{n=1}^{N} a_r^{(n)}$
et $\hat{\xi} = \hat{\lambda}_r \prod_{n=1}^{N} \hat{a}_r^{(n)}$

3.5.2.4 Expériences de validation

Pour valider l'algorithme, j'ai réalisé des expériences sur des tenseurs en faisant varier la taille de leurs dimensions et leur nombre d'éléments non-nuls, sur un serveur Dell PowerEdge R740 (Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz, 20 cœurs, 256Go RAM). Je compare le temps d'exécution de la décomposition CANDECOMP/PARAFAC avec ceux des librairies distribuées de l'état de l'art : HaTen2 [79], BigTensor [139], SamBaTen [63] et CSTF [23]. Hadoop 2.6.0 a été utilisé pour exécuter HaTen et BigTensor.

Les tenseurs utilisés pour les expériences ont été créés de manière aléatoire, avec 3 dimensions de même taille, allant de 100 à 100 000 éléments par dimension. La sparsité va de 10^{-1} à 10^{-9}, et les tenseurs ont été créés seulement si le nombre d'éléments non-nuls résultants de ces critères est supérieur à $3 \times size$ et inférieur ou égal à 1 milliard (par exemple pour des dimensions de taille 100, les tenseurs peuvent avoir au plus respectivement 10^6 et 10^9 éléments, donc avec une sparsité de 10^{-1} ils ne peuvent pas atteindre 1 milliard d'éléments mais respectivement 10^5 et 10^8 éléments non-nuls). J'ai exécuté la décomposition pour 5 itérations (afin de ne pas mettre en défaut les librairies qui n'ont pas de calcul de critère d'arrêt), et ai mesuré le temps d'exécution. Les résultats sont montrés dans la figures 3.9. Le code source des expériences et de l'outil utilisé pour créer les tenseurs sont disponibles en ligne [9].

Mon implémentation est clairement plus efficace que celles de l'état de l'art, avec une accélération atteignant plusieurs ordres de magnitude. CSTF obtient des temps comparables pour ce qui concerne les plus petits tenseurs, mais se fait vite dépasser pour les tenseurs plus larges, et termine même sur une erreur pour

9. `https://github.com/AnnabelleGillet/MuLOT/tree/main/experiments`

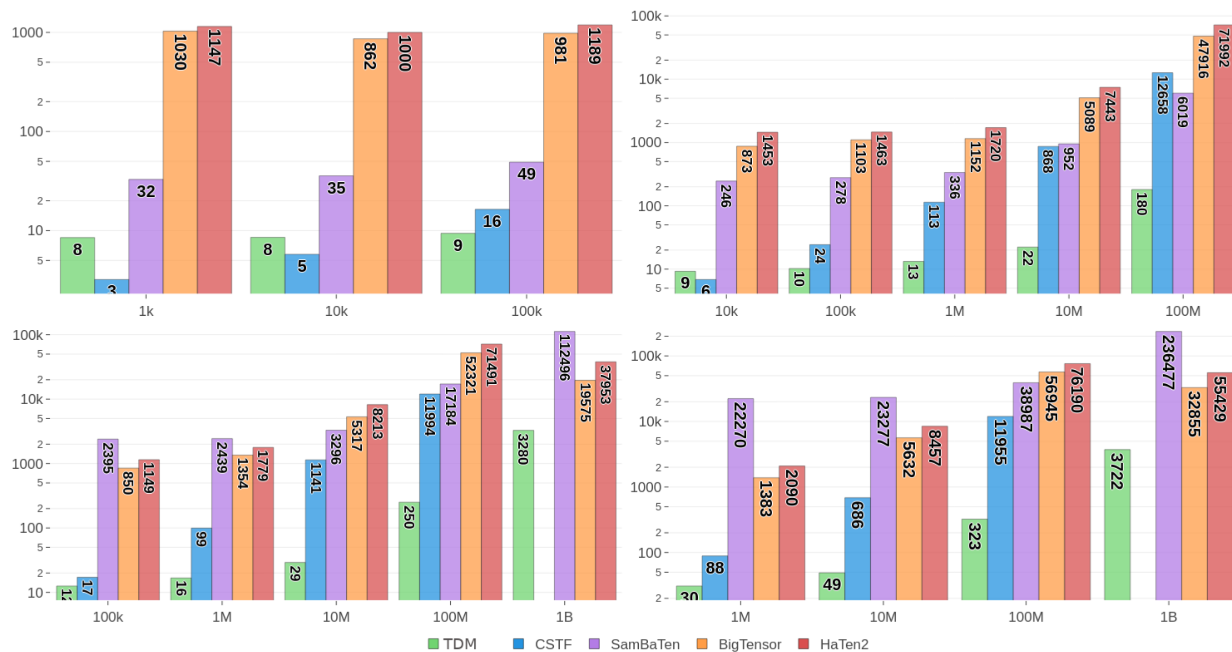


Figure 3.9 – Temps d'exécution pour des tenseurs d'ordre 3, avec les dimensions de taille 100 (en haut à gauche), 1 000 (en haut à droite), 10 000 (en bas à gauche) et 100 000 (en bas à droite). CSTF produit une exception *Out Of Memory* pour des tenseurs ayant 1 milliard d'éléments

les tenseurs ayant 1 milliard d'éléments. Les temps d'exécution de la décomposition avec TDM sont proches d'être linéaires suivant le nombre d'éléments non-nuls. Les techniques d'optimisation appliquées montrent des résultats efficaces même pour des tenseurs ayant 1 milliard d'éléments, avec un temps d'exécution maximal pour un tenseur d'ordre 3 ayant des dimensions de taille 100 000 de 62 minutes, alors que la librairie la plus proche, BigTensor, prend 547 minutes. Le temps d'exécution de la décomposition avec TDM est aussi raisonnable sur des petits tenseurs, puisqu'il ne dépasse pas 10 secondes sur des tenseurs avec des dimensions de taille 100.

3.6 Conclusion

Ce chapitre propose une solution à deux problématiques majeures liées à l'analyse de données. La première problématique est celle de proposer un **modèle pivot permettant de représenter des sources de données multiples**. En effet, les données massives sont reconnues non seulement pour leur volume, mais aussi pour leur variété, et ce jusque dans leur format. Cette variété a fait naître des systèmes de stockage spécialisés, qui peuvent être regroupés dans un *polystore*. L'exploitation des *polystores* est faisable, entre autres, à l'aide d'un modèle pivot suffisamment flexible pour représenter la plupart des modèles de données, et qui présente également des capacités d'analyse de données. La deuxième problématique est celle de **placer les données au centre des analyses**, en facilitant leur manipulation mais aussi en contrôlant les opérations pour éviter les erreurs. L'interprétabilité des résultats est doublement impactée par cette problématique, non

seulement à cause des erreurs fonctionnelles en attribuant une signification erronée aux données, mais aussi à cause des *mappings* que l'utilisateur est obligé de maintenir lorsque l'outil d'analyse ne prend pas en compte et n'offre pas de solution pour manipuler des données réelles.

Le Tensor Data Model répond à ces problématiques, en ajoutant la notion de schéma de données aux tenseurs. Les tenseurs sont des objets mathématiques multi-dimensionnels qui peuvent jouer le rôle de modèles pivots. En plus du schéma, un ensemble d'opérateurs de manipulation de données renforce les capacités d'interactions du modèle avec les données, et les opérateurs tensoriels natifs aux tenseurs fournissent des capacités analytiques.

L'implémentation de TDM supporte des propriétés de sûreté du typage ainsi que d'inférence de schéma, qui augmentent la sécurité lors de la manipulation des tenseurs en limitant non seulement les erreurs techniques mais aussi les erreurs fonctionnelles. De plus, l'optimisation de la librairie TDM est une préoccupation importante, prise en compte au niveau de la librairie globale mais surtout lors de l'implémentation de la décomposition CANDECOMP/PARAFAC, ce qui permet de surpasser les librairies de l'état de l'art.

Le modèle TDM ouvre de nombreuses perspectives d'évolution. Sa formalisation pourrait être améliorée en se servant du système F [149], une extension du lambda-calcul typé, capable de représenter les fonctions tout en prenant les types en considération. En effet, il sert déjà de support théorique aux *implicits* de Scala [131]. De cette manière, la sûreté du typage et l'inférence de schéma seraient intégrées à part entière directement dans le modèle, en plus d'être déjà présentes dans l'implémentation.

Au niveau des opérateurs de manipulation des données, les opérateurs standards d'OLAP, tels que ROLL UP ou DRILL DOWN, peuvent être ajoutés, puisqu'il est possible de voir les cubes OLAP comme des tableaux multi-dimensionnels, que les tenseurs sont capables de représenter directement. De cette manière, TDM bénéficierait des capacités d'OLAP à manipuler la granularité des dimensions et à agréger les résultats.

Au niveau des opérateurs tensoriels, d'autres décompositions peuvent être ajoutées, afin de proposer des techniques d'analyse plus variées, notamment Tucker et DEDICOM. La décomposition Tucker [167] est proche de la décomposition CANDECOMP/PARAFAC, sauf que le rang n'est pas forcément le même pour chaque dimension. En plus des matrices de facteurs, Tucker produit un tenseur cœur qui permet de mettre en correspondance les matrices de facteurs entre elles. DEDICOM [71] est une autre décomposition, qui vise plus spécifiquement les données de type social puisqu'elle considère que deux des dimensions sont de taille identique et représentent les mêmes individus, les autres dimensions servant à représenter le détail des interactions entre les individus.

En ce qui concerne l'utilisation de la librairie, et plus particulièrement de la décomposition CANDECOMP/PARAFAC, une méthode pour choisir le rang de manière pertinente permettrait d'exploiter plus efficacement la décomposition (ce point est abordé plus en détail dans le chapitre 5). CORCONDIA [28] est une de ces méthodes, mais nécessite de d'abord calculer la décomposition pour ensuite mesurer la qualité de son rang, ce qui peut être un frein pour travailler avec des tenseurs volumineux.

L'optimisation de la librairie TDM montre des résultats appréciables à large échelle, mais l'exécution de la décomposition CANDECOMP/PARAFAC pourrait être améliorée pour les tenseurs de taille réduite, bien que le temps d'exécution actuel soit convenable. Pour cela, l'utilisation de la librairie breeze [10] à la place de Spark éviterait de passer par la phase de distribution des données, et améliorerait ainsi le temps d'exécution global.

10. Librairie d'algèbre linéaire en Scala `https://github.com/scalanlp/breeze`

Publications relatives au chapitre

*****

Concernant les publications réalisées en relation avec ce chapitre, elles montrent l'évolution chronologique du travail concernant TDM. La publication [B1] définit le modèle théorique ainsi que les premières expérimentations de la manipulation tensorielle des données, la publication [B2] marque les débuts de la librairie TDM avec sa sûreté du typage et son inférence de schéma et la publication [B4] concerne l'optimisation de la librairie ainsi que l'intégration de la décomposition CANDECOMP/PARAFAC. La référence [B3] est un peu à part puisqu'elle concerne une présentation longue de la librairie à un public de professionnels et d'industriels utilisant le langage Scala. La librairie a reçu de bons retours à cette conférence.

[B1] Éric Leclercq, Annabelle Gillet, Thierry Grison, Marinette Savonnet. **Polystore and Tensor Data Model for Logical Data Independence and Impedance Mismatch in Big Data Analytics**. In : *Transactions on Large-Scale Data-and Knowledge-Centered Systems XLII (TLDKS)*. Springer, 2019, pp. 51–90.

[B2] Annabelle Gillet, Éric Leclercq, Marinette Savonnet, Nadine Cullot. **Empowering big data analytics with polystore and strongly typed functional queries**. In : *Proceedings of the 24th Symposium on International Database Engineering & Applications (IDEAS)*. 2020, pp. 1–10.

[B3] Annabelle Gillet, Éric Leclercq. **TDM : Breaking through dimensions with tensors**. In : *ScalaCon (`http://www.scalacon.org/`)*. 2021.

[B4] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **MuLOT : Multi-level optimization of the canonical polyadic tensor decomposition at large-scale**. In : *25th European Conference on Advances in Databases and Information Systems (ADBIS)*. 2021, pp. 1–15.

Chapitre

4

Implémentation de l'architecture Hydre et du modèle TDM

De l'importance de la qualité technique dans la recherche

*****

4.1 Introduction : sûreté, optimisation et reproductibilité 92
4.2 L'architecture Hydre : une implémentation du patron Lambda+ Architecture 94
  4.2.1 Les spécifications de l'architecture 94
  4.2.2 L'architecture physique 99
4.3 Implémentation de TDM 100
  4.3.1 Les *implicits* 100
  4.3.2 La sûreté du typage 101
  4.3.3 L'inférence de schéma 102
  4.3.4 L'utilisation de TDM 103
4.4 Conclusion 104

Dans la recherche en informatique, implémenter les solutions proposées et rendre le code disponible représentent plusieurs avantages. Cela permet de prouver que la solution agit bien comme elle le devrait, et renforce donc la crédibilité de la recherche associée. Cela permet aussi de vérifier l'applicabilité et les limites en termes de temps d'exécution mais aussi de ressources. Enfin, cela permet d'améliorer le système itératif de la recherche, en laissant les autres chercheurs accéder librement à l'implémentation, et donc en leur laissant également la possibilité de vérifier le travail et de continuer à construire à partir de celui-ci.

En considérant ces avantages, il est donc nécessaire d'apporter un soin particulier à l'implémentation des propositions, afin d'éviter les erreurs qui pourraient s'y glisser et remettre en cause le reste de la solution, mais aussi pour pouvoir apporter des optimisations.

Ce chapitre est fortement orienté vers la technique, et apporte plus de détails aux implémentations concernant principalement les chapitres 2 et 3. Il est construit de la manière suivante : la section 1 introduit les notions importantes que les implémentations doivent refléter, la section 2 présente les détails techniques de l'architecture Hydre du chapitre 2, la section 3 met l'accent sur l'implémentation de TDM et de ses différents mécanismes du chapitre 3, et la section 4 conclut le chapitre.

4.1 Introduction : sûreté, optimisation et reproductibilité

Dans son article de 2020 [88], Kim met en avant que l'analyse de données a fait naître des besoins de détection des "erreurs de données" en plus des "erreurs de programmes", mais que les programmes ainsi que les outils et méthodes associés, par exemple pour le débogage, ne sont pas adaptés aux besoins de l'analyse de données. En conséquence, les *workflows* d'analyse nécessitant parfois plusieurs heures de calculs sont exécutés tout en ayant peu de garanties sur l'obtention d'un résultat conforme aux attentes. Bien que Kim se concentre principalement sur les techniques de débogage, d'autres facettes de l'analyse de données peuvent être remises en cause face à ce manque de sûreté.

Par exemple, Python est bien souvent reconnu comme le langage le plus demandé pour un poste de *data scientist* [1]. Il est souvent privilégié de par sa simplicité d'utilisation, sa facilité d'apprentissage et la rapidité des développements initiaux, mais cache pourtant de nombreux défauts qui le rendent en réalité peu adapté pour le développement de projets, et encore moins pour ceux liés à la science des données. En plus de ses soucis de performance [2], le plus préoccupant reste les problèmes de sûreté qui peuvent se retrouver également dans d'autres langages.

Une grande partie des problèmes de sûreté sont issus du typage dynamique, qui ne possède pas d'étape de compilation pour fixer le type des variables avant l'exécution. Dans son livre sur la programmation en Scala [132], M. Odersky justifie le choix du typage statique pour son langage par sa capacité à garantir à tout instant qu'une variable est bien du type attendu. C'est une garantie forte, car sans cela il faudrait réaliser une infinité de tests pour arriver à cette conclusion, qui vérifieraient chaque appel à chaque instant pour obtenir cette même garantie, puisque les tests permettent uniquement de valider le bon déroulement d'un scénario en fonction des paramètres fournis mais pas pour toutes les configurations. Cela est d'autant plus vrai en travaillant avec des données, du fait de leurs transformations successives qui peuvent modifier des types d'attributs. Cette vision est également soutenue par B. Meyer, qui écrit [128, p.597] que :

> La fiabilité introduite par l'utilisation du typage statique vient de la possibilité de détecter des erreurs qui, sinon, ne se seraient manifestées qu'au moment de l'exécution, et seulement lors de certaines exécutions. La règle qui impose de déclarer les entités et les fonctions [...] introduit une redondance dans le texte logiciel, ce qui permet au compilateur [...] de détecter des incohérences entre le but annoncé et l'utilisation effective d'une entité, d'une caractéristique ou d'une expression.

Les langages qui possèdent un typage statique comportent une compilation préalable à l'exécution pour

1. https://emploi.developpez.com/actu/314428/Python-serait-la-competence-la-plus-demandee-pour-la-science-des-donnees-au-detriment-de-R-selon-une-analyse-portant-sur-plus-de-15-000-offres-d-emploi-de-specialistes-des-donnees/
2. https://hackernoon.com/concurrent-programming-in-python-is-not-what-you-think-it-is-b6439c3f3e6a

pouvoir vérifier correctement les contraintes de type. En plus de pouvoir ajouter des sûretés supplémentaires, la compilation est aussi un moyen d'optimiser le *bytecode* produit [3]. Le compilateur peut avoir une vue d'ensemble sur les expressions utilisées, et peut alors en modifier l'ordre ou utiliser des équivalences plus adaptées, ce qui améliore les performances sans que l'utilisateur ait besoin de connaître lui-même les détails des mécanismes d'un langage pour arriver à un résultat similaire.

L'optimisation des programmes gagne de l'importance à mesure que les données à traiter deviennent de plus en plus volumineuses, et allongent proportionnellement le temps d'exécution, parfois de manière moins optimale que linéaire. Il est donc essentiel de rechercher des moyens de rendre l'exécution plus efficace, et les propriétés telles que le passage à l'échelle doivent au minimum être étudiées pour anticiper le comportement du programme lorsque la taille des données en entrée varie.

Les optimisations peuvent adopter plusieurs formes. Celle associée à la réorganisation des opérations avant leur exécution n'est d'ailleurs pas une nouveauté, puisqu'elle se rapproche des plans d'exécution des requêtes dans les SGBD [134]. D'autres systèmes ont eux aussi adopté ce mécanisme, comme Spark avec son optimiseur Catalyst [4], qui améliore sensiblement les performances lors de la manipulation de données. Mais on peut également trouver d'autres formes d'optimisations, telles que la parallélisation du programme s'il le permet, la refactorisation du code pour supprimer les opérations inutiles ou encore l'utilisation d'outils adaptés.

La qualité du code joue également un rôle dans son optimisation, puisqu'il devient alors plus simple d'identifier les sections à problèmes et d'intervenir pour les améliorer, sans que cela n'impacte le programme entier. De plus, en supplément des avantages évoqués ci-dessus, produire un code soigné et structuré est bénéfique d'un point de vue scientifique pour contribuer à la reproductibilité des travaux de recherche.

La reproductibilité est essentielle en recherche [18]. Elle permet d'exposer ses expériences et résultats afin d'ajouter de la transparence, de permettre à d'autres d'évaluer le travail mais aussi de faciliter son extension. La reproductibilité implique que suffisamment d'éléments soient donnés pour que les mêmes résultats puissent être obtenus par d'autres personnes que les auteurs des travaux.

Toutefois, mettre un programme en ligne sur un dépôt, parfois accompagné de données, ne suffit pas à en faire un travail reproductible de qualité. En effet, des instructions supplémentaires doivent être fournies pour indiquer comment et dans quelles conditions exécuter le programme, mais le code doit aussi être suffisamment soigné pour être lisible et compréhensible, et de cette manière faciliter sa critique et son utilisation.

De nombreuses initiatives sont menées pour encourager la reproductibilité, telles que la création du site *Papers with Code* [5], qui visait tout d'abord à héberger les articles concernant le *machine learning* accompagnés du code nécessaire pour valider les résultats de l'article, puis qui s'est étendu à d'autres domaines comme l'informatique plus largement, la physique ou les mathématiques. Ce site compte aujourd'hui plus de 50 000 articles. À l'inverse, *Papers without Code* [6] cherche à recenser les articles qui ne fournissent pas de code ou pas assez d'informations pour être reproductibles, afin de contacter les auteurs pour les encourager à fournir les éléments nécessaires pour rendre leur travail reproductible.

Ce chapitre reflète la manière dont la sûreté, l'optimisation et la reproductibilité ont été intégrés tout au long de mon travail de thèse, et ce quelles que soient les contributions. Il met l'accent plus particulièrement

3. `https://www.brown.edu/news/2021-07-01/tuplex`
4. `https://medium.com/microsoftazure/apache-spark-deep-dive-4e01022afc32`
5. `www.paperswithcode.com`
6. `www.paperswithoutcode.com`

sur les détails techniques de l'architecture Hydre, mais aussi sur les propriétés de sûreté et d'inférence de schéma de TDM, ainsi que sur la qualité du code produit lors de ces travaux.

4.2 L'architecture Hydre : une implémentation du patron Lambda+ Architecture

Afin d'effectuer des analyses dans le contexte du projet Cocktail, il faut tout d'abord pouvoir collecter les données servant à constituer un corpus d'étude. Il faut ainsi être capable d'absorber un flux important de plusieurs milliers de messages par seconde. En effet, le volume des données de Twitter est important (500.10^6 tweets sont publiés quotidiennement [7]), et les tweets contiennent de nombreuses informations qu'il faut extraire comme l'utilisateur émetteur, la date d'émission, les hashtags utilisés, les utilisateurs mentionnés, la localisation, les URL citées, etc. En conséquence, une plateforme d'analyse à haute performance doit être capable d'absorber, en temps réel, un flux important de tweets, d'effectuer des analyses elles aussi en temps réel et de stocker les données en vue d'analyses plus complexes réalisées en temps différé. L'architecture Hydre a été conçue dans ce but.

4.2.1 Les spécifications de l'architecture

L'architecture Hydre, évoquée dans le chapitre 2, implémente le patron de la Lambda+ Architecture et profite de ses avantages, afin de répondre aux besoins de collecte, gestion et analyse des données issues de Twitter. Le découplage de ses différents composants permet une forte résistance aux pannes et facilite aussi la mise à jour des composants sans impacter le reste de l'architecture.

4.2.1.1 Architecture générale

Pour rappel, la figure 4.1 présente l'architecture Hydre globale. Elle est majoritairement développée à l'aide du langage Scala [132], et agrège différents composants :

- le composant de gestion des flux des messages, qui utilise Kafka pour véhiculer les différents messages entre les composants de l'architecture ;
- le système de collecte des données (*data traffic controller*), faisant appel aux APIs de Twitter pour collecter les tweets au format JSON et les envoyer sous forme de messages Kafka ;
- le *master dataset* qui stocke les données brutes dans un entrepôt Hadoop HDFS pour permettre la reprise sur pannes, et pour pouvoir traiter à nouveaux les données en cas d'évolutions de schéma ;
- le composant de *streaming ETL* permet d'insérer les tweets dans les différents SGBD du *polystore* ;
- le composant de *real-time insights* est chargé de calculer des indicateurs macroscopiques sur la collecte afin d'avoir une vue d'ensemble en temps réel des tendances ;
- le composant *storage* est implémenté par un *polystore*, qui met à disposition les tweets modélisés avec différents schémas ayant chacun ses propres caractéristiques. Il est exploité par différents outils d'analyse, pouvant récupérer les données dans un modèle adapté aux algorithmes.

L'activité du *data traffic controller* est décomposée en collectes, qui elles-mêmes peuvent être décomposées en corpus. Un corpus permet de regrouper des catégories de mots-clés autour d'une thématique résultant d'un choix de critères de collecte (mots-clés, comptes, hashtags, etc.). Chaque machine de collecte ne peut être associée qu'à un seul corpus d'une seule collecte.

7. `http://www.internetlivestats.com/twitter-statistics/`

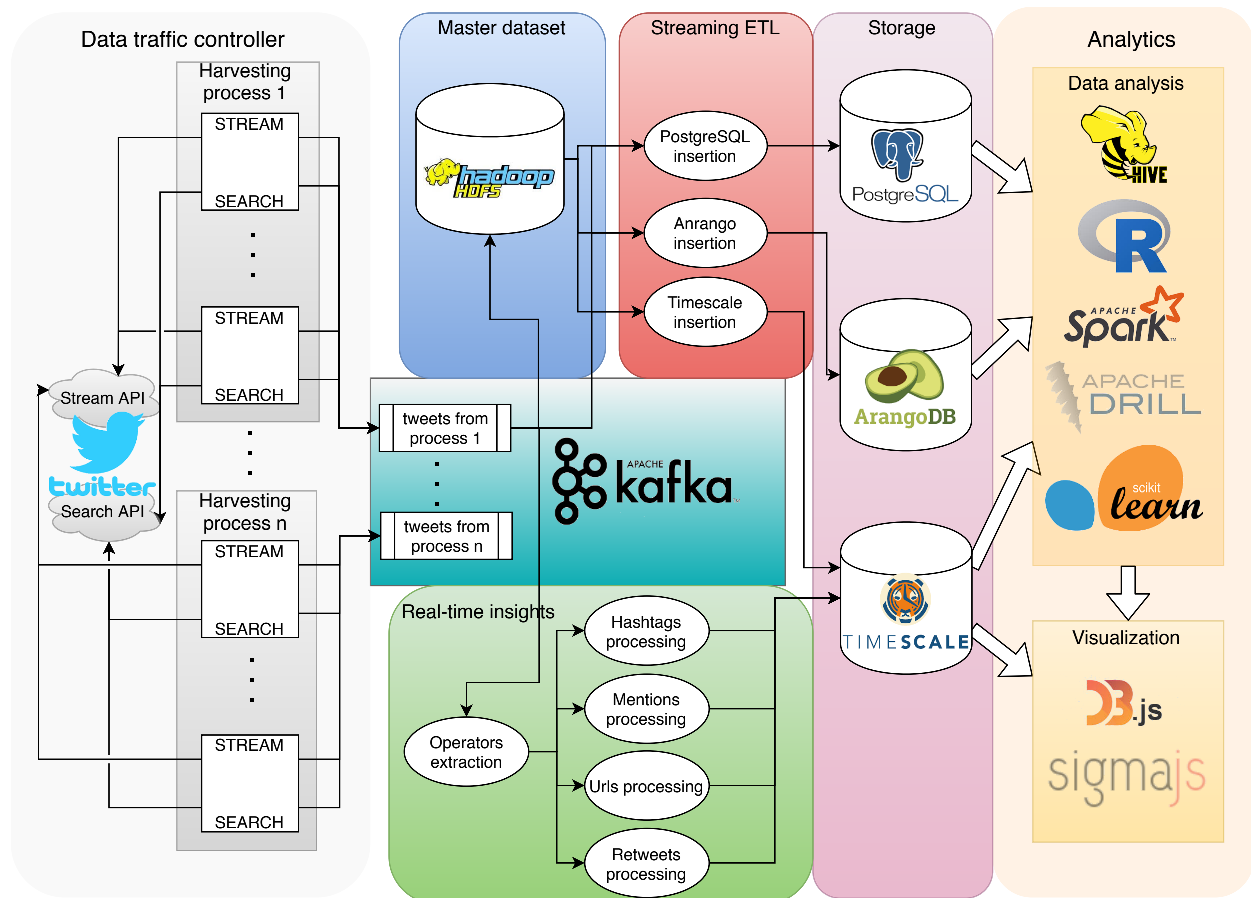


FIGURE 4.1 – L'architecture Hydre détaillée

4.2.1.2 La collecte des données

Afin de collecter des tweets, Twitter met à disposition deux APIs : l'API *search*[8] et l'API *stream*[9]. Bien que le but de ces APIs soit le même, leur fonctionnement est très différent. En effet, la première permet de collecter des tweets ayant été produits au maximum 7 jours plus tôt et qui correspondent à la requête qui a été fournie à l'API, tandis que la deuxième permet d'enregistrer une requête auprès de Twitter, qui renverra ensuite tous les tweets nouvellement produits correspondant aux critères de collecte spécifiés. Les critères sont insérés dans des requêtes qui filtrent les tweets sur des mots-clés, des hashtags, des utilisateurs, etc. Les deux APIs nous fournissent les tweets au format JSON. Dans leur version gratuite, ces APIs imposent toutefois quelques limitations par compte utilisé. En ce qui concerne l'API *search*, il est seulement possible d'envoyer au plus 180 requêtes de 10 critères par créneaux de 15 minutes, en récupérant 100 tweets maximum à chaque fois. L'API *stream*, quant à elle, limite le nombre de tweets récupérés par l'application à 1% du trafic total de Twitter par machine connectée, et si les tweets ne sont pas consommés assez rapidement par

8. https://developer.twitter.com/en/docs/tweets/search/overview
9. https://developer.twitter.com/en/docs/tweets/filter-realtime/overview

l'application, la connexion avec Twitter peut être coupée. Pour communiquer avec ces APIs, nous utilisons la librairie Twitter4j [10].

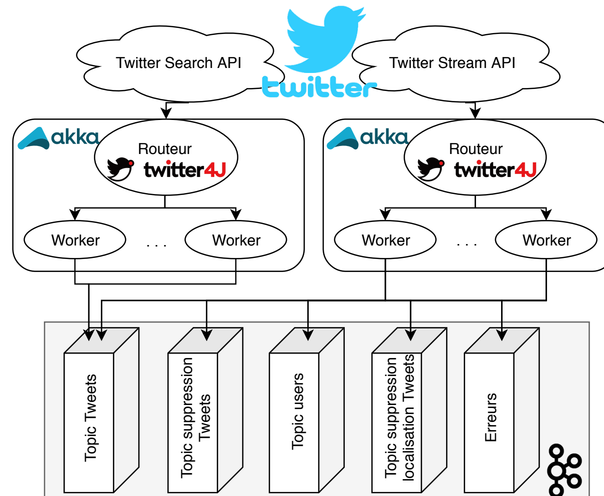


Figure 4.2 – Collecte de tweets et articulation avec les topics Kafka

Pour gérer les tweets que nous récoltons, nous utilisons des acteurs Akka [64] disponibles avec Scala. Akka est une implémentation de l'*Actor Model*, servant à réaliser des applications distribuées à base d'échanges de messages de type (K, V) où K est la clé et V la valeur (le contenu du tweet). Si la rapidité n'est pas une caractéristique primordiale pour l'API *search* au vu de ses limitations, elle prend de l'importance pour l'API *stream* dans le cas où le flux de tweets est conséquent. Nous utilisons la fonctionnalité *Routing* d'Akka, en définissant un certain nombre de *Workers* auxquels le routeur transmet les messages qu'il reçoit pour qu'ils puissent les traiter (figure 4.2). Le nombre de *Workers* peut être fixe, comme c'est le cas pour notre traitement à la réception des tweets provenant de l'API *search*, ou il peut être adaptable dynamiquement au volume de messages que le routeur reçoit, ce que nous utilisons pour traiter les tweets provenant de l'API *stream*, permettant de ce fait de supporter la propriété d'élasticité. L'architecture supporte le traitement de plusieurs corpus simultanément, chaque corpus dispose d'une ou plusieurs machines de collecte en fonction de la volumétrie de tweets attendue pour le corpus (figure 4.1, partie grisée).

4.2.1.3 Système de gestion des flux

Le système Apache Kafka [101] est utilisé pour servir d'intermédiaire entre la partie collecte et les autres composants de l'architecture. Kafka est un *Middleware* Orienté Messages (MOM) initialement développé par LinkedIn et rendu *open source* en 2011. Il est conçu pour traiter les données massives qui arrivent avec une grande vélocité, et est une référence dans le domaine puisqu'il est adopté par de nombreuses entreprises. Les messages (sous forme de couple clé valeur) sont publiés dans des topics par les producteurs. Contrairement à la plupart des MOM, Kafka ne supprime pas les messages dès qu'ils sont consommés, mais les enregistre sur disque jusqu'à ce que le volume ou le délai de rétention paramétrés aient été atteints. Ce fonctionnement

10. `http://twitter4j.org/en/index.html`

permet également de traiter les messages dans plusieurs applications, avec chacune leur propre rythme de consommation des messages. Le parallélisme est assuré en fragmentant les topics en partitions, dans lesquelles les messages sont envoyés en fonction de leur clé. Les consommateurs s'organisent alors en groupes, chaque consommateur se voyant affecter une ou plusieurs partitions, sans qu'une même partition ne soit partagée entre deux consommateurs d'un même groupe. Un serveur Kafka est composé de plusieurs *brokers* qui se partagent les partitions d'un topic, mais également des réplicats de partitions contribuant à l'obtention d'une forte résistance aux pannes. Dans l'architecture Hydre, Kafka est l'élément central. Les tweets collectés sont envoyés rapidement en tant que messages, puis les différents composants viennent les consommer. La rétention des messages, calculée en fonction de la capacité des disques et de la volumétrie du flux de données, permet de maintenir et de faire évoluer les composants consommateurs sans impacter le reste de l'architecture. Kafka permet d'assurer un fort découplage entre tous les composants rendant ainsi possibles des interventions sur des parties de l'architecture pour des mises à jour ou des correctifs.

4.2.1.4 Le composant *real-time insights*

Le rôle principal du composant *real-time insights* est, dans notre cas, de produire des indicateurs macroscopiques sur une collecte en cours et de détecter des évènements, comme les hashtags les plus utilisés, leur fréquence, ou les mentions faites d'un utilisateur. Ces indicateurs prennent la forme de séries temporelles ayant une granularité temporelle d'une heure. Cette couche est développée à l'aide de Kafka Streams [100], et les résultats sont insérés dans une base de données TimescaleDB [11].

Le fonctionnement de ce composant (figure 4.1 partie verte) débute par un processus qui se charge de séparer les différents éléments des tweets (les hashtags, les mentions, etc.). Ces éléments sont envoyés dans différents topics Kafka dans le but d'être traités chacun par un processus spécifique. Les processus sont associés à un *state store* de Kafka, qui leur permet de garder un état en mémoire. Nous nous en servons pour compter combien de fois un même élément a été rencontré sur une plage de temps donnée. Le compte mis à jour est ensuite inséré dans TimescaleDB.

Ce composant nous sert aussi à mettre à jour des informations concernant les tweets ou les utilisateurs : par exemple dans le cas d'un retweet, nous traitons aussi le tweet d'origine dont les attributs représentant le nombre de *likes* ou de retweets ont été mis à jour, et pour les utilisateurs cela nous permet de voir l'évolution du nombre de *followers* ou les changements de *screen name*.

4.2.1.5 Le *master dataset* et le composant de *streaming ETL*

Lorsque de nouveaux tweets sont collectés et envoyés *via* Kafka, ils sont stockés dans le *master dataset* sous forme brute (c'est-à-dire le format JSON). Afin de remplir cette tâche, nous utilisons Hadoop, qui est un *framework* permettant de distribuer de larges volumes de données sur un *cluster* et d'y effectuer des traitements en utilisant le paradigme *map-reduce*. L'ajout de machines au *cluster* entraîne un passage à l'échelle efficace grâce au principe de scalabilité horizontale, de ce fait augmentant à la fois la capacité de calcul et de stockage. De nombreux autres outils tiers constituent un véritable éco-système. Ils permettent d'abstraire certains mécanismes et proposent des fonctionnalités spécifiques.

Dans le composant de *streaming ETL*, un ensemble de processus est chargé de récupérer les nouvelles données depuis Kafka et, pour chaque SGBD composant notre *polystore*, de les insérer tout en respectant le

11. www.timescale.com

schéma défini. De cette manière, les données que nous récoltons sont transformées dans plusieurs modèles afin de pouvoir ensuite être exploitées par les algorithmes d'analyse. Différents types de bases de données sont utilisés pour correspondre aux besoins des algorithmes : TimescaleDB pour les séries temporelles, ArangoDB [12] pour les modèles graphes et PostgreSQL [13] pour stocker les données relationnelles des tweets comme la géo-localisation ou les informations des utilisateurs. Grâce à l'organisation des éléments de notre architecture, il est aisé de rajouter une autre base de données, de modifier ou ajouter un schéma d'insertion si cela est nécessaire. Il suffit en effet de modifier ou de rajouter un processus d'insertion dans le composant de *streaming ETL* afin que ces changements soient pris en compte.

4.2.1.6 La gestion de l'architecture

FIGURE 4.3 – Interface de *monitoring* des machines de collecte

Le **superviseur** est l'application manipulant toute l'architecture. Il permet d'affecter des machines de collecte et des critères de collecte à un corpus d'une collecte. Il sert également à démarrer et arrêter les machines de collecte, et à leur diffuser les critères. Il gère aussi les traitements réalisés dans les différents composants de l'architecture, en démarrant ou arrêtant les processus pour chaque corpus des collectes. Les machines de collecte ainsi que les composants de l'architecture exposent leurs différentes fonctionnalités à travers des *Web Services*, dont le superviseur se sert pour contrôler l'ensemble de l'architecture.

Un **système de *monitoring*** a été mis en place en utilisant Prometheus [14] et Grafana [15]. Il permet de surveiller l'état de Kafka, des différents composants de l'architecture, ainsi que des machines de collecte. De plus, en utilisant dans les machines de collecte le *framework* Kamon [16], qui consiste à créer des *metrics* de

12. www.arangodb.com
13. www.postgresql.org
14. https://prometheus.io/
15. https://grafana.com/
16. https://kamon.io/

monitoring personnalisées, il est possible de donner un aperçu du nombre de tweets collectés par chaque machine (figure 4.3).

4.2.2 L'architecture physique

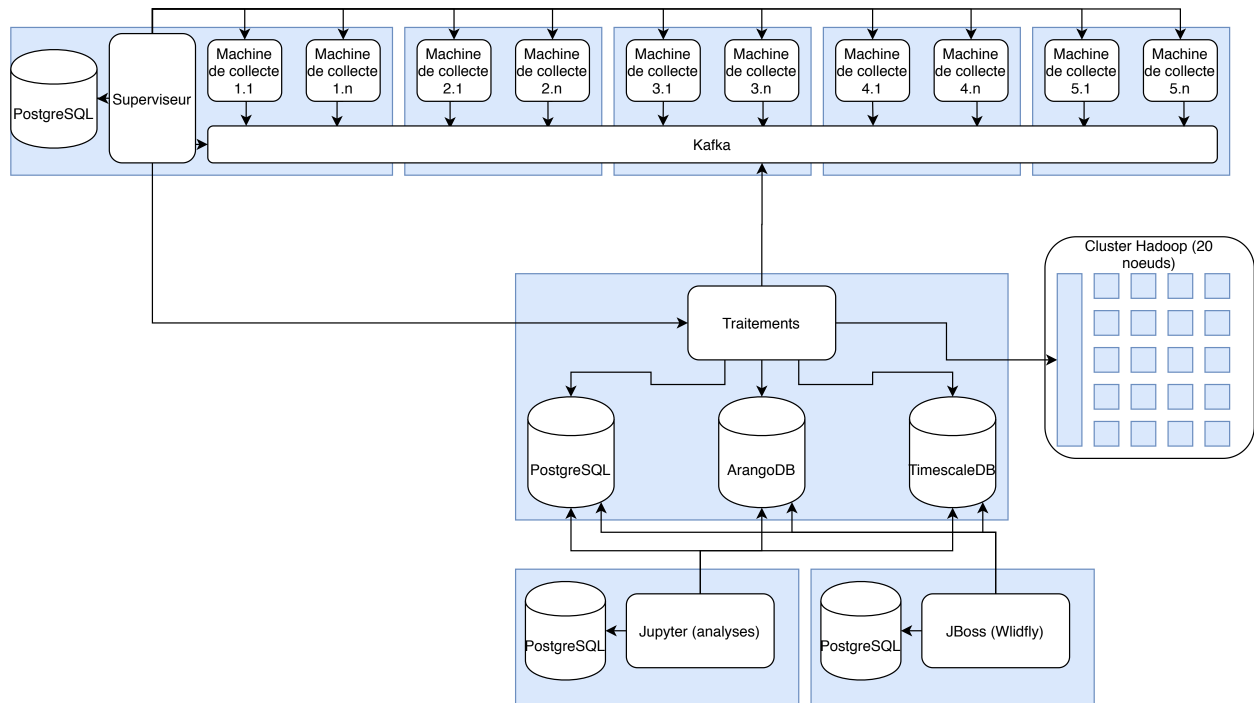


FIGURE 4.4 – Architecture physique utilisée pour le déploiement d'Hydre

L'architecture complète a été déployée sur un ensemble de machines physiques (figure 4.4). Les différentes machines de collecte sont réparties sur 5 serveurs identiques. Ces serveurs hébergent également un *cluster* Kafka, avec un facteur de réplication de 3, ce qui permet de conserver un fonctionnement normal même en ayant 2 serveurs à l'arrêt. Le superviseur ainsi que la base de données qui lui est associée sont hébergés sur un de ces serveurs. L'application de traitements de l'architecture (composants *streaming ETL* et *real-time insights*), ainsi que le *polystore*, sont hébergés sur une seule machine. Un *cluster* Hadoop de 20 nœuds constitue le *master dataset*.

Pour exploiter les données du *polystore*, 2 machines sont utilisées : une machine hébergeant un serveur d'applications JBoss Wildfly et une base de données associée afin de produire des synthèses des données disponibles, et une autre machine hébergeant des *notebooks* Jupyter avec également une base de données dédiée, qui permet de réaliser des analyses exploratoires sur les données collectées. Leur exploitation est détaillée dans le chapitre 5.

4.3 Implémentation de TDM

TDM est disponible en tant que librairie *open-source*. Il est donc nécessaire de présenter le code de manière claire, de le commenter, et de le structurer conformément aux bonnes pratiques afin de faciliter sa compréhension et son utilisation. En conséquence, la librairie est accompagnée de tests permettant de valider son comportement, et le *code coverage*, c'est-à-dire le pourcentage de lignes de code couvertes par les tests, est de 90% [17]. Les expériences et les cas d'exemple qui ont été réalisés avec TDM sont également en libre accès [18], pour à la fois faciliter la reproduction des expériences mais également donner des exemples pratiques de l'utilisation de TDM.

De plus, une attention particulière a été portée à la sûreté du typage ainsi qu'à la capacité d'inférence de schéma de la librairie TDM. Ces propriétés, reposant largement sur les *implicits*, sont expliquées dans la suite de cette section, plus techniquement que dans le chapitre 3, dans lequel nous avons insisté sur les principes des mécanismes nécessaires à la sûreté du typage et l'inférence de schéma.

4.3.1 Les *implicits*

Les *implicits* sont un mécanisme du langage Scala qui permet de déléguer une partie de la logique du code au compilateur [19]. Initialement, ce mécanisme peut s'utiliser pour déclarer une variable en tant qu'*implicit*, puis à appeler des méthodes ayant des paramètres *implicits* compatibles sans avoir besoin de fournir explicitement cette variable.

Par exemple, si l'on souhaite réaliser un programme simple, avec deux fonctions, une pour dire bonjour et une autre pour dire au-revoir, il est possible de définir ces fonctions de la manière suivante :

```
1 def sayHello()(implicit name: String) = {
2   println(s"Hello $name!")
3 }
4 def sayGoodbye()(implicit name: String) = {
5   println(s"Goodbye $name!")
6 }
```

On peut alors appeler ces fonctions sans fournir de paramètre, en déclarant au préalable une variable *implicit* de type `String` :

```
1 implicit val myName = "world"
2 sayHello() // Affiche "Hello world!"
3 sayGoodbye() // Affiche "Goodbye world!"
```

Les *implicits*, utilisés de cette façon, peuvent servir à définir un contexte global et à s'en servir dans différentes fonctions, sans pour autant l'appeler explicitement tant qu'il est inclus dans le *scope* courant. L'avantage est de garder le code propre, d'éviter les répétitions d'un même paramètre dans tous les appels

17. `https://coveralls.io/github/AnnabelleGillet/TDM`
18. `https://github.com/AnnabelleGillet/TDM-experiments`
19. Il est à noter que les exemples de la suite de cette section sont réalisés avec la version 2 de Scala, la version 3 récemment disponible intègre de nombreux nouveaux mécanismes très utiles au typage (comme l'union de types). Toutefois, Spark n'est pour le moment pas disponible pour Scala 3, et TDM est donc développé avec Scala 2.

de fonctions, mais en même temps de laisser suffisamment de liberté pour tout de même fournir un paramètre explicitement en cas de besoin. En reprenant l'exemple précédent, il est possible d'exécuter le code suivant :

```
1 implicit val myName = "world"
2 sayHello() // Affiche "Hello world!"
3 sayGoodbye()("world2") // Affiche "Goodbye world2!"
```

Les utilisations sont alors très variées : on peut créer une configuration globale, un format standard de données à traiter, les caractéristiques de l'environnement, etc. tout en gardant la possibilité de reprendre la main en fournissant un paramètre autre que celui prévu par défaut. Mais l'intérêt des *implicits* ne s'arrête pas là, et en poussant les fonctionnalités du langage il est possible d'arriver à des résultats garantissant une forte flexibilité.

4.3.2 La sûreté du typage

Scala étant un langage portant une grande considération aux types, des constructions spécifiques existent, et permettent de mettre en place des vérifications plus poussées. Par exemple, l'opérateur `=:=` vérifie qu'un type est bien le même qu'un autre type. Il est possible de le coupler avec les *implicits* de la manière suivante :

```
1 def sayAmount[T](amount: T)(implicit checkType: T =:= Int) = {
2   println(s"The amount is $amount")
3 }
4
5 sayAmount(10) // Affiche "The amount is 10"
```

Cet extrait de code nous permet de vérifier que le type `T` est bien `Int`, et pas autre chose. Par exemple, la ligne suivante provoque une erreur de compilation, puisque le paramètre est de type `Long` :

```
1 sayAmount(10L) // Ne compile pas, message d'erreur : "Cannot prove that Long =:= Int."
```

Si à ce niveau l'utilisation de l'*implicit* n'a que peu d'intérêt puisque le paramètre `amount` pourrait être renseigné comme étant directement du type `Int`, l'exemple peut être poussé plus loin pour montrer une utilité. On peut par exemple l'étendre pour accepter un paramètre soit de type `Int`, soit de type `String`, sans pour autant définir deux fois la fonction. Il est possible de procéder de la manière suivante :

```
1 @implicitNotFound("Type ${T} is not authorized. Types authorized: Int or String.")
2 sealed trait AuthorizedType[T] extends Serializable
3
4 final object AuthorizedType {
5   implicit def IntAuthorized[T](implicit eq: T =:= Int): AuthorizedType[T] = new
      AuthorizedType[T] {}
6   implicit def StringAuthorized[T](implicit eq: T =:= String): AuthorizedType[T] =
      new AuthorizedType[T] {}
7 }
```

Plusieurs éléments de cet extrait de code méritent d'être commentés :

- le `sealed trait` de la ligne 2 indique que le `trait AuthorizedType` ne pourra pas être étendu

en dehors du fichier dans lequel le `trait` a été défini, fournissant de cette manière un contrôle fin sur les ajouts relatifs à ce `trait` (par exemple si une librairie est utilisée dans un programme, le programme ne pourra pas étendre les `sealed trait` de la librairie) ;

— l'annotation `implicitNotFound` de la ligne 1 permet de définir un message d'erreur personnalisé lorsqu'un *implicit* de type `AuthorizedType` ne peut pas être résolu par le compilateur.

L'objet `AuthorizedType` est défini de la ligne 4 à la ligne 7, et contient les fonctions déclarées comme *implicits*, qui servent à obtenir une instance de type `AuthorizedType` lorsque leur propre *implicit* contrôlant le type paramétré est résolu, entraînant de ce fait la résolution des *implicits* de type `AuthorizedType`. Il ne reste ensuite plus qu'à créer une fonction avec un paramètre *implicit* de type `AuthorizedType` et de s'en servir comme suit :

```
1 def sayAmountOfAuthorizedType[T](amount: T)(implicit checkType: AuthorizedType[T]) = {
2   println(s"The amount is $amount")
3 }
4
5 sayAmountOfAuthorizedType(10) // Affiche "The amount is 10"
6 sayAmountOfAuthorizedType("10") // Affiche "The amount is 10"
```

En testant la fonction `sayAmountOfAuthorizedType` avec un type non autorisé, on constate que le message d'erreur du compilateur correspond à celui définit avec l'annotation `implicitNotFound` :

```
1 sayAmountOfAuthorizedType(10L) // Ne compile pas, message d'erreur : "Type Long is
    not authorized. Types authorized: Int or String."
```

En supplément de l'opérateur `=:=`, Scala définit nativement d'autres constructions similaires. Mais la librairie shapeless [65] constitue une référence incontournable au niveau de la manipulation des types, et propose de nombreux opérateurs nouveaux, parfois couplés aux types de la librairies tels que les `HList`. L'utilisation des *implicits* comme présentée ci-dessus nous sert à obtenir une sûreté du typage avancée dans TDM. Il est toutefois possible de pousser les *implicits* encore plus loin afin d'obtenir le mécanisme d'inférence de schéma, très important lors de la manipulation de données.

4.3.3 L'inférence de schéma

Certains opérateurs de manipulation de données, tels que la projection ou la jointure naturelle, entraînent une modification de schéma. Comme l'objet de départ contient un nombre variable de colonnes ou dimensions (pour un `DataFrame` ou un tenseur TDM), il est difficile de pouvoir faire déduire au compilateur le nouveau schéma obtenu à la suite de l'application d'un de ces opérateurs. D'ailleurs, Spark ne le gère pas, puisque son `Dataset`, sa structure de données la plus poussée en terme de sûreté du typage, demande à l'utilisateur de fournir explicitement la `case class` (une sorte de classe simplifiée en Scala) correspondant au nouveau schéma obtenu, ce qui de ce fait augmente le risque d'erreurs si la `case class` ne correspond pas au résultat (inversion de colonnes, mauvaise correspondance de types, etc.). Dans l'implémentation de TDM j'ai porté un soin particulier à cet aspect de la sûreté des programmes et j'ai réussi à mettre en place un mécanisme d'inférence de schéma, comme évoqué dans le chapitre 3, grâce aux `HList` de shapeless et aux *implicits*. Ce mécanisme est détaillé de manière plus technique ci-après.

Pour rappel, une `HList` de shapeless est une liste constituée d'éléments de types hétérogènes, dans

laquelle le détail de chaque type est conservé, sans avoir besoin de faire appel à un super-type commun qui dilue l'information. Il est possible de s'en servir de la manière suivante pour obtenir l'union des types de deux `HList` :

```
import shapeless._

def unionHList[L1 <: HList, L2 <: HList, LOut <: HList](l1: L1, l2: L2)(implicit
    union: Union.Aux[L1, L2, LOut], ta: Typeable[LOut]) = {
  println(s"${ta.describe}")
}

val hList1 = "foo" :: true :: HNil // Type inféré : String :: Boolean :: HNil
    (sous-type d'HList)
val hList2 = true :: 123L :: HNil // Type inféré : Boolean :: Long :: HNil (sous-type
    d'HList)

unionHList(hList1, hList2) // Affiche "String :: Boolean :: Long :: HNil"
```

La ligne 3 est la signature de la fonction. Plusieurs éléments sont intéressants. Les deux `HList` sur lesquelles nous souhaitons réaliser une union des types sont définies comme paramètres, en spécifiant que leur type est un sous-type de `HList`. Deux paramètres *implicits* sont également définis. Celui de type `Typeable` sert uniquement dans l'exemple pour pouvoir appeler la fonction `describe` et afficher le type résultat. L'autre paramètre sert à produire l'union des deux `HList`. Le `Aux` du type `Union` sert à faire référence à ce qu'on appelle le *aux pattern*[20], qui peut être résumé comme une technique servant à accéder à un type interne à un autre type. Dans notre exemple, il sert à accéder au type `LOut`, qui représente l'union des types des deux `HList`. Si l'on crée deux `HList` pour les passer en paramètres de la fonction ainsi définie, on constate que l'union des types a été correctement réalisée par le compilateur, et ce sans avoir eu à fournir explicitement des informations supplémentaires lors de l'appel de la fonction. C'est sur ce mécanisme que repose l'inférence de schéma de TDM, grâce aux dimensions définies de manière unique avec les *phantom types* présentés dans le chapitre 3.

4.3.4 L'utilisation de TDM

Comme les détails des mécanismes de sûreté du typage et d'inférence de schéma sont cachés à l'utilisateur, cela se traduit par une simplicité lors de l'utilisation de TDM. Le code ci-dessous illustre le chargement d'un fichier CSV dans un `DataFrame`, qui subit quelques manipulations avant d'être chargé dans un tenseur. La partie de code qui utilise TDM commence à la ligne 21, et consiste en la création des dimensions (lignes 21 à 23), la création du tenseur (lignes 25 à 29), l'exécution de la décomposition CANDECOMP/-PARAFAC (ligne 31) et l'extraction du résultat de la décomposition pour chacune des dimensions (lignes 33 à 35). Cet extrait de code correspond à l'analyse du jeu de données regroupant les interactions entre les enfants d'une école primaire, étudié plus en détail dans le chapitre 5.

```
var df = spark.read.format("csv").option("header","false")
      .option("sep", "\t").load("datasets/primaryschool.csv")
      .toDF("time", "student1", "student2", "class1", "class2")
```

20. https://gigiigig.github.io/posts/2015/09/13/aux-pattern.html

```
4       // Time slices of 5 minutes
5       .withColumn("time", floor(col("time") / (3600 / 12)).cast("integer"))
6       .withColumn("val", lit(1))
7
8   // Union the DataFrame with itself to have student1 -> student2 and student2 ->
        student1
9   df = df.union(
10     df.select("time", "student1", "student2", "class1", "class2", "val")
11  ).dropDuplicates
12
13  // Concatene the class to the student id to make them more readable
14  df = df.withColumn("student1", concat(col("class1"), lit("-"), col("student1")))
15      .withColumn("student2", concat(col("class2"), lit("-"), col("student2")))
16      .drop("class1")
17      .drop("class2")
18
19  // TDM part
20  // Create dimensions
21  object Student1 extends TensorDimension[String]
22  object Student2 extends TensorDimension[String]
23  object Time extends TensorDimension[Int]
24  // Create tensor
25  val tensor = TensorBuilder[Int](df)
26              .addDimension(Student1, "student1")
27              .addDimension(Student2, "student2")
28              .addDimension(Time, "time")
29              .build("val")
30  // Run CP decomposition
31  val kruskal = tensor.canonicalPolyadicDecomposition(13, norm = Norm.L2,
        computeCorcondia = true, minFms = 0.999)
32  // Extract result
33  val student1Tensor = kruskal.extract(Student1)
34  val student2Tensor = kruskal.extract(Student2)
35  val timeTensor = kruskal.extract(Time)
```

4.4 Conclusion

Ce chapitre met en avant l'importance qui a été accordée à la qualité des implémentations liées aux contributions de cette thèse. Cette qualité concerne deux points principaux, qui sont la connaissance et l'application des technologies et méthodes les plus adaptées au but recherché, mais aussi la mise en œuvre des bonnes pratiques et l'ajout de documentation pour faciliter la compréhension du travail exposé.

Les contributions techniques principales, à savoir l'architecture Hydre et la librairie TDM, ont été détaillées dans ce chapitre. L'architecture Hydre utilise de nombreuses briques logicielles pour assurer les fonctionnalités nécessaires à la collecte, au stockage et à l'analyse des données de Twitter, comprenant l'utilisation du *stream processing* et la mise en place d'un *polystore*. La libraire TDM de son côté bénéficie de l'utilisation de mécanismes puissants reposant sur les *implicits* du langage Scala, de manière à s'assurer de la

validité du programme avant son exécution, qui pourrait consommer beaucoup de ressources avant d'échouer, mais aussi à éviter les erreurs fonctionnelles ayant un impact sur l'interprétation mais étant difficilement identifiables. De plus, les différentes expériences et exemples d'utilisation réalisés sont rendus disponibles en *open-source* afin de montrer comment utiliser TDM dans un contexte d'analyses.

Publications relatives au chapitre

*****

Les contributions présentées dans les chapitres précédents ont toutes une part technique importante, qui est intégrée dans la plupart des publications qui leur sont dédiées. Ces contributions prennent différentes formes, et de ce fait leur contenu est adapté en fonction du public, la présentation de TDM à ScalaCon en mai 2021 devant plus de 500 professionnels et industriels utilisant le langage Scala (présentation [C5]) étant celle qui contient le plus d'éléments techniques. La présentation [C2] est une vulgarisation de l'architecture, tandis que la présentation [C3] est destinée aux personnels et utilisateurs des centres de calcul.

[C1] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **Lambda Architecture pour une analyse à haute performance des données des réseaux sociaux**. In : *INFormatique des ORganisations et Systèmes d'Information et de Décision (INFORSID)*. 2019, pp. 1–16.

[C2] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **Une plateforme haute performance pour l'exploitation des données massives**. In : *DataBFC2 (`https://databfc2.sciencesconf.org/`)*. 2019.

[C3] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **Lambda+ Architecture — Un patron d'architecture haute performance pour le traitement des Big Data**. In : *Journées Calcul et Données (JCAD, `https://jcad2020.sciencesconf.org/`)*. 2020.

[C4] Annabelle Gillet, Éric Leclercq, Marinette Savonnet, Nadine Cullot. **Empowering big data analytics with polystore and strongly typed functional queries**. In : *Proceedings of the 24th Symposium on International Database Engineering & Applications (IDEAS)*. 2020, pp. 1–10.

[C5] Annabelle Gillet, Éric Leclercq. **TDM : Breaking through dimensions with tensors**. In : *ScalaCon (`http://www.scalacon.org/`)*. 2021.

[C6] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **Évolution et formalisation de la Lambda Architecture pour des analyses à hautes performances — Application aux données de Twitter**. In : *Revue ouverte d'ingénierie des systèmes d'information (ISI)* Numéro 1 (2021), pp. 1–26.

[C7] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **Lambda+, the renewal of the Lambda Architecture : Category Theory to the rescue**. In : *33rd Conference on Advanced Information Systems Engineering (CAiSE)*. Springer LNCS, 2021, pp. 1–15.

[C8] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **MuLOT : Multi-level optimization of the canonical polyadic tensor decomposition at large-scale**. In : *25th European Conference on Advances in Databases and Information Systems (ADBIS)*. 2021, pp. 1–15.

Chapitre

5

Un observatoire des données sociales : méthodes et techniques de collecte et d'analyse

*****

5.1 Introduction . . . 110
 5.1.1 Le besoin d'un observatoire des données sociales . . . 110
 5.1.2 La démarche . . . 111
 5.1.3 Les cas d'étude . . . 112
5.2 De la collecte à la constitution des corpus et leur mise à disposition . . . 113
 5.2.1 La collecte des données . . . 113
 5.2.2 La constitution des corpus d'étude . . . 113
 5.2.3 La mise à disposition des corpus . . . 114
5.3 Techniques d'analyses . . . 114
 5.3.1 Les analyses macroscopiques . . . 116
 5.3.2 Les analyses mésoscopiques . . . 118
 5.3.3 Les analyses microscopiques . . . 121
5.4 Conclusion . . . 131

L'analyse des données débute souvent par une question particulière, qui conduira ensuite à la recherche ou à la construction d'un corpus d'étude, sur lequel divers algorithmes pourront être exécutés afin d'éclairer la question initiale. Dans le cadre du projet Cocktail, nous réalisons des processus complets d'analyse, depuis la collecte des données jusqu'à la présentation des résultats à des chercheurs en sciences sociales et en sciences de la communication qui prennent ensuite en charge la partie interprétation.

Ce chapitre relate de l'organisation mise en place dans le cadre de l'observatoire pour exécuter plusieurs types d'analyses à des niveaux différents. Cette organisation prend en compte les interactions avec les

chercheurs en sciences sociales et les experts métiers ayant peu ou pas de connaissances en science des données. Bien que cette démarche ait été définie et utilisée spécifiquement dans le projet Cocktail, elle est assez flexible pour être généralisée à d'autres projets ou circonstances. Ce chapitre présente également les différentes techniques d'analyses utilisées, qui sont illustrées à travers plusieurs cas d'étude.

Le chapitre est construit de la manière suivante : la section 1 décrit les besoins conduisant à la création d'un observatoire, la démarche de réalisation des analyses ainsi que les cas d'étude réalisés dans le cadre du projet Cocktail, la section 2 détaille le processus de collecte puis de mise à disposition des données, la section 3 expose les différentes techniques d'analyse utilisées à différents niveaux de granularité, et la section 4 conclut le chapitre.

5.1 Introduction

Un observatoire visant à analyser les données requiert, en plus des moyens techniques, une collaboration entre les analystes et les experts métiers, afin que les premiers puissent fournir des résultats pertinents aux questions des seconds. Il est donc nécessaire de définir une démarche adaptée, qui permette de réaliser des analyses de qualité.

5.1.1 Le besoin d'un observatoire des données sociales

Dans le domaine des sciences humaines et sociales, l'apparition des Réseaux Sociaux Numériques (RSN) a permis d'avoir accès à de gros volumes de données à grandes dimensions et a eu les mêmes conséquences que l'apparition du microscope ou du télescope en biologie et en astronomie. Ces données ont provoqué une rupture d'échelle dans les analyses effectuées par les chercheurs et ont introduit le besoin d'une approche computationnelle des sciences sociales (*Computational Social Sciences*) [106, 40, 33]. Ce nouveau paradigme, comme le constatent les auteurs des différents articles cités précédemment, nécessite une interdisciplinarité forte.

Du point de vue de la science des données, les questions de recherche des chercheurs en sciences humaines et sociales sont abordées avec l'objectif de créer des modèles exécutables [1] et de réaliser des expérimentations sur un corpus de données défini pour une question particulière. Les résultats de l'exécution des modèles sont ensuite transmis aux chercheurs en sciences humaines et sociales pour raffinement et interprétation. Ces résultats servent soit à éclairer leur question de recherche, soit à réfuter leurs propres modèles. Ces interactions introduisent ainsi un cycle de développement des analyses.

La dualité entre les indicateurs macroscopiques et les analyses exploratoires présentée dans le chapitre 2 est de grande importance dans un observatoire traitant de données massives. En effet, il faut être capable d'identifier les données servant à répondre à une question de recherche, vérifier la validité des corpus, et exécuter des analyses variées sur ces corpus. L'architecture Hydre, qui implémente le patron de la Lambda+ Architecture, a été développée dans cet objectif. Elle permet de calculer des indicateurs macroscopiques aidant à détecter les tendances dans les données, mais aussi de laisser les données à disposition pour gagner en

1. Ce sont des abstractions logicielles permettant, dans le contexte des sciences sociales computationelles, de réaliser des analyses sur des masses de données. En effet, les méthodes statistiques traditionnelles ne peuvent s'appliquer facilement à des données non représentatives et ayant un bruit important. De plus, les données des réseaux sociaux forment des réseaux complexes [19] qui présentent des particularités [20, 9] comme des phénomènes qui suivent des lois de puissances, sans échelle, soumis à l'assortativité et à l'homophilie.

flexibilité grâce aux analyses exploratoires. Le *polystore* servant de système de stockage à l'architecture est le point de départ des analyses exploratoires. Comme ces dernières ne sont pas définies à l'avance et doivent s'adapter à chaque contexte de cas d'étude, la diversité des modèles de données du *polystore* réduit le nombre de manipulations nécessaires avant d'obtenir des données adaptées aux algorithmes.

5.1.2 La démarche

La démarche de création des modèles exécutables est une problématique nécessitant de traduire les questions de recherche sous une forme de *workflows*. L'exécution d'un *workflow* sur les données revient à appliquer un algorithme ou un enchaînement de différents algorithmes à partir de données mises en forme spécifiquement pour cet objectif. La figure 5.1, détaillée ci-après, donne un aperçu de la collaboration mise en place dans le projet Cocktail, et correspond à un contexte général d'analyses inter-disciplinaires.

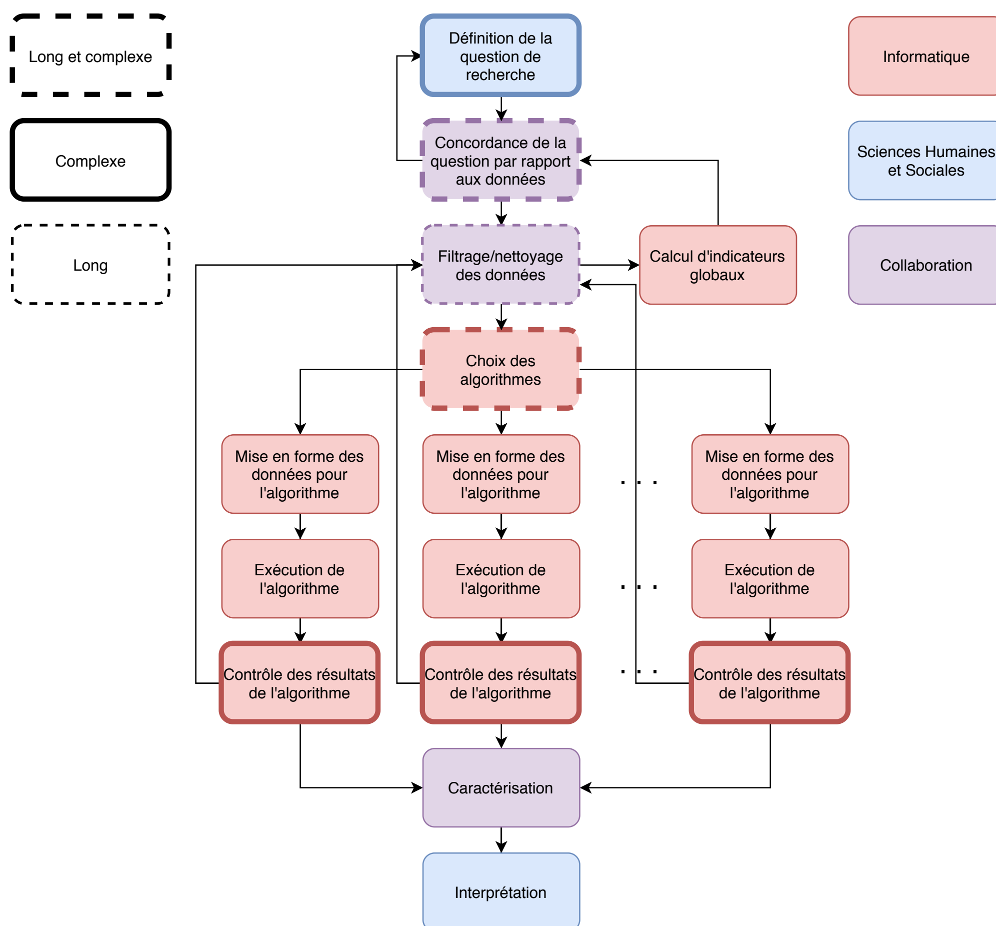


FIGURE 5.1 – *Workflow* général d'une analyse

Un *workflow* d'analyse débute par une question de recherche émise par les chercheurs en sciences humaines et sociales. La concordance de la question par rapport aux données disponibles doit ensuite être vérifiée, avant de pouvoir extraire et nettoyer un sous-ensemble de données qui formera alors le corpus

d'étude. La validité du corpus peut être vérifiée grâce aux indicateurs globaux, qui sont également utilisés pour raffiner la phase de nettoyage si le corpus d'étude n'est pas satisfaisant.

Les spécialistes en sciences des données ont pour rôle de sélectionner le ou les algorithmes permettant d'éclairer la question scientifique et d'en assurer l'enchaînement. Ils peuvent également développer un nouvel algorithme en s'appuyant sur des théories formelles existantes (théorie des graphes, algèbre linéaire et multi-linéraire, processus markoviens, etc.). Lors de la modification ou de la création d'algorithmes, des éléments d'informatique théorique sont mobilisés pour effectuer des preuves sur les algorithmes (terminaison, convergence) et étudier leur comportement (complexité en espace, en temps, passage à l'échelle).

Le *polystore* de l'architecture Hydre offre de multiples modèles de représentation des données (relationnel, graphe, clé-valeurs, séries temporelles, etc.) implantés dans différents systèmes de stockage. Optimiser les performances de l'extraction et des opérations effectuées par les algorithmes nécessite d'exploiter au mieux les modèles et les systèmes. De la même manière que pour les systèmes de stockage des données, les algorithmes peuvent être implantés sur différents environnements (CPU, GPU, *cluster*, etc.) et s'appuyer sur des paradigmes de programmation différents (impératif procédural, impératif objet, vectoriel, fonctionnel, etc.). On distingue donc trois types de paramètres que l'on doit étudier pour déterminer la meilleure implantation possible d'un algorithme d'analyse :

- le fonctionnement interne de l'algorithme : structure de données, fondation théorique ;
- le ou les systèmes de stockage pour alimenter l'algorithme : modèles, schémas, parallélisme des requêtes ;
- le paradigme de programmation de la plateforme d'exécution. L'architecture physique décrite au chapitre 4 nous permet d'avoir une grande latitude de choix pour la plateforme d'exécution.

Suite à l'application des algorithmes, les résultats obtenus sont ensuite transmis aux chercheurs en sciences humaines et sociales pour interprétation, validation, raffinement ou réfutation. Le processus peut être itératif dans sa globalité ou par étape en fonction des résultats intermédiaires.

5.1.3 Les cas d'étude

Au cours du projet Cocktail, plusieurs corpus d'étude ont été créés afin d'éclairer une question de recherche spécifique. Certains corpus déjà existants peuvent se rajouter aux cas d'étude, puisqu'ils ont servi à développer de nouvelles analyses. Ces cas d'étude peuvent être résumés de la manière suivante :

- Twitter aux Élections Européennes (TEE) : initialement collecté lors des Élections Européennes de 2014, ce corpus a été complété en 2019 lors de la campagne électorale. Cette double composition du corpus permet d'étudier les évolutions entre les deux périodes ;
- Twitter aux Élections Présidentielles (TEP) : semblable au corpus précédent, le corpus TEP cible les Élections Présidentielles françaises de 2017, et permet d'étudier la perception du public des différents partis mais aussi les méthodes de communication utilisées par ces derniers à propos de sujets de société ;
- Lubrizol : suite à l'incendie qui s'est déclaré à l'usine Lubrizol à Rouen le 26 septembre 2019, ce corpus a été créé afin d'étudier les retombées au niveau de la santé des habitants, et plus spécifiquement la communication institutionnelle sur la santé ;
- Vegan : le mouvement vegan ayant gagné en importance au sein des habitudes alimentaires, ce corpus vise à étudier le comportement et les interactions de ceux appartenant à ce mouvement ;
- COVID : évènement majeur des années 2020 et 2021, ce corpus cible plus particulièrement la vaccination contre le COVID, les sujets et thèmes émergeant à ce propos sur Twitter.

5.2 De la collecte à la constitution des corpus et leur mise à disposition

Afin de pouvoir réaliser des analyses et interpréter les résultats, il faut tout d'abord collecter des données pertinentes et les nettoyer pour construire des corpus permettant de répondre à des questions spécifiques. Les corpus doivent être accessibles aux *data analysts* pour qu'ils puissent mener à bien les analyses, et les résultats doivent être mis à disposition des experts métiers pour qu'ils les interprètent.

5.2.1 La collecte des données

À partir de critères de collecte, constitués de comptes, hashtags et mots-clés identifiés par les experts métiers, une collecte de données de Twitter est lancée afin de constituer un corpus global. Cette collecte est effectuée par l'architecture Hydre, présentée dans le chapitre 2. Actuellement, nous avons trois collectes différentes dans le projet Cocktail, représentant un volume total de données brutes supérieur à 13To :

- **alimentaire** : qui vise à étudier les habitudes de consommation, le rapport à l'agriculture, ainsi que les différents mouvements liés à l'alimentation tels que les vegans ;
- **santé** : qui est centrée principalement autour du papillomavirus et de sa perception sur le réseau social ;
- **vaccination anti-COVID** : qui vise à étudier la communication des pro- et des anti-vaccin, afin de voir les oppositions.

Les deux premières collectes sont actives depuis mi-2019 et plus de 3 000 critères de collecte ont été spécifiés.

Des indicateurs macroscopiques sont calculés en continu et en temps réel sur les collectes en cours à l'aide du composant *real-time insights* de Hydre[2]. Ils permettent :

- de détecter des anomalies dans le comportement habituel de la collecte ;
- d'ajouter et d'affiner les critères de collecte ;
- de définir des analyses sur un sujet plus précis, d'identifier des questions de recherche, etc.

Ce dernier point motive la construction des corpus d'étude. Un corpus d'étude représente un sous-ensemble de tweets d'un corpus global, dont le but est de répondre à une question de recherche spécifique.

5.2.2 La constitution des corpus d'étude

Un corpus d'étude se base sur le corpus global, mais a besoin d'être nettoyé pour conserver uniquement les données qui correspondent au phénomène qui doit être analysé ou à la question qui doit être abordée. De cette manière, les phases de filtrage et nettoyage permettent de diminuer le bruit qui empêche une analyse précise.

La figure 5.2 montre le processus de création d'un corpus d'étude. Durant l'étape itérative de nettoyage des données, différents filtres, comme des intervalles de dates, des comptes, des hashtags ou des mots-clés sont utilisés, et peuvent être combinés afin d'obtenir un corpus satisfaisant. Des indicateurs macroscopiques sont calculés sur les corpus d'étude afin de pouvoir donner une appréciation de la qualité ou de la pertinence des données. Ces indicateurs macroscopiques prennent par exemple la forme de top-k des hashtags ou des comptes (en fonction d'un type d'interaction donné) les plus populaires.

2. Par exemple le nombre de tweets collectés, le nombre d'occurrences d'un hashtag ou d'un compte, avec une granularité temporelle d'une heure.

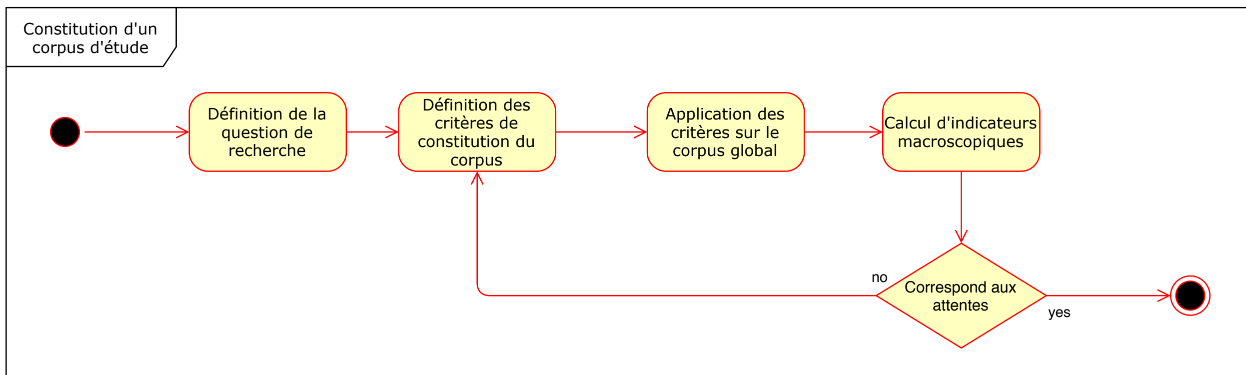


FIGURE 5.2 – Détail des étapes de la création d'un corpus d'étude

Comme détaillé dans l'introduction générale, cette étape est chronophage et demande une collaboration étroite entre le *data analyst*, qui a les compétences pour nettoyer les données, et l'expert métier, qui a les connaissances du domaine permettant d'appliquer le nettoyage au contexte de l'étude.

5.2.3 La mise à disposition des corpus

La mise à disposition des données contenues dans les corpus se fait dans deux types de cas :

— pour les analystes, afin qu'ils puissent avoir accès aux données, les explorer et exécuter divers algorithmes dans le but de répondre à une question de recherche ;

— pour les experts métiers, pour qu'ils puissent à la fois avoir un aperçu global du corpus (les hashtags qui le composent, les utilisateurs actifs, etc.) mais aussi avoir accès aux résultats obtenus par les analystes pour pouvoir les interpréter.

Afin d'obtenir une flexibilité importante dans le modèle de ces données, elles sont stockées dans le *polystore*, composé de trois SGBD : un SGBD relationnel (PostgreSQL), un SGBD graphe (ArangoDB) et un SGBD séries temporelles (TimescaleDB). Les schémas de données respectifs sont donnés dans les figures 5.3, 5.4 et 5.5.

Pour faciliter l'accès et la manipulation des données, des *notebooks* Jupyter sont mis en place. Ils permettent de garder une trace des expériences réalisées, tout en les rendant accessibles à tous, afin de renforcer le travail collaboratif. De plus, ils permettent d'incorporer des rendus visuels qui aident les experts métiers à comprendre et interpréter le résultat des analyses, et leur donnent l'occasion de faire des manipulations simples sur les données, leur laissant ainsi plus d'indépendance qu'en fournissant les résultats sous un format statique. Afin de faciliter l'utilisation du *polystore* et des algorithmes classiques d'analyse, j'ai développé une série de modèles de *notebooks* qui permettent de mettre rapidement en place l'étude d'un corpus. Les analyses qui peuvent être effectuées sont détaillées dans la section 5.3.

5.3 Techniques d'analyses

Les analyses réalisées grâce à l'observatoire peuvent être réparties en trois catégories :

— les analyses **macroscopiques**, qui donnent un aperçu global des corpus et permettent de cibler les questions qui doivent être investiguées ;

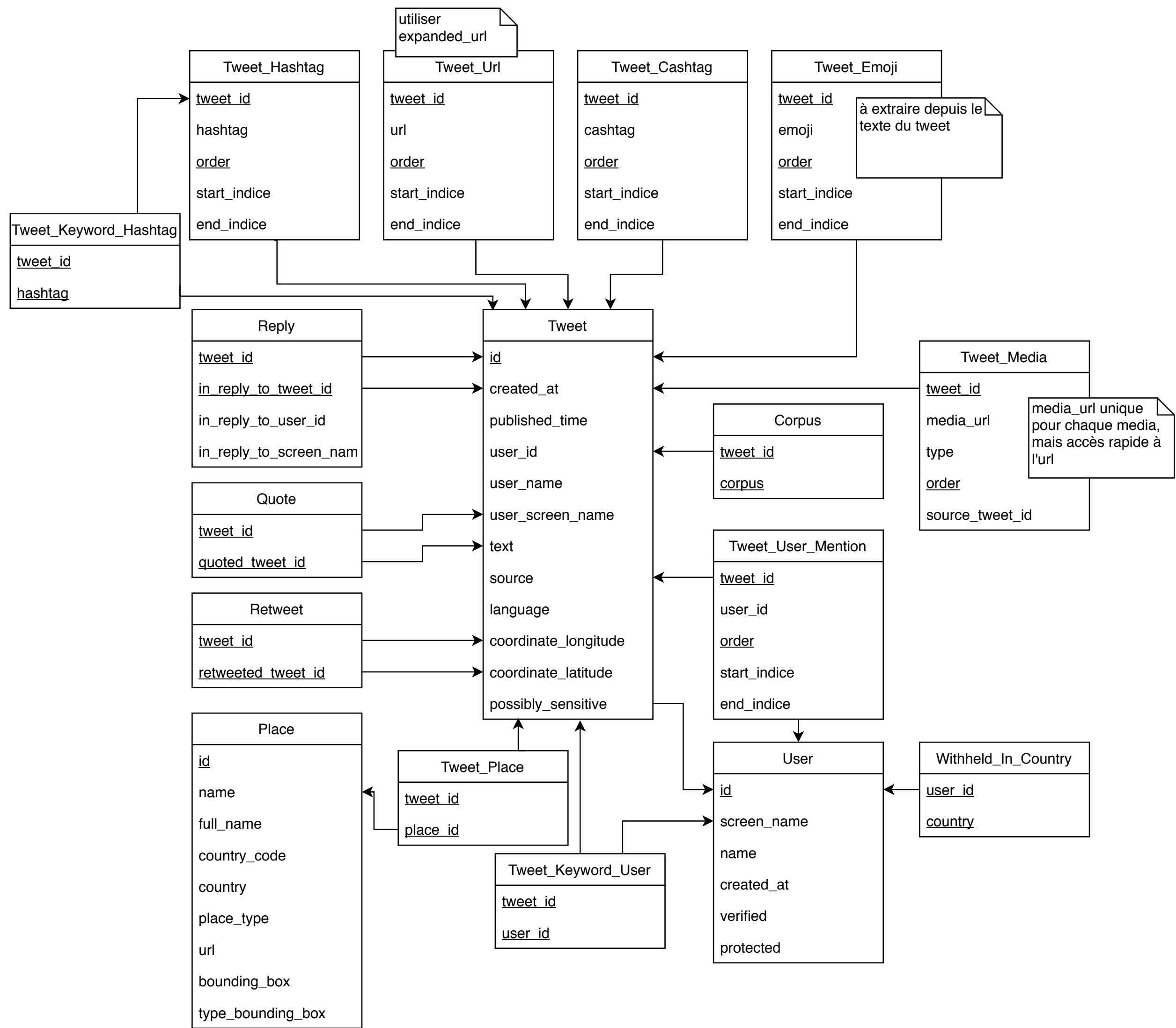


Figure 5.3 – Modèle relationnel du stockage des tweets

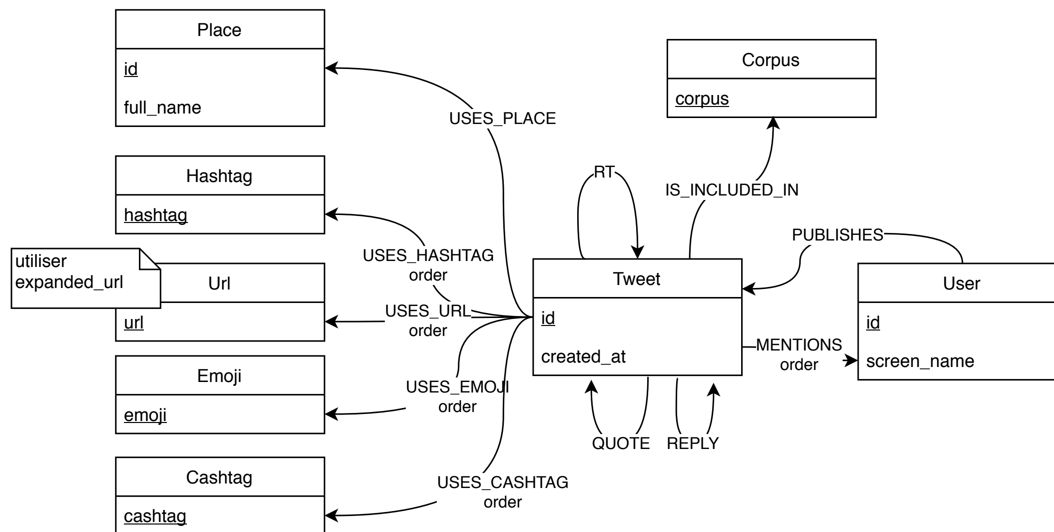


FIGURE 5.4 – Modèle graphe du stockage des tweets

— les analyses **mésoscopiques**, qui sont plus précises que les analyses macroscopiques et montrent les caractéristiques d'un corpus en termes de communautés et de centralités ;
— les analyses **microscopiques**, qui ciblent un point particulier dans le but de le contextualiser ou de mettre en évidence des éléments singuliers.

Afin de pouvoir comprendre les analyses, il faut savoir que les tweets peuvent prendre différentes formes, qui sont les retweets (qui consistent à rediffuser un tweet produit par un utilisateur sans modification), les quotes (qui sont des retweets contenant un commentaire ajouté), les réponses (qui sont rattachées à un autre tweet publié) ou les tweets originaux (qui ne correspondent à aucune des formes précédentes). Les tweets peuvent également avoir des opérateurs, qui peuvent être des mentions (une référence à un autre utilisateur) ou des hashtags (un mot précédé d'un dièse qui agit comme une catégorisation du tweet). Ces différents éléments de structure peuvent constituer des points de vue complémentaires lors des analyses. Par exemple il est possible de réaliser des analyses sur les hashtags les plus importants, la diffusion des informations avec les retweets, les personnes influentes, etc.

5.3.1 Les analyses macroscopiques

Les analyses macroscopiques fournissent des informations à gros grains sur une collecte générale ou sur un corpus d'étude, dans le but de distinguer les tendances mais aussi certaines anomalies qui peuvent nécessiter des investigations plus poussées, comme un hashtag se démarquant des autres. Le résultat de ce type d'analyse est mis à disposition des experts métiers afin qu'ils puissent l'explorer et soulever les éléments dignes d'intérêt.

Dans le projet Cocktail, les résultats de ce type d'analyse sont mis à disposition des experts métiers de deux manières différentes :

— avec un serveur d'applications, mis en place par un alternant de Master 2, qui héberge une application qui fournit les informations concernant les collectes globales en intégrant directement les nouvelles

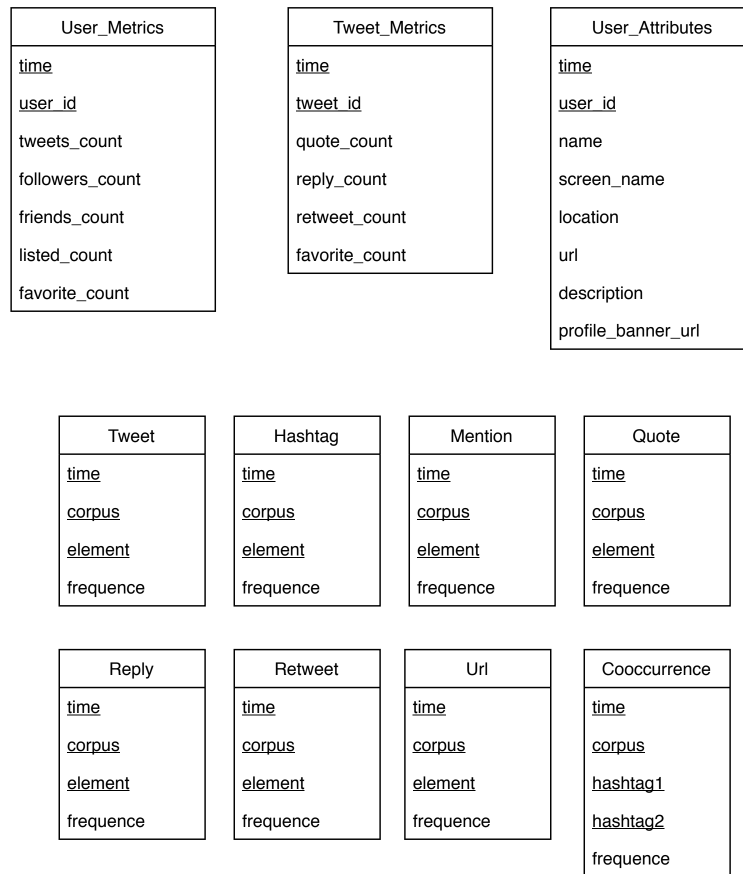
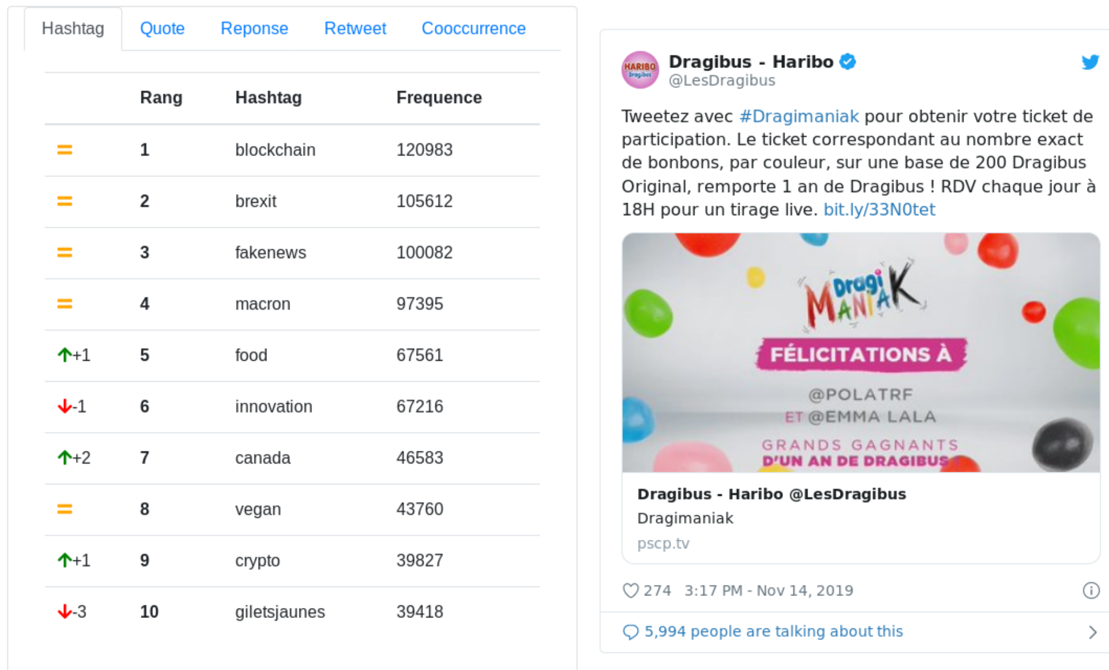


FIGURE 5.5 – Modèle séries temporelles du stockage des tweets

données ;
- avec des *notebooks* Jupyter, qui sont conçus pour laisser une liberté de manipulation et permettre aux experts métiers de réaliser des modifications légères de code pour fouiller davantage dans les données, comme par exemple la modification d'une variable représentant un hashtag pour afficher les informations correspondantes. Cette mise à disposition concerne uniquement les corpus d'étude.

Le serveur d'application se sert des données collectées et mises en forme par les composants *real-time insights* et *streaming ETL* de l'architecture Hydre, et les expose aux experts métiers. Leurs possibilités d'interaction avec ces données sont limitées, afin d'éviter une sollicitation trop importante des SGBD à la fois au niveau des insertions et de l'interrogation. Ils peuvent toutefois avoir un aperçu des tendances des collectes, en ayant accès à des informations telles que les tops-k des hashtags (figure 5.6), des mentions ou encore des comptes les plus retweetés ou quotés, les séries temporelles du nombre de tweets collectés ou d'éléments individuels comme les hashtags (figure 5.7) ou les mentions, etc.

Les analyses macroscopiques exposées à travers les *notebooks* Jupyter sont plus ciblées que celles du serveur d'applications. En effet, elles concernent un corpus d'étude en particulier. Comme pour le serveur



	Rang	Hashtag	Frequence
=	1	blockchain	120983
=	2	brexit	105612
=	3	fakenews	100082
=	4	macron	97395
↑+1	5	food	67561
↓-1	6	innovation	67216
↑+2	7	canada	46583
=	8	vegan	43760
↑+1	9	crypto	39827
↓-3	10	giletsjaunes	39418

FIGURE 5.6 – Top-k des hashtags, leur évolution et visualisation d'un tweet populaire

d'applications, on retrouve principalement des tops-k et des séries temporelles (figure 5.8), qui servent à la fois à vérifier que les données du corpus correspondent au sujet d'étude, c'est-à-dire que les critères utilisés pour filtrer le corpus ne sont pas hors-sujets ou porteurs de doubles sens, à donner les tendances du corpus concernant les hashtags les plus utilisés et les comptes les plus populaires, mais aussi à identifier des points d'intérêt qui peuvent être ensuite étudiés plus en détail. Ces informations macroscopiques sont ensuite complétées par des analyses mésoscopiques.

5.3.2 Les analyses mésoscopiques

Les analyses mésoscopiques font appel à divers algorithmes, principalement sur des graphes, afin d'extraire les structures communautaires et les éléments centraux des données. Afin de faciliter l'intégration de nouvelles analyses et de leurs résultats, un méta-schéma a été créé (figure 5.9). Il recense les graphes et leur filiation, ce qui permet de conserver un arbre de parenté lorsqu'une succession d'analyses ciblent une partie de plus en plus détaillée d'un graphe (par exemple la décomposition en communautés d'une communauté elle-même obtenue d'un graphe principal). Grâce à ce schéma, il est aussi possible de conserver les mesures attribuées aux nœuds, telles que le numéro de la communauté d'appartenance ou la valeur de la centralité. Ce schéma centralise les informations, évite les tables redondantes, et facilite la visualisation et la mise à disposition des résultats, puisqu'ils suivent tous la même logique et requièrent uniquement le paramétrage du graphe et du type de la mesure avec lesquels travailler.

À partir des données de Twitter, plusieurs graphes sont construits en début d'exploration du corpus, afin d'obtenir davantage de précisions sur sa structure et son contenu. Ces graphes sont :

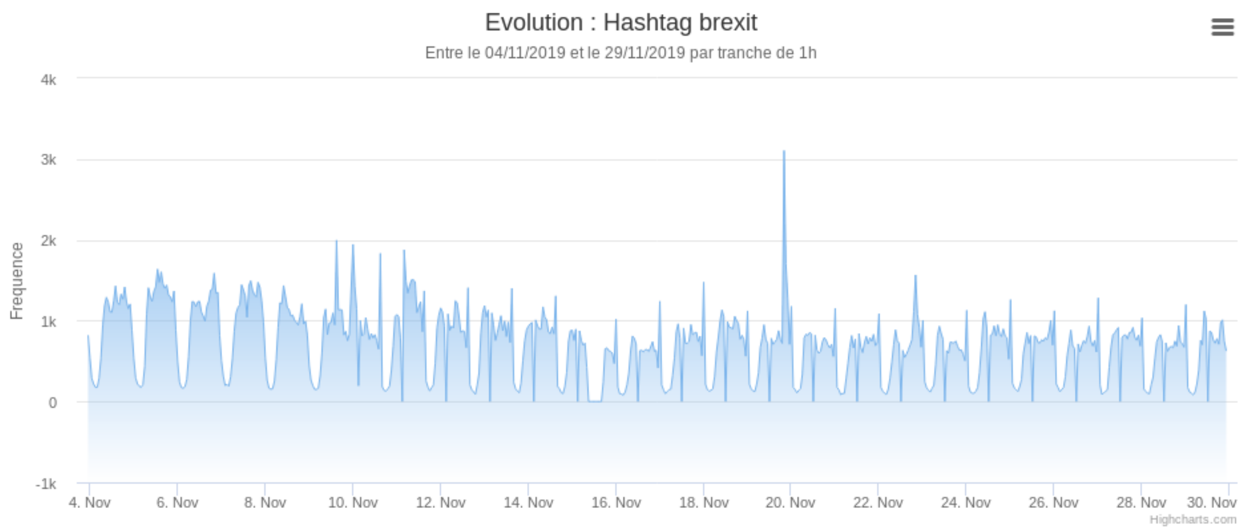


Figure 5.7 – Série temporelle d'un hashtag

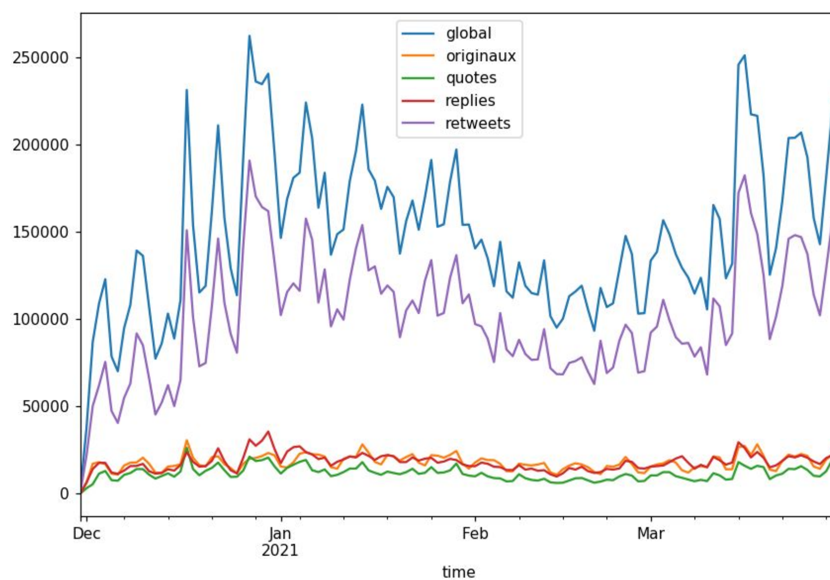


Figure 5.8 – Série temporelle de la publication des hashtags dans le corpus concernant la vaccination contre le COVID

— *user-user* : ce graphe lie les utilisateurs entre eux. Les liens peuvent être définis suivant plusieurs interactions, à savoir les mentions, les retweets, les quotes ou les réponses. Chacune de ces interactions peut amener des informations supplémentaires ;
— hashtag-hashtag : ce graphe représente les cooccurrences de hashtags, sans prendre en compte les émetteurs des tweets qui les contiennent ;

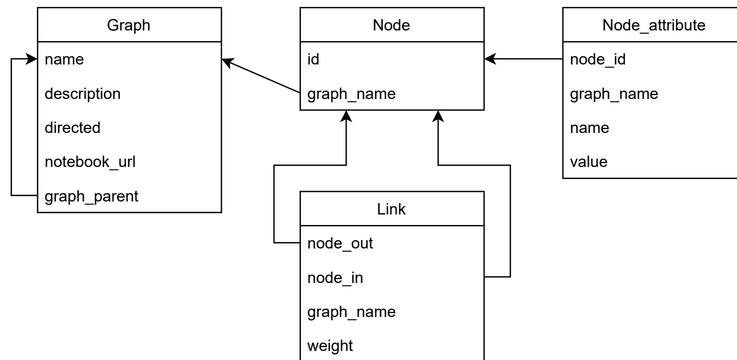


FIGURE 5.9 – Méta-schéma pour stocker les résultats d'analyse

— *user*-hashtag : ce graphe lie les utilisateurs aux hashtags qu'ils utilisent. Il permet d'avoir une vision thématique des utilisateurs.

Pour exploiter ces graphes, différents types d'algorithmes sont utilisés, dont le but est principalement de trouver des communautés ou d'attribuer des valeurs de centralité aux nœuds. Ces algorithmes peuvent être combinés ou utilisés incrémentalement (par exemple pour calculer les centralités à l'intérieur des communautés), et ce dans l'objectif de mettre en avant de nouvelles informations qui sont cachées lorsque les analyses sont réalisées individuellement.

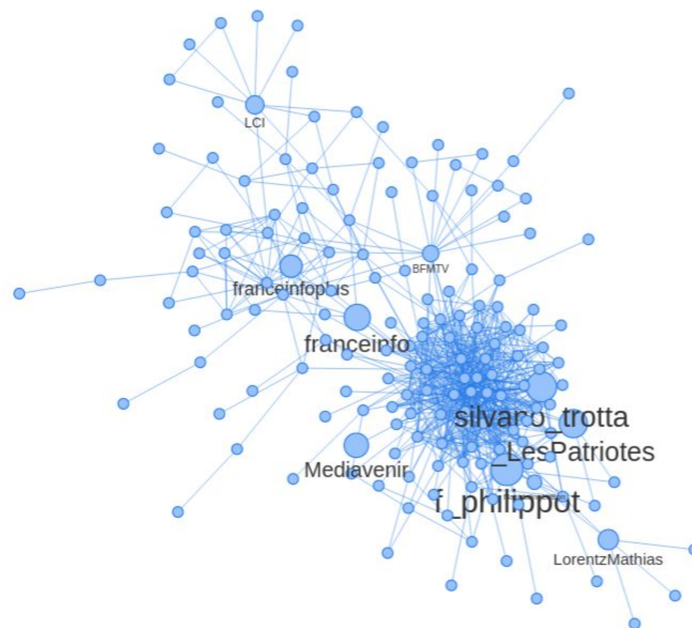


FIGURE 5.10 – Centralité des utilisateurs dans le corpus concernant la vaccination contre le COVID

Les différentes visualisations de graphes sont réalisées à l'aide de Pyvis [141], qui permet de personnaliser l'affichage mais également d'interagir avec le rendu de la visualisation. Dans le contexte de l'observatoire, cela

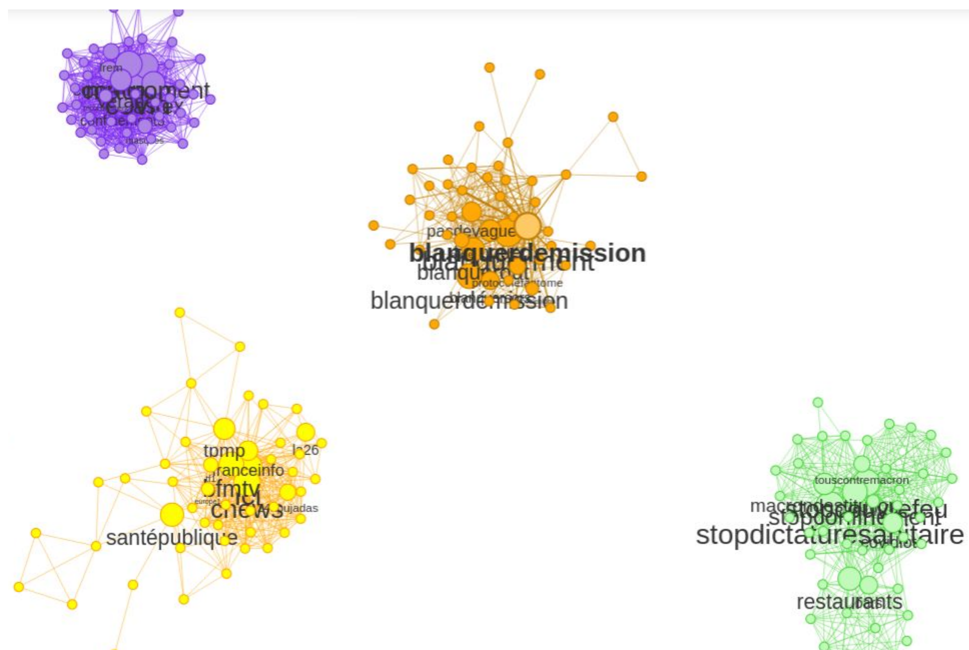


Figure 5.11 – Extrait des communautés de hashtags dans le corpus concernant la vaccination contre le COVID

est très utile, puisque par défaut il est possible de mettre en avant les informations les plus importantes, tout en permettant aux experts métiers de manipuler eux-même le graphe pour explorer les informations complémentaires sans surcharger la visualisation (par exemple, les nœuds les plus centraux sont mis en avant, mais en survolant les autres nœuds les experts métiers peuvent voir les entités auxquelles ils correspondent ainsi que le rang qui leur est attribué grâce à l'algorithme de centralité).

Les algorithmes classiques de centralité, tels que HITS [91] ou PageRank [135], sont d'abord utilisés sur le graphe global pour mettre en avant les utilisateurs (figure 5.10) et hashtags populaires. D'autres algorithmes, cette fois visant à découvrir les communautés contenues dans les graphes, sont ensuite appliqués, et permettent de structurer les graphes en fonction des thématiques ou de groupes d'utilisateurs proches. Parmi ces algorithmes de détection de communautés, on retrouve Louvain [24], Walktrap [145] ou encore Infomap [152]. Dans chaque communauté, les algorithmes de centralité sont également appliqués pour avoir une idée de son organisation interne (figure 5.11).

5.3.3 Les analyses microscopiques

Les analyses microscopiques sont des analyses plus fines et ciblées que les précédentes. Elles ne sont généralement pas connues en avance, et nécessitent une phase de sélection de la technique et de l'algorithme à utiliser en fonction de la question de recherche à laquelle on souhaite répondre. Cette section décrit plusieurs analyses qui ont pu être réalisées dans le contexte de l'observatoire. Elles s'intéressent à des éléments particuliers, comme par exemple les hashtags, les comptes utilisateurs, des profils de comptes, des patterns de communication ou encore des éléments à l'intérieur des communautés, afin de détecter des singularités, des anomalies, etc.

5.3.3.1 Étude des discours de haine

En partant du corpus TEP2017, les chercheurs en sciences sociales ont voulu étudier les discours de haine qui se sont propagés sur Twitter pendant la période des élections du 23 avril 2017 au 07 mai 2017 inclus. Pour cela, ils nous ont fourni une liste d'une centaine de mots caractérisant des insultes, que nous avons utilisée pour construire le corpus d'étude.

Pour l'élaboration du corpus, la phase de filtrage et nettoyage s'est effectuée en deux étapes :

1. utilisation des mots des tweets pour effectuer le filtrage et délimiter le corpus :
   — réduction des mots à leur racine en utilisant l'algorithme dit *Porter Snowball stemming* implanté par la méthode `to_tsvector` du SGBD PostgreSQL ;
   — **filtrage** au moyen d'une liste de mots révélateurs de discours de haine (thématique, qualificatif ou amplificateur) appliquée sur les mots racines des tweets pour trouver les tweets qui contiennent ces mots ;
2. **réduction du corpus de tweets** en conservant les tweets qui contiennent au moins 2 des mots de la liste (la comparaison est faite sur la racine des mots, à la fois dans les tweets et pour les mots de la liste).

Un corpus constitué de 49 929 tweets a ainsi été obtenu, dont 14 513 originaux (c'est-à-dire les tweets qui ne sont pas un retweet, une quote ou une réponse).

Nous avons réalisé une analyse sémantique latente (LSA) sur ce corpus, avec l'utilisation de la décomposition en valeurs singulières (SVD) pour réduire les dimensions des espaces étudiés les uns par rapport aux autres. La réduction de dimensions permet ensuite d'appliquer des algorithmes de *clustering* classiques tels que *k-means*. La LSA est utilisée pour regrouper des documents et extraire des thématiques. Elle utilise des mots réduits à un radical et modélise chaque document avec un vecteur qui compte le nombre d'apparitions d'un mot dans le document, et ceci pour tous les mots. Ce nombre est souvent normalisé en utilisant la pondération tf-idf (*term frequency, inverse document frequency*). Dans notre cas, les tweets sont trop petits pour être considérés comme des documents et pouvoir faire l'objet d'une normalisation de type *tf-idf*.

Le but de l'application de la LSA dans le contexte de notre analyse était de regrouper les tweets en fonction des mots qui les composent, et les mots en fonction de leur utilisation dans les tweets, puis faire de même pour les utilisateurs et les mots. Pour cela, deux matrices ont été formées, en prenant en compte uniquement les tweets originaux :

— **matrice tweets-mots** : qui indique le nombre d'utilisations des mots dans chaque tweet ;
— **matrice *users*-mots** : qui indique le nombre d'utilisations des mots dans tous les tweets d'un utilisateur.

Une fois ces matrices construites, nous avons pu appliquer la SVD. La SVD opère sur une matrice quelconque M et la décompose en un produit de 3 matrices :

$$M = U\Sigma V^t$$

En résultat, si la matrice M a n lignes (n tweets) et m colonnes (m mots), la matrice U est une matrice de n lignes et r colonnes, Σ est une matrice diagonale contenant les valeurs singulières, qui permettent de connaître quelles sont les dimensions les plus importantes, V^t est une matrice de r lignes et m colonnes. Ainsi, U et V ré-expriment les tweets et les mots dans un espace intermédiaire constitué de r dimensions. Pour effectuer une réduction de dimensions et conserver l'information principale, une règle empirique est de garder les r' valeurs singulières qui contribuent à 80-90% de la somme totale de ces valeurs singulières.

Une première tentative d'application de la SVD sur les données brutes a fourni un résultat non-pertinent : il y avait trop de valeurs singulières significatives pour opérer une réduction de la dimension des espaces, que ce soit pour la matrice tweets-mots ou pour la matrice *users*-mots. Une deuxième tentative a consisté à rajouter le poids des retweets dans les matrices, ce qui a rendu le résultat plus pertinent. En conservant les valeurs singulières qui contribuent à 80% de la somme totale des valeurs singulières, on obtient 59 valeurs singulières pour la première matrice et 56 pour la seconde.

Nous avons ensuite utilisé l'algorithme *k-means* pour vérifier la pertinence des regroupements et donc la cohérence des résultats. Cet algorithme permet de regrouper les éléments exprimés dans un espace à r' dimensions en *clusters* suivant leur proximité. Pour appliquer l'algorithme, il est nécessaire de fournir le paramètre k, c'est-à-dire le nombre de *clusters* à former. Différentes méthodes peuvent être utilisées pour aider à déterminer ce paramètre :

— **la méthode *elbow*** : qui, dans chaque *cluster*, mesure la distance entre chaque élément et le centre du *cluster*. Ces distances sont additionnées, et une valeur basse indique donc un nombre idéal de *clusters* ;

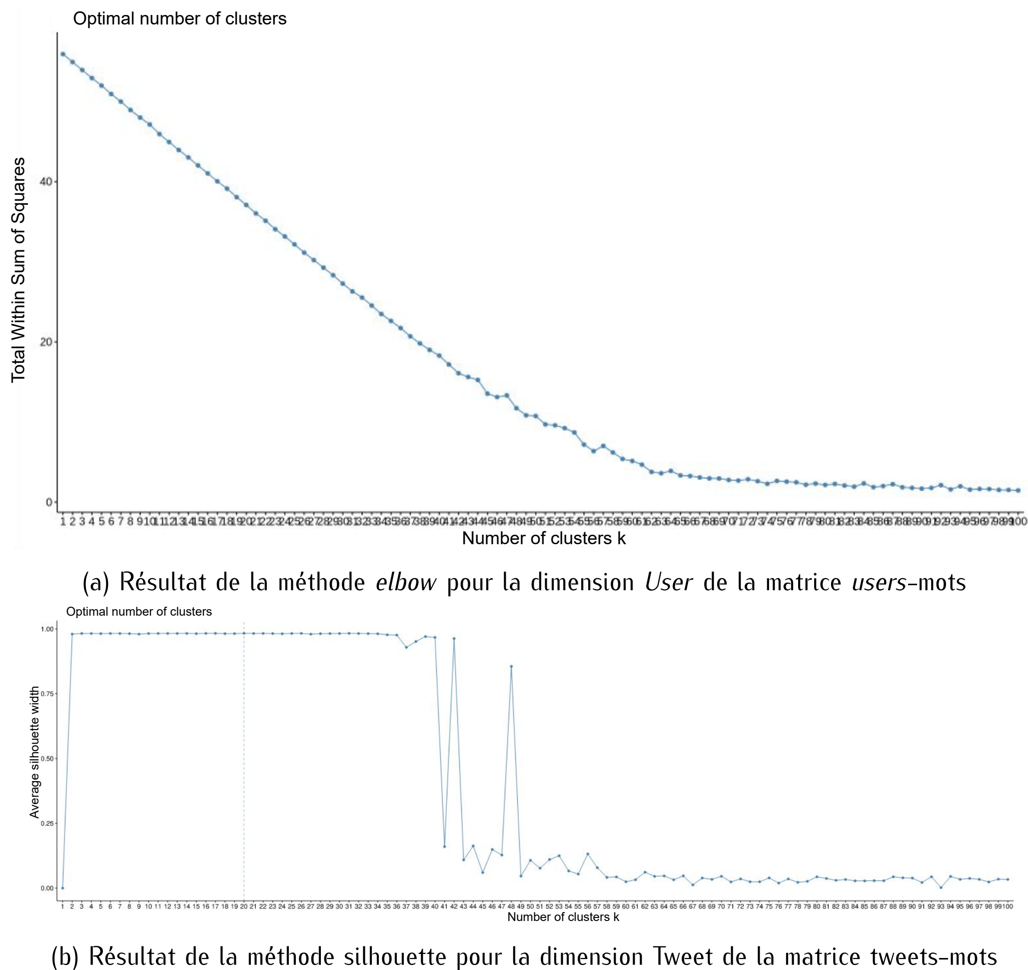
— **la méthode silhouette** : compare la distance de chaque élément aux éléments de son propre *cluster*, par rapport à la distance entre cet élément et les éléments des autres *clusters*. Plus le score est élevé, plus les *clusters* sont compacts et éloignés.

Appliquées sur le corpus, ces deux méthodes ont donné des résultats différents. La méthode *elbow* donne approximativement $k = 70$ pour les tweets et $k = 90$ pour les mots de la matrice tweets-mots, et $k = 60$ pour les *users* et $k = 90$ pour les mots de la matrice *users*-mots. La méthode silhouette en donne beaucoup moins : $k = 20$ pour les tweets et $k = 52$ pour les mots avec la SVD appliquée sur la matrice tweets-mots, et $k = 15$ pour les utilisateurs et $k = 72$ pour les mots avec la SVD appliquée sur la matrice *users*-mots.

Ces résultats peuvent s'expliquer par le fait que les *clusters* ne sont pas bien dissociés. Ceci est confirmé par l'étude des *clusters* calculés avec les valeurs de k fournies par la méthode silhouette, qui font apparaître beaucoup de *clusters* réduits à 1 élément, et c'est également constaté dans une plus faible mesure avec la méthode *elbow*, qui elle arrive à découper les gros *clusters* au prix d'une séparation moins tranchée. En effet, les expériences ne confirment pas la formation de *clusters* bien séparés. Selon les résultats des méthodes silhouette et *elbow*, k est plus difficile à déterminer pour la dimension de mots. Cela est probablement dû à un bruit trop important parmi les mots conservés pour construire les matrices ou à une homogénéité trop forte des données.

À la suite de cette étude, nous avons cherché à identifier les limites de cette méthode pour ce type d'analyse. Une première limite est que les données en entrée ne semblent pas suffisamment nettoyées : la méthode utilisée pour obtenir la racine des mots ne permet pas d'éliminer les mots non utiles pour l'analyse, ce qui induit beaucoup de bruit dans le corpus. Dans ce contexte, une intervention d'experts du domaine semble nécessaire pour nettoyer les mots utilisés lors de la construction des matrices servant pour l'exécution de la SVD. Ce nettoyage permettrait de constituer une liste étendue à partir de celle fournie au début de l'analyse, qui tienne compte de la variété des mots que l'on peut trouver dans le corpus, ceci afin d'aider les algorithmes à travailler sur des données plus ciblées.

Une seconde limite pour expliquer les résultats est que les données sont trop homogènes. On trouve finalement un peu de discours de haine partout, indépendamment des thématiques des autres discours. Le discours de haine ne serait donc pas ciblé : les mêmes mots sont utilisés pour exprimer une haine sur beaucoup de sujets très différents, ce qui ne peut pas se détecter facilement car les algorithmes recherchent des régularités (proximité, densité, motifs, etc.).



(a) Résultat de la méthode *elbow* pour la dimension *User* de la matrice *users*-mots

(b) Résultat de la méthode silhouette pour la dimension Tweet de la matrice tweets-mots

FIGURE 5.12 – Extrait des méthodes *elbow* et silhouette pour déterminer le nombre de *clusters* pour l'algorithme *k-means*

5.3.3.2 Polarisation et frontières des communautés

Ces analyses ont pour but d'étudier la polarisation des communautés, leurs frontières, et plus particulièrement de caractériser les modes de communication entre les communautés afin de mieux comprendre les oppositions. La méthode proposée, basée sur les graphes, est uniquement structurelle, et ne se sert pas d'éléments d'analyse du langage, qui sont difficilement applicables dans le cas de tweets du fait du contenu restreint de ces derniers. Ce travail a été réalisé par Alexis Guyot lors de son année de Master 2 Bases de Données et Intelligence Artificielle parcours recherche et a donné lieu à une publication commune [66].

La première étape de la méthode consiste à identifier la zone interne et la zone frontière de chaque communauté. Pour cela, les communautés sont étudiées par paires, et on considère qu'un nœud fait partie de la zone interne lorsqu'il n'a aucun voisin qui soit un nœud de l'autre communauté, ou qu'il fait partie de la zone frontière s'il a au moins un voisin qui fasse partie de sa propre zone interne et un autre voisin qui soit membre de l'autre communauté [62].

ID	Taille	Catégorie	Top-hashtags
20792	10 604	Pro-vaccins	AFP, Mutation, Confinement3, CouvreFeu, Ecoles, EHPAD, PasseportVert, DictatureSanitaire, Israël, JeMeFaisVacciner, Pasteur
12884	4 722	Anti-vaccins	Ivermectine, DictatureSanitaire, JeNeMeConfineraiPas, Raoult, Hydroxychloroquine, EtLesSoins, Plandémie, VéranDémission, LesPierresCrieront, GreatReset, Ethique, BeBraveWHO, JeNeMeVaccineraiPas
17387	2 736	Réactions médias	AFP, Confinement3, DictatureSanitaire, CouvreFeu, SudRadio, Le79Inter, Ivermectine, Santé, EDPHAD, LCI, Cnews, ZeroCovid, La26, PasseportVaccinal, Variant
11672	2 638	Anti-Blanquer	BlanquerMent, BlanquerDemission, Blanquer, PasDeVague, CovidLong, Ecoles, GarderieNationale, ProtocoleFantome, ProtocoleBidon, ParentsEnColere
16302	1 649	Politique	LR, LCI, UE, RN, DictatureSanitaire, OlivierVeran, LFI, PS, EmmanuelMacron, Medias, LREM, Melenchon
16079	976	Québecois	PolQC, Covid19qc, PolCan, CAQ, Québec, PLQ, QCPolic, EduQC, Montreal
15179	882	Anti-gouvernement	LREM, StopDictatureSanitaire, Macronie, JeSuisLibre, MacronDestitution, RéveillezVous, TousContreMacron, BlanquerDemission, Gouvernement
14931	384	Sans étiquette	Raoult, LREM, Ivermectine, DictatureSanitaire, PasseportVaccinal, Macronie, JeNeMeConfineraiPas, Trump, Cnews, VeranDemission

Table 5.1 – Communautés significatives découvertes par l'algorithme de Louvain et leur caractérisation par les hashtags

L'antagonisme d'une communauté envers une autre est alors calculé grâce à la formule suivante :

$$S = \frac{1}{\left|B_{i,j}\right|} \sum_{v \in B_{i,j}} \left[\frac{\sum_{e \in E_{iv}} w(e)}{\sum_{e \in E_{iv}} w(e) + \sum_{e \in E_{bv}} w(e)} - 0.5 \right]$$

avec $\left|B_{i,j}\right|$ le nombre de nœuds dans la zone frontière, E_{iv} l'ensemble des liens entre le nœud v et un nœud de la zone interne de sa communauté, E_{bv} l'ensemble des liens entre le nœud v et un nœud de l'autre communauté, et $w(e)$ le poids du lien e.

Le score d'antagonisme S est compris dans l'intervalle [-0.5, 0.5]. Une valeur positive signifie que la frontière interagit plus avec l'intérieur de sa communauté qu'avec l'extérieur, et inversement. Ce score peut s'interpréter de la manière suivante : plus un utilisateur va défendre l'opinion de sa communauté auprès des autres communautés, plus il est susceptible d'être porteur d'antagonisme. Les scores d'antagonisme entre chaque paire de communautés peuvent être représentés dans une matrice afin d'en faciliter l'interprétation.

Cette méthode a été appliquée sur le corpus relatif à la vaccination contre le COVID, pour une période allant du 1er décembre 2020 au 31 mars 2021. Les liens du graphe créés à partir de ce corpus représentent

les quotes entre les utilisateurs. Le graphe nettoyé (les liens avec un poids de 1 ont été supprimés) contient 24 591 utilisateurs et 55 703 interactions.

L'algorithme Louvain a été exécuté pour trouver les communautés significatives, puis les hashtags les plus utilisés par la communauté ont été répertoriés afin de pouvoir catégoriser chaque communauté, en supprimant les hashtags neutres se retrouvant dans la plupart des communautés, tels que #COVID (table 5.1).

La matrice représentant les scores d'antagonisme de ces communautés (figure 5.13) peut être lue de la manière suivante : les valeurs de la matrice sont les scores d'antagonisme en provenance de la communauté identifiée par la colonne vers la communauté identifiée par la ligne.

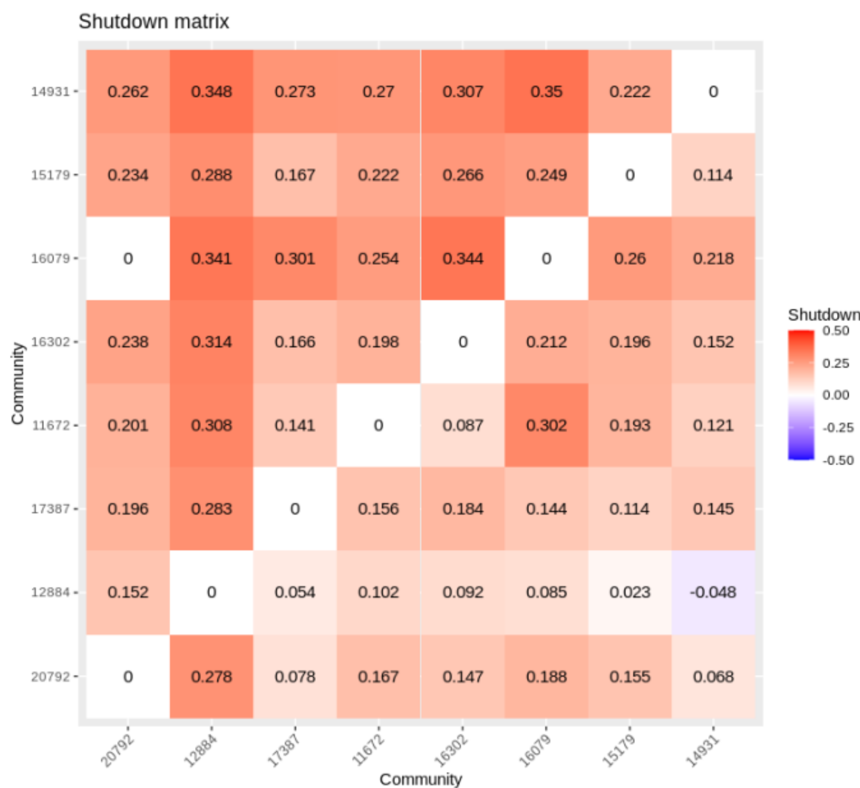


FIGURE 5.13 – Matrice d'antagonisme

Grâce à l'application de cette méthode, nous avons constaté, entre autres, que les utilisateurs faisant partie de la communauté anti-vaccin (numéro 12 884) interagissent beaucoup entre eux et peu avec l'extérieur, ce qui indique que cette communauté est probablement renfermée sur elle-même (et est donc susceptible d'être antagoniste), mais que pourtant les autre communautés ont une valeur d'antagonisme proche de 0 par rapport à la communauté anti-vaccin, ce qui indique qu'ils communiquent presque autant avec les membres de leur propre communauté qu'avec les membres de la communauté anti-vaccin.

La détection d'antagonisme dans un graphe issu de données d'un réseau social est une approche intéressante pour étudier les interactions entre les communautés plutôt qu'uniquement à l'intérieur d'une

communauté. D'autres mesures sont issues de ce travail, comme le score de porosité par exemple, qui indique à quel point une communauté est renfermée sur elle-même par rapport à l'ensemble des autres communautés, et qui peut donc être un indicateur de la présence d'une chambre d'écho dans une communauté.

5.3.3.3 Utilisation de la décomposition CANDECOMP/PARAFAC

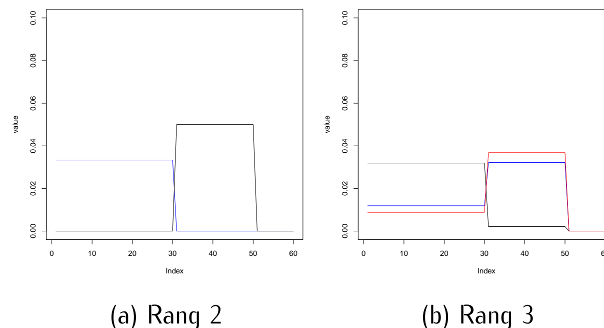
La décomposition CANDECOMP/PARAFAC peut fournir des résultats intéressants à la fois pour identifier des communautés, mais également pour détecter des anomalies. En effet, la décomposition produit un vecteur par dimension et par rang, dans lequel les plus fortes valeurs correspondent aux éléments d'intérêt dans ce rang. Avant de voir plus en détail comment un même opérateur peut servir pour ces deux objectifs, il faut tout d'abord comprendre l'impact que peut avoir le choix du rang sur la décomposition.

Trouver le meilleur rang pour la décomposition CANDECOMP/PARAFAC est un problème complexe [72]. Certains tentent de le déduire, comme c'est le cas avec le *core consistency diagnostic* (CORCONDIA) [28], et d'autres tentent de le contourner grâce à un processus répété constitué de deux étapes [15] :

1. calculer la décomposition de rang 1 ;
2. déflater le tenseur (*tensor deflation*) en supprimant le *cluster* qui vient d'être obtenu grâce à la décomposition.

Bien qu'une décomposition de rang 1 produise toujours un résultat satisfaisant, la déflation peut altérer le résultat de la décomposition. Cela peut être démontré avec un exemple empirique.

Nous construisons un tenseur d'ordre 3 et de taille $60 \times 60 \times 60$. Ce tenseur contient trois *clusters* qui ne se chevauchent pas : un de taille $30 \times 30 \times 30$, le deuxième de taille $20 \times 20 \times 20$ et le dernier de taille $10 \times 10 \times 10$. La décomposition CANDECOMP/PARAFAC est appliquée sur ce tenseur, d'abord avec le rang fixé à 2 puis à 3. Les résultats de l'application de la décomposition sont montrés dans la figure 5.14, et nous permettent de déduire que le meilleur rang est 2.



(a) Rang 2 (b) Rang 3

FIGURE 5.14 – Décomposition d'un tenseur de taille $60 \times 60 \times 60$ avec trois *clusters*

Le *cluster* de taille $30 \times 30 \times 30$ est ensuite supprimé, afin de simuler la déflation du tenseur à la suite d'une décomposition de rang 1, puis une décomposition de rang 2 est appliquée sur le tenseur déflaté (figure 5.15). Le troisième *cluster* est maintenant visible dans le résultat de la décomposition.

Cette expérience montre que la déflation d'un tenseur impacte les décompositions réalisées après la déflation, et n'est de ce fait pas une solution idéale pour trouver le rang parfait pour la décomposition.

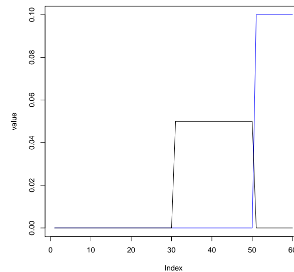


FIGURE 5.15 – Décomposition de rang 2 sur un tenseur déflaté de taille $60 \times 60 \times 60$ avec deux *clusters*

Toutefois, ce mécanisme peut être utilisé pour découvrir des *clusters* moins importants, ce qui peut être intéressant pour les analyses, particulièrement quand elles concernent des données qui suivent une loi de puissance.

Un autre mécanisme intéressant de la décomposition CANDECOMP/PARAFAC est l'ordre dans lequel les itérations du rang 1 de la décomposition trouvent les différents *clusters* du tenseur. Pour mieux comprendre ce mécanisme, nous construisons un tenseur d'ordre 3 de taille $60 \times 60 \times 60$, avec trois *clusters* de taille $20 \times 20 \times 20$ qui ne se chevauchent pas. Chaque *cluster* se différencie des autres grâce à la valeur que prennent ses éléments : le premier est peuplé de valeurs 5, le deuxième de valeurs 10, et le troisième de valeurs 15. Ces *clusters* sont tous détectés par une décomposition de rang 3. La décomposition CANDECOMP/PARAFAC de rang 1 est exécutée 1 000 fois sur ce tenseur, et nous notons quel *cluster* a été trouvé à chaque itération (table 5.2). Le poids du *cluster* dans le tenseur semble donc jouer un rôle dans la probabilité de le trouver lorsqu'une décomposition de rang 1 est appliquée.

Valeur des éléments du *cluster*	Fréquence d'apparition
15	62%
10	32%
5	6%

TABLE 5.2 – Probabilité d'apparition de chaque *cluster* avec une décomposition de rang 1

En se basant sur ces observations, nous pouvons déduire qu'il existe une probabilité plus élevée d'obtenir uniquement des *clusters* importants si le rang de la décomposition n'est pas assez élevé, et qu'une déflation peut être nécessaire pour trouver des *clusters* d'intérêt mais cachés par des signaux forts.

La décomposition CANDECOMP/PARAFAC développée pour TDM a été mise à l'épreuve pour des analyses concernant la détection de communautés et d'anomalies sur deux jeux de données réels [3] : un concernant les interactions entre des enfants dans une école primaire, et l'autre concernant l'activité d'un réseau informatique contenant des attaques (DARPA 1998).

Le jeu de données des interactions entre les enfants dans une école primaire [56], disponible en ligne [4], a déjà été analysé à l'aide de la décomposition CANDECOMP/PARAFAC par Gauvin et al. [55]. Ce jeu de

3. Les expériences sont disponibles en ligne `https://github.com/AnnabelleGillet/TDM-experiments`
4. `http://www.sociopatterns.org/datasets/primary-school-temporal-network-data/`

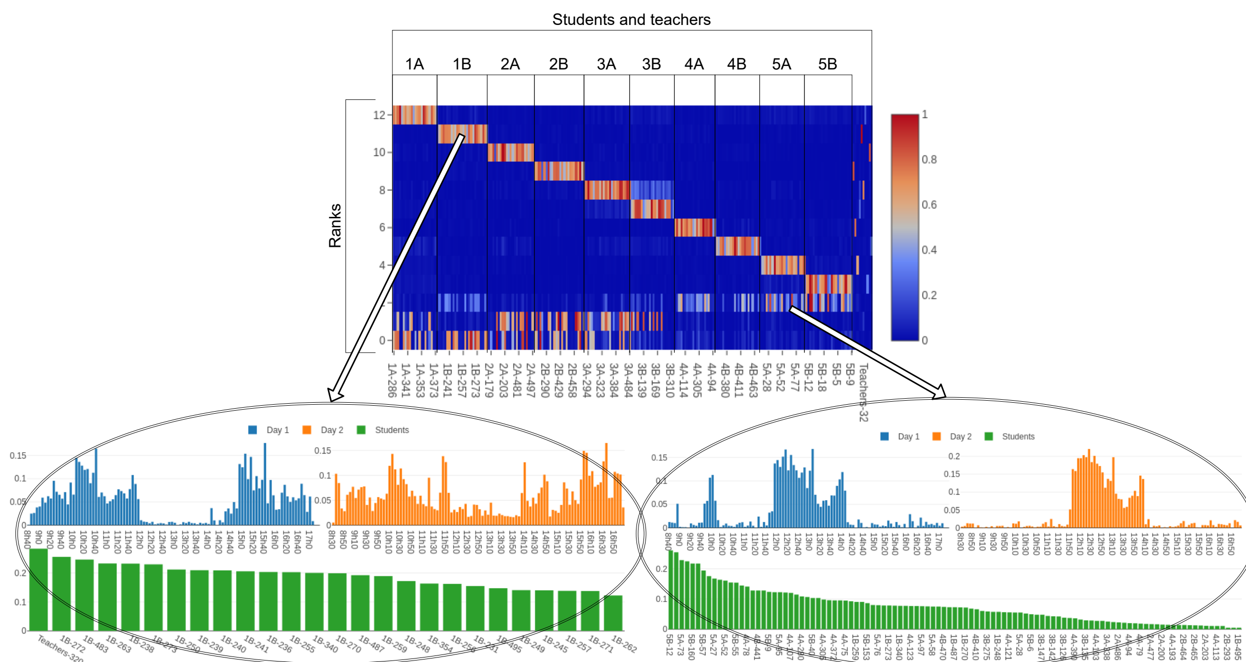


FIGURE 5.16 – Communautés détectées dans le jeu de données de l'école primaire

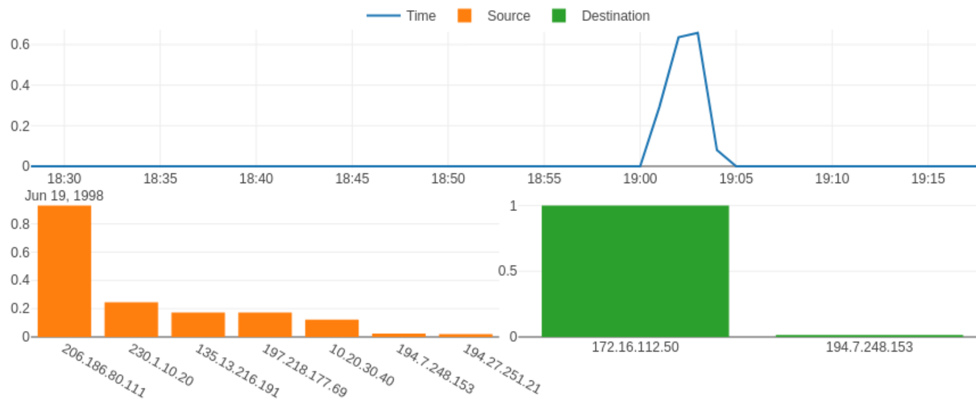
données a été enregistré grâce à des appareils RFID portés par 242 enfants répartis en 10 classes, et 10 enseignants sur une période de deux jours. Une interaction est enregistrée lorsqu'elle a une durée de plus de 20 secondes.

Un tenseur d'ordre 3 est construit à partir de ce jeu de données, avec deux des dimensions représentant les personnes et la troisième le temps avec une granularité temporelle de 5 minutes. Le rang de la décomposition est choisi à l'aide de CORCONDIA, et est fixé à 13. Le résultat de la décomposition est montré dans la figure 5.16, dans laquelle chaque ligne est un rang et chaque colonne une personne. Nous pouvons voir dans cette figure que 10 des 13 rangs représentent les classes et contiennent chacun l'enseignant affecté à cette classe, et si nous regardons le détail temporel de ce genre de *cluster* (bas de la figure), on remarque que la temporalité correspondante concerne les heures de classe. Trois *clusters* se démarquent des autres en regroupant des enfants de plusieurs classes, mais en regardant le détail temporel on constate que ces *clusters* apparaissent principalement pendant la pause de midi et aux intercours, ce qui explique le mélange des classes.

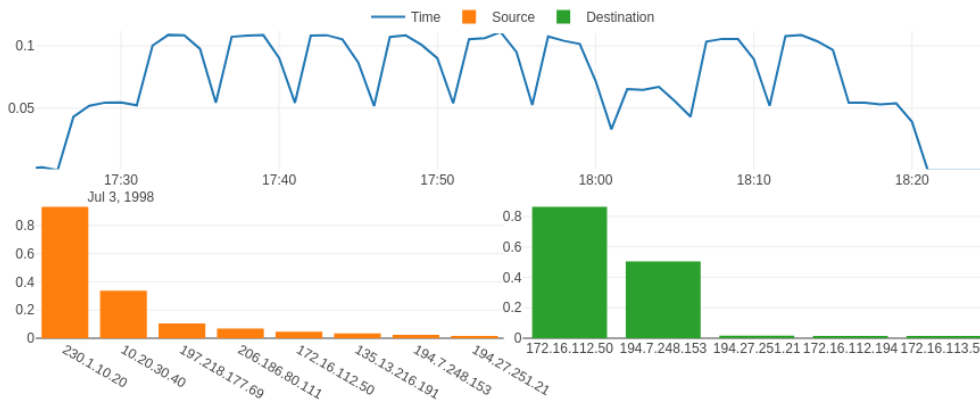
Le jeu de données DARPA 1998 [111], également disponible en ligne [5], recense le trafic réseau sur 7 semaines. Il contient 17 898 adresses IP source et 13 569 adresses IP destination. Les interactions sont accompagnées du moment auquel elles ont lieu, à la milliseconde près. Les attaques qui ont été subies durant cette période sont connues, et constituent la vérité de terrain.

Pour analyser ce jeu de données, un tenseur d'ordre 3 est utilisé, avec une dimension qui représente les adresses IP source, une autre les adresses IP destination, et la dernière le temps avec une granularité

5. `https://www.ll.mit.edu/r-d/datasets/1998-darpa-intrusion-detection-evaluation-dataset`



(a) Attaque warezmaster



(b) Attaque neptune

FIGURE 5.17 – Extrait des attaques détectées dans le jeu de données DARPA1998

temporelle d'une minute. Un extrait des rangs obtenus est donné dans la figure 5.17. Les attaques représentées dans ces rangs ont été validées avec la liste officielle des attaques. Il est intéressant de constater que la décomposition CANDECOMP/PARAFAC, en plus d'identifier correctement les adresses IP source et destination, permettent d'obtenir un instant exact et une signature temporelle de ces attaques. En effet, l'attaque warezmaster (figure 5.17a) s'est produite sur un temps court pour réaliser un *upload* de warez, tandis que l'attaque neptune (figure 5.17b) était de type déni de service sur plusieurs ports, composée de phases de 5 minutes répétées pendant une heure.

En conclusion, la décomposition CANDECOMP/PARAFAC est efficace pour détecter les communautés et les anomalies dans un jeu de données. La différence pour passer de l'un à l'autre réside dans la modélisation du tenseur. Dans le jeu de données de l'école primaire, toutes les valeurs étaient à 1 (pour modéliser l'existence de l'interaction), ce qui fait qu'un *cluster* peut gagner de l'importance en ayant beaucoup d'éléments qui le composent. Avec le jeu de données DARPA 1998, toutes les connexions ont été comptées pour chaque minute

représentée dans le tenseur, ce qui veut dire qu'un élément particulier pouvait avoir un poids élevé à lui seul, et être représenté dans un rang. Cette méthode a été utilisée sur le corpus COVID du projet Cocktail [58], mais ne figure pas dans ce chapitre car l'étape d'interprétation n'a pas encore été réalisée par les experts métiers.

5.4 Conclusion

Ce chapitre a montré comment des processus de collecte, de stockage et d'analyse de données peuvent être assemblés pour former un observatoire. Celui que nous avons mis en place dans le cadre du projet Cocktail concerne les données du réseau social Twitter. Son fonctionnement nécessite des interventions d'experts métiers pour sélectionner les critères pertinents de la collecte, mais aussi pour identifier les questions de recherche qui méritent d'être étudiées en fonction de ce que peuvent montrer les premières analyses macroscopiques.

Les analyses réalisées avec cet observatoire suivent une granularité qui permet tout d'abord de valider le corpus créé pour l'étude, puis de plonger dans ses données pour identifier les éléments importants, pour enfin mettre en place des analyses plus fines et dépendantes de la situation, ayant différents objectifs et extrayant différentes informations des données.

L'objectif final du projet Cocktail est l'industrialisation la plateforme et le développement d'une interface utilisateur par les entreprises partenaires du projet, pour pouvoir proposer des services aux entreprises, et ce avec 3 niveaux de granularité pour les analyses (indicateur macroscopiques, analyses standards et analyses expertes). Dans ce contexte, le code de l'architecture Hydre fait l'objet d'un transfert de technologie vers l'entreprise en charge de l'industrialisation de l'observatoire.

Par ailleurs, d'un point de vue recherche, les analyses microscopiques et les *notebooks* Jupyter ont été utilisés par les chercheurs en sciences humaines et sociales et ont contribué à réaliser plusieurs publications dans les 18 derniers mois. D'autres doctorants de l'équipe exploitent l'observatoire et ses fonctionnalités pour produire des publications qui concernent leur thématique de recherche.

Publications relatives au chapitre

*****

Concernant les publications réalisées en relation avec ce chapitre, elles montrent les analyses produites à l'aide de l'observatoire et qui ont servi à l'écriture d'un article. L'observatoire qui a été mis en place résultant du travail de ma thèse sert donc pour d'autres travaux auxquels j'ai collaboré, à savoir la thèse d'Hiba Abou Jamra (publication [D2]) et le stage orienté recherche d'Alexis Guyot (publication [D5]). Par rapport au projet Cocktail, le travail effectué a conduit à la rédaction de deux livrables de *work packages* (publications [D3] et [D4]).

[D1] Annabelle Gillet, Alexander Frame, Gilles Brachotte, Éric Leclercq, Marinette Savonnet. **Analysis of the 2014 and 2019 European elections using Twitter data (sans actes)**. In : *Modèles & Analyse des Réseaux : Approches Mathématiques & Informatiques (MARAMI)*. 2019, pp. 1–4.

[D2] Hiba Abou Jamra, Annabelle Gillet, Éric Leclercq, Marinette Savonnet. **Analyse des discours sur Twitter dans une situation de crise — Étude de l'incident à l'usine Lubrizol de Rouen**. In : *INFormatique des ORganisations et Systèmes d'Information et de Décision (INFORSID)*. 2020, pp. 1–16.

[D3] Hiba Abou Jamra, Annabelle Gillet, Éric Leclercq, Marinette Savonnet. *Observatoire des signaux faibles et des tendances dans le discours alimentaire : le cocktail Twitter — Livrable Workpackage WP1 — Spécifications fonctionnelles et techniques de COCKTAIL – fonctionnalités, acteurs et scénarios (ANR-15-IDEX-0003)*. Rapp. tech. Laboratoire d'Informatique de Bourgogne, 2020.

[D4] Annabelle Gillet, Éric Leclercq, Nadine Cullot. *Observatoire des signaux faibles et des tendances dans le discours alimentaire : le cocktail Twitter — Livrable Workpackage WP3 — Prototypage de Cocktail – Collecte et stockage des données issues de Twitter (ANR-15-IDEX-0003)*. Rapp. tech. Laboratoire d'Informatique de Bourgogne, 2020.

[D5] Alexis Guyot, Annabelle Gillet, Éric Leclercq. **Frontières des communautés polarisées : application à l'étude des théories complotistes autour des vaccins**. In : *INFormatique des ORganisations et Systèmes d'Information et de Décision (INFORSID)*. 2021, pp. 1–16.

Chapitre

6

Conclusion et perspectives

*****

6.1 Contributions 136
6.1.1 La Lambda+ Architecture et la formalisation d'architectures basée sur la théorie des catégories 136
6.1.2 Le *Tensor Data Model* 137
6.1.3 L'implémentation des contributions 138
6.1.4 La mise en place de l'observatoire 138
6.2 Perspectives 138
6.2.1 La Lambda+ Architecture et la formalisation d'architectures basée sur la théorie des catégories 139
6.2.2 Le *Tensor Data Model* 139
6.2.3 Synthèse 140

Face à la prédominance des données massives, les opportunités d'en extraire des informations qui peuvent être converties en valeur sont nombreuses. Toutefois, cette extraction de valeur nécessite de lever plusieurs verrous importants, qui incluent la construction d'architectures logicielles supportant des propriétés essentielles telles que la résistance aux pannes, l'exactitude des traitements et les calculs en temps réel, tout en prenant en considération les caractéristiques des données massives, principalement la vélocité et le volume. Un autre verrou concerne la variété des données à intégrer dans les outils d'analyse, que ce soit par rapport au nombre d'attributs disponibles ou à la diversité des modèles, ainsi que la capacité à évaluer la véracité des données manipulées, pour pouvoir enfin prétendre à l'extraction de leur valeur.

Afin de lever ces verrous, notre approche consiste à prendre en compte les problématiques induites par les données massives à tous les niveaux de la chaîne de traitement des données, depuis leur création jusqu'à l'interprétation des résultats d'analyse, en passant par leur stockage et leur mise à disposition. L'efficacité de chacune de ces étapes n'est pas suffisante pour traiter les données massives, et il faut également qu'elles puissent garantir une certaine sûreté afin d'éviter au maximum les erreurs de manipulation et de traitement, qui se reporteraient alors dans toutes les étapes suivantes, entraînant soit des coûts et efforts supplémentaires pour identifier et corriger ces erreurs, soit des interprétations erronées.

Les contributions de cette thèse se placent dans ce contexte, et interviennent à plusieurs endroits de la chaîne de traitement des données massives, à savoir lors de la conception d'architectures logicielles et lors du développement de couches d'abstraction pour supporter la variété des données, afin de guider la collecte, le stockage et l'analyse des données massives. Les deux contributions majeures de cette thèse sont :

- le **patron Lambda+ Architecture** qui combine la possibilité de calculer des indicateurs macroscopiques en temps réel et de réaliser des analyses exploratoires. Ce patron est accompagné d'une **formalisation reposant sur la théorie des catégories, visant à étudier la conservation des propriétés lors de la composition de composants d'une architecture** ;
- le ***Tensor Data Model*** qui ajoute un schéma à l'objet mathématique tenseur, apportant de cette manière une modélisation multi-dimensionnelle flexible et de puissants opérateurs d'analyse, comme la décomposition tensorielle CANDECOMP/PARAFAC. Le modèle bénéficie d'une implémentation qui intègre une grande sûreté de typage et un mécanisme d'inférence de schéma.

En plus de ces contributions majeures, deux autres contributions sont présentes :

- le **développement d'implémentations correspondant aux travaux de recherche**, qui se servent de technologies adaptées et intègrent la sûreté nécessaire à leur utilisation. La mise à disposition du code et des expériences de validation contribuent à la reproductibilité de mes travaux ;
- les **processus mis en place afin de constituer un observatoire** traitant de données sociales, dont le but est de collecter, stocker et analyser les données massives. Ces processus incluent la préparation et la mise à disposition des données avant de pouvoir réaliser diverses analyses à plusieurs niveaux de granularité.

6.1 Contributions

Cette section présente une synthèse des différentes contributions développées tout au long de la thèse.

6.1.1 La Lambda+ Architecture et la formalisation d'architectures basée sur la théorie des catégories

La Lambda+ Architecture s'inspire de la Lambda Architecture, dont le but est de traiter les données massives avec les propriétés de temps réel, d'exactitude des traitement et de résistance aux pannes. Bien que la résistance aux pannes soit avérée, les propriétés de temps réel et d'exactitude des traitements ne peuvent être obtenues simultanément, et sont valides uniquement dans des couches distinctes de l'architecture, à savoir la couche *speed* pour la propriété temps réel et la couche *batch* pour l'exactitude des traitements. Il est donc essentiel d'être capable de connaître les impacts sur l'architecture globale que vont avoir les compositions de composants, d'autant plus lors de l'évolution, de la maintenance ou de la fusion d'architectures.

La perte de ces propriétés dans la Lambda Architecture globale a été prouvée grâce à une formalisation utilisant la théorie des catégories, qui permet d'étudier les propriétés conservées ou perdues lorsque les composants d'une architecture sont assemblés dans une même composition. Cette formalisation est idéale pour étudier les architectures, puisque la théorie des catégories met nativement l'accent sur les compositions, qui sont fortement présentes dans les architectures lorsque les composants (qui ont chacun un rôle, des propriétés, des paradigmes et des technologies différents) interagissent.

Cette formalisation a été également appliquée sur le patron Lambda+ Architecture, pour montrer que, contrairement à la Lambda Architecture, les propriétés de temps réel et d'exactitude des traitements sont supportées dans la globalité de l'architecture. La Lambda+ Architecture propose également d'autres évolutions par rapport à la Lambda. En effet, elle étend les cas d'utilisation de la Lambda, qui étaient fortement limités car réduits au calcul de requêtes prédéfinies. Pour cela, elle intègre la flexibilité nécessaire au traitement des données massives, en calculant d'un côté des indicateurs macroscopiques en temps réel, qui correspondent à des besoins bien connus et identifiés, et d'un autre côté en permettant de réaliser des analyses exploratoires sur les données, nécessaires à la découverte de nouvelles informations ou à l'exploration d'anomalies détectées grâce aux indicateurs macroscopiques. Les évolutions des systèmes de *stream processing* sont également intégrées dans la Lambda+ Architecture, ce qui permet de supporter l'exactitude des traitements dans les processus utilisant le *stream processing*, et par la même occasion de supprimer la complexité induite par la duplication des traitements dans la couche *speed* et dans la couche *batch*.

6.1.2 Le *Tensor Data Model*

Le *Tensor Data Model* (TDM) propose d'utiliser l'objet mathématique tenseur pour construire un modèle de données. Les tenseurs sont des objets abstraits multi-dimensionnels, qui sont capables de jouer le rôle de modèles pivots par rapport à de nombreux autres modèles. Ils possèdent également de puissants opérateurs, tels que les décompositions tensorielles, qui permettent d'étudier les relations latentes. Toutefois, les tenseurs sous leur forme mathématique sont éloignés des données. Leurs dimensions, ainsi que les éléments des dimensions, sont représentés uniquement par des entiers, ce qui force l'utilisateur à maintenir lui-même une correspondance entre ces entiers et leur réelle signification. Or, cette démarche est propice aux erreurs, et est éloignée des bonnes pratiques habituelles de la programmation. L'application des opérateurs, qui peut modifier le nombre de dimensions et le nombre d'éléments dans une dimension d'un tenseur, peuvent également contribuer à induire des erreurs dans la correspondance entre entiers et signification réelle, sans parfois même que l'utilisateur ne s'en aperçoive.

TDM se positionne comme une surcouche aux *polystores*, et ajoute la notion de schéma aux tenseurs. Ce schéma est accompagné d'opérateurs de manipulation de données, qui contribuent à garder les données au cœur du modèle. La fusion entre les tenseurs et un schéma de données renforce la position de TDM en tant que modèle pivot, idéal pour faire face à la variété des modèles de données présente dans les *polystores*. De plus, les opérateurs tensoriels tels que les décompositions participent aux capacités analytiques de TDM.

L'implémentation de TDM suit l'esprit du modèle. Elle est développée en Scala, et met en place une sûreté de typage ainsi qu'une inférence de schéma grâce aux mécanismes internes du langage tels que les *phantom types*, les *implicits* ou la librairie shapeless. Cela permet, lors de la compilation, d'éviter des erreurs techniques mais également fonctionnelles, en vérifiant que les paramètres des opérateurs ne mènent pas à une incohérence, mais surtout en inférant le schéma du tenseur résultat, même lorsque l'opérateur entraîne une modification du schéma (avec une jointure par exemple).

Les performances sont aussi prises en compte dans l'implémentation. Apache Spark est utilisé afin d'optimiser les différents calculs, mais aussi afin de pouvoir profiter de ses capacités d'exécution distribuée. De cette manière, et en intégrant des optimisations multi-niveaux, il a été possible de développer la décomposition CANDECOMP/PARAFAC à large échelle, tout en améliorant significativement le temps d'exécution par rapport aux implémentations de l'état de l'art. Les performances sont également visibles plus globalement dans la librairie TDM, puisque j'ai démontré que TDM n'entraîne pas de différence en temps d'exécution par rapport aux mêmes opérations réalisées avec Spark seul.

6.1.3 L'implémentation des contributions

Une attention particulière a été portée lors de l'implémentation des contributions. En effet, la sûreté, l'optimisation et la reproductibilité sont essentielles pour que la diffusion des résultats se fasse dans de bonnes conditions. La diffusion des résultats doit permettre à ceux qui les consultent de les comprendre, de les reproduire et de les réutiliser.

L'architecture Hydre, qui est une application du patron Lambda+ Architecture, met donc en œuvre des technologies en adéquation avec les besoins des composants, et qui permettent de supporter les propriétés attendues. Le déploiement de l'architecture sur plusieurs serveurs physiques (un *cluster* Kafka, un *cluster* Hadoop et plusieurs autre serveurs) lui permet de collecter des tweets en continu pour le projet Cocktail, pour ensuite les stocker en vue d'être analysés, mais aussi de produire en temps réel des indicateurs macroscopiques utiles aux experts métiers.

La librairie TDM fusionne optimisations et sûreté d'utilisation. Sa présentation à la conférence ScalaCon a eu de bons retours, ce qui nous permet de conforter son utilité en tant que modèle pivot et outil d'analyse. De plus, les différentes expériences et exemples d'utilisation réalisés avec la librairie sont également rendus disponibles, dans le but de guider ceux qui souhaitent s'en servir.

6.1.4 La mise en place de l'observatoire

Les données collectées par l'architecture Hydre font partie de l'observatoire du projet Cocktail. Le choix des critères utilisés pour les collectes est fait par des experts métiers, qui ciblent les sujets d'intérêt par rapport au domaine d'étude. On distingue deux types d'utilisateurs différents pour cet observatoire : les experts métiers, qui identifient les questions de recherche et interprètent les résultats, et les analystes, qui, en fonction des questions de recherche, sélectionnent et exécutent des algorithmes sur les données après avoir créé et nettoyé un corpus correspondant à l'étude à réaliser.

Les besoins de ces utilisateurs sont donc différents, mais nécessitent tous d'avoir accès aux données, sous différentes formes. Les experts métiers doivent pouvoir consulter des indicateurs macroscopiques, servant à confirmer la bonne orientation de la collecte mais aussi à déceler des points d'intérêt sur lesquels réaliser des analyses. Les analystes doivent pouvoir accéder à toutes les données sous une forme suffisamment flexible pour leur permettre d'utiliser les algorithmes les plus adaptés. Pour cela, les données sont stockées dans un *polystore* sous différents schémas, laissant ainsi la possibilité à l'analyste de se servir des données qui l'intéressent.

Les analyses suivent plusieurs niveaux de granularité. Les indicateurs macroscopiques sont calculés en temps réel et lors de la création des corpus d'étude. Ils servent à la fois à identifier les tendances, mais aussi à déceler certaines anomalies qui peuvent guider les analyses. La détection de communautés et le calcul de la centralité se situent au niveau mésoscopique. Ils permettent de structurer les données en fonction de différents éléments, comme les comptes ou les hashtags pour Twitter. Enfin, les analyses microscopiques ciblent un élément particulier, et nécessitent la sélection puis l'exécution d'un algorithme spécifique, comme nous l'avons réalisé pour l'étude des discours de haine ou de la polarisation des communautés.

6.2 Perspectives

Les perspectives qui découlent des contributions sont de trois types : des apports théoriques, des perspectives directes qui sont une extension naturelle des résultats, ou des perspectives à plus long terme

visant à unifier les points de vue architecture et analyses pour le traitement des données massives.

6.2.1 La Lambda+ Architecture et la formalisation d'architectures basée sur la théorie des catégories

La formalisation se servant de la théorie des catégories a révélé un potentiel important, et de ce fait plusieurs perspectives d'évolution se dessinent. Tout d'abord, à des fins de vérification, la formalisation doit être appliquée à plusieurs styles, patrons et implémentations d'architectures afin de valider son fonctionnement plus largement, mais aussi de détecter des éventuelles faiblesses et points d'amélioration. Le système de modélisation des propriétés, de son côté, peut être étendu afin d'intégrer des échelles de valeurs. Cela pourrait être utilisé par exemple pour indiquer le temps d'exécution d'un composant, puis par composition d'obtenir le temps d'exécution global d'un élément traité dans toute l'architecture. Enfin, la formalisation pourrait être aussi utilisée pour naviguer entre différents niveaux d'abstraction d'une architecture, et représenter un composant plus en détail. Ce mécanisme pourrait servir à pousser la granularité jusqu'au niveau du code, et de cette manière intégrer également les appels de fonctions et placer les *design patterns* au même niveau que les patrons ou styles d'architectures. De plus, l'utilisation des foncteurs pleins (des foncteurs surjectifs) pourrait servir à vérifier qu'une implémentation suit un style ou un patron d'architecture.

En adoptant un autre point de vue, la Lambda+ Architecture peut être vue comme une architecture de *data lake*. En effet, elle réceptionne les données, et les intègre dans un système de stockage pour ensuite les exploiter. Mais elle propose également un composant intéressant dans le cas d'un *data lake* : le composant *real-time insights*. Une façon d'organiser les *data lakes* est de produire des métadonnées qui peuvent ensuite être utilisées pour naviguer dans le *data lake*, trouver les données d'intérêt et les exploiter [67]. Le composant *real-time insights* pourrait servir à proposer un système de création de métadonnées en temps réel, lors de l'intégration des données. Combiné à différentes techniques, telles que les annotations ou les ontologies, cela permettrait de fournir automatiquement un aperçu des données contenues dans le *data lake*.

6.2.2 Le *Tensor Data Model*

Concernant TDM, une des évolutions prévues visant à renforcer sa sûreté de typage d'un point de vue théorique consiste à adopter le système F [149] pour formaliser les opérateurs. En effet, il est utilisé par M. Odersky pour prouver les mécanismes avancés de Scala concernant le typage, en incluant les *implicits* [131]. Le système F est une extension d'ordre supérieur du lambda-calcul typé, et il est idéal pour formaliser les fonctions tout en prenant les types en considération. L'utiliser sur le modèle TDM renforcerait donc considérablement sa formalisation, et porterait la propriété de sûreté du typage, voire également celle d'inférence de schéma, directement à l'intérieur du modèle.

La librairie TDM continuera elle aussi à recevoir des mises à jour, qui peuvent entraîner une extension du modèle. Elles concernent plusieurs points. Tout d'abord, la diversification des opérateurs de manipulation de données. Comme les tenseurs ont la capacité de représenter différents modèles, il est possible de porter les opérateurs spécifiques de ces modèles directement dans TDM. Par exemple, le parallèle entre le modèle OLAP, souvent représenté en tant que cube, et TDM est direct. Les opérateurs associés, tels que DRILL DOWN ou ROLL UP, pourraient donc faire partie intégrante de TDM. Les opérateurs de manipulation de graphes pourraient aussi être ajoutés sur les tenseurs et dans TDM, comme par exemple l'algèbre proposée dans [60]. Ensuite, d'autres décompositions tensorielles permettraient de diversifier les analyses réalisables avec TDM. Des décompositions telles que Tucker ou DEDICOM seront donc implémentées, tout en prenant en

considération les optimisations multi-niveaux qui peuvent être apportées à ces décompositions pour rendre plus efficace leur application sur des tenseurs volumineux. Un dernier point d'amélioration de la librairie consiste à développer une version particulière de la décomposition CANDECOMP/PARAFAC pour les tenseurs de taille réduite. Bien que l'exécution de la décomposition actuelle sur des petits tenseurs présente des temps d'exécution convenables, ils pourraient être davantage améliorés en utilisant des librairies telles que breeze [1] plutôt que Spark, qui peut induire de l'*overhead* sur des données peu volumineuses lors de la distribution des données.

Au niveau de la décomposition CANDECOMP/PARAFAC, son utilisation sur des données des réseaux sociaux est en cours d'expérimentation. Le fait d'obtenir des résultats mettant en relation plusieurs dimensions ouvrent de nombreuses possibilités en termes d'analyse. Par exemple, avec les données de Twitter il est possible de mettre en correspondance des hashtags, des périodes de temps, des utilisateurs qui produisent du contenu et des utilisateurs qui rediffusent ce contenu. Toutefois, ces analyses sont également confrontées à des problématiques qui doivent être résolues pour améliorer leur qualité. En effet, si le choix du rang sur des tenseurs standards bénéficie de quelques références dans la littérature pour le guider [28], les tenseurs contenant des données suivant une loi de puissance, comme c'est bien souvent le cas pour les données des réseaux sociaux, entraînent des difficultés lors de l'évaluation de ce rang. D'après les expérimentations sur le rang du chapitre 5, les *clusters* ayant un poids relativement moins importants que les plus gros *clusters* du tenseur ne sont pas détectés sans déflation préalable, bien qu'ils puissent tout de même contenir des résultats intéressants. Une approche en cours d'étude consisterait à fonctionner par couches, en réalisant une décomposition avec un rang idéal estimé grâce à une méthode comme CORCONDIA, pour ensuite effectuer une déflation du tenseur, cette fois non pas en fonction du résultat d'une décomposition de rang 1 [15], mais de rang n, dans le but de supprimer le signal le plus fort du tenseur et de pouvoir accéder aux signaux plus faibles. Cette approche aurait l'avantage de pouvoir attribuer à chaque *cluster* l'intensité du signal qu'il représente, depuis les signaux les plus forts et facilement observables même sans analyses particulières, jusqu'aux signaux faibles, difficilement détectables, et potentiels précurseurs d'évènements.

6.2.3 Synthèse

Avoir une vision globale de tous les traitements qui peuvent être réalisés sur les données, de leur collecte à leur analyse, est essentiel afin de mieux comprendre les problématiques et les enjeux généraux. C'est un élément que j'ai pu prendre en compte lors de ma thèse, grâce à mes travaux tout d'abord sur les architecture logicielles avec la Lambda+ Architecture, puis sur les outils d'analyse avec le *Tensor Data Model*. Ces deux contributions ont ouvert de nombreuses perspectives, toutes intéressantes à explorer, qui vont constituer le fil conducteur de mes travaux pour les prochaines années, tout en maintenant les apports croisés entre les fondements théoriques et les implémentations techniques. J'ai pu réellement mettre en pratique l'apport de la théorie à la technique mais aussi les aspects de science expérimentale de l'informatique, en constatant que la maîtrise des éléments théoriques est une étape essentielle qui demande un travail de fond important.

1. Librairie d'algèbre linéaire en Scala `https://github.com/scalanlp/breeze`

Liste générale des publications

*****

Dans cette liste, les références [G1], [G3], [G7] et [G10] sont des présentations à des colloques sélectionnées sur résumé. La vidéo de la présentation associée à [G7] est disponible en ligne, celle de [G10] sera disponible en novembre.

[G1] Annabelle Gillet, Alexander Frame, Gilles Brachotte, Éric Leclercq, Marinette Savonnet. **Analysis of the 2014 and 2019 European elections using Twitter data (sans actes)**. In : *Modèles & Analyse des Réseaux : Approches Mathématiques & Informatiques (MARAMI)*. 2019, pp. 1–4.

[G2] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **Lambda Architecture pour une analyse à haute performance des données des réseaux sociaux**. In : *INFormatique des ORganisations et Systèmes d'Information et de Décision (INFORSID)*. 2019, pp. 1–16.

[G3] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **Une plateforme haute performance pour l'exploitation des données massives**. In : *DataBFC2 (`https://databfc2.sciencesconf.org/`)*. 2019.

[G4] Éric Leclercq, Annabelle Gillet, Thierry Grison, Marinette Savonnet. **Polystore and Tensor Data Model for Logical Data Independence and Impedance Mismatch in Big Data Analytics**. In : *Transactions on Large-Scale Data-and Knowledge-Centered Systems XLII (TLDKS)*. Springer, 2019, pp. 51–90.

[G5] Hiba Abou Jamra, Annabelle Gillet, Éric Leclercq, Marinette Savonnet. **Analyse des discours sur Twitter dans une situation de crise — Étude de l'incident à l'usine Lubrizol de Rouen**. In : *INFormatique des ORganisations et Systèmes d'Information et de Décision (INFORSID)*. 2020, pp. 1–16.

[G6] Hiba Abou Jamra, Annabelle Gillet, Éric Leclercq, Marinette Savonnet. *Observatoire des signaux faibles et des tendances dans le discours alimentaire : le cocktail Twitter — Livrable Workpackage WP1 — Spécifications fonctionnelles et techniques de COCKTAIL – fonctionnalités, acteurs et scénarios (ANR-15-IDEX-0003)*. Rapp. tech. Laboratoire d'Informatique de Bourgogne, 2020.

[G7] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **Lambda+ Architecture — Un patron d'architecture haute performance pour le traitement des Big Data**. In : *Journées Calcul et Données (JCAD, `https://jcad2020.sciencesconf.org/`)*. 2020.

[G8] Annabelle Gillet, Éric Leclercq, Nadine Cullot. *Observatoire des signaux faibles et des tendances dans le discours alimentaire : le cocktail Twitter — Livrable Workpackage WP3 — Prototypage de Cocktail – Collecte et stockage des données issues de Twitter (ANR-15-IDEX-0003)*. Rapp. tech. Laboratoire d'Informatique de Bourgogne, 2020.

[G9] Annabelle Gillet, Éric Leclercq, Marinette Savonnet, Nadine Cullot. **Empowering big data analytics with polystore and strongly typed functional queries**. In : *Proceedings of the 24th Symposium on International Database Engineering & Applications (IDEAS)*. 2020, pp. 1–10.

[G10] Annabelle Gillet, Éric Leclercq. **TDM : Breaking through dimensions with tensors**. In : *ScalaCon (`http://www.scalacon.org/`)*. 2021.

[G11] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **Évolution et formalisation de la Lambda Architecture pour des analyses à hautes performances — Application aux données de Twitter**. In : *Revue ouverte d'ingénierie des systèmes d'information (ISI)* Numéro 1 (2021), pp. 1–26.

[G12] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **Lambda+, the renewal of the Lambda Architecture : Category Theory to the rescue**. In : *33rd Conference on Advanced Information Systems Engineering (CAiSE)*. Springer LNCS, 2021, pp. 1–15.

[G13] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **MuLOT : Multi-level optimization of the canonical polyadic tensor decomposition at large-scale**. In : *25th European Conference on Advances in Databases and Information Systems (ADBIS)*. 2021, pp. 1–15.

[G14] Alexis Guyot, Annabelle Gillet, Éric Leclercq. **Frontières des communautés polarisées : application à l'étude des théories complotistes autour des vaccins**. In : *INFormatique des ORganisations et Systèmes d'Information et de Décision (INFORSID)*. 2021, pp. 1–16.

Bibliographie

*****

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard. **Tensorflow : A system for large-scale machine learning**. In : *12th USENIX Symposium on Operating Systems Design and Implementation*. 2016, pp. 265–283.

[2] Gregory D Abowd, Robert Allen, David Garlan. **Formalizing style to understand descriptions of software architecture**. In : *ACM Transactions on Software Engineering and Methodology (TOSEM)* 4.4 (1995), pp. 319–364.

[3] Ashvin Agrawal, Rony Chatterjee, Carlo Curino, Avrilia Floratou, Neha Gowdal, Matteo Interlandi, Alekh Jindal, Kostantinos Karanasos, Subru Krishnan, Brian Kroth. **Cloudy with high chance of DBMS : A 10-year prediction for Enterprise-Grade ML**. In : *Conference on Innovative Data Systems Research (CIDR)*. 2020.

[4] Subutai Ahmad, Alexander Lavin, Scott Purdy, Zuha Agha. **Unsupervised real-time anomaly detection for streaming data**. In : *Neurocomputing* 262 (2017), pp. 134–147.

[5] Nawaaz Ahmed, Nikolay Mateev, Keshav Pingali, Paul Stodghill. **A framework for sparse matrix code synthesis from high-level specifications**. In : *SC'00 : Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE. 2000, pp. 58–58.

[6] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, Sam Whittle. **MillWheel : fault-tolerant stream processing at internet scale**. In : *VLDB Endowment* 6.11 (2013), pp. 1033–1044.

[7] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt. **The dataflow model : a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing**. In : (2015).

[8] Tyler Akidau, Slava Chernyak, Reuven Lax. **Streaming Systems : The What, Where, When, and how of Large-scale Data Processing**. " O'Reilly Media, Inc.", 2018.

[9] Réka Albert, Albert-László Barabási. **Statistical mechanics of complex networks**. In : *Reviews of modern physics* 74.1 (2002), pp. 47.

[10] Andrew L Alexander, Jee Eun Lee, Mariana Lazar, Aaron S Field. **Diffusion tensor imaging of the brain**. In : *Neurotherapeutics* 4.3 (2007), pp. 316–329.

[11] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl. **The stratosphere platform for big data analytics**. In : *The VLDB Journal* 23.6 (2014), pp. 939–964.

[12] Rana Alotaibi, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, Stamatis Zampetakis. **Towards Scalable Hybrid Stores : Constraint-Based Rewriting to the Rescue**. In : *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 1660–1677.

[13] Rana Alotaibi, Bogdan Cautis, Alin Deutsch, Moustafa Latrache, Ioana Manolescu, Yifei Yang. **ESTOCADA : towards scalable polystore systems**. In : *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 2949–2952.

[14] Nada Amin, Tiark Rompf, Martin Odersky. **Foundations of path-dependent types**. In : *ACM SIGPLAN Notices* 49.10 (2014), pp. 233–249.

[15] Miguel Araujo, Spiros Papadimitriou, Stephan Günnemann, Christos Faloutsos, Prithwish Basu, Ananthram Swami, Evangelos E Papalexakis, Danai Koutra. **Com2 : fast automatic discovery of temporal ('comet') communities**. In : *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2014, pp. 271–283.

[16] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi. **Spark sql : Relational data processing in spark**. In : *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 2015, pp. 1383–1394.

[17] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, Mike Paleczny. **Workload analysis of a large-scale key-value store**. In : *ACM SIGMETRICS Performance Evaluation Review*. T. 40. 1. ACM. 2012, pp. 53–64.

[18] Vaibhav Bajpai, Mirja Kühlewind, Jörg Ott, Jürgen Schönwälder, Anna Sperotto, Brian Trammell. **Challenges with reproducibility**. In : *Proceedings of the Reproducibility Workshop*. 2017, pp. 1–4.

[19] Albert-László Barabási. **Network science**. Cambridge University Press, 2016.

[20] Albert-László Barabási, Réka Albert. **Emergence of scaling in random networks**. In : *Science* 286.5439 (1999), pp. 509–512.

[21] Ian Basaille. **Médias sociaux et gestion de communautés-applications dans le domaine de la gestion de la relation client**. 2017.

[22] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, Kenneth Knowles. **One SQL to rule them all-an efficient and syntactically idiomatic approach to management of streams and tables**. In : *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 1757–1772.

[23] Zachary Blanco, Bangtian Liu, Maryam Mehri Dehnavi. **Cstf : Large-scale sparse tensor factorizations on distributed platforms**. In : *Proceedings of the 47th International Conference on Parallel Processing*. 2018, pp. 1–10.

[24] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, Etienne Lefebvre. **Fast unfolding of communities in large networks**. In : *Journal of statistical mechanics : theory and experiment* 2008.10 (2008), pp. P10008.

[25] Reza Bosagh Zadeh, Xiangrui Meng, Alexander Ulanov, Burak Yavuz, Li Pu, Shivaram Venkataraman, Evan Sparks, Aaron Staple, Matei Zaharia. **Matrix computations and optimization in apache spark**. In : *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2016, pp. 31–38.

[26] Oscar Boykin, Sam Ritchie, Ian O'Connell, Jimmy Lin. **Summingbird : A framework for integrating batch and online mapreduce computations**. In : *Proceedings of the VLDB Endowment* 7.13 (2014), pp. 1441–1451.

[27] Eric Brewer. **A certain freedom : thoughts on the CAP theorem**. In : *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. 2010, pp. 335–335.

[28] Rasmus Bro, Henk AL Kiers. **A new efficient method for determining the number of components in PARAFAC models**. In : *Journal of Chemometrics : A Journal of the Chemometrics Society* 17.5 (2003), pp. 274–286.

[29] William H Brown, Raphael C Malveau, Hays W" Skip" McCormick, Thomas J Mowbray. **AntiPatterns : refactoring software, architectures, and projects in crisis**. John Wiley & Sons, Inc., 1998.

[30] Manfred Broy. **Can practitioners neglect theory and theoreticians neglect practice ?** In : *Computer* 44.10 (2011), pp. 19–24. ISSN : 0018-9162.

[31] Manfred Broy, María Victoria Cengarle. **UML formal semantics : lessons learned**. In : *Software & Systems Modeling* 10.4 (2011), pp. 441–446.

[32] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, Kostas Tzoumas. **Apache flink : Stream and batch processing in a single engine**. In : *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[33] Ray M Chang, Robert J Kauffman, YoungOk Kwon. **Understanding the paradigm shift to computational social science in the presence of big data**. In : *Decision Support Systems* 63 (2014), pp. 67–80.

[34] Tongfei Chen. **Typesafe abstractions for tensor operations**. In : *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*. ACM. 2017, pp. 45–50.

[35] Eric C Chi, Tamara G Kolda. **On tensors, sparsity, and nonnegative factorizations**. In : *SIAM Journal on Matrix Analysis and Applications* 33.4 (2012), pp. 1272–1299.

[36] Andrzej Cichocki, Rafal Zdunek, Anh Huy Phan, Shunichi Amari. **Nonnegative matrix and tensor factorizations : applications to exploratory multi-way data analysis and blind source separation**. John Wiley & Sons, 2009.

[37] Tony Clark, Balbir S Barn. **Event driven architecture modelling and simulation**. In : *International Symposium on Service Oriented System*. IEEE. 2011, pp. 43–54.

[38] Paul C Clements. **A survey of architecture description languages**. In : *International Workshop on Software Specification and Design*. IEEE. 1996, pp. 16–25.

[39] Fengyu Cong, Qiu-Hua Lin, Li-Dan Kuang, Xiao-Feng Gong, Piia Astikainen, Tapani Ristaniemi. **Tensor decomposition of EEG signals : a brief review**. In : *Journal of neuroscience methods* 248 (2015), pp. 59-69.

[40] R. Conte, N. Gilbert, G. Bonelli, C. Cioffi-Revilla, G. Deffuant, J. Kertesz, V. Loreto, S. Moat, J. -P. Nadal, A. Sanchez, A. Nowak, A. Flache, M. San Miguel, D. Helbing. **Manifesto of computational social science**. In : *The European Physical Journal Special Topics* 214.1 (2012), pp. 325-346. ISSN : 1951-6401.

[41] Iain D Craig. **Blackboard systems**. In : *Artificial Intelligence Review* 2.2 (1988), pp. 103-118.

[42] Michael J Crawley. **The R book**. John Wiley & Sons, 2012.

[43] Alfredo Cuzzocrea, Ladjel Bellatreche, Il-Yeol Song. **Data warehousing and OLAP over big data : current challenges and future research directions**. In : *Proceedings of the sixteenth international workshop on Data warehousing and OLAP*. 2013, pp. 67-70.

[44] Subhasis Dasgupta, Kevin Coakley, Amarnath Gupta. **Analytics-driven data ingestion and derivation in the AWESOME polystore**. In : *2016 IEEE International Conference on Big Data (Big Data)*. IEEE. 2016, pp. 2555-2564.

[45] John Deacon. **Model-view-controller (MVC) architecture**. In : (2009).

[46] Zinovy Diskin, Tom Maibaum. **Category theory and model-driven engineering : From formal semantics to design patterns and beyond**. In : *Model-Driven Engineering of Information Systems : Principles, Techniques, and Practice* 173 (2014).

[47] James Dixon. **Pentaho, Hadoop, and data lakes**. In : *blog, Oct* (2010).

[48] Samuel Eilenberg, Saunders MacLane. **General theory of natural equivalences**. In : *Transactions of the American Mathematical Society* 58.2 (1945), pp. 231-294.

[49] Raul Castro Fernandez, Peter R Pietzuch, Jay Kreps, Neha Narkhede, Jun Rao, Joel Koshy, Dong Lin, Chris Riccomini, Guozhang Wang. **Liquid : Unifying Nearline and Offline Big Data Integration**. In : *Conference on Innovative Data System Research (CIDR'15)*. 2015.

[50] Brendan Fong, David I Spivak. **An invitation to applied category theory : seven sketches in compositionality**. Cambridge University Press, 2019.

[51] Brian Foote, Joseph Yoder. **Big ball of mud**. In : *Pattern languages of program design* 4 (1997), pp. 654-692.

[52] Vijay Gadepally, Peinan Chen, Jennie Duggan, Aaron Elmore, Brandon Haynes, Jeremy Kepner, Samuel Madden, Tim Mattson, Michael Stonebraker. **The BigDAWG Polystore System and Architecture**. In : *High Performance Extreme Computing Conference*. IEEE. 2016, pp. 1-6.

[53] Vijay Gadepally, Jeremy Kepner, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Lauren Edwards, Matthew Hubbell, Peter Michaleas, Julie Mullen. **D4M : Bringing associative arrays to database engines**. In : *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2015, pp. 1-6. DOI : `10.1109/hpec.2015.7322472`.

[54] Noé Gaumont, Maziyar Panahi, David Chavalarias. **Reconstruction of the socio-semantic dynamics of political activist Twitter networks—Method and application to the 2017 French presidential election**. In : *PloS one* 13.9 (2018), pp. e0201879.

[55] Laetitia Gauvin, André Panisson, Ciro Cattuto. **Detecting the community structure and activity patterns of temporal networks : a non-negative tensor factorization approach**. In : *PloS one* 9.1 (2014), pp. e86028.

[56] Valerio Gemmetto, Alain Barrat, Ciro Cattuto. **Mitigation of infectious disease at school : targeted class closure vs school closure**. In : *BMC infectious diseases* 14.1 (2014), pp. 1-10.

[57] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **Lambda Architecture pour une analyse à haute performance des données des réseaux sociaux**. In : *INFormatique des ORganisations et Systèmes d'Information et de Décision (INFORSID)*. 2019, pp. 1-16.

[58] Annabelle Gillet, Éric Leclercq, Nadine Cullot. **MuLOT : Multi-level optimization of the canonical polyadic tensor decomposition at large-scale**. In : *25th European Conference on Advances in Databases and Information Systems (ADBIS)*. 2021, pp. 1-15.

[59] Nazli Goharian, Ankit Jain, Qian Sun. **Comparative analysis of sparse matrix algorithms for information retrieval**. In : *computer* 2 (2003), pp. 0–4.

[60] Luiz Gomes-Jr, Bernd Amann, André Santanchè. **Beta-algebra : Towards a relational algebra for graph analysis**. In : *EDBT/ICDT 2015 Joint Conference*. T. 157. 2015.

[61] Aditya Gudibanda, Tom Henretty, Muthu Baskaran, James Ezick, Richard Lethin. **All-at-once decomposition of coupled billion-scale tensors in Apache Spark**. In : *High Performance extreme Computing Conference*. IEEE. 2018, pp. 1-8.

[62] Pedro Calais Guerra, Wagner Meira Jr, Claire Cardie, Robert Kleinberg. **A measure of polarization on social media networks based on community boundaries**. In : *Seventh international AAAI conference on weblogs and social media*. 2013.

[63] Ekta Gujral, Ravdeep Pasricha, Evangelos E Papalexakis. **Sambaten : Sampling-based batch incremental tensor decomposition**. In : *International Conference on Data Mining*. SIAM. 2018, pp. 387-395.

[64] Munish Gupta. **Akka essentials**. Packt Publishing Ltd, 2012.

[65] Dave Gurnell. **The Type Astronaut's Guide to Shapeless**. Lulu. com, 2017.

[66] Alexis Guyot, Annabelle Gillet, Éric Leclercq. **Frontières des communautés polarisées : application à l'étude des théories complotistes autour des vaccins**. In : *INFormatique des ORganisations et Systèmes d'Information et de Décision (INFORSID)*. 2021, pp. 1-16.

[67] Rihan Hai, Christoph Quix, Matthias Jarke. **Data lake concept and systems : a survey**. In : *arXiv preprint arXiv :2106.09592* (2021).

[68] Chen Hai-yan, Dong Wei, Wang Ji, Chen Huo-wang. **Verify UML statecharts with SMV**. In : *Wuhan University Journal of Natural Sciences* 6.1-2 (2001), pp. 183-190.

[69] Neil B Harrison, Paris Avgeriou. **Leveraging architecture patterns to satisfy quality attributes**. In : *European conference on software architecture*. Springer. 2007, pp. 263-270.

[70] Richard A Harshman. **Foundations of the PARAFAC procedure : Models and conditions for an "explanatory" multimodal factor analysis**. In : (1970).

[71] Richard A Harshman. **Models for analysis of asymmetrical relationships among N objects or stimuli**. In : *First Joint Meeting of the Psychometric Society and the Society of Mathematical Psychology, Hamilton, Ontario, 1978*. 1978.

[72] Christopher J Hillar, Lek-Heng Lim. **Most tensor problems are NP-hard**. In : *Journal of the ACM (JACM)* 60.6 (2013), pp. 1–39.

[73] Dylan Hutchison, Bill Howe, Dan Suciu. **Lara : A Key-Value Algebra underlying Arrays and Relations**. In : *arXiv preprint arXiv :1604.03607* (2016).

[74] Dylan Hutchison, Bill Howe, Dan Suciu. **LaraDB : A minimalist kernel for linear and relational algebra computation**. In : *ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*. ACM. 2017, pp. 2–12.

[75] Jean-François Im, Kishore Gopalakrishna, Subbu Subramaniam, Mayank Shrivastava, Adwait Tumbde, Xiaotian Jiang, Jennifer Dai, Seunghyun Lee, Neha Pawar, Jialiang Li. **Pinot : Realtime OLAP for 530 Million Users**. In : *ACM SIGMOD*. 2018, pp. 583–594.

[76] William H Inmon. **Building the data warehouse**. John wiley & sons, 2005.

[77] William H Inmon. **Data Lake Architecture : Designing the Data Lake and avoiding the garbage dump**. Technics publications, 2016.

[78] Hayden Jananthan, Ziqi Zhou, Vijay Gadepally, Dylan Hutchison, Suna Kim, Jeremy Kepner. **Polystore mathematics of relational algebra**. In : *International Conference on Big Data (Big Data)*. IEEE. Déc. 2017, pp. 3180–3189. DOI : 10.1109/BigData.2017.8258298.

[79] Inah Jeon, Evangelos E Papalexakis, U Kang, Christos Faloutsos. **Haten2 : Billion-scale tensor decompositions**. In : *International Conference on Data Engineering*. IEEE. 2015, pp. 1047–1058.

[80] Pontus Johnson, Mathias Ekstedt, Ivar Jacobson. **Where's the theory for software engineering ?** In : *IEEE software* 29.5 (2012), pp. 96–96. ISSN : 0740-7459.

[81] Paris C Kanellakis. **Elements of relational database theory**. In : *Formal models and semantics*. Elsevier, 1990, pp. 1073–1156.

[82] Evdokia Kassela, Nikodimos Provatas, Ioannis Konstantinou, Avrilia Floratou, Nectarios Koziris. **General-Purpose vs. Specialized Data Analytics Systems : A Game of ML & SQL Thrones**. In : *2019 IEEE International Conference on Big Data (Big Data)*. IEEE. 2019, pp. 317–326.

[83] Jeremy Kepner, Vijay Gadepally, Hayden Jananthan, Lauren Milechin, Siddharth Samsi. **AI Data Wrangling with Associative Arrays**. In : *arXiv preprint arXiv :2001.06731* (2020).

[84] Pwint Phyu Khine, Zhao Shun Wang. **Data lake : a new ideology in big data era**. In : *ITM web of conferences*. T. 17. EDP Sciences. 2018, pp. 03025.

[85] Mijung Kim. **TensorDB and tensor-relational model (TRM) for efficient tensor-relational operations**. Thèse de doct. Arizona State University, 2014.

[86] Mijung Kim, Selçuk Candan. **Approximate tensor decomposition within a tensor-relational algebraic framework**. In : *ACM international conference on Information and knowledge management (CIKM)*. ACM. 2011, pp. 1737–1742.

[87] Mijung Kim, Selçuk Candan. **TensorDB : In-database tensor manipulation with tensor-relational query plans**. In : *ACM International Conference on Conference on Information and Knowledge Management (CIKM)*. ACM. 2014, pp. 2039–2041.

[88] Miryung Kim. **Software engineering for data analytics**. In : *IEEE Software* 37.4 (2020), pp. 36–42.

[89] Mariam Kiran, Peter Murphy, Inder Monga, Jon Dugan, Sartaj Singh Baveja. **Lambda architecture for cost-effective batch and speed big data processing**. In : *IEEE International Conference on Big Data*. IEEE. 2015, pp. 2785–2792.

[90] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, Saman Amarasinghe. **The tensor algebra compiler**. In : *OOPSLA* (2017), pp. 1–29.

[91] Jon M Kleinberg. **Authoritative sources in a hyperlinked environment.** In : *SODA*. T. 98. Citeseer. 1998, pp. 668–677.

[92] Martin Kleppmann. **A Critique of the CAP Theorem**. In : *arXiv preprint arXiv :1509.05393* (2015).

[93] Martin Kleppmann, Jay Kreps. **Kafka, samza and the unix philosophy of distributed data**. In : (2015).

[94] Mieczyslaw M Kokar, Jerzy A Tomasik, Jerzy Weyman. **Data vs. decision fusion in the category theory framework**. In : 2001.

[95] Tamara G Kolda, Brett W Bader. **Tensor decompositions and applications**. In : *SIAM review* 51.3 (2009), pp. 455–500.

[96] Boyan Kolev, Oleksandra Levchenko, Esther Pacitti, Patrick Valduriez, Ricardo Vilaça, Rui Gonçalves, Ricardo Jiménez-Peris, Pavlos Kranas. **Parallel polyglot query processing on heterogeneous cloud data stores with LeanXcale**. In : *International Conference on Big Data*. IEEE. 2018, pp. 1757–1766.

[97] Boyan Kolev, Patrick Valduriez, Carlyna Bondiombouy, Ricardo Jiménez-Peris, Raquel Pau, José Pereira. **CloudMdsQL : querying heterogeneous cloud data stores with a common language**. In : *Distributed and parallel databases* 34.4 (2016), pp. 463–503.

[98] Jean Kossaifi, Yannis Panagakis, Anima Anandkumar, Maja Pantic. **Tensorly : Tensor learning in python**. In : *The Journal of Machine Learning Research* 20.1 (2019), pp. 925–930.

[99] Jay Kreps. **Questioning the Lambda Architecture**. In : *O'Reilly RADAR,online article, July* (2014). URL : `https://www.oreilly.com/ideas/questioning-the-lambda-architecture`.

[100] Jay Kreps. **Introducing kafka streams : Stream processing made simple**. In : *Confluent Blog, March* (2016).

[101] Jay Kreps, Neha Narkhede, Jun Rao. **Kafka : A distributed messaging system for log processing**. In : *Proceedings of the NetDB*. T. 11. 2011, pp. 1–7.

[102] Liwei Kuang, Fei Hao, Laurence T Yang, Man Lin, Changqing Luo, Geyong Min. **A tensor-based approach for big data representation and dimensionality reduction**. In : *IEEE transactions on emerging topics in computing* 2.3 (2014), pp. 280–291.

[103] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, Siddarth Taneja. **Twitter heron : Stream processing at scale**. In : *ACM SIGMOD*. 2015, pp. 239–250.

[104] Butler W Lampson. **Hints for computer system design**. In : *Proceedings of the ninth ACM symposium on Operating systems principles*. 1983, pp. 33–48.

[105] Doug Laney. **3D data management : Controlling data volume, velocity and variety**. In : *META group research note* 6.70 (2001), pp. 1.

[106] David Lazer, Alex Pentland, Lada Adamic, Sinan Aral, Albert-László Barabási, Devon Brewer, Nicholas Christakis, Noshir Contractor, James Fowler, Myron Gutmann. **Computational social science**. In : *Science* 323.5915 (2009), pp. 721-723.

[107] Daniel Le Métayer. **Describing software architecture styles using graph grammars**. In : *IEEE Transactions on software engineering* 24.7 (1998), pp. 521-533.

[108] Chung-Ho Lee, Chi-Yi Lin. **Implementation of Lambda Architecture : A Restaurant Recommender System over Apache Mesos**. In : *Int. Conf. on Advanced Information Networking and Applications (AINA)*. IEEE. 2017, pp. 979-985.

[109] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, Ce Zhang. **Mlog : Towards declarative in-database machine learning**. In : t. 10. 12. VLDB Endowment, 2017, pp. 1933-1936.

[110] Jimmy Lin. **The Lambda and the Kappa**. In : *IEEE Internet Computing* 21.5 (2017), pp. 60-66.

[111] Richard Lippmann, Robert K Cunningham, David J Fried, Isaac Graf, Kris R Kendall, Seth E Webster, Marc A Zissman. **Results of the DARPA 1998 Offline Intrusion Detection Evaluation.** In : *Recent advances in intrusion detection*. T. 99. 1999, pp. 829-835.

[112] Zhen Hua Liu, Jiaheng Lu, Dieter Gawlick, Heli Helskyaho, Gregory Pogossiants, Zhe Wu. **Multi-model database management systems-a look forward**. In : *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*. Springer, 2018, pp. 16-29.

[113] Jiaheng Lu, Irena Holubová. **Multi-model databases : a new journey to handle the variety of data**. In : *ACM Computing Surveys (CSUR)* 52.3 (2019), pp. 1-38.

[114] Hermano Lustosa, Anderson da Silva, Daniel da Silva, Patrick Valduriez, Fábio Porto. **SAVIME : An Array DBMS for Simulation Analysis and ML Models Predictions**. In : *Journal of Information and Data Management* 11.3 (2020), pp. 247-264.

[115] Hermano Lustosa, Noel Lemus, Fábio Porto, Patrick Valduriez. **Tars : An array model with rich semantics for multidimensional data**. In : *Forum and Demos at ER*. 1979. 2017, pp. 114-127.

[116] Gerard Maas, Francois Garillot. **Stream Processing with Apache Spark : Mastering Structured Streaming and Spark Streaming**. O'Reilly Media, 2019.

[117] Mohamed A Mabrok, Michael J Ryan. **Category theory as a formal mathematical foundation for model-based systems engineering**. In : *Appl. Math. Inf. Sci* 11 (2017), pp. 43-51.

[118] Antonio Maccioni, Riccardo Torlone. **Augmented access for querying and exploring a polystore**. In : *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE. 2018, pp. 77-88.

[119] Takanori Maehara, Kohei Hayashi, Ken-ichi Kawarabayashi. **Expected tensor decomposition with stochastic gradient descent**. In : *Proceedings of the AAAI Conference on Artificial Intelligence*. T. 30. 1. 2016.

[120] Zoran Majkić. **Big Data Integration Theory**. Springer, 2014.

[121] Alexander Malkis, Diego Marmsoler. **A model of service-oriented architectures**. In : *Brazilian Symposium on Components, Architectures and Reuse Software*. IEEE. 2015, pp. 110-119.

[122] Diego Marmsoler, Alexander Malkis, Jonas Eckhardt. **A model of layered architectures**. In : *arXiv preprint arXiv :1503.04916* 178 (2015), pp. 47-61. ISSN : 2075-2180.

[123] Nathan Marz. *How to beat the CAP theorem*. 2011. URL : `http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html`.

[124] Nathan Marz, James Warren. **Big Data : Principles and best practices of scalable real-time data systems**. Manning, 2015.

[125] Wes McKinney. **pandas : a foundational Python library for data analysis and statistics**. In : *Python for high performance and scientific computing* 14.9 (2011), pp. 1-9.

[126] Wes McKinney, PD Team. **Pandas-Powerful python data analysis toolkit**. In : *Pandas—Powerful Python Data Anal Toolkit* 1625 (2015).

[127] John Meehan, Stan Zdonik, Shaobo Tian, Yulong Tian, Nesime Tatbul, Adam Dziedzic, Aaron Elmore. **Integrating real-time and batch processing in a polystore**. In : *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*. IEEE. 2016, pp. 1-7.

[128] Bertrand Meyer, Bertrand Meyer, Bertrand Meyer, Bertrand Meyer. **Conception et programmation orientées objet**. Eyrolles, 2000.

[129] Amr A Munshi, Yasser Abdel-Rady I Mohamed. **Data lake lambda architecture for smart grids big data analytics**. In : *IEEE Access* 6 (2018), pp. 40463-40471.

[130] Dmitry Namiot, Manfred Sneps-Sneppe. **On micro-services architecture**. In : *International Journal of Open Information Technologies* 2.9 (2014), pp. 24-27.

[131] Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, Sandro Stucki. **Simplicitly : Foundations and applications of implicit function types**. In : *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1-29.

[132] Martin Odersky, Lex Spoon, Bill Venners. **Programming in scala**. Artima Inc, 2008.

[133] Bruno CdS Oliveira, Adriaan Moors, Martin Odersky. **Type classes as objects and implicits**. In : *ACM SIGPLAN Notices* 45.10 (2010), pp. 341-360.

[134] M Tamer Ozsu, Patrick Valduriez. **Principles of distributed database systems**. T. 2. Springer, 1999.

[135] Lawrence Page, Sergey Brin, Rajeev Motwani, Terry Winograd. *The PageRank citation ranking : Bringing order to the web.* Rapp. tech. Stanford InfoLab, 1999.

[136] Meilir Page-Jones. **Comparing techniques by means of encapsulation and connascence**. In : *Communications of the ACM* 35.9 (1992), pp. 147-151.

[137] Evangelos E Papalexakis, Christos Faloutsos, Nicholas D Sidiropoulos. **ParCube : Sparse parallelizable CANDECOMP-PARAFAC tensor decomposition**. In : *ACM Transactions on Knowledge Discovery from Data (TKDD)* 10.1 (2015), pp. 1-25.

[138] Evangelos E Papalexakis, Christos Faloutsos, Nicholas D Sidiropoulos. **Tensors for data mining and data fusion : Models, applications, and scalable algorithms**. In : *Transactions on Intelligent Systems and Technology (TIST)* 8.2 (2016), pp. 16.

[139] Namyong Park, Byungsoo Jeon, Jungwoo Lee, U Kang. **Bigtensor : Mining billion-scale tensor made easy**. In : *ACM International on Conference on Information and Knowledge Management*. 2016, pp. 2457-2460.

[140] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga. **PyTorch : An imperative style, high-performance deep learning library**. In : *Advances in Neural Information Processing Systems*. 2019, pp. 8024-8035.

[141] Giancarlo Perrone, Jose Unpingco, Haw-minn Lu. **Network visualizations with Pyvis and VisJS**. In : *arXiv preprint arXiv :2006.04951* (2020).

[142] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E Gonzalez, Joseph M Hellerstein, Anthony D Joseph, Aditya Parameswaran. **Towards scalable dataframe systems**. In : *arXiv preprint arXiv :2001.00888* (2020).

[143] Anh Huy Phan, Petr Tichavsky, Andrzej Cichocki. **Fast alternating LS algorithms for high order CANDECOMP/PARAFAC tensor factorizations**. In : *Transactions on Signal Processing* 61.19 (2013), pp. 4834-4846.

[144] Maksim Podkorytov, Michael Gubanov. **Hybrid. Poly : A Consolidated Interactive Analytical Polystore System**. In : *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE. 2019, pp. 1996-1999.

[145] Pascal Pons, Matthieu Latapy. **Computing communities in large networks using random walks**. In : *International symposium on computer and information sciences*. Springer. 2005, pp. 284-293.

[146] Christos Psarras, Lars Karlsson, Jiajia Li, Paolo Bientinesi. **The landscape of software for tensor computations**. In : *arXiv preprint arXiv :2103.13756* (2021).

[147] Junaid Qadir, Anwaar Ali, Raihan Rasool, Andrej Zwitter, Arjuna Sathiaseelan, Jon Crowcroft. **Crisis analytics : big data-driven crisis response**. In : *Journal of International Humanitarian Action* 1.1 (2016), pp. 1-21.

[148] Stephan Rabanser, Oleksandr Shchur, Stephan Günnemann. **Introduction to tensor decompositions and their applications in machine learning**. In : *arXiv preprint arXiv :1711.10781* (2017).

[149] John C Reynolds. **Towards a theory of type structure**. In : *Programming Symposium*. Springer. 1974, pp. 408-425.

[150] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky. **Theano : A Python framework for fast computation of mathematical expressions**. In : *arXiv :1605.02688* (2016).

[151] Mark Richards, Neal Ford. **Fundamentals of Software Architecture**. O'Reilly, 2020.

[152] Martin Rosvall, Carl T Bergstrom. **Maps of random walks on complex networks reveal community structure**. In : *Proceedings of the national academy of sciences* 105.4 (2008), pp. 1118-1123.

[153] Alexander Rush. *Tensor Considered Harmful*. Rapp. tech. Harvard NLP, 2010. URL : `http://nlp.seas.harvard.edu/NamedTensor`.

[154] Pegdwendé Sawadogo, Jérôme Darmont. **On data lake architectures and metadata management**. In : *Journal of Intelligent Information Systems* (2020), pp. 1-24.

[155] Pegdwendé Sawadogo, Tokio Kibata, Jérôme Darmont. **Metadata Management for Textual Documents in Data Lakes**. In : *21st International Conference on Enterprise Information Systems (ICEIS 2019)*. 2019.

[156] Nicholas D Sidiropoulos, Lieven De Lathauwer, Xiao Fu, Kejun Huang, Evangelos E Papalexakis, Christos Faloutsos. **Tensor decomposition for signal processing and machine learning**. In : *Transactions on Signal Processing* 65.13 (2017), pp. 3551–3582.

[157] Darja Solodovnikova, Laila Niedrite. **Handling Evolution in Big Data Architectures**. In : *Baltic Journal of Modern Computing* 8.1 (2020), pp. 21–47.

[158] David I Spivak. **Category theory for the sciences**. MIT Press, 2014.

[159] Paul Springer, Tong Su, Paolo Bientinesi. **HPTT : a high-performance tensor transposition C++ library**. In : *ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. 2017, pp. 56–62.

[160] Michael Stonebraker. **In search of database consistency**. In : *Communications of the ACM* 53.10 (2010), pp. 8–9.

[161] Michael Stonebraker, Ugur Cetintemel. **"One size fits all" : an idea whose time has come and gone**. In : *International Conference on Data Engineering (ICDE'05)*. IEEE. 2005, pp. 2–11.

[162] Jimeng Sun, Dacheng Tao, Christos Faloutsos. **Beyond streams and graphs : dynamic tensor analysis**. In : *ACM SIGKDD International Conference on Knowledge Discovery and Data mining*. ACM. 2006, pp. 374–383.

[163] Gabriele Taentzer, Rick Salay, Daniel Strüber, Marsha Chechik. **Transformations of software product lines : A generalizing framework based on category theory**. In : *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE. 2017, pp. 101–111.

[164] K Toon. *A Database-Centric Approach to J2EE Application Development*. 2004.

[165] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham. **Storm@ twitter**. In : *ACM SIGMOD*. 2014, pp. 147–156.

[166] David Toth. **Database engineering from the category theory viewpoint**. In : 2008, pp. 37.

[167] Ledyard R Tucker. **Some mathematical notes on three-mode factor analysis**. In : *Psychometrika* 31.3 (1966), pp. 279–311.

[168] Stefan Van Der Walt, S Chris Colbert, Gael Varoquaux. **The NumPy array : a structure for efficient numerical computation**. In : *Computing in Science & Engineering* 13.2 (2011), pp. 22.

[169] Panos Vassiliadis, Alkis Simitsis. **Near real time ETL**. In : *New trends in data warehousing and data analysis*. Springer, 2009, pp. 1–31.

[170] Fangjin Yang, Gian Merlino, Nelson Ray, Xavier Léauté, Himanshu Gupta, Eric Tschetter. **The RAD-Stack : Open Source Lambda Architecture for Interactive Analytics**. In : *Proceedings of the 50th Hawaii International Conference on System Sciences*. 2017.

[171] Rose Yu, Stephan Zheng, Anima Anandkumar, Yisong Yue. **Long-term forecasting using tensor-train rnns**. In : *Arxiv* (2017).

[172] Matei Zaharia, Bill Chambers. **Spark : The Definitive Guide**. O'Reilly Media, 2018.

[173] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, Ion Stoica. **Discretized streams : an efficient and fault-tolerant model for stream processing on large clusters**. In : *USENIX Hot Topics in Cloud Computing*. 2012.

Index des termes

*****

C

CAP theorem 16
Connascence 14

D

Data lake 4, 5, 16, 26, 27, 139
Data warehouse 4, 25–27, 50, 51

I

Implicit 8, 75–77, 80, 88, 100–104, 137, 139

K

Kappa Architecture 21, 23

L

Lambda Architecture 8, 13, 15–17, 20, 21, 23, 25, 42, 45, 46, 136, 137

M

Matrice
- Produit de Hadamard 58, 83–85
- Produit de Khatri-Rao 59, 82, 84, 85
- Produit de Kronecker 59

P

Patron d'architecture 8, 12, 13, 46, 110, 136, 137, 139
Phantom type 75, 76, 78, 103, 137
Polystore 8, 26–28, 50–55, 87, 94, 97, 99, 104, 111, 112, 114, 137, 138

S

shapeless 75, 76, 78, 80, 102, 137
Stream processing 2, 12, 14–20, 23, 25, 26, 28, 46, 104, 137
- Apache Beam 20
- Apache Flink 20
- Apache Heron 19
- Apache Samza 20
- Apache Storm 19
- Cloud Dataflow 19
- Débit 18
- Garantie de traitement 18
  - *At-least-once* 19, 21
  - *At-most-once* 19, 21
  - *Effectively-once* 19, 21, 23
  - *Exactly-once* 19
- Kafka Streams 20, 97
- Latence 18
- MillWheel 19
- Spark Streaming 19
- Temporalité 18

Style d'architecture 12, 18, 21–24, 46, 139

T

Tenseur 8, 9, 50, 52–88, 102, 103, 127–131, 136, 137, 139, 140
- Différence 66, 71, 72, 79
- Décomposition CANDECOMP/PARAFAC 50, 54, 59–62, 72, 73, 79, 81–84, 86, 88, 89, 103, 127, 128, 130, 136, 137, 140
  - Extraction 73
  - Reconstruction 73
- Fibre 57
- Intersection 66, 69, 70, 72, 79
- Jointure naturelle 66, 70, 71, 77, 79, 80, 82, 102
- Matricisation 57, 58
- Produit mode-n 58
- Produit tensoriel 57, 83

Projection 66, 67, 77, 79, 80, 102
Restriction 66, 68, 80
Sélection 66, 67, 80, 82
Tranches 57
Union 66, 68, 69, 79, 80
Théorie des catégories 8, 15, 22, 29, 31–33, 35, 42, 46, 136, 139
Catégorie 22, 29–40, 42, 46
Foncteur 15, 22, 29–31, 33–39, 41, 42, 46, 139
Morphisme 15, 22, 29–39, 41, 42
Morphisme identité 30, 33, 38, 39, 42
Power set 31, 32, 34, 36, 37
Produit de catégories 31–33, 35–37, 39, 42
Préordre 29, 31–33, 35

Index des auteurs

*****

A

Abadi, Martín 53, 61
Abou Jamra, Hiba 133
Abowd, Gregory D 12, 21
Adamic, Lada 110
Agha, Zuha 26
Agrawal, Ashvin 51
Ahmad, Subutai 26
Ahmed, Nawaaz 83
Akidau, Tyler 18–20
Al-Rfou, Rami 53
Alain, Guillaume 53
Albert, Réka 110
Alexander, Andrew L 52
Alexandrov, Alexander 20
Ali, Anwaar 3
Allen, Robert 12, 21
Almahairi, Amjad 53
Alotaibi, Rana 26, 51, 54
Amann, Bernd 139
Amarasinghe, Saman 61
Amari, Shunichi 56, 58, 83, 86
Amin, Nada 77
Anandkumar, Anima 52, 53, 62
Angermueller, Christof 53
Antiga, Luca 53, 61
Aral, Sinan 110
Araujo, Miguel 60, 127, 140
Arcand, William 51
Armbrust, Michael 56
Astikainen, Piia 60
Atikoglu, Berk 74
Avgeriou, Paris 13

B

Bader, Brett W 52, 57, 83, 86
Bahdanau, Dzmitry 53
Bajpai, Vaibhav 93
Balikov, Alex 19
Ballas, Nicolas 53
Barabási, Albert-László 110
Barham, Paul 53, 61
Barn, Balbir S 13
Barrat, Alain 128
Basaille, Ian 7
Baskaran, Muthu 62
Bastien, Frédéric 53
Basu, Prithwish 60, 127, 140
Baveja, Sartaj Singh 14
Bayer, Justin 53
Begoli, Edmon 20
Bekiroğlu, Kaya 19
Belikov, Anatoly 53
Bellatreche, Ladjel 26
Belopolsky, Alexander 53
Bergeron, Bill 51
Bergmann, Rico 20
Bergstrom, Carl T 121
Bestor, David 51
Bhagat, Nikunj 19
Biboudis, Aggelos 88, 139
Bientinesi, Paolo 61
Blanco, Zachary 62, 86
Blanvillain, Olivier 88, 139
Blondel, Vincent D 121
Bondiombouy, Carlyna 54
Bonelli, G. 110

Bosagh Zadeh, Reza 83
Boykin, Oscar 17
Brachotte, Gilles 141
Bradbury, James 53, 61
Bradley, Joseph K 56
Bradshaw, Robert 19
Brewer, Devon 110
Brewer, Eric 16
Brin, Sergey 121
Bro, Rasmus 88, 127, 140
Brown, William H 6
Broy, Manfred 14, 21
Bursztyn, Damian 26, 51
Byun, Chansup 51

C

Candan, Selçuk 55
Carbone, Paris 20
Cardie, Claire 124
Cattuto, Ciro 60, 128
Cautis, Bogdan 54
Cengarle, María Victoria 21
Cetintemel, Ugur 5, 51
Chambers, Bill 53
Chambers, Craig 19
Chanan, Gregory 53, 61
Chang, Ray M 110
Chatterjee, Rony 51
Chavalarias, David 3
Chechik, Marsha 22
Chen, Jianmin 53, 61
Chen, Peinan 26, 51, 54
Chen, Tongfei 53
Chen, Yiru 51
Chen, Zhifeng 53, 61
Chernyak, Slava 18, 19
Chi, Eric C 86
Chou, Stephen 61
Christakis, Nicholas 110
Cichocki, Andrzej 56, 58, 60, 83, 86
Cioffi-Revilla, C. 110
Clark, Tony 13
Clements, Paul C 21
Coakley, Kevin 55
Colbert, S Chris 53, 62
Cong, Fengyu 60
Conte, R. 110
Contractor, Noshir 110
Craig, Iain D 13
Crawley, Michael J 56
Crowcroft, Jon 3
Cui, Bin 51
Cullot, Nadine 28, 47, 89, 107, 131, 133, 141
Cunningham, Robert K 129
Curino, Carlo 51
Cuzzocrea, Alfredo 26

D

da Silva, Anderson 56
da Silva, Daniel 56
Dai, Jennifer 14, 17
Darmont, Jérôme 27
Das, Tathagata 19
Dasgupta, Subhasis 55
Davis, Andy 53, 61
De Lathauwer, Lieven 59
Deacon, John 13
Dean, Jeffrey 53, 61
Deffuant, G. 110
Dehnavi, Maryam Mehri 62, 86
Deutsch, Alin 26, 51, 54
Devin, Matthieu 53, 61
Diskin, Zinovy 22
Dixon, James 26
Donham, Jake 19
Dugan, Jon 14
Duggan, Jennie 26, 51, 54
Dziedzic, Adam 25

E

Eckhardt, Jonas 13, 21
Edwards, Lauren 51
Eilenberg, Samuel 22, 29
Ekstedt, Mathias 14, 21
Elmore, Aaron 25, 26, 51, 54
Ewen, Stephan 20
Ezick, James 62

F

Faloutsos, Christos 52, 59, 60, 62, 86, 127, 140
Fernandez, Raul Castro 25, 51
Fernández-Moctezuma, Rafael J 19
Field, Aaron S 52
Flache, A. 110
Floratou, Avrilia 21, 51
Fong, Brendan 29
Foote, Brian 12
Ford, Neal 5, 12, 18

Fowler, James 110
Frachtenberg, Eitan 74
Frame, Alexander 141
Franklin, Michael J 56
Freytag, Johann-Christoph 20
Fried, David J 129
Fu, Maosong 19
Fu, Xiao 59

G

Gade, Krishna 19
Gadepally, Vijay 26, 51, 52, 54, 56
Garillot, Francois 17
Garlan, David 12, 21
Gaumont, Noé 3
Gauvin, Laetitia 60, 128
Gawlick, Dieter 51, 52
Gemmetto, Valerio 128
Ghemawat, Sanjay 53, 61
Ghodsi, Ali 56
Gilbert, N. 110
Gillet, Annabelle 28, 47, 89, 107, 124, 131, 133, 141
Gimelshein, Natalia 53, 61
Goharian, Nazli 83
Gomes-Jr, Luiz 139
Gong, Xiao-Feng 60
Gonzalez, Joseph E 5, 6, 75
Gonçalves, Rui 26, 51, 52, 54
Gopalakrishna, Kishore 14, 17
Gowdal, Neha 51
Graf, Isaac 129
Grison, Thierry 89
Gross, Sam 53, 61
Gubanov, Michael 55
Gudibanda, Aditya 62
Guerra, Pedro Calais 124
Guillaume, Jean-Loup 121
Gujral, Ekta 62, 86
Gupta, Amarnath 55
Gupta, Himanshu 14, 16
Gupta, Munish 96
Gurnell, Dave 75, 102
Gutmann, Myron 110
Guyot, Alexis 124, 133
Günnemann, Stephan 59, 60, 127, 140

H

Haberman, Josh 19
Hai, Rihan 139
Hai-yan, Chen 21
Hao, Fei 55
Haridi, Seif 20
Harrison, Neil B 13
Harshman, Richard A 59, 88
Hayashi, Kohei 59
Haynes, Brandon 26, 51, 54
Heise, Arvid 20
Helbing, D. 110
Hellerstein, Joseph M 5, 6, 75
Helskyaho, Heli 51, 52
Henretty, Tom 62
Hillar, Christopher J 127
Holubová, Irena 51
Howe, Bill 55, 56
Huai, Yin 56
Huang, Kejun 59
Hubbell, Matthew 51
Hueske, Fabian 20
Huo-wang, Chen 21
Hutchison, Dylan 52, 55, 56
Hyde, Julian 20

I

Im, Jean-François 14, 17
Inmon, William H 26, 27
Interlandi, Matteo 51
Irving, Geoffrey 53, 61
Isard, Michael 53, 61

J

Jackson, Jason 19
Jacobson, Ivar 14, 21
Jain, Ankit 83
Jananthan, Hayden 52, 56
Jarke, Matthias 139
Jeon, Byungsoo 62, 86
Jeon, Inah 60, 62, 86
Ji, Wang 21
Jiang, Song 74
Jiang, Xiaotian 14, 17
Jiménez-Peris, Ricardo 26, 51, 52, 54
Jindal, Alekh 51
Johnson, Pontus 14, 21
Joseph, Anthony D 5, 6, 75

K

Kaftan, Tomer 56
Kamil, Shoaib 61

Kanellakis, Paris C 66
Kang, U 60, 62, 86
Kao, Odej 20
Karanasos, Kostantinos 51
Karlsson, Lars 61
Kassela, Evdokia 21
Katsifodimos, Asterios 20
Kauffman, Robert J 110
Kawarabayashi, Ken-ichi 59
Kedigehalli, Vikas 19
Kellogg, Christopher 19
Kendall, Kris R 129
Kepner, Jeremy 26, 51, 52, 54, 56
Kertesz, J. 110
Khine, Pwint Phyu 4
Kibata, Tokio 27
Kiers, Henk AL 88, 127, 140
Killeen, Trevor 53, 61
Kim, Mijung 55
Kim, Miryung 92
Kim, Suna 52, 56
Kiran, Mariam 14
Kjolstad, Fredrik 61
Kleinberg, Jon M 121
Kleinberg, Robert 124
Kleppmann, Martin 16, 20
Knight, Kathryn 20
Knowles, Kenneth 20
Kokar, Mieczyslaw M 22
Kolda, Tamara G 52, 57, 83, 86
Kolev, Boyan 26, 51, 52, 54
Konstantinou, Ioannis 21
Koshy, Joel 25, 51
Kossaifi, Jean 53, 62
Koutra, Danai 60, 127, 140
Koziris, Nectarios 21
Kranas, Pavlos 26, 51, 52, 54
Kreps, Jay 20, 21, 25, 51, 96, 97
Krishnan, Subru 51
Kroth, Brian 51
Kuang, Li-Dan 60
Kuang, Liwei 55
Kulkarni, Sanjeev 19
Kwon, YoungOk 110
Kühlewind, Mirja 93

L

Lambiotte, Renaud 121
Lampson, Butler W 12, 15
Laney, Doug 3
Latapy, Matthieu 121
Latrache, Moustafa 54
Lavin, Alexander 26
Lax, Reuven 18, 19
Lazar, Mariana 52
Lazer, David 110
Le Métayer, Daniel 22
Leclercq, Éric 28, 47, 89, 107, 124, 131, 133, 141
Lee, Chung-Ho 14, 17
Lee, Doris 5, 6, 75
Lee, Jee Eun 52
Lee, Jungwoo 62, 86
Lee, Seunghyun 14, 17
Lefebvre, Etienne 121
Leich, Marcus 20
Lemus, Noel 56
Lerer, Adam 53, 61
Leser, Ulf 20
Lethin, Richard 62
Levchenko, Oleksandra 26, 51, 52, 54
Li, Haoyuan 19
Li, Jiajia 61
Li, Jialiang 14, 17
Li, Xupeng 51
Lian, Cheng 56
Lim, Lek-Heng 127
Lin, Chi-Yi 14, 17
Lin, Dong 25, 51
Lin, Jimmy 5, 16, 17
Lin, Man 55
Lin, Qiu-Hua 60
Lin, Zeming 53, 61
Lippmann, Richard 129
Liu, Bangtian 62, 86
Liu, Davies 56
Liu, Fengyun 88, 139
Liu, Zhen Hua 51, 52
Loreto, V. 110
Lu, Haw-minn 120
Lu, Jiaheng 51, 52
Lugato, David 61
Luo, Changqing 55
Lustosa, Hermano 56
Léauté, Xavier 14, 16

M

Ma, William 5, 6, 75
Maas, Gerard 17

Mabrok, Mohamed A 22
Maccioni, Antonio 55
Macke, Stephen 5, 6, 75
MacLane, Saunders 22, 29
Madden, Samuel 26, 51, 54
Maehara, Takanori 59
Maibaum, Tom 22
Majkić, Zoran 22
Malkis, Alexander 13, 21
Malveau, Raphael C 6
Manolescu, Ioana 26, 51, 54
Markl, Volker 20
Marmsoler, Diego 13, 21
Marz, Nathan 13, 15
Massa, Francisco 53, 61
Mateev, Nikolay 83
Mattson, Tim 26, 51, 54
McKinney, Wes 5, 56
McVeety, Sam 19
Meehan, John 25
Meira Jr, Wagner 124
Meng, Xiangrui 56, 83
Merlino, Gian 14, 16
Meyer, Bertrand 6, 92
Michaleas, Peter 51
Milechin, Lauren 52
Miller, Heather 88, 139
Mills, Daniel 19
Min, Geyong 55
Mittal, Sailesh 19
Mo, Xiangxi 5, 6, 75
Moat, S. 110
Mohamed, Yasser Abdel-Rady I 3, 14, 16
Monga, Inder 14
Moors, Adriaan 76
Motwani, Rajeev 121
Mowbray, Thomas J 6
Mullen, Julie 51
Munshi, Amr A 3, 14, 16
Murphy, Peter 14

N

Nadal, J. -P. 110
Namiot, Dmitry 13
Narkhede, Neha 25, 51, 96
Niedrite, Laila 21
Nordstrom, Paul 19
Nowak, A. 110

O

O'Connell, Ian 17
Odersky, Martin 76, 77, 88, 92, 94, 139
Oliveira, Bruno CdS 76
Ott, Jörg 93
Ozsu, M Tamer 93

P

Pacitti, Esther 26, 51, 52, 54
Page, Lawrence 121
Page-Jones, Meilir 14
Paleczny, Mike 74
Panagakis, Yannis 53, 62
Panahi, Maziyar 3
Panisson, André 60, 128
Pantic, Maja 53, 62
Papadimitriou, Spiros 60, 127, 140
Papalexakis, Evangelos E 52, 59, 60, 62, 86, 127, 140
Parameswaran, Aditya 5, 6, 75
Park, Namyong 62, 86
Pasricha, Ravdeep 62, 86
Paszke, Adam 53, 61
Patel, Jignesh M 19
Pau, Raquel 54
Pawar, Neha 14, 17
Pentland, Alex 110
Pereira, José 54
Perrone, Giancarlo 120
Perry, Frances 19
Petersohn, Devin 5, 6, 75
Phan, Anh Huy 56, 58, 60, 83, 86
Pietzuch, Peter R 25, 51
Pingali, Keshav 83
Podkorytov, Maksim 55
Pogossiants, Gregory 51, 52
Pons, Pascal 121
Porto, Fábio 56
Provatas, Nikodimos 21
Psarras, Christos 61
Pu, Li 83
Purdy, Scott 26

Q

Qadir, Junaid 3
Quix, Christoph 139

R

Rabanser, Stephan 59
Ramasamy, Karthik 19

Rao, Jun 25, 51, 96
Rasool, Raihan ur 3
Ray, Nelson 14, 16
Reynolds, John C 88, 139
Riccomini, Chris 25, 51
Richards, Mark 5, 12, 18
Ristaniemi, Tapani 60
Ritchie, Sam 17
Rompf, Tiark 77
Rosvall, Martin 121
Rush, Alexander 52, 53
Ryan, Michael J 22

S

Salay, Rick 22
Samsi, Siddharth 52
San Miguel, M. 110
Sanchez, A. 110
Santanchè, André 139
Sathiaseelan, Arjuna 3
Savonnet, Marinette 89, 133, 141
Sawadogo, Pegdwendé 27
Schmidt, Eric 19
Schönwälder, Jürgen 93
Shchur, Oleksandr 59
Shenker, Scott 19
Shrivastava, Mayank 14, 17
Shukla, Amit 19
Sidiropoulos, Nicholas D 52, 59, 62
Simitsis, Alkis 25, 51
Sneps-Sneppe, Manfred 13
Solodovnikova, Darja 21
Song, Il-Yeol 26
Sparks, Evan 83
Sperotto, Anna 93
Spivak, David I 29
Spoon, Lex 76, 92, 94
Springer, Paul 61
Staple, Aaron 83
Stodghill, Paul 83
Stoica, Ion 19
Stonebraker, Michael 5, 16, 26, 51, 54
Strüber, Daniel 22
Stucki, Sandro 88, 139
Su, Tong 61
Subramaniam, Subbu 14, 17
Suciu, Dan 55, 56
Sun, Jimeng 52
Sun, Qian 83
Swami, Ananthram 60, 127, 140

T

Taentzer, Gabriele 22
Taneja, Siddarth 19
Tao, Dacheng 52
Tatbul, Nesime 25
Team, PD 56
Tian, Shaobo 25
Tian, Yulong 25
Tichavsky, Petr 60
Tomasik, Jerzy A 22
Toon, K 13
Torlone, Riccardo 55
Toshniwal, Ankit 19
Toth, David 22
Trammell, Brian 93
Tschetter, Eric 14, 16
Tucker, Ledyard R 88
Tumbde, Adwait 14, 17
Tzoumas, Kostas 20

U

Ulanov, Alexander 83
Unpingco, Jose 120

V

Valduriez, Patrick 26, 51, 52, 54, 56, 93
Van Der Walt, Stefan 53, 62
Varoquaux, Gael 53, 62
Vassiliadis, Panos 25, 51
Venkataraman, Shivaram 83
Venners, Bill 76, 92, 94
Vilaça, Ricardo 26, 51, 52, 54

W

Wang, Guozhang 25, 51
Wang, Zhao Shun 4
Warren, James 15
Webster, Seth E 129
Wei, Dong 21
Weyman, Jerzy 22
Whittle, Sam 19
Winograd, Terry 121
Wu, Wentao 51
Wu, Zhe 51, 52

X

Xin, Doris 5, 6, 75

Xin, Reynold S 56
Xu, Yuehai 74

Y

Yang, Fangjin 14, 16
Yang, Laurence T 55
Yang, Yifei 54
Yavuz, Burak 83
Yoder, Joseph 12
Yu, Rose 52
Yue, Yisong 52

Z

Zaharia, Matei 19, 53, 83
Zampetakis, Stamatis 26, 51
Zdonik, Stan 25
Zdunek, Rafal 56, 58, 83, 86
Zhang, Ce 51
Zheng, Stephan 52
Zhou, Ziqi 52, 56
Zissman, Marc A 129
Zwitter, Andrej 3

Résumé

Titre : Modélisation et développement d'un observatoire générique pour la collecte et l'analyse de données massives
Mots-clés : Données massives, Architectures logicielles, Théorie des catégories, Tenseurs, *Stream processing*, Modèles de données

Les données massives fascinent, aussi bien grâce à la valeur qu'elles recèlent pouvant apporter un avantage significatif lors de la prise de décision, qu'à cause des défis que leur exploitation représente. Ces défis sont présents à plusieurs niveaux de la chaîne d'analyse des données. Au niveau de la création des architectures logicielles, le volume et la vélocité requièrent au minimum des performances suffisantes pour ingérer et stocker les données. La variété des données a aussi un impact, puisqu'une multitude de nouveaux systèmes de stockage ont vu le jour, chacun correspondant à un besoin spécifique. Les *polystores* sont des systèmes intégrant cette diversité, afin de gagner en flexibilité par rapport aux *data warehouses*, désormais trop rigides. Cette diversification vient toutefois avec un coût, celui de la difficulté à prendre en charge les différents modèles de données lors des analyses.

Cette thèse se place dans ce contexte, en proposant la Lambda+ Architecture, un patron d'architecture qui améliore la Lambda Architecture pour l'adapter aux données massives et supporter simultanément l'exactitude des traitements et les calculs en temps réel. La théorie des catégories sert de base formelle pour étudier la conservation des propriétés et ouvre de nouvelles perspectives pour les architectures logicielles qui reposent sur des compositions de composants. La seconde contribution est le Tensor Data Model, un modèle pivot agissant comme une surcouche aux *polystores*. Basé sur les tenseurs, il leur ajoute la notion de schéma, afin de bénéficier d'opérateurs de manipulation de données en plus des opérateurs tensoriels, ainsi que d'un système de sûreté du typage et d'inférence de schéma, en plus de performances satisfaisantes. Chacune de ces contributions bénéficie d'une implémentation, et elles sont regroupées dans un observatoire visant à analyser des données sociales issues de Twitter et à mettre les résultats à disposition d'experts métier.

Abstract

Title: Modelling and development of a generic observatory to harvest and analyze Big Data
Keywords: Big Data, Software Architectures, Category Theory, Tensors, Stream processing, Data models

Big Data fascinate, both because of the value they hold that can provide a significant advantage in decision-making, and because of the challenges that their exploitation represents. These challenges are present at several levels of analytics workflows. At the level of the creation of software architectures, the volume and the velocity require at least enough performance to handle the ingestion and storage of data. The data variety has also an impact, as several new storage systems have emerged, each one corresponding to a specific need. The polystores are systems that integrate this diversity, to gain flexibility compared to the data warehouses, now too rigid. However, this diversification comes at a cost, that of the difficulty of taking into consideration the various data models in analyzes.

This thesis is placed in this context, and proposes the Lambda+ Architecture, a architecture pattern that improves the Lambda Architecture to make it suitable for processing of Big Data while supporting simultaneously the correctness and the real-time properties. The category theory is used as formal basis to study the conservation of properties and opens new perspectives for software architectures that rely on compositions of components. The second contribution is the Tensor Data Model, a pivot model that act as an overlay to polystores. Based on tensors, it adds the notion of schema to them, to benefit from data manipulation operators on top of tensorial operators, as well from a strong type safety and schema inference systems, with good performance. Each one of these contributions benefit from an implementation, and the are gathered into an observatory that aims to analyze social data from Twitter and to make the results available for business experts.