Évolution et formalisation de la Lambda Architecture pour des analyses à hautes performances - Application aux données de Twitter

Evolution and formalization of the Lambda Architecture for high performance analytics - Application to Twitter data

Annabelle Gillet, Éric Leclercq, and Nadine Cullot

Laboratoire d'Informatique de Bourgogne - EA 7534, Université de Bourgogne Franche-Comté, Dijon, France, annabelle.gillet@depinfo.u-bourgogne.fr, eric.leclercq@u-bourgogne.fr, nadine.cullot@u-bourgogne.fr

RÉSUMÉ. Extraire de la valeur des données des réseaux sociaux est une tâche complexe induite par leur vélocité, volume et variabilité. Les utilisateurs s'approprient le dispositif et développent des usages multiples, ce qui renforce la variabilité sémantique. Les résultats des analyses doivent être produits au plus tôt (de manière optimale en temps réel) pour en renforcer la pertinence. Pour y parvenir, des connaissances métiers sont essentielles et elle sont généralement acquises lors d'analyses exploratoires. En conséquence, les plateformes de collecte, stockage et analyse des données des réseaux sociaux doivent supporter des flux de données importants, des analyses en temps réel et des analyses exploratoires. Des styles et des patrons d'architecture permettent de prendre en compte ces spécificités, afin de proposer des techniques de prise en charge de ces données, et ainsi de faciliter leur traitement. Ces architectures ont besoin d'être formalisées, pour étudier de quelle manière les propriétés essentielles sont respectées, connaître leur comportement, et anticiper les effets que peuvent avoir les composants lorsqu'ils sont regroupés au sein d'une même architecture, et ce avant même de les développer puis de les mettre en production. Dans cet article, nous proposons un patron d'architecture, la Lambda+ Architecture, inspiré de la Lambda Architecture et adapté au traitement des données massives. Nous proposons également un cadre formel pour la spécification d'architectures se basant sur la théorie des catégories, ainsi qu'une implémentation de notre patron pour analyser les données issues de Twitter.

ABSTRACT. Extracting value from social network data is a task whose complexity is driven by speed, volume and variability of data. Users develop multiple uses of these systems, that enhance the semantic variability. Analytics results must be produce as soon as possible (optimally in real-time) to be more relevant. Thus, business knowledge is essential and can usually be acquired by doing exploratory analysis. Accordingly, systems that harvest, store and analyze data from social networks have to support important streams of data, real-time analysis and exploratory analysis. Architecture styles and pattern allow to take these specificities into consideration, by proposing techniques to handle those data, and thus to facilitate their processing. These architectures have to be formalized, to study if essential properties are fulfilled, to know their behaviour, and to anticipate the effects that components can have on other components when they are gathered into a same architecture, and this even before developing and putting in production the architecture. In this article, we propose an architecture pattern, the Lambda+ Architecture, inspired from the Lambda Architecture and adapted to the processing of Big Data. We propose a formalization for architectures based on category theory, and an implementation of our pattern to analyze Twitter data.

MOTS-CLÉS. Patron d'architecture, Théorie des catégories, Lambda Architecture

KEYWORDS. Architecture pattern, Category theory, Lambda Architecture

1. Introduction

Les réseaux sociaux numériques (RSN) produisent une masse de données importante, et ce de manière continue. À titre d'exemple, 6 000 tweets sont produits sur Twitter chaque seconde. L'exploitation de ces données peut apporter une importante plus-value dans différents domaines comme le marketing, la politique ou les sciences sociales. Extraire de la valeur ou des connaissances des données issues des RSN sont des tâches complexes car la qualité des données est variable, difficile à évaluer, les usages sont multiples et ils induisent une grande variabilité sémantique et une grande variété des contenus. Par exemple, l'utilisation des hashtags peut porter plusieurs significations distinctes, dépendantes de

l'intention de l'émetteur. Parmi ces significations, on peut retrouver le fait de vouloir catégoriser un tweet, ou encore de viser une certaine communauté afin d'étendre la discussion à l'intérieur de cette communauté [ZM18, Gia18].

Pour arriver à extraire de la valeur des données des RSN, et en fonction des observations à réaliser ou des questions posées, plusieurs algorithmes sont mis en œuvre. Ils peuvent se baser sur différentes modélisations des données : on peut vouloir étudier le comportement d'un utilisateur au cours du temps, auquel cas une modélisation en série temporelle est adaptée, on peut vouloir étudier le regroupement d'utilisateurs en communautés, ainsi que leur organisation et leurs interactions, dans ce cas une modélisation en graphe est nécessaire. Les différents algorithmes peuvent être appliqués de manière isolée ou de manière successive, organisés en *workflows* d'analyse [AJGSL20], dans le but de fournir des indications à différents niveaux de granularité (macroscopique, mésoscopique et microscopique). Certains indicateurs extraits de la masse de données possèdent une plus grande valeur lorsqu'ils sont identifiés dans un délai court suivant leur production. C'est par exemple le cas pour la détection de fraudes bancaires [AKA14] ou de détection d'anomalies [ALPA17]. Pour cela, les données ont besoin d'être traitées en temps réel, principalement à l'aide des systèmes de *stream processing*, et d'adapter les algorithmes à ce paradigme de programmation particulier.

1.1. *Le besoin d'architectures logicielles*

Pour prendre en compte les spécificités des données à traiter (diversité, vélocité, etc.), les besoins en terme d'analyse et leurs évolutions, il est nécessaire de concevoir des architectures logicielles ayant des propriétés correspondant aux exigences. Supporter à la fois des analyses exploratoires et des analyses en temps réel complexifie significativement la construction de telles architectures. De plus, elle doivent supporter différents paradigmes de programmation pour implanter la diversité des algorithmes qui serviront à révéler la valeur des données.

La définition et le développement de telles architectures est une tâche complexe, qui nécessite de connaître les propriétés que le système doit avoir (par exemple la capacité de passage à l'échelle) et de définir les composants ainsi que leurs interactions [Lam83]. 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* [FY97], qui rend extrêmement complexe toute modification. Pour éviter cette situation, les styles et les patrons d'architecture permettent de guider la définition d'une architecture.

Les styles sont des spécifications à niveau de granularité élevé, guidant les interactions entre les composants [AAG95]. 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 particulière. Certains styles sont bien connus, comme par exemple l'architecture en couches (*layered architecture*) [MME15] 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 et avec un coût 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 [NSS14]. 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*) [CB11]. 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 permettent de définir une abstraction spécifique d'un style d'architecture, pour obtenir des propriétés essentielles [HA07]. 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 [Cra88] propose de mettre les données à disposition des processus dans un répertoire commun, afin de pouvoir accéder et utiliser les données pour des calculs, en mettant ensuite à disposition dans ce même répertoire les résultats obtenus. Ce patron simule l'interaction d'experts travaillant conjointement à la résolution d'un problème sur un tableau commun. Un autre patron est le Modèle-Vue-Contrôleur [Dea09], 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. La Lambda Architecture [Mar11] est aussi un patron d'architecture, bien connu dans le contexte Big Data, 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 patron proposé dans cet article reprend les idées clés de la Lambda Architecture (voir section 2). Il intègre les capacités actuelles des systèmes de *stream processing*, tout en conservant les propriétés essentielles de tolérance aux pannes et de traitements en temps réel, et en contournant les limites de la Lambda, principalement induites avec la duplication des traitements, qui compliquent les évolutions et les opérations de maintenance.

La plupart des articles de recherche sur les architectures logicielles visant à traiter des Big Data présentent des architectures spécifiques, ancrées dans leur contexte d'utilisation, et avec un ensemble de technologies prédéfinies [YMR+17, KMM+15, MM18, LL17, IGS+18]. Cela mène à une forte *connascence* entre l'architecture, les outils et les propriétés. La *connascence* est un terme introduit par Page dans [PJ92], il mesure ou traduit la complexité induite par les relations de dépendance des composants entre eux. Plusieurs types de *connascences* statiques ou dynamiques ont été identifiés pour évaluer les architectures.

Les architectures sont des entités complexes, qui peuvent évoluer, subir des modifications, être fusionnées avec d'autres architectures, etc. Les imposantes architectures distribuées peuvent contenir des composants ayant leur propre style ou patron, formant ainsi une composition de plusieurs petites architectures. Il existe donc un besoin de formalisation important dans le monde des architectures [Bro11, JEJ12], formalisation qui doit pouvoir retranscrire les effets d'une composition de composants, leurs impacts sur les propriétés, anticiper les conséquences d'une évolution.

1.2. *Contribution*

Nous proposons 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. Nous présentons une implémentation du patron Lambda+ Architecture à travers Hydre, une plateforme servant à collecter, stocker et analyser les données issues de Twitter. Nous proposons un cadre pour la formalisation d'architectures se basant 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. C'est une approche prometteuse pour répondre à ces besoins et 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.

Dans la suite de l'article, nous présentons en section 2 la Lambda Architecture, le contexte de sa création, ses apports et ses limites, ainsi que des implémentations et des adaptations de la Lambda Architecture, et nous montrons au travers de ces descriptions que ce patron ne répond pas à toutes les attentes et ne peut pas être considéré comme un patron global d'architecture, mais seulement comme un moyen de compenser les faiblesses d'une technologie naissante. Dans cette section, nous présentons également les principales approches de formalisation de la description des architectures et nous motivons l'utilisation de la théorie des catégories. La section 3 est consacrée à la description du patron que nous proposons : la Lambda+ Architecture. La section 4 décrit son implémentation et utilisation dans un projet de recherche pluri-disciplinaire. La section 5 détaille un *framework* de formalisation d'architectures au moyen de la théorie des catégories. La section 6 dresse un bilan des propositions et présente des perspectives de recherche à court et moyen termes.

2. Lambda Architecture et travaux connexes

Cette section décrit les principes de la Lambda Architecture, des exemples d'implémentations, l'environnement technologique de sa définition et synthétise ses limites.

2.1. *Principe de la Lambda Architecture*

La Lambda Architecture est un patron d'architecture logicielle décrit par Nathan Marz [Mar11], qu'il a plus tard approfondi dans un ouvrage plus complet [MW15]. Ce patron d'architecture vise à traiter les données massives en temps réel, tout en assurant un résultat exact à l'issue de ces traitements. Pour cela, la Lambda est composée de trois couches (*layers*) : la *Batch layer*, la *Serving layer* et la *Speed layer* (figure 1). Le principe général est le suivant : la *Batch layer* permet de fournir périodiquement une vue exacte sur les données, mais avec un temps de traitement ou de transformation important. La *Serving layer* met ensuite ces vues à disposition des utilisateurs, avec pour objectif d'optimiser l'accès aux données. La *Speed layer* sert à compenser le temps de traitement de la *Batch layer*, en proposant un traitement des données en temps réel, mais sans garantir l'exactitude des résultats.

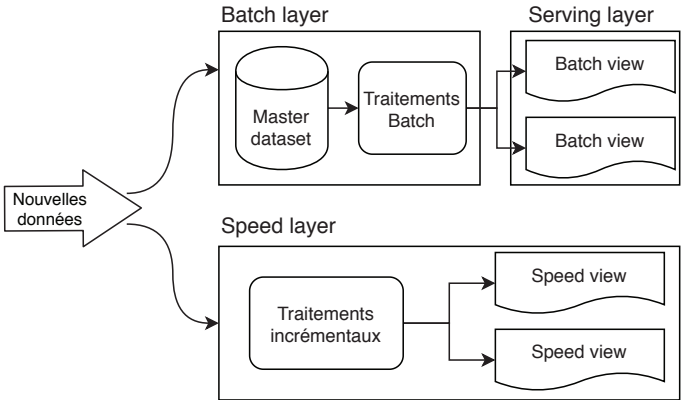

Figure 1. *Principe général de la Lambda Architecture*

La *Batch layer* a deux rôles principaux : 1) stocker les nouvelles données sous un format brut, et 2) réaliser des calculs, des transformations de modèles sur ces données. Le stockage des données brutes ne se fait pas sous la forme d'états mais sous la forme d'une suite d'événements ou de messages qui permettent d'obtenir un état actualisé des différents éléments lorsque ces évènements sont traités dans leur ensemble. La *Batch layer* stocke toutes les données qu'elle reçoit dans le *master dataset*, c'est-à-dire un entrepôt permanent de données brutes. Les calculs et les transformations de modèles sont opérés à partir du *master dataset*, souvent de manière distribuée afin de garantir le passage à l'échelle.

La *Serving layer* se charge ensuite de récupérer les résultats de ces traitements, et les met à la disposition des utilisateurs. Le but étant d'optimiser le temps d'accès, la modélisation des données est réalisée dans ce sens (utilisation des index, schéma non normalisé, etc.).

La *Speed layer* s'occupe de traiter les données en temps réel, au moyen d'algorithmes qui travaillent de manière incrémentale, souvent réalisés avec des technologies de *stream processing*. Elle rend disponible les nouvelles données qui n'ont pas encore été traitées par la *Batch layer*. La performance prime donc sur l'exactitude (ou la véracité) des traitements réalisés, cette contrainte étant laissée à la charge de la *Batch layer*.

Avec ces spécifications, les avantages de la Lambda Architecture sont une forte résistance aux pannes, la garantie d'un résultat correct avec la *Batch layer* et une faible latence avec la *Speed layer*. Elle a inspiré de nombreuses architectures, telles que RADStack [YMR+17], une plateforme *open source* produisant des analyses interactives, utilisant 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.

Munshi et al. [MM18] 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.

Lee et al. [LL17] spécifient une Lambda Architecture pour construire un système de recommandation de restaurants. Leur apport est l'utilisation d'Apache Mesos pour abstraire les composants matériels de la plateforme, faciliter son déploiement et sa mise à l'échelle. Les technologies utilisées sont Kafka pour la *Speed Layer* et les traitements temps réel, Hadoop HDFS pour le stockage des données brutes, et Spark pour la *Batch Layer*.

Le système Pinot a été développé à LinkedIn [IGS+18]. Il a été 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 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.

Kiran et al. [KMM+15] s'intéressent aux performances que peut offrir la Lambda Architecture, afin de minimiser les coûts qui peuvent être associés au déploiement des applications en mode *cloud*. Leur approche est mise en pratique avec le déploiement de leur architecture sur Amazon AWS, afin d'analyser des données issues de capteurs.

La Lambda Architecture est un patron reconnu pour réaliser des traitements sur les Big Data. Cependant, sa définition informelle nécessite souvent une interprétation, et elle est donc plus proche d'un principe que d'un patron d'architecture. Bien que la résistance aux pannes soit une caractéristique avérée de la Lambda, le temps réel et l'exactitude 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, un historique de la création de la Lambda Architecture est nécessaire.

2.2. *Émergence de la Lambda Architecture*

Lorsque Marz a présenté la Lambda en 2011, les systèmes de *stream processing*, nécessaires à l'élaboration de la couche *Speed*, n'en étaient qu'à leurs balbutiements [ACL18], y compris Apache Storm porté par Marz [TTS+14, Mar14]. D'autres acteurs principaux du domaine ont ensuite contribué avec différents systèmes, dont Spark Streaming [ZDL+12] en 2012, qui s'appuie sur le moteur Spark afin de proposer un système de *stream processing* en *micro-batch*, Apache Heron [KBF+15] conçu en 2015 par Twitter qui avait besoin d'un système plus adapté à leur cas d'utilisation, ayant de meilleures performances et pouvant passer à l'échelle plus facilement que Storm. MillWheel [ABB+13], Apache Flink [CKE+15], Dataflow [ABC+15] sont d'autres systèmes de *stream processing* qui ont été développés dans cette période.

À leurs débuts, ces systèmes se concentraient principalement sur leurs performances, en délaissant la propriété d'exactitude de calcul. De ce fait, deux garanties de traitement étaient disponibles : la garantie *at-most-once*, impliquant que les données n'étaient traitées qu'une seule fois maximum, mais qu'elles pouvaient donc ne pas être traitées ; et la garantie *at-least-once*, impliquant que les données étaient traitées au moins une fois, mais qu'elles pouvaient donc être traitées plusieurs fois. En l'état, le manque de précision de la couche *Speed* pouvait seulement être compensée en corrigeant les résultats avec un traitement par lot périodique. Cependant, les systèmes de *stream processing* ont été améliorés, et une

nouvelle garantie de traitement fut ajoutée : la garantie *exactly-once*, fusionnant les deux garanties précédentes, pour ne traiter les données qu'une et une seule fois. Cette garantie est toutefois à nuancer. Elle ne concerne que les effets internes au système de *stream processing*, ce qui signifie que si un traitement a un effet de bord (comme l'écriture dans un fichier), ce traitement pourra être effectué plusieurs fois. De par ce comportement, cette garantie peut parfois être appelée *effectively-once*, qui est un terme plus précis pour la qualifier. Dans ce cas, si des effets de bord sont présents dans les traitements, ils doivent être idempotents, c'est-à-dire qu'ils produiront toujours le même résultat pour les mêmes données en entrée, même s'ils sont exécutés plusieurs fois, pour que la garantie de traitement soit applicable de manière globale.

2.3. *Discussion*

La Kappa Architecture est un autre patron qui adopte une vision différente de la Lambda Architecture. Elle peut être vue comme une simplification dans le sens où elle considère que tous les traitements opèrent sur des flux (*everything is a stream*). Comme Kreps le décrit dans son article [Kre14], plutôt que d'avoir une couche *Batch* et une couche *Speed*, il est possible de conserver uniquement une seule couche *Speed* et d'organiser les flux. Pour ce faire, les données doivent être conservées sous la forme d'une suite de messages (*logs*). La Kappa permet permet d'éviter la complexité de la Lambda, mais ne conserve pas sa capacité de résistance aux pannes, et ne permet pas non plus d'élargir les cas d'utilisation possibles et d'inclure les analyses exploratoires.

Lors de sa conception, la Lambda Architecture a servi à compenser les faiblesses d'une technologie naissante. Cependant, maintenant que les systèmes de *stream processing* sont assez matures pour corriger cette faiblesse d'eux-mêmes, l'utilité de la Lambda s'en trouve réduite. Ses points faibles principaux sont : 1) un coût et une complexité de développement élevés à cause de la duplication des traitements dans les couches *Speed* et *Batch* en utilisant des paradigmes différents, qui n'est désormais plus nécessaire, 2) la restriction des cas d'utilisation nécessitant de connaître a priori les traitements à exécuter sur les données, ce qui en fait un patron d'architecture peu adapté aux situations nécessitant des analyses exploratoires, et 3) un manque de spécifications claires de l'architecture, qui entraîne des interprétations et des variations lors de l'application du patron.

Le besoin d'une théorie et d'une formalisation pour l'élaboration des architectures logicielles a gagné en importance ces dernières années [Bro11, JEJ12]. 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. Établir un cadre formel dans ce domaine demande 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 de l'ingénierie.

Deux articles récents [KPK+19, SN20] 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. C'est le cas par exemple pour les technologies de stockage de données ou de traitement de flux.

Les *Architecture Description Languages* (ADL) [Cle96] ont une place importante dans la formalisation d'architecture. Les ADL *boxes and lines* ainsi que les ADL basés sur UML [BC11] ne permettent pas de vérifier facilement les propriétés, à cause de leur faible niveau de formalisation. Nous nous concentrons sur les ADL ayant des bases théoriques bien établies.

Abowd et al. [AAG95] propose 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.

Malkis et Marmsoler [MM15, MME15] ont travaillé sur la formalisation des styles d'architectures. Ils se sont appuyés 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.

Le Métayer [LM98] 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. 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 [MR17] 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 application 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.

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'abstractions, et les interactions entre les composants ont autant d'importance que les composants eux-mêmes. La théorie des catégories est une approche prometteuse, puisqu'elle donne de l'importance aux relations avec les morphismes et aux compositions. Elle permet également d'utiliser les foncteurs, simplifiant le passage d'un niveau d'abstraction à un autre.

3. Description de 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, les systèmes de *streaming* le supportant directement. 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 Big Data, en proposant deux fonctionnalités principales complémentaires : 1) le stockage des données orienté vers une utilisation prévue pour des analyses exploratoires, et 2) 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.

Nous proposons de définir formellement la Lambda+ Architecture. Tout d'abord, comme indiqué dans la section 2, 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, passage à l'échelle et facilité d'évolution. Ce style privilégie les communications asynchrones entre les composants, en organisant les flux de messages à l'aide de files et de topics disponibles avec les *middlewares* orientés messages. Chaque composant traite les messages, et envoie éventuellement son résultat dans une autre file 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 représente les cinq composants de la Lambda+ Architecture.

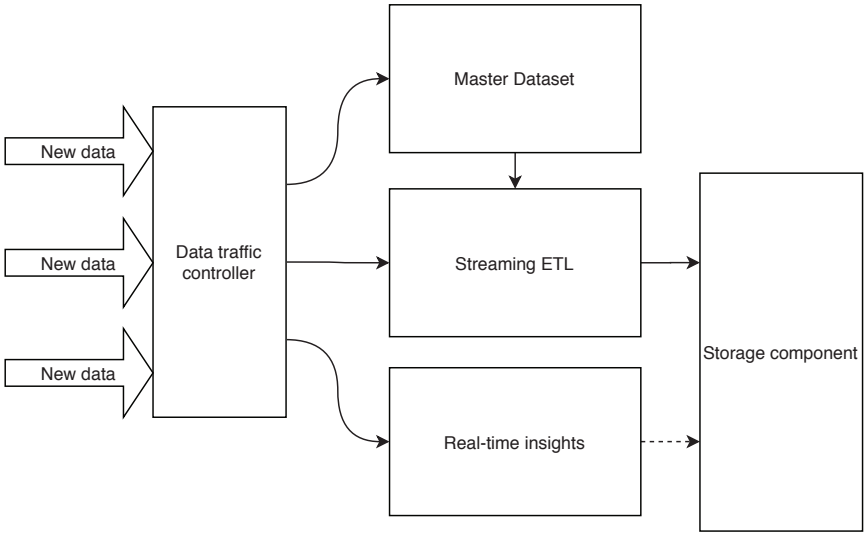

Figure 2. *Aperçu de la Lambda+ Architecture*

3.1. *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.

3.2. *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 [VS09, FPK+15, MZT+16].

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 *streaming* 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 : 1) en temps réel lorsqu'il réceptionne les données du *data traffic controller* ; et 2) en temps différé lors d'une extraction 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*.

3.3. *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.

3.4. *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 *streaming* permettent d'intégrer 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 [ALPA17].

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 de cette manière 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, 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.

3.5. *Le composant storage*

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*. 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.

Les *data warehouses* sont apparus bien avant l'avènement des Big Data, et sont une technologie mature et largement adoptés par les entreprises [Inm05]. 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 : 1) elles concernent un sujet particulier, afin de faciliter les requêtes d'informatique décisionnelle ; 2) elles sont intégrées pour éviter les inconsistances induites lors de l'utilisation de sources multiples ; 3) elles possèdent une dimension temporelle, afin d'exécuter des requêtes pour des périodes différentes ; 4) 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.

Les *polystores* sont des systèmes intégrants divers SGBD, systèmes de stockage, ainsi que plusieurs langages de manipulation de données ou des langages de programmation se basant sur différents paradigmes [GCD+16]. 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 [KLP+18, ABD+19].

Les *data lakes* sont moins bien définis que les *data warehouses* ou que les *polystores* [Dix10]. 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 marécage, 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 marécage, et sur l'accompagnement de la localisation de données pouvant servir à une analyse. Les approches les plus populaires consistent à intégrer des métadonnées dans le *data lake* [SKD19, SD20], ou à partitionner le *data lake* en fonction de divers critères (comme regrouper les mêmes formats, les mêmes sujets, etc.) en *data ponds* [Inm16]. L'organisation en *data ponds* a quelques points communs avec les *polystores*.

4. Une application de la Lambda+ : l'architecture Hydre

Nous avons mis en œuvre le patron Lambda+ pour développer une architecture utilisée dans le projet de recherche interdisciplinaire COCKTAIL [1], réunissant des chercheurs en sciences humaines et sociales, en sciences de la communication, en agroalimentaire et en informatique, ainsi que deux Entreprises des Services du Numérique régionales. Ce projet cherche à étudier les discours de l'alimentaire et de la santé dans les réseaux sociaux, et à détecter des signaux faibles.

Pour ce faire, le projet collecte en continu des données de Twitter à partir de critères (hashtags, comptes, etc.) spécifiés par les chercheurs spécialistes du domaine. Plus de 3 000 critères sont actuellement utilisés pour cette étude. Le but de cette collecte est double : 1) réaliser des analyses exploratoires pouvant faire appel à des algorithmes variés, nécessitant des données adaptées à la question de recherche soulevée par les chercheurs en sciences sociales. Ces analyses font partie d'un processus dynamique et itératif, en incluant l'interprétation du résultat des analyses grâce aux connaissances métier ; et 2) calculer des indicateurs macroscopiques en temps réel, permettant d'identifier les tendances générales de la collecte, de détecter des événements pouvant servir à amorcer les analyses exploratoires.

1. `https://projet-cocktail.fr/`

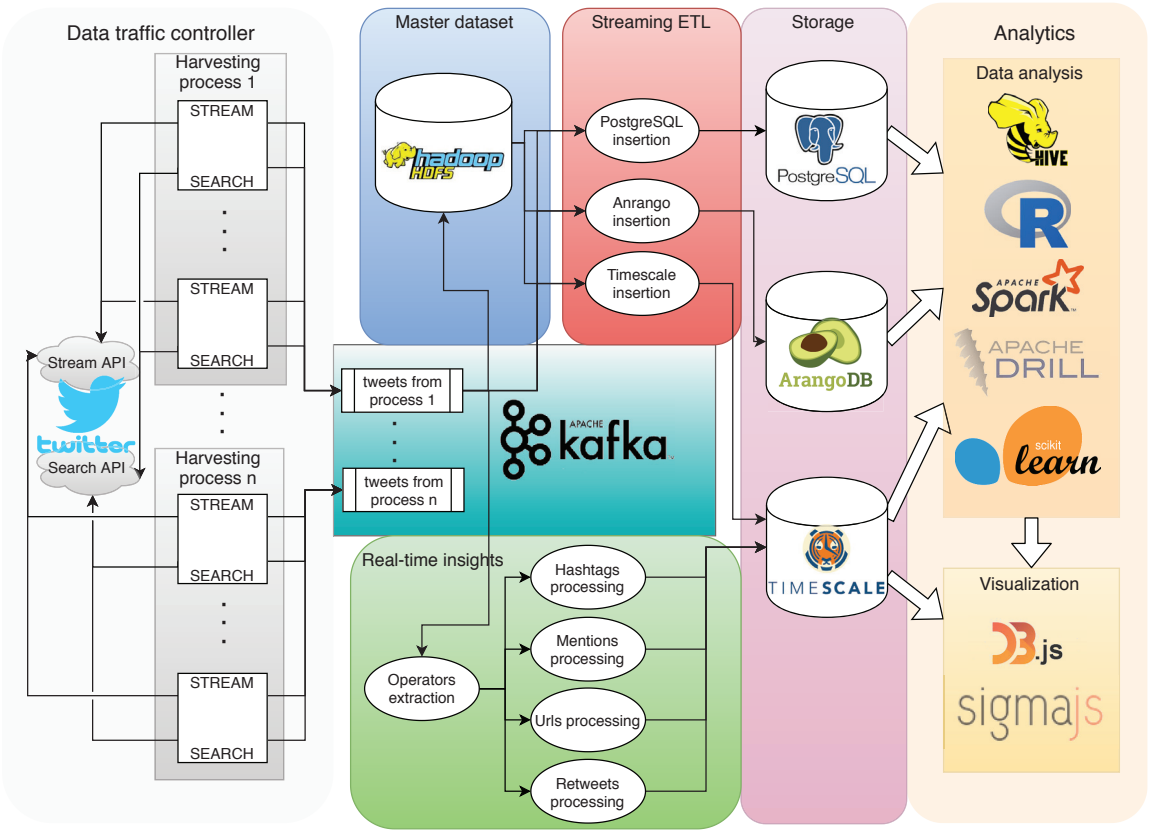

Figure 3. *L'architecture Hydre*

Afin de collecter, stocker et analyser les données issues de Twitter, l'architecture Hydre a été spécifiée et développée (figure 3). 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. L'implémentation de cette architecture a été présentée plus en détail dans un article précédent [GLC19], en incluant des mesures de performances démontrant ses capacités de passage à l'échelle, ainsi que sa capacité d'absorption du flux moyen théorique de Twitter de 6 000 tweets/s. Plusieurs améliorations ont été apportées à l'architecture par rapport à la première version développée et décrite dans [GLC19], notamment au niveau de l'ajout du composant de *streaming ETL*. Au niveau matériel, elle utilise un *cluster* Hadoop de 20 nœuds, un *cluster* Kafka de 5 nœuds et 4 autres serveurs utilisés pour 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 récupérés 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 4To.

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 réalisées, comme par exemple les hashtags ou utilisateurs populaires. Les résultats sont stockés dans le SGBD séries temporelles, 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*.

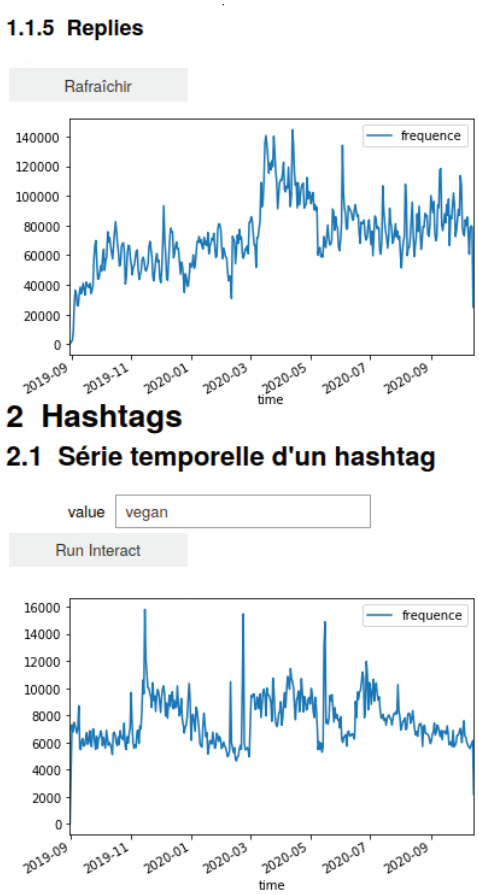

Figure 4. *Extrait d'un notebook Jupyter d'affichage des indicateurs macroscopiques de la collecte*

Les données des collectes sont mises à disposition des chercheurs à travers des interfaces exposées par un serveur d'application, ainsi que par des *notebooks* Jupyter pré-configurés (figure 4). 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 [AJGSL20, GFB+19].

5. Formalisation

La description de l'architecture Hydre fait apparaître plusieurs points qu'il est essentiel de traiter pour pouvoir implanter des architectures. Il s'agit par exemple de spécifier les exigences et les propriétés techniques attendues pour chaque composant, mais aussi pour leurs compositions, et ceci à différents niveaux de granularité. Ainsi, cette spécification doit permettre d'assurer la cohérence globale de l'architecture définie. De plus, la formalisation doit aussi pouvoir permettre d'évaluer les impacts des évolutions, à partir des liens et dépendances explicitement déclarés.

L'état de l'art nous a montré que les outils de spécification d'architecture et de validation de leurs propriétés utilisent des fondations théoriques telles que la logique du premier ordre, les automates à états finis, la satisfaisabilité booléenne. Cependant, les architectures ont différents niveaux d'abstractions, et les interactions entre les composants ont autant d'importance que les composants eux-mêmes. Ces fondations formelles ne couvrent pas intégralement l'ensemble de ces besoins. La théorie des catégories a prouvé sa flexibilité en ayant été appliquée dans divers domaines [FS19].

En se concentrant sur les relations et les compositions, la théorie des catégories met en œuvre de puissants mécanismes qui peuvent être appliqués pour formaliser des architectures :

1. Le comportement des foncteurs associé aux produits de catégories et aux préordres permettent de déduire :
   (a) la formation des propriétés complexes en connaissant les propriétés simples dont elles dépendent ;
   (b) la valeur que va prendre un ensemble de composants pour une propriété donnée en connaissant la valeur que prend chaque composant individuellement pour cette propriété.
2. Les foncteurs peuvent être utilisés pour traverser les niveaux d'abstraction, en agissant comme des ponts permettant de passer d'une partie détaillée de l'architecture à l'architecture complète.
3. Les foncteurs pleins peuvent être utilisés pour prouver qu'une implémentation d'architecture suit un certain style ou patron.

5.1. *Notions essentielles de la théorie des catégories*

Née en 1940 [EM45], la théorie des catégories permet d'étudier les relations entre différentes structures, et donc de passer d'un modèle à l'autre ou encore de naviguer entre différents niveaux d'abstraction. Elle peut être appliquée à une multitude de domaines, comme les mathématiques, la physique ou l'informatique [Spi14]. Cette théorie se base sur deux éléments principaux : les catégories et les foncteurs.

DÉFINITION 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 $f : x \to y$;*
3. *pour chaque $x \in$ Ob(C), un morphisme particulier id_x : le morphisme identité de x, $id_x : x \to x$;*
4. *pour chaque triplet $x, y, z \in$ Ob(C), une **composition** $\circ$: Hom$_C(y, z) \times$ Hom$_C(x, y) \to$ Hom$_C(x, z)$. Pour deux morphismes $f : x \to y$ et $g : y \to z$, la composition est notée $g \circ f : x \to z$.*

Et deux lois :

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

Dans les schémas, 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 schémas, les morphismes identités ne sont pas représentés bien que toujours existants.

DÉFINITION 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 $F : C \to C'$, 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 : $F :$ Hom$_C(x, y) \to$ Hom$_{C'}(F(x), F(y))$, pour chaque paire $x, y \in$ Ob(C).*

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 $g : x \to y$, $h : y \to z$.*

L'effet d'un foncteur sur les morphismes impose des contraintes fortes sur la validité d'un foncteur, sur lesquelles il est possible de se baser pour déduire plusieurs propriétés. En effet, pour chaque objet envoyé d'une catégorie à l'autre, les morphismes le transformant en un autre objet doivent aussi être conservés. La figure 5 illustre plusieurs cas de foncteurs d'une catégorie $C1$ vers 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 $F(b1) = c2$ pour les objets, et $F(f1) = g2$ pour le morphisme. Sur la troisième figure, on a $F(a1) = a2$ et $F(b1) = a2$ également. Pour valider la définition du foncteur, $F(f1) = id_{a2}$. 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 $C2$ qui permette de valider le foncteur.

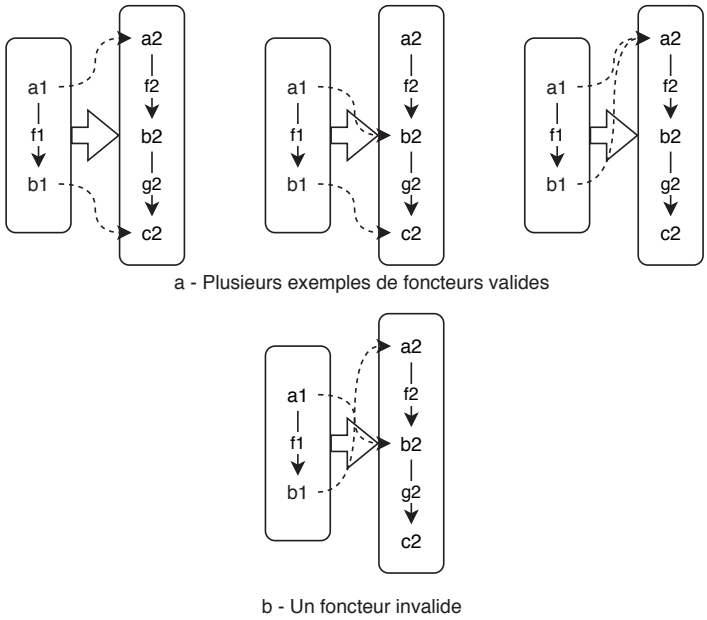

Figure 5. *Exemple de l'implication de* $F(h \circ g) = F(h) \circ F(g)$

5.2. *Définitions avancées*

Quelques définitions d'éléments de la théorie des catégorie nécessaires à la formalisation d'architectures sont présentées dans cette section.

DÉFINITION 3.– *Préordre*

Les préordres sont 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* $f = g$*.*

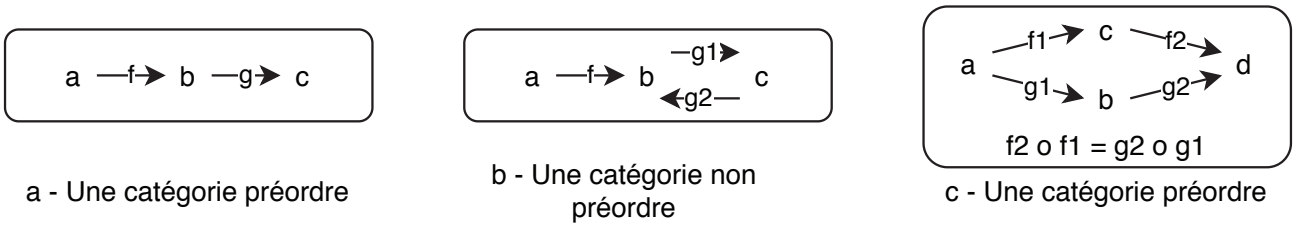

Figure 6. *Comportement des préordres*

Les préordres apportent une forme de hiérarchie dans une catégorie. En effet, entre chaque paire $x, y \in \text{Ob}(C)$, il ne peut exister qu'un seul morphisme, incluant les compositions. Sur la figure 6, on voit 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 la formalisation, lorsque le cas de chemins parallèles comme dans la partie c se présentera, nous omettrons d'écrire l'équation.

DÉFINITION 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. Une catégorie représentant un power set est un préordre.

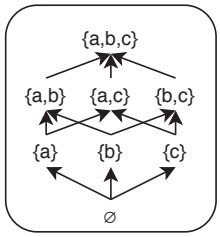

Figure 7. *Un power set pour l'ensemble* $\{a, b, c\}$

Un exemple de *power set* pour l'ensemble $\{a, b, c\}$ est donné dans la figure 7.

DÉFINITION 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 $(x, y) \to (x', y')$ sont les paires (f, g) avec $f : x \to x' \in Hom_{C1}(x, x')$ et $g : y \to y' \in Hom_{C2}(y, y')$.

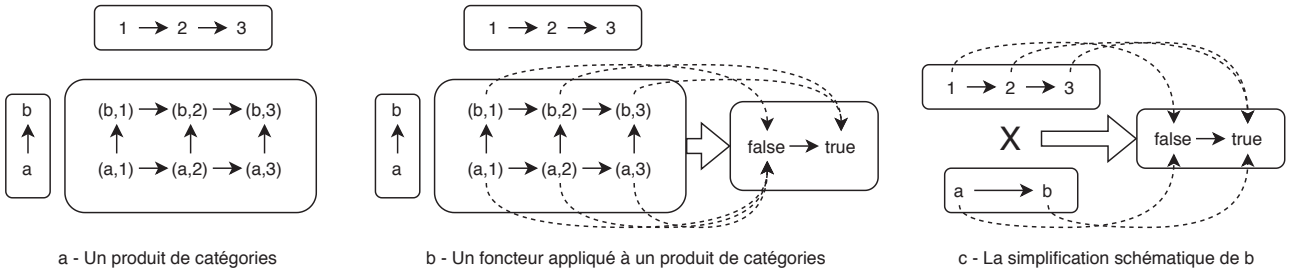

Figure 8. *Fonctionnement et simplification schématique du produit de catégories*

La figure 8 représente le fonctionnement du produit de catégories. La partie a montre la formation des objets et des morphismes dans le nouvelle catégorie créée, la partie b montre comment un foncteur peut être appliqué sur le résultat d'un produit de catégorie, et la partie c montre la simplification schématique de la partie b que nous utiliserons dans la suite de cet article par soucis 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.

DÉFINITION 6.– *Foncteur plein*

On dit qu'un foncteur $F : C \rightarrow D$ défini comme $Hom_F(c, c') : Hom_C(c, c') \rightarrow Hom_D(F(c), F(c'))$ avec $c, c' \in Ob(C)$ est plein si $Hom_F(c, c')$ est surjectif pour toute paire $c, c' \in Ob(C)$.

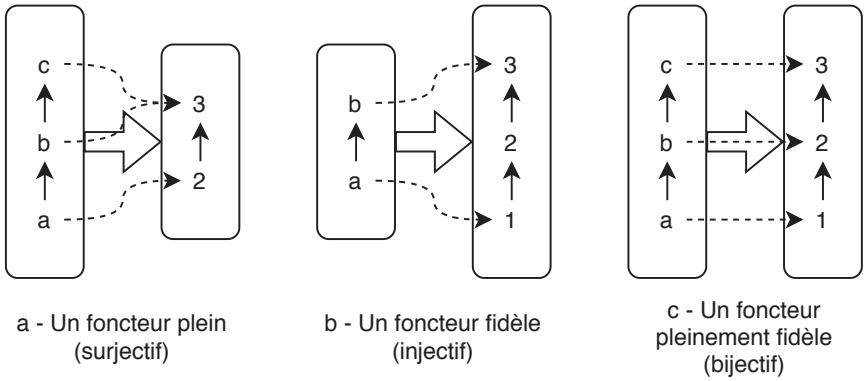

Figure 9. *Des exemples de foncteurs fidèle, plein et pleinement fidèle*

En plus des foncteurs pleins, il existe deux autres types de foncteurs représentés dans la figure 9 : les foncteurs fidèles (qui sont injectifs) et les foncteurs pleinement fidèles (qui sont bijectifs).

5.3. *Principes de la formalisation*

Nous proposons une formalisation d'architectures se basant sur la théorie des catégories, détaillée ci-après. Nous définissons trois catégories principales et deux foncteurs :

DÉFINITION 7.– *La catégorie $Components$*

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

DÉFINITION 8.– *La catégorie $Architecture$*

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

DÉFINITION 9.– *La catégorie $ComponentsPS$*

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

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

DÉFINITION 10.– *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 dans l'architecture.

DÉFINITION 11.– *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 constitué uniquement d'un seul composant. De cette manière, il est possible d'étudier le comportement des propriétés par rapport à un ensemble de composants.

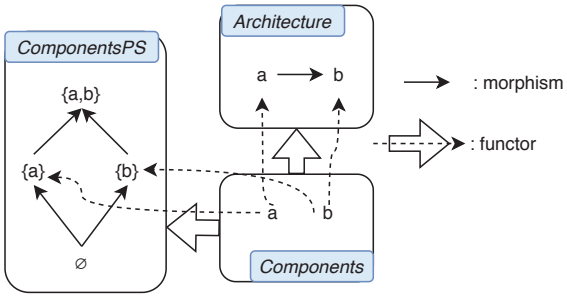

Figure 10. *Spécification d'une architecture simple avec deux composants a et b*

La figure 10 présente la formalisation d'une architecture simple, contenant deux composants a et b, représentés dans la catégorie $Components$. La catégorie $Architecure$ 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.

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.

DÉFINITION 12.– *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 ne contenant que 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.

La figure 11 permet de visualiser comment la propriété d'exactitude des résultats (représentée par la catégorie $Correctness$) est supportée dans une architecture avec deux composants a et b utilisant du *stream processing*. Dans ce cas, l'exactitude dépend de la propriété de garantie de traitement du *stream*

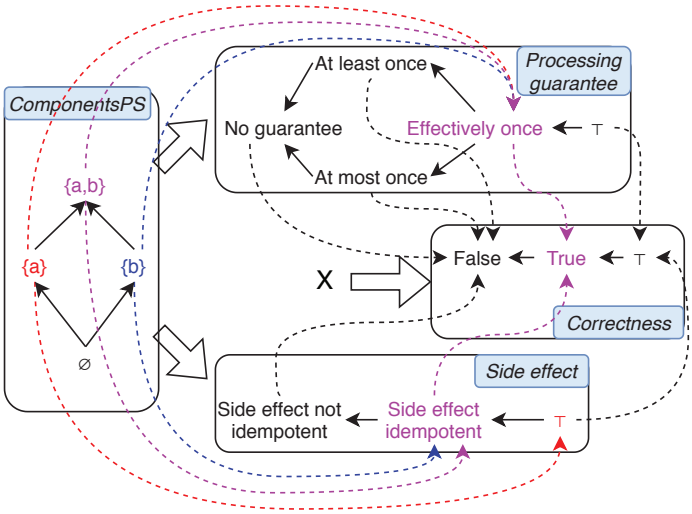

Figure 11. *Spécification et déduction d'une propriété complexe*

processing (catégorie $ProcessingGuarantee$), ainsi que des effets de bord éventuels lors de l'exécution des traitements (catégorie $SideEffect$). Afin de simplifier la figure, les catégories $Components$ et $Architecture$ sont omises, et le résultat du produit de catégories est représenté par le symbole X. Le résultat de la transformation entraînée par le foncteur entre le produit et la propriété d'exactitude peut être retrouvé en conservant, pour chaque pair x, y avec $x \in ProcessingGuarantee$ et $y \in SideEffect$, la valeur la plus basse indiquée par les flèches en pointillés (cf figure 8). Dans cet exemple, les deux composants de l'architecture possèdent la valeur *effectively-once* pour la propriété de garantie de traitement. Pour la propriété d'effet de bord, le composant a n'en a pas et n'est donc pas concerné par la propriété, il prend alors la valeur $\top$, tandis que le composant b produit des effets de bord idempotents. Pour connaître la valeur de chaque composant pour la propriété d'exactitude, il faut regarder la valeur la plus basse pour les paires *effectively-once* et $\top$, et *effectively-once* et *side effect idempotent*. Pour ces deux cas, la valeur la plus basse est $true$, ce qui veut dire que la propriété d'exactitude est supportée.

REMARQUE. Grâce à cette formalisation, des connaissances supplémentaires peuvent être extraites. La catégorie $ComponentsPS$ ainsi que les catégories de propriétés sont des préordres. Les valeurs de propriétés pour les composants individuels sont fixées. En s'appuyant sur la définition des foncteurs, et plus particulièrement sur $F : \mathrm{Hom}_C(x, y) \rightarrow \mathrm{Hom}_{C'}(F(x), F(y))$, 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. 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.

La figure 12 montre la formalisation appliquée au patron Lambda+ Architecture. Pour simplifier le schéma, la catégorie $ComponentsPS$ contient seulement les sous-ensembles de composants qui sont valides dans l'architecture. Par exemple, la combinaison du *data traffic controller*, du *master dataset* et du composant *storage* n'est pas valide, car il n'est pas possible de relier le *storage* au *master dataset* ou au *data traffic controller* sans le composant de *streaming ETL* en intermédiaire. Pour décrire un style ou un patron plutôt qu'une implémentation d'architecture, la valeur des propriétés de niveau 1 pour chaque composant n'a pas à être spécifiée, mais l'existence du foncteur est indiquée, afin de les définir lors de l'implémentation.

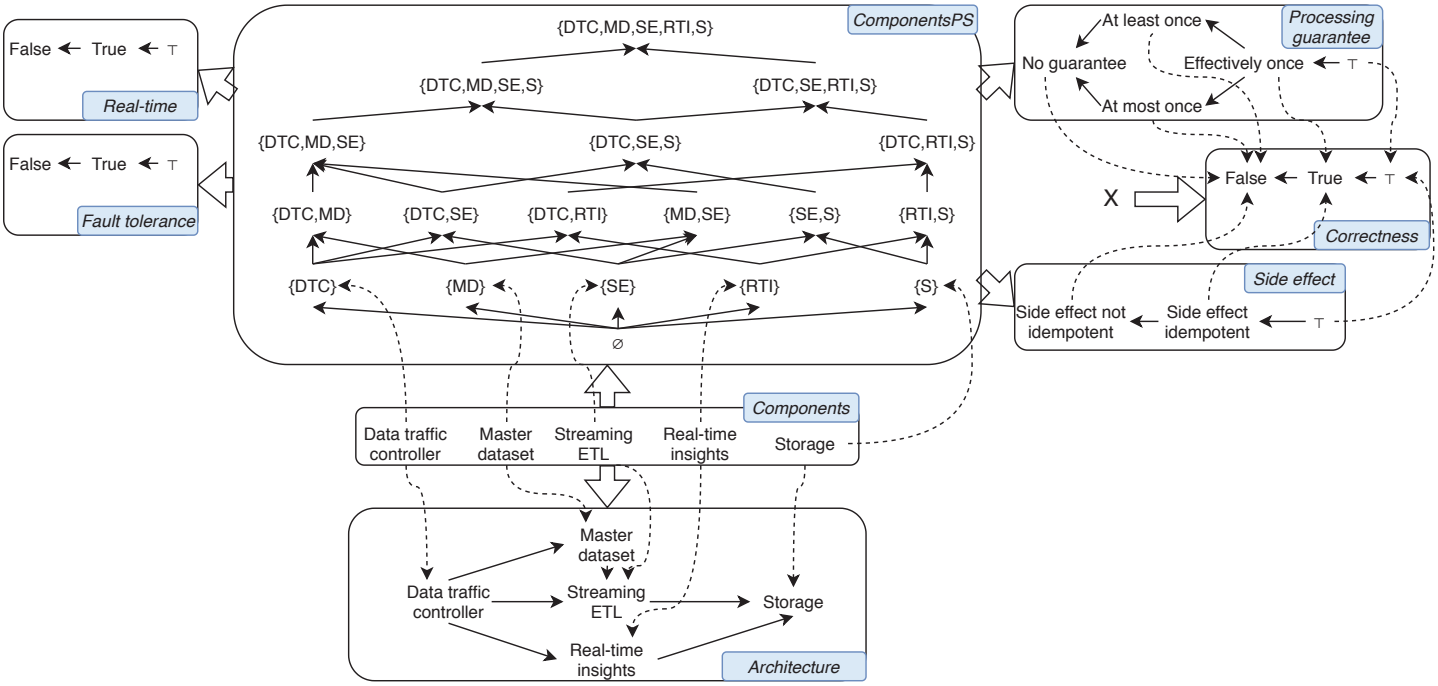

Figure 12. *La formalisation du patron Lambda+ Architecture*

Un autre point fort de la théorie des catégories que nous pouvons exploiter dans la formalisation d'architectures est sa capacité à naviguer entre différents niveaux d'abstraction. Pour ce faire, les sous-composants nécessaires au fonctionnement interne d'un composant sont représentés par des objets dans une nouvelle catégorie, qui est liée à la catégorie $Components$ par un foncteur envoyant les sous-composants vers le composant dont ils font partie. Cette nouvelle catégorie créée a le même fonctionnement que la catégorie $Components$, et est reliée par des foncteurs à des catégories similaires aux catégories $Architecture$ et $ComponentsPS$, correspondant au niveau d'abstraction voulu. Ainsi, il est possible d'étudier l'évolution des propriétés pour un niveau d'abstraction différent de l'architecture.

Un exemple de ce fonctionnement appliqué à l'architecture Hydre est donné dans la figure 13. Pour garder la figure lisible, seuls les processus de traitement des hashtags et des mentions sont conservés. Les deux autres processus de traitement d'URL et de retweets (voir figure 3) ayant la même logique de traitement sont omis. Ces sous-composants possèdent la propriété de résistance aux pannes (catégorie $FaultTolerance$) et de traitement en temps réel (catégorie $RealTime$), puisqu'ils sont développés avec Kafka Streams qui permet de supporter ces propriétés, et que le flux de données peut être traité à nouveau en cas d'erreur, soit directement depuis les topics Kafka si le délai de rétention n'est pas dépassé, soit depuis le *master dataset* si c'est le cas. Les résultats produits respectent également la propriété d'exactitude (catégorie $Correctness$), puisque la garantie de traitement (catégorie $Processing\ Guarantee$) est en *effectively-once* et que les effets de bord (catégorie $SideEffect$) sont idempotents.

À partir de la catégorie $Architecture$ d'une implémentation d'architecture, il est aussi possible de vérifier si cette implémentation suit un style ou un patron d'architecture particulier. Pour cela, il suffit de démontrer l'existence d'un foncteur plein allant de la catégorie $Architecture$ de l'implémentation à la catégorie $Architecture$ du style ou patron. En utilisant un foncteur plein plutôt qu'on foncteur pleinement fidèle, cela permet de faire correspondre une implémentation plus détaillée à un style ou patron plus général.

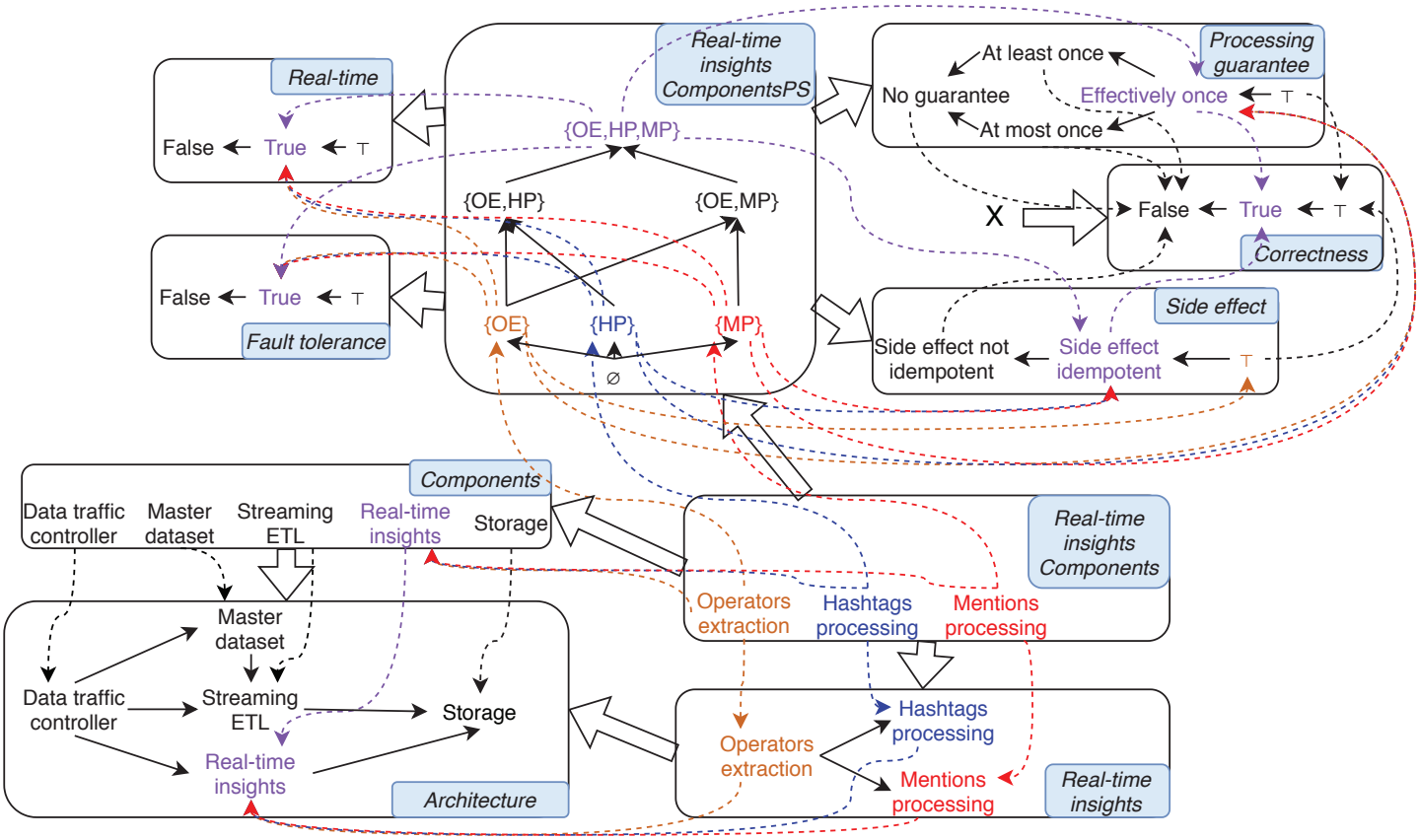

Figure 13. *Un extrait de la formalisation de l'architecture Hydre*

6. Conclusion

Dans cet article, nous avons présenté la Lambda+ Architecture, un patron visant à étendre la Lambda Architecture à des cas d'utilisation sur données massives plus variés, tout en évitant ses défauts et ses complexités. Nous avons appliqué ce patron à un cas réel dans le cadre d'un projet interdisciplinaire, dont l'objectif est l'étude des discours sur Twitter. Hydre, la plateforme réalisée, permet de collecter, stocker et analyser des tweets, en produisant en temps réel des indicateurs sur les collectes, mais aussi en réalisant des analyses exploratoires sur des sujets spécifiques.

Nous proposons également un cadre théorique pour la formalisation d'architectures, utilisant la théorie des catégories comme support. Cette formalisation permet d'étudier la conservation des propriétés lors de la composition d'architectures, de déduire la valeur d'une propriété complexe en connaissant la valeur de propriétés simples, de naviguer entre les niveaux d'abstraction, et de vérifier si une implémentation d'architecture respecte un certain style ou patron. Nous montrons comment cette proposition a été mise en oeuvre sur l'architecture Hydre.

Comme perspectives, nous prévoyons d'une part d'appliquer cette formalisation à d'autres styles et patrons d'architecture afin de vérifier qu'elle se généralise bien, tout en faisant apparaître les spécificités de chaque style et patron. D'autre part, nous souhaitons étendre cette formalisation sur plusieurs points : 1) déduire automatiquement quels sont les sous-ensembles valides (constitués de composants pouvant être reliés entre eux) dans la catégorie $ComponentsPS$, dans le but de simplifier la formalisation en évitant d'avoir à tester les propriétés sur des ensembles de composants qui ne peuvent pas exister, et 2) étendre la définition des propriétés, afin d'inclure des valeurs numériques, des intervalles autorisés, par

exemple pour contrôler si une architecture respecte la propriété de temps réel en connaissant les temps d'exécution dans les différents composants.

Remerciements

Ce travail est soutenu par le programme « Investissements d'Avenir », projet ISITE-BFC (contrat ANR-15-IDEX-0003). Le projet Cocktail est piloté scientifiquement par Gilles Brachotte, laboratoire CIMEOS EA-4177, Université de Bourgogne.

Bibliographie

[AAG95] Gregory D Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(4) :319–364, 1995.

[ABB+13] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel : fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11) :1033–1044, 2013.

[ABC+15] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model : a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. 2015.

[ABD+19] Rana Alotaibi, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, and Stamatis Zampetakis. Towards Scalable Hybrid Stores : Constraint-Based Rewriting to the Rescue. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1660–1677, 2019.

[ACL18] Tyler Akidau, Slava Chernyak, and Reuven Lax. *Streaming Systems : The What, Where, When, and How of Large-scale Data Processing*. " O'Reilly Media, Inc.", 2018.

[AJGSL20] Hiba Abou Jamra, Annabelle Gillet, Marinette Savonnet, and Éric Leclercq. Analyse des discours sur Twitter dans une situation de crise. In *INFormatique des ORganisations et des Systèmes d'Information et de Décision (INFORSID)*, pages 1–16, 2020.

[AKA14] Djeffal Abdelhamid, Soltani Khaoula, and Ouassaf Atika. Automatic bank fraud detection using support vector machines. In *The International Conference on Computing Technology and Information Management (ICCTIM)*, page 10. Citeseer, 2014.

[ALPA17] Subutai Ahmad, Alexander Lavin, Scott Purdy, and Zuha Agha. Unsupervised real-time anomaly detection for streaming data. *Neurocomputing*, 262 :134–147, 2017.

[BC11] Manfred Broy and María Victoria Cengarle. Uml formal semantics : lessons learned. *Software & Systems Modeling*, 10(4) :441–446, 2011.

[Bro11] Manfred Broy. Can practitioners neglect theory and theoreticians neglect practice ? *Computer*, 44(10) :19–24, 2011.

[CB11] Tony Clark and Balbir S Barn. Event driven architecture modelling and simulation. In *Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE)*, pages 43–54. IEEE, 2011.

[CKE+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink : Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[Cle96] Paul C Clements. A survey of architecture description languages. In *Proceedings of the 8th international workshop on software specification and design*, pages 16–25. IEEE, 1996.

[Cra88] Iain D Craig. Blackboard systems. *Artificial Intelligence Review*, 2(2) :103–118, 1988.

[Dea09] John Deacon. Model-view-controller (MVC) architecture. 2009.

[Dix10] James Dixon. Pentaho, hadoop, and data lakes. *blog, Oct*, 2010.

[EM45] Samuel Eilenberg and Saunders MacLane. General theory of natural equivalences. *Transactions of the American Mathematical Society*, 58(2) :231–294, 1945.

[FPK+15] Raul Castro Fernandez, Peter R Pietzuch, Jay Kreps, Neha Narkhede, Jun Rao, Joel Koshy, Dong Lin, Chris Riccomini, and Guozhang Wang. Liquid : Unifying Nearline and Offline Big Data Integration. In *Conference on Innovative Data System Research (CIDR'15)*, 2015.

[FS19] Brendan Fong and David I Spivak. *An invitation to applied category theory : seven sketches in compositionality*. Cambridge University Press, 2019.

[FY97] Brian Foote and Joseph Yoder. Big ball of mud. *Pattern languages of program design*, 4 :654–692, 1997.

[GCD+16] Vijay Gadepally, Peinan Chen, Jennie Duggan, Aaron Elmore, Brandon Haynes, Jeremy Kepner, Samuel Madden, Tim Mattson, and Michael Stonebraker. The BigDAWG Polystore System and Architecture. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2016.

[GFB+19] Annabelle Gillet, Alexander Frame, Gilles Brachotte, Éric Leclercq, and Marinette Savonnet. Analysis of the 2014 and 2019 European elections using Twitter data. *Modèles & Analyse des Réseaux : Approches Mathématiques & Informatiques*, 2019.

[Gia18] Korina Giaxoglou. #JeSuisCharlie ? Hashtags as narrative resources in contexts of ecstatic sharing. *Discourse, context & media*, 22 :13–20, 2018.

[GLC19] Annabelle Gillet, Éric Leclercq, and Nadine Cullot. Lambda Architecture pour une analyse à haute performance des données des réseaux sociaux. In *INFormatique des ORganisations et des Systèmes d'Information et de Décision (INFORSID)*, 2019.

[HA07] Neil B Harrison and Paris Avgeriou. Leveraging architecture patterns to satisfy quality attributes. In *European conference on software architecture*, pages 263–270. Springer, 2007.

[IGS+18] Jean-François Im, Kishore Gopalakrishna, Subbu Subramaniam, Mayank Shrivastava, Adwait Tumbde, Xiaotian Jiang, Jennifer Dai, Seunghyun Lee, Neha Pawar, Jialiang Li, et al. Pinot : Realtime olap for 530 million users. In *Proceedings of the 2018 International Conference on Management of Data*, pages 583–594, 2018.

[Inm05] William H Inmon. *Building the data warehouse*. John wiley & sons, 2005.

[Inm16] Bill Inmon. *Data Lake Architecture : Designing the Data Lake and avoiding the garbage dump*. Technics publications, 2016.

[JEJ12] Pontus Johnson, Mathias Ekstedt, and Ivar Jacobson. Where's the theory for software engineering ? *IEEE software*, 29(5) :96–96, 2012.

[KBF+15] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron : Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250, 2015.

[KLP+18] Boyan Kolev, Oleksandra Levchenko, Esther Pacitti, Patrick Valduriez, Ricardo Vilaça, Rui Gonçalves, Ricardo Jiménez-Peris, and Pavlos Kranas. Parallel polyglot query processing on heterogeneous cloud data stores with LeanXcale. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 1757–1766. IEEE, 2018.

[KMM+15] Mariam Kiran, Peter Murphy, Inder Monga, Jon Dugan, and Sartaj Singh Baveja. Lambda architecture for cost-effective batch and speed big data processing. In *IEEE International Conference on Big Data*, pages 2785–2792. IEEE, 2015.

[KPK+19] Evdokia Kassela, Nikodimos Provatas, Ioannis Konstantinou, Avrilia Floratou, and 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)*, pages 317–326. IEEE, 2019.

[Kre14] Jay Kreps. Questioning the Lambda Architecture. *O'Reilly RADAR, online article, July*, 2014.

[Lam83] Butler W Lampson. Hints for computer system design. In *Proceedings of the ninth ACM symposium on Operating systems principles*, pages 33–48, 1983.

[LL17] Chung-Ho Lee and Chi-Yi Lin. Implementation of Lambda Architecture : A Restaurant Recommender System over Apache Mesos. In *IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 979–985. IEEE, 2017.

[LM98] Daniel Le Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on software engineering*, 24(7) :521–533, 1998.

[Mar11] Nathan Marz. How to beat the CAP theorem, 2011.

[Mar14] Nathan Marz. History of apache storm and lessons learned, 2014.

[MM15] Alexander Malkis and Diego Marmsoler. A model of service-oriented architectures. In *2015 IX Brazilian Symposium on Components, Architectures and Reuse Software*, pages 110–119. IEEE, 2015.

[MM18] Amr A Munshi and Yasser Abdel-Rady I Mohamed. Data lake lambda architecture for smart grids big data analytics. *IEEE Access*, 6 :40463–40471, 2018.

[MME15] Diego Marmsoler, Alexander Malkis, and Jonas Eckhardt. A model of layered architectures. *arXiv preprint arXiv :1503.04916*, 178 :47–61, 2015.

[MR17] Mohamed A Mabrok and Michael J Ryan. Category theory as a formal mathematical foundation for model-based systems engineering. *Appl. Math. Inf. Sci*, 11 :43–51, 2017.

[MW15] Nathan Marz and James Warren. *Big Data : Principles and best practices of scalable real-time data systems*. Manning, 2015.

[MZT+16] John Meehan, Stan Zdonik, Shaobo Tian, Yulong Tian, Nesime Tatbul, Adam Dziedzic, and Aaron Elmore. Integrating real-time and batch processing in a polystore. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–7. IEEE, 2016.

[NSS14] Dmitry Namiot and Manfred Sneps-Sneppe. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9) :24–27, 2014.

[PJ92] Meilir Page-Jones. Comparing techniques by means of encapsulation and connascence. *Communications of the ACM*, 35(9) :147–151, 1992.

[SD20] Pegdwendé Sawadogo and Jérôme Darmont. On data lake architectures and metadata management. *Journal of Intelligent Information Systems*, pages 1–24, 2020.

[SKD19] Pegdwendé Sawadogo, Tokio Kibata, and Jérôme Darmont. Metadata management for textual documents in data lakes. In *21st International Conference on Enterprise Information Systems (ICEIS 2019)*, 2019.

[SN20] Darja Solodovnikova and Laila Niedrite. Handling evolution in big data architectures. *Baltic Journal of Modern Computing*, 8(1) :21–47, 2020.

[Spi14] David I Spivak. *Category theory for the sciences*. MIT Press, 2014.

[TTS+14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156, 2014.

[VS09] Panos Vassiliadis and Alkis Simitsis. Near real time ETL. In *New trends in data warehousing and data analysis*, pages 1–31. Springer, 2009.

[YMR+17] Fangjin Yang, Gian Merlino, Nelson Ray, Xavier Léauté, Himanshu Gupta, and Eric Tschetter. The RAD-Stack : Open source lambda architecture for interactive analytics. In *Proceedings of the 50th Hawaii International Conference on System Sciences*, 2017.

[ZDL+12] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams : an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing*, 2012.

[ZM18] Michele Zappavigna and James R Martin. #Communing affiliation : Social tagging as a resource for aligning around values in social media. *Discourse, Context & Media*, 22 :4–12, 2018.

