Détection d'évènements complexes dans les flux d'évènements massifs

William Braik

ACADÉMIE DE BORDEAUX
UNIVERSITÉ DE **B**ORDEAUX
Sciences et **T**echnologies

THÈSE

Présentée au Laboratoire Bordelais de Recherche en Informatique pour obtenir le grade de Docteur de l'Université de Bordeaux

Spécialité	:	**Informatique**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Mathématiques et Informatique**

Détection d'évènements complexes dans les flux d'évènements massifs

Complex event detection over large event streams

par

William BRAIK

Soutenue le 15 mai 2017, devant le jury composé de :

Président du jury
Didier DONSEZ, Professeur ..... Université Grenoble 1, France

Directeur de thèse
Xavier BLANC, Professeur ..... Université de Bordeaux, France

Rapporteurs
Didier DONSEZ, Professeur ..... Université Grenoble 1, France
Laurent PAUTET, Professeur ..... Télécom ParisTech, France

Examinateurs
David AUBER, Maître de conférences ..... Université de Bordeaux, France
Sonia BEN MOKHTAR, Chargée de recherche ..... INSA de Lyon, France
Floréal MORANDAT, Maître de conférences ..... ENSEIRB-MATMECA, France

université de BORDEAUX

Table des matières

CHAPITRE

1 Introduction

Ce chapitre d'introduction présente le contexte de cette thèse CIFRE, réalisée avec la société AKKA et en partenariat avec le site ecommerce Cdiscount. Récemment, Cdiscount a porté un intérêt particulier à l'analyse en temps réel [1] du comportement de ses clients. Dans ce cadre, nous commencons par présenter les problématiques de la thèse ainsi que les contributions réalisées.

1.1 Contexte

La société Cdiscount, partenaire d'AKKA et pionnière dans le domaine de l'ecommerce en France, base l'essentiel de ses activités de vente autour de son site web marchand. Un aspect crucial de ce site concerne les données de traçage générées par l'activité des clients au cours de leurs sessions de navigation.

Environ deux millions de clients visitent quotidiennement le site Cdiscount. Grâce à un système de traçage intégré au site, chaque client émet en temps réel des évènements dont le rôle est de décrire avec précision les différentes actions réalisées lors de la navigation. Les évènements représentent typiquement les clics sur les produits, les ajouts au panier, les achats, etc. L'ensemble des évènements produits en continu par le système de traçage forme un flux d'évènements massif.

L'analyse de cette masse de données permet d'orienter les stratégies *marketing* et les prises de décision de l'entreprise en révélant les comportements et les habitudes d'achat des clients. Considérons le scénario d'un client ayant consulté plusieurs fois la même fiche

1. La notion de *temps réel* est historiquement utilisé en informatique pour parler des systèmes où les contraintes temporelles sont garanties. Cependant, dans notre contexte, nous utilisons ce terme pour faire référence au fait de traiter les données à la volée et avec un temps de réponse court.

produit dans un intervalle de temps réduit, sans avoir ajouté le dit produit au panier. Dans ce cas, il est possible d'inférer en temps réel l'hésitation du client, qui représente une opportunité de vente à laquelle le site ecommerce est en mesure de réagir immédiatement, en proposant par exemple un bon d'achat ou une aide interactive au client. L'analyse du comportement, et plus généralement des scénarios de navigation des clients en temps réel permet ainsi au site marchand de déclencher des ventes supplémentaires, tout en contribuant à l'amélioration de l'expérience utilisateur.

1.2 Problématiques

Des plateformes dites *Big Data* permettent déjà d'analyser de grands volumes de données, et d'identifier des profils comportementaux à partir de l'historique des sessions de navigation [Wu *et al.*, 2014]. Cependant, l'inconvénient majeur de ces plateformes est qu'elles sont, par conception, incapables de traiter les données en temps réel. En effet, leur modèle de traitement de données par lots (*Batch Processing*) ne permet pas d'exploiter immédiatement les résultats, et limitent donc la prise de décision. C'est dans l'optique de dépasser cette limite que Cdiscount a entrepris l'évolution du modèle de traitement des données de traçage, vers un modèle de traitement de flux (*Stream Processing*), plus adapté aux besoins de réactivité et d'interactivité du site ecommerce.

Le problème soulevé par Cdiscount, qui fait l'objet de cette thèse, est donc un problème d'identification de scénarios complexes en temps réel dans des flux infinis. Ce type de problème relève du domaine du *traitement d'évènements complexes* (*Complex Event Processing*, ou CEP [Cugola et Margara, 2012b]). Le CEP vise à identifier, de manière continue, des combinaisons d'évènements prédéfinies dans un flux d'évènements. Dans notre contexte, ces combinaisons d'événements décrivent des scénarios spécifiques, tels que l'hésitation d'achat d'un produit, et qui doivent être détectés au niveau de chaque client.

La particularité de la thèse tient essentiellement de (i) l'application du domaine CEP dans un cadre ecommerce, et (ii) des exigences de performance formulées par Cdiscount, et qui sont relatives à la détection de scénarios.

Plus spécifiquement, les problématiques ciblées tiennent en trois points essentiels.

— L'expression fine des scénarios à identifier : pour identifier des évènements complexes à forte valeur ajoutée dans le contexte d'un site web marchand, il est nécessaire de pouvoir exprimer finement différents types de scénarios de navigation. La problématique consiste donc à proposer un langage d'expression de ces scénarios, adapté au site ecommerce.

— L'identification en temps réel des scénarios exprimés : le processus d'identification doit être réalisé de manière efficace et continue, afin de répondre à l'objectif de détection rapide des scénarios prédéfinis. Par exemple, Cdiscount souhaite qu'un scénario donné puisse être identifié chez un client avec un délai d'environ une seconde, afin de pouvoir établir suffisamment d'interactivité avec le client.

— Une capacité de passage à l'échelle et de tolérance aux pannes : la performance et la disponibilité du système doivent être garanties, quelque soit la nature du flux d'événements à analyser. À relativement court terme, le débit du flux d'évènements produit par le site Cdiscount devrait atteindre environ 10 000 évènements par seconde. Mais ce chiffre est en constante augmentation, du fait d'un nombre de visiteurs de plus en plus important, et d'un système de traçage de plus en plus perfectionné. L'objectif est donc de garantir que le système CEP dispose, à tout instant, de suffisamment de ressources de calcul pour traiter efficacement le flux d'évènements. D'autre part, le système CEP doit pouvoir tolérer les pannes pouvant survenir de manière aléatoire, afin d'assurer son fonctionnement continuel.

1.3 AUROS: un système CEP pour les sites ecommerce

Dans cette thèse, nous présentons AUROS, un système CEP permettant d'identifier en temps réel des scénarios complexes dans des flux d'événements, et conçu pour répondre aux problématiques décrites dans la section précédente.

Tout d'abord, AUROS propose un langage permettant d'exprimer des scénarios complexes à identifier dans le flux. Le choix d'implémenter ce nouveau langage a été motivé par la volonté de concevoir un système CEP accessible aux utilisateurs *non-techniques*, tels que l'équipe *marketing* de Cdiscount. D'autre part, le langage est suffisamment expressif pour spécifier de nombreux scénarios à identifier, et notamment ceux décrits par Cdiscount.

Concrètement, AUROS permet la spécification d'un ensemble de règles de détection, décrivant chacune un scénario spécifique à identifier, pour chaque client du site, dans le flux d'évènements. Pour cela, l'utilisateur d'AUROS définit les caractéristiques des évènements attendus, leur ordre d'occurrence, et éventuellement un ensemble de contraintes spécifiant les relations entre les différents évènements, comme par exemple l'intervalle de temps qui les séparent. Le langage proposé permet également d'associer une action à chaque règle de détection. Cette action est exécutée lors de la détection d'un scénario spécifié par la règle et permet, par exemple, d'enregistrer ces détections dans une base de données ou d'émettre des alertes en temps réel.

Pour répondre à la problématique d'efficacité, AUROS propose un modèle d'exécution de règle qui est l'automate fini. Toute règle de détection est compilée par le système en un automate fini déterministe. L'état initial de l'automate représente l'initialisation d'une nouvelle instance correspondant à un scénario, et l'état final l'identification effective d'une occurrence de ce scénario par le système CEP.

Les transitions de l'automate permettent d'identifier les contraintes entre les différents évènements constituant un scénario. Au fur et à mesure du processus de détection, AUROS maintient l'état des automates correspondant à chaque règle et ce, pour chaque client naviguant sur le site marchand. Cette approche permet le traitement en temps réel du flux

d'évènements, car la mise à jour de l'état dans un automate est une opération peu coûteuse. Cependant il est courant pour un système CEP de gérer simultanément de nombreuses règles de détection et de nombreux clients. Cela amène donc un nombre important d'états à coexister au sein du système, puisque le nombre d'états est fonction à la fois du nombre de clients qui produisent les évènements, et des spécificités de chaque règle.

Pour traiter ce flux massif, le système CEP que nous présentons dans cette thèse, AUROS, se base sur la plateforme Spark, qui propose un modèle de traitement de données sur architecture distribuée similaire à *MapReduce* [Dean et Ghemawat, 2008]. Spark, comme *MapReduce*, est un modèle de traitement distribué des données s'exécutant sur une grappe de nœuds de calcul (*cluster*). En permettant de provisionner de nouveaux nœuds de calcul pour augmenter la puissance de traitement du cluster, ce modèle distribué garantit le passage à l'échelle du système en fonction du volume de données à traiter. Bien qu'ils reposent tous deux sur le paradigme Big Data, Spark se distingue de *MapReduce* par son modèle de traitement des données en mémoire [Zaharia *et al.*, 2012], plus efficace que le modèle *MapReduce* qui implique des accès disque fréquents, et donc des latences de traitement considérables. Enfin, l'efficacité du modèle Spark permet à la plateforme d'être compatible avec le modèle de traitement de flux [Zaharia *et al.*, 2013], ce qui n'est pas le cas de *MapReduce*.

En synthèse, le système AUROS comprend plusieurs composantes :

— Le langage AUROS permettant à l'utilisateur du système de spécifier des règles de détection.

— Un modèle d'automate, permettant d'évaluer les règles spécifiées, et ainsi d'identifier en temps réel des scénarios spécifiques dans le flux d'événements en entrée.

— Un processus permettant au système de passer à l'échelle, en distribuant automatiquement le traitement du flux et l'identification des scénarios pour tous les clients de Cdiscount.

Les contributions de cette thèse portent donc sur trois composantes. (i) Nous proposons tout d'abord un nouveau langage de spécification de règles de détection, qui constitue l'interface utilisateur du système AUROS, (ii) puis nous présentons un compilateur permettant de transformer toute règle spécifiée dans le langage AUROS en modèle d'automate fini correspondant, et enfin (iii) nous proposons une plateforme permettant de distribuer le traitement du flux d'évènements d'une part, et l'identification des scénarios pour chaque client d'autre part, en s'appuyant sur le modèle Spark.

1.4 Plan

L'objectif de cette thèse est de présenter AUROS, un système CEP compatible avec des contraintes précises d'expressivité de règles de détection, de performance, et de mise à l'échelle. Nous montrons la possibilité d'implémenter, dans le paradigme particulier qu'est

Big Data, un algorithme de détection de règles distribué capable de traiter efficacement le flux d'évènements massif produit par le site Cdiscount.

Dans le chapitre 2, nous commençons par présenter l'état de l'art du domaine CEP. Pour cela, nous étudions plusieurs systèmes CEP existants et les comparons à AUROS sur différents aspects. Dans cet état de l'art, nous présentons également les principales plateformes Big Data existantes, et justifions notre choix de Spark.

Ensuite, dans le chapitre 3, nous décrivons le langage proposé par AUROS pour spécifier des règles de détection, notamment à travers différents scénarios de navigation extraits de cas d'utilisation concrets exprimés par Cdiscount.

Dans le chapitre 4, nous discutons du modèle d'exécution de règles d'AUROS, basé sur l'automate fini déterministe, et permettant une détection efficace et en temps réel des scénarios spécifiés.

Puis dans le chapitre 5, nous présentons l'architecture distribuée sur laquelle AUROS s'appuie pour garantir le passage à l'échelle et la tolérance aux pannes. Nous expliquerons comment les automates correspondants aux règles spécifiées sont déployés sur une architecture distribuée Spark, et comment celui-ci distribue les traitements sur différents nœuds de calculs.

Dans le chapitre 6 d'évaluation d'AUROS, nous montrons la conformité de notre implémentation d'un système CEP avec les exigences formulées par Cdiscount à travers plusieurs expérimentations.

Enfin, nous concluons dans le chapitre 7 en synthétisant les travaux effectuées dans le cadre de la thèse, discutons les résultats obtenus, et mentionnons les potentielles améliorations futures d'AUROS.

CHAPITRE

2 État de l'art

Dans ce chapitre, nous introduisons les concepts clés du domaine du *Complex Event Processing* (CEP), qui est central à nos travaux. Puis, nous présentons les divers travaux de recherche menés dans ce domaine, à travers ses différents aspects. Ces aspects portent respectivement sur (i) les langages permettant d'exprimer les règles de détection, (ii) les modèles d'exécution des règles permettant l'identification d'évènements complexes dans le flux, et (iii) les différentes architectures et environnements d'exécution sur lesquels systèmes CEP se basent.

Sommaire

2.1 Le domaine du Complex Event Processing

Un système CEP vise essentiellement à détecter, dans un flux d'évènements infini, des séquences prédéfinies d'évènements [Cugola et Margara, 2012b]. Une fois qu'une séquence prédéfinie est détectée par le système CEP, celui produit une combinaison d'évènements, permettant éventuellement de déclencher certaines actions associées (génération d'une alerte, stockage de la combinaison détectée en base de données, etc.).

Le concept central dans le domaine CEP est donc l'*évènement*.

> **Évènement.** Un évènement reflète la réalisation d'une action à un instant donné. Dans notre contexte, les évènements correspondent exclusivement aux actions de navigation réalisées par les clients d'un site web marchand tel que Cdiscount. Ainsi, nous considérons qu'un évènement est caractérisé par : (i) l'identifiant du client à l'origine de la création de l'évènement (attribut `cid`) (ii) la date de création de l'évènement (attribut `date`) (iii) le type d'évènement, qui permet de déterminer précisément l'action réalisée par le client sur le site (attribut `type`) (iv) un ensemble d'informations sur le contexte de l'action (dictionnaire d'attributs `data`).

Chaque client produit en temps réel, lors de la navigation, des évènements de types différents. L'agrégation des évènements produits par l'ensemble des clients constitue alors un *flux d'évènements*.

> **Flux d'évènements.** Un flux d'évènements regroupe un ensemble d'évènements. À haut niveau, un flux peut être vu comme une file infinie d'évènements, à laquelle de nouveaux évènements sont continuellement ajoutés. Dans notre contexte, l'ordre des évènements dans le flux correspond à leur date d'émission par le système de traçage du site Cdiscount. Le débit du flux, c'est-à-dire le nombre d'évènements produits par unité de temps, dépend donc directement du nombre d'actions réalisées par les clients du site à un instant donné.

Comme dit précédemment, le rôle d'un système CEP est d'identifier dans le flux des *séquences d'évènements*.

> **Séquence d'évènements.** Une séquence d'évènements regroupe, de manière ordonnée, l'ensemble des évènements qui correspondent à l'occurrence d'un scénario spécifique. Le système CEP détecte continuellement de nouvelles séquences d'évènements dans le flux.

L'utilisateur du système prédéfinit les différents scénarios à identifier dans le flux via un ensemble de *règles de détection*.

> **Règle de détection.** Les règles de détection sont créées par l'utilisateur du système CEP, et correspondent chacune à un scénario spécifique à détecter. Elles décrivent précisément les caractéristiques des séquences d'évènements à identifier dans le flux. Pour cela, chaque règle décrit les différents évènements qui composent la séquence, ainsi que les relations qui doivent exister entre eux. Une règle de détection est continuellement évaluée par le système CEP, jusqu'à sa désactivation explicite par l'utilisateur.

Chaque règle de détection permet donc au système CEP d'identifier dans le flux de nombreuses séquences. Ces séquences, une fois identifiées, peuvent être exploitées par le système CEP pour produire des *évènements complexes*.

> **Évènement complexe.** Un évènement complexe est produit par le système CEP pour chaque séquence d'évènements identifiée par une règle de détection. L'évènement complexe peut contenir tout ou partie des informations provenant de la séquence d'évènements, ainsi que d'autres informations synthétisées.

Un système CEP consomme donc un flux d'évènements en entrée, y détecte des séquences décrites par des règles de détection, et produit en sortie des évènements complexes. La particularité de ce processus est qu'il est réalisé continuellement et en temps réel. La figure 2.1 schématise le concept d'un système CEP.

Pour illustrer le fonctionnement d'un système CEP, prenons un exemple simple extrait d'un cas d'utilisation réel exprimé par Cdiscount.

LISTING 2.1 – Exemple de flux d'évènements produit par un unique client du site Cdiscount

```
[
(c1,1,Search,{query="TV", sid=42}),
(c1,2,View,{productID="TV_344", sid=42}),
(c1,3,View,{productID="TV_531", sid=42}),
(c1,4,Search,{query="TV 4K", sid=97}),
(c1,5,View,{productID="TV_789", sid=97}),
(c1,6,Add,{productID="TV_789"})
]
```

Le flux d'évènements du listing 2.1 représente l'ensemble des évènements produits par un unique client, dont l'identifiant est `c1`. Par souci de simplicité, les dates des évènements sont représentées par des entiers successifs. Ce client effectue plusieurs actions sur le site, que nous décrivons dans leur ordre chronologique :

1. Il commence par effectuer une première recherche (type d'évènement `Search`) basée sur le mot clé `"TV"` (attribut `query`). À cette recherche est associée l'identifiant 42 (attribut `sid`).

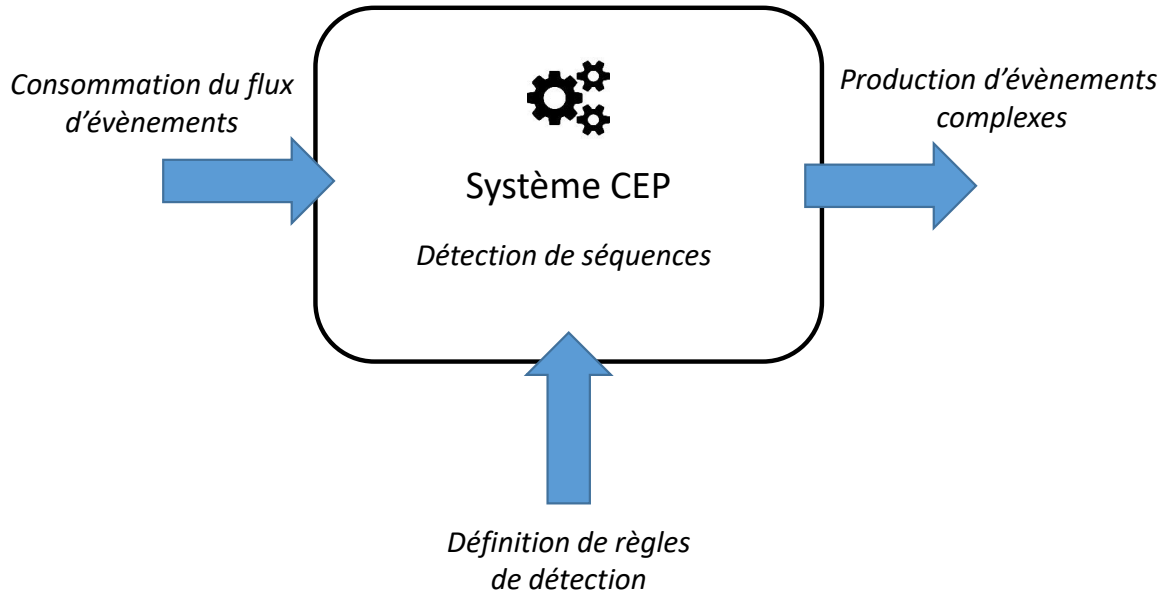

FIGURE 2.1 – Vue de haut niveau d'un système CEP. Un système CEP reçoit en entrée un flux d'évènements, dans lequel il détecte des séquences. Les caractéristiques des séquences à identifier sont paramétrées par un ensemble de règles de détection, qui sont définies par l'utilisateur du système. En sortie du système CEP, celui-ci produit des évènements complexes correspondant à chaque séquence identifiée dans le flux.

2. Puis, il visionne deux produits sélectionnés parmi les résultats de la recherche (type d'évènement `View`, identifiant de recherche `42`). Les évènements `View` comportent l'identifiant des produits visionnés (attribut `productID`).
3. Il effectue ensuite une seconde recherche sur le mot clé `"TV 4K"`.
4. Enfin, le client visionne un produit proposé dans les résultats de cette deuxième recherche, et ajoute celui-ci dans son panier (type d'évènement `Add`).

Imaginons qu'un utilisateur du système CEP, par exemple un membre de l'équipe Marketing de Cdiscount, spécifie une règle de détection visant à identifier les *hésitations d'achat*. Il considère qu'un scénario d'hésitation d'achat correspond à effectuer, dans un intervalle de temps limité, au moins deux recherches successives menant à un ajout au panier, mais sans concrétisation par un achat (évènement de type `Purchase`).

En exprimant une règle de détection spécifiant un tel comportement, l'utilisateur du système CEP peut identifier dans le flux du listing 2.1 l'hésitation d'achat du client `c1`. En effet, le système CEP, en évaluant la règle, peut détecter une séquence d'évènements comportant les deux évènements de recherche (`Search`) suivis d'un évènement d'ajout au panier (`Add`). Cette séquence d'évènements correspond à l'occurrence d'un scénario d'hésitation d'achat. En produisant un évènement complexe correspondant, le système CEP

"signale" ce comportement à l'équipe Marketing de Cdiscount, qui peut éventuellement réagir par une action correspondante, dans le but de déclencher la vente.

Les définitions que nous avons présenté, ainsi que l'illustration par un exemple du rôle d'un système CEP montrent que sa conception nécessite de répondre aux trois points suivants :

— Proposer un langage d'expression des règles de détection. Dans notre contexte, ce langage sera utilisé par les l'équipe marketing de Cdiscount qui souhaite prédéfinir des comportements afin de pouvoir les observer.

— Proposer un modèle d'exécution permettant la détection efficace des évènements complexe en fonction des règles de détection.

— Assurer un passage à l'échelle en faisant en sorte que le système puisse supporter des flux massifs.

Dans la suite de ce chapitre, nous présentons les travaux de recherche relatifs à ces trois aspects.

2.2 Langages de spécification de règles de détection

Cette section présente les différents travaux proposant des langages d'expression de règles de détection. Puis, elle précise les aspects attendus du langage que nous présentons dans cette thèse.

2.2.1 Le langage *PADRES*

PADRES [Li et Jacobsen, 2005] est un système *publish-subscribe* qui propose un langage pour exprimer des souscriptions à des évènements complexes. Bien que *PADRES* ne soit pas un système CEP, son langage peut être considéré comme un langage CEP car il permet de s'abonner à des types d'évènements (souscriptions atomiques) mais aussi à des combinaisons d'évènements.

Comme tout système *publish-suscribe*, *PADRES* permet d'exprimer des souscriptions atomiques correspondant à un ou plusieurs flux d'évènements. *PADRES* se rapproche cependant des systèmes CEP car son langage permet également de combiner ces souscriptions atomiques au sein de règles, notamment via des opérateurs logiques et temporels. Il est possible de filtrer les évènements sélectionnés par une règle en fonction de leurs attributs. De plus, les règles peuvent être combinées entre elles pour former des règles plus complexes.

Les opérateurs logiques (`and`, symbole `&` et `or`, symbole `||`) permettent d'exprimer des contraintes concernant la nature des évènements auxquels souscrire, mais ne précisent pas leur ordre temporel.

L'ordre temporel est exprimé grâce à un opérateur de séquence (symbole ;) qui précise l'ordre dans lequel les évènements doivent être sélectionnés par la règle. Un paramètre `timespan` peut être associé à l'opérateur de séquence pour spécifier un intervalle de temps maximum entre deux évènements se suivant. Il est aussi possible d'associer à une partie d'une règle (bloc) un paramètre `within` pour spécifier l'intervalle de temps maximum entre le premier et le dernier évènement du bloc. Enfin, pour spécifier des contraintes temporelles précises, *PADRES* permet d'accéder à l'attribut `time`, qui présent dans tout évènement, et qui contient leur date d'occurrence.

L'expressivité du langage *PADRES* n'est pas formellement définie, mais vise à être comparable au *Business Process Execution Language* (`BPEL`) [Farahbod *et al.*, 2004]. *BPEL* est un langage standard pour la description des interactions entre différents processus métier via des services web.

Pour autant, *PADRES* ne permet pas de spécifier des négations dans les règles. Il n'est donc pas possible d'exclure des évènements ou des combinaisons d'évènements lorsque l'on exprime une souscription via une règle.

Le langage ne propose pas non plus l'utilisation de fonctions d'agrégation, permettant par exemple de calculer un résultat à partir d'un ensemble d'évènements (par exemple, calculer une moyenne).

2.2.2 Les langages *Sase* et *Sase+*

Sase [Gyllstrom *et al.*, 2006] est un système CEP qui propose un langage dont la syntaxe est fortement inspirée du langage *SQL*, mais adaptée à la spécification de règles de détection.

Une règle de détection *Sase* commence par la définition du flux d'évènements à traiter (clause `FROM`). Puis, une règle est spécifiée par quatre clauses : `PATTERN`, `WHERE`, `WITHIN` et `RETURN`.

La clause `PATTERN` définit l'ensemble des évènements à détecter, ainsi que leurs contraintes logiques et temporelles. Pour cela, des opérateurs logiques de conjonction et disjonction peuvent être utilisés. L'opérateur de séquence (`SEQ`) permet de définir l'ordre d'occurrence des évènements.

La clause `WHERE` spécifie l'ensemble de contraintes associées à la règle. Il s'agit essentiellement de filtres et de conditions s'appliquant aux attributs des évènements.

La clause `WITHIN` permet quant à elle de spécifier une fenêtre temporelle qui s'applique à l'ensemble de la règle de détection. Il s'agit d'une fenêtre fixe, permettant de définir l'intervalle de temps maximum entre le premier et le dernier évènement de la règle.

Enfin, la clause `RETURN`, placée en fin de règle, permet de définir l'évènement complexe à produire lorsque la règle identifie une séquence dans le flux. Celui-ci dérive directement de la règle spécifié dans la clause 'PATTERN', car il est construit en référençant, en transformant ou en agrégeant les attributs extraits des évènements composant la règle de détection.

La limite principale du langage *Sase* provient de l'impossibilité de spécifier des séquences d'évènements dont la taille est indéterminée. Il n'est donc pas possible de spécifier des itérations d'évènements (par exemple, plusieurs recherches successives). Cette limite est cependant effacée dans le langage successeur de *Sase*, nommé *Sase+* [Agrawal *et al.*, 2008]. *Sase+* propose en effet un nouvel opérateur (appelé *Kleene Plus* – en référence aux expressions régulières), qui permet de capturer des itérations d'évènements au sein de séquences, dont la taille est indéterminée *a priori*.

Une séquence *Kleene* peut, de la même manière qu'un évènement, être référencée dans la clause `WHERE`. Pour manipuler ces séquences, *Sase+* ajoute des accesseurs adaptés, permettant notamment d'accéder aux évènements d'une séquence *Kleene* par leur index, ainsi que des fonctions d'agrégation pour agréger les évènements de la séquence.

Cependant, *Sase+* ne permet pas de spécifier des négations dans les règles. Il ne permet pas non plus de définir un intervalle de temps maximum entre deux évènements.

2.2.3 Le langage *TESLA*

TESLA [Cugola et Margara, 2010] est le langage de spécification de règles présenté pour le système CEP *T-Rex* [Cugola et Margara, 2012a]. La structure d'une règle *TESLA* est similaire à celle de *Sase*, et s'inspire elle aussi du langage *SQL*.

Une règle *TESLA* est composée de plusieurs clauses. La première clause, `define`, permet de définir tous les évènements qui constituent l'évènement complexe à détecter.

La clause `from` définit les contraintes temporelles entre les évènements. En outre, il n'est possible que de définir des contraintes séquentielles entre les évènements. Les opérateurs logiques ne sont pas supportés mais il est possible de spécifier des négations pour exclure des évènements. Il est également possible de définir des fenêtres temporelles via le mot clé `Within`. Enfin, il est possible de filtrer et d'agréger des évènements en fonction de leurs propriétés. *TESLA* propose notamment des fonctions prédéfinies telles que Count ou Avg.

La clause `where` décrit quant à elle comment calculer les valeurs des attributs de l'évènement complexe (défini dans la clause `define`), à partir des attributs des évènements détectés (défini dans la clause `from`).

Enfin, la clause `consuming`, optionnelle, permet de spécifier la stratégie de consommation, un paramètre de l'algorithme de détection dont nous parlons plus en détail dans la section suivante.

Les principales limitation de *TESLA* sont donc qu'il ne propose pas d'opérateurs logiques de conjonction et de disjonction, et qu'il ne propose pas non plus d'opérateur d'itération.

2.2.4 Le *Cayuga Event Language*

Cayuga Event Language (*CEL*) est le langage de spécification de règles présenté avec le système de monitoring *Cayuga* [Demers *et al.*, 2006].

Contrairement aux systèmes CEP précédemment décrits, *Cayuga* fait en sorte que les évènements puissent posséder deux timestamps : un timestamp de début et un timestamp de fin. Cela permet ainsi de représenter des évènements qui durent dans le temps et pas seulement des évènements instantanés. Le flux *Cayuga* ordonne les évènements par rapport au timestamp de fin.

CEL adopte une syntaxe proche de *SQL*. Les règles sont elles aussi constituées de plusieurs clauses.

La clause `SELECT`, optionnelle, permet de définir l'évènement complexe qui sera retourné par le système. Cet évènement peut être constitué des évènements identifié dans le flux ou même construit en transformant les informations venant du flux.

La clause `PUBLISH` permet d'identifier et de nommer un flux de sortie pour la publication des évènements complexes. Ce flux peut être à son tour consommé par d'autres règles, créant ainsi des pipelines.

Au coeur d'une règle *CEL* se trouve la clause `FROM`. Elle spécifie via un ensemble d'opérateurs les contraintes à respecter pour détecter l'évènement complexe. L'opérateur `FILTER` permet notamment de sélectionner un évènement en fonction de ses attributs. L'opérateur `NEXT` est l'opérateur de séquence de *CEL*: il permet de capturer une paire d'évènements consécutifs dans le temps, qui peuvent soit provenir du même flux, soit de deux flux différents. Un prédicat peut être associé à cet opérateur pour définir des contraintes de sélection à satisfaire au niveau du second évènement. L'opérateur `FOLD` est la version itérative du précédent opérateur : il permet de détecter des séquences d'évènements composées du même type d'évènement, se répétant un nombre quelconque de fois. Plusieurs paramètres peuvent être associés à cet opérateur : un prédicat de sélection équivalent au paramètre de `NEXT`, un prédicat d'arrêt de l'itération, et éventuellement une fonction d'agrégation à exécuter à chaque pas de l'itération.

L'expressivité de *CEL* est équivalente à celle de l'algèbre *Cayuga*, qui est formellement définie par les auteurs. Les principale limitation de *CEL* est qu'il ne permet pas la spécification de fenêtres temporelles: il n'est donc pas possible de limiter le champ d'application d'une règle dans le temps. *CEL* ne propose pas non plus de spécifier des négations dans les règles.

2.2.5 Le *Siddhi Query Language*

Le langage *Siddhi Query Language* est présenté avec le système *Siddhi* [Suhothayan *et al.*, 2011], dont le périmètre inclut mais dépasse le cadre strict du CEP, puisqu'il permet également la manipulation de flux. Nous décrivons ici uniquement les aspects du langage liés à la spécification de règles de détection.

La syntaxe de ce langage est basée sur *SQL*, d'où le nom de *Siddhi Query Language* (*SiddhiQL*). Les évènements *Siddhi* sont typés : l'utilisateur définit au préalable le schéma des flux en entrée dans la clause `define stream`.

Une règle *SiddhiQL* commence par une clause `FROM` dans laquelle l'évènement complexe à détecter est spécifié. Pour cela, plusieurs opérateurs peuvent être utilisés. L'opérateur de séquence non-contigu (symbole `->`) prend en compte l'ordre d'occurrence des évènements, mais n'impose pas la contiguïté des évènements dans le flux. Les évènements non reconnus sont dans ce cas simplement ignorés. L'opérateur de séquence contigu (symbole `,`) est la version contiguë de l'opérateur précédent, et impose donc que les évènements de la séquence se suivent directement dans le flux (l'évaluation de la règle est alors invalidée dés qu'un évènement non reconnu est capté). Les opérateurs logiques (`and` et `or`) permettent de spécifier des combinaisons d'évènements sans prendre en compte leur ordre d'occurrence. Pour spécifier l'itération d'un type d'évènement donné, *SiddhiQL* permet, dans le cas non-contigus, de suffixer à un évènement une construction indiquant combien de fois l'évènement doit se répéter consécutivement dans la séquence. Dans le cas contigus, *SiddhiQL* permet d'utiliser les opérateurs classiques d'expressions régulières (`*`, `+` et `?`) pour définir des itérations. *SiddhiQL* permet enfin de définir une fenêtre associée à la règle, via l'opérateur `within`.

La clause `SELECT` permet de sélectionner, et éventuellement de transformer via des fonctions, les attributs des évènements composant l'évènement complexe qui sera retourné. Les évènements individuels qui composent une itération peuvent être accédés via leur index dans la sous-séquence correspondante. En fin de règle, la clause `INSERT INTO` permet de créer un flux de sortie dans lequel l'évènement complexe spécifié dans la clause `SELECT` est inséré. Ce flux de sortie peut être réutilisé en entrée d'une autre règle, car les règles *Siddhi* sont composables récursivement.

Pour partitionner les évènements en fonction d'un ou plusieurs attributs, des partitions de flux peuvent être définies via le mot clé `partition`. Les partitions sont isolées les unes des autres, et la règle s'applique alors séparément et parallèlement pour chaque.

Le langage *SiddhiQL* est un langage très complet supportant un grand nombre d'opérateurs et de fonctionnalités CEP.

2.2.6 Vers un nouveau langage d'expression de règles de détection

Le tableau 2.1 montre un comparatif des cinq langages précédemment décrits par rapport aux opérateurs et fonctionnalités qu'ils proposent. On voit clairement que seul le langage *SiddhiQL* est complet en comparaison des autres langages.

Pour autant, ce langage a été jugé trop complexe par Cdiscount, qui souhaite un langage plus simple, adapté à leurs besoins, et accessible aux utilisateurs non-techniques tels que l'équipe Marketing. Le langage de spécification de règles que nous présentons dans cette thèse, appelé AUROS, répond aux besoins spécifiques de Cdiscount. Nous proposons notre propre syntaxe, qui n'est pas basée sur *SQL*.

TABLEAU 2.1 – Comparaison de 5 langages CEP

Langage	Conj.	Disj.	Séquence	Itération	Agrégations	Négation	Intervalles	Fenêtres
PADRES	✓	✓	✓				✓	
Sase+	✓	✓	✓	✓	✓			✓
TESLA			✓		✓	✓		✓
CEL	✓	✓	✓	✓				
SiddhiQL	✓	✓	✓	✓	✓	✓	✓	✓

Au niveau des fonctionnalités de détection, AUROS s'inspire de l'ensemble des systèmes précédemment décrits. Il propose notamment les opérateurs logiques, un opérateur de séquence, un opérateur d'itération (inspiré de *Kleene Plus* de *Sase+*), ainsi que des négations.

AUROS permet également d'associer une fenêtre aux règles de détection, et de contraindre temporellement les détections en spécifiant les intervalles de temps maximum (ou minimum) entre les évènements, même s'ils ne se suivent pas directement (comme cela est le cas avec *PADRES*). Enfin, le flux d'entrée peut être partitionné par rapport à une clé de partitionnement, qui est composée d'un ou plusieurs attributs.

Le langage AUROS n'apporte donc pas de nouvelles fonctionnalités de détection par rapport aux langages existants, mais propose néanmoins une expressivité largement en concordance avec ceux-ci. Sa syntaxe correspond aux standards de l'entreprise pour laquelle nous avons conçu ce langage. Le chapitre 3 décrit en détail le langage AUROS.

2.3 Modèles d'évaluation des règles de détection

Cette section présente les travaux de recherche qui proposent des modèles d'exécution des règles de détection sur des flux d'évènements. Tous les modèles d'exécution que nous avons étudiés se basent sur l'automate fini comme modèle sémantique. Les états de l'automate permettent de l'état des séquences en cours d'identification, alors que les transitions permettent de sélectionner les évènements correspondants à la règle de détection.

Les différences entre les approches existantes viennent essentiellement de la façon dont sont exécutés les automates, et de la façon dont ils exploitent les évènements du flux pour identifier des séquences correspondant à la règle de détection sous-jacente. *Cugola et al.* identifient ces différences par la *stratégie de sélection* et et la *stratégie de consommation* mise en place par le modèle d'exécution [Cugola et Margara, 2012b].

— La stratégie de sélection précise quels évènements du flux doivent être sélectionnés lors de l'évaluation d'une règle de détection. Prenons l'exemple d'une règle simple, permettant de détecter un client effectuant une recherche via le moteur de recherche du site Cdiscount, puis consultant une des fiches produit proposées en résultat de cette recherche. Lorsqu'une même recherche est répétée plusieurs fois de suite, plusieurs évènements de type `Recherche` sont émis (ce scénario est typiquement ob-

servé lorsque l'internaute rafraichit la page des résultats de recherche dans son navigateur). Un évènement de type `Consultation` est ensuite émis au moment où le client consulte une fiche produit. Lorsque l'évènement de type `Consultation` suit un ou plusieurs évènements de type `Recherche`, une séquence d'évènements est alors détectée par la règle. Cependant, dans le cas où plusieurs évènements de type `Recherche` précèdent l'évènement `Consultation`, une ambiguïté existe quant à décider quel(s) évènement(s) de type `Recherche` doivent être sélectionnés par la règle. La règle a donc le choix de sélectionner soit seulement le premier évènement `Recherche`, soit seulement le deuxième, soit tous, etc. La stratégie de sélection précise pour une règle donnée la sémantique de sélection d'évènements qui doit être appliquée, et permet ainsi au système CEP de gérer avec cohérence les cas ambigus.

— La stratégie de consommation permet de décider si les évènements sélectionnés par une règle de détection donnée (via la stratégie de sélection) doivent être consommés ou non. Un évènement consommé par une règle ne peut pas être sélectionné par d'autres règles : il est supprimé du flux et est rendu inaccessible par les règles concurrentes. Prenons à nouveau l'exemple de la règle précédente, permettant de détecter une recherche suivie de la consultation d'une fiche produit. Lorsque un client émet un évènement `Recherche` puis un évènement `Consultation`, une séquence est identifiée par la règle. Si la règle consomme ces évènements, ceux-ci disparaissent du flux : un futur évènement `Consultation` n'aura donc aucun effet. En revanche, si la règle ne consomme pas les évènements, ils sont maintenus dans le flux, et un futur évènement `Consultation` sera associé à l'évènement `Recherche` précédent pour détecter une seconde séquence. La stratégie de consommation définit donc l'impact de l'identification d'une séquence sur les potentielles séquences identifiables dans le futur, par l'ensemble des règles.

Les sections suivantes présentent les modèles d'exécution de règles de détection qui sont employés par des systèmes CEP existants. Nous montrons que le processus consistant à compiler les règles de détection vers des modèles d'automates finis correspondants est prédominant dans l'état de l'art. Nous avons fait le choix de présenter les modèles *Sase+*, *T-Rex* et *Cayuga* car ils sont décrits avec précision dans la littérature existante. Les systèmes CEP *Siddhi* et *PADRES*, dont nous avons étudié les langages dans la section précédente, utilisent également des automates pour l'exécution des règles de détection, mais ne sont pas aussi précis dans la description des modèles spécifiques qu'ils utilisent.

2.3.1 Le modèle *Sase+*

Sase+ [Agrawal *et al.*, 2008] représente chaque règle de détection par un modèle d'automate fini non-déterministe appelé *NFAb*. *NFAb* est couplé à une structure de données spécifique, le *matchbuffer*, servant à mémoriser les évènements sélectionnés dans le flux lors de l'évaluation de la règle par le modèle.

Les états et les transitions de l'automate *NFAb* correspondent aux évènements et aux contraintes définies dans la règle de détection. L'état initial de l'automate marque le début de l'évaluation de la règle, et l'état final marque l'identification d'une séquence. Entre ces deux états, un ensemble d'états et de transitions marquent les étapes intermédiaires de la détection. Plus spécifiquement, chaque transition du modèle *NFAb* porte une formule logique permettant de déterminer, pour un état donné, si un évènement doit être sélectionné ou non par la règle.

Lorsqu'un évènement est sélectionné, il est mémorisé dans le *matchbuffer* (action `take`). Pour les évènements non sélectionnés, l'action réalisée par le modèle dépend de la stratégie d'évaluation définie par l'utilisateur au niveau de la règle. *Sase+* propose une stratégie d'évaluation dite *contiguë*, qui impose que les évènements d'une règle se suivent directement dans le flux d'évènements, c'est-à-dire sans aucun autre évènement entre eux. Si, à tout instant, un évènement non reconnu par la règle apparait dans le flux, alors l'évaluation de la règle en cours est invalidée, et l'automate *NFAb* réinitialisé.

Sase+ propose également une stratégie d'évaluation dite *non contiguë*, qui contrairement à la stratégie précédente, permet de tolérer les évènements du flux non reconnus par la règle. Ces évènements sont alors simplement ignorés, c'est-à-dire qu'ils ne sont pas mémorisés dans le matchbuffer (action `ignore`), et *NFAb* reste dans son état courant, en attente de l'évènement suivant.

Agrawal et al. décrivent les différentes étapes du processus de compilation qui permettent de traduire une règle de détection vers un modèle *NFAb* correspondant. Cette compilation permet notamment de déterminer le nombre d'états et de transitions du modèle, ainsi que les formules logiques qui sont associées à chaque transition, en fonction de l'ensemble des conditions et de la stratégie de sélection associées à la règle.

Le modèle *NFAb* de *Sase+* est non-déterministe car les formules logiques associées aux transitions provenant d'un même état ne sont pas nécessairement mutuellement exclusives. Cette nature non-déterministe peut alors engendrer de nombreuses branches d'exécution de l'automate simultanées. Ces branches d'exécution correspondent chacune à une séquence en cours d'identification (cf. définition d'une séquence dans la section 2.1), et sont constituées d'un *matchbuffer* et de leur état courant dans l'automate *NFAb*.

La complexité de l'évaluation du modèle *NFAb* s'exprime ainsi par le nombre maximum théorique de séquences qu'il peut engendrer pour une règle donnée. Il est montré que selon la stratégie de sélection adoptée par la règle, le nombre de séquences peut être exponentiel par rapport à la taille de la fenêtre temporelle sur laquelle la règle s'applique. Le rôle de cette fenêtre temporelle est de limiter le nombre de séquences en cours à maintenir dans le système, puisque les séquences expirées sont supprimées.

Cependant, afin de contrebalancer les effets néfastes du non-déterminisme dus à la multiplication des séquences qui doivent être maintenues en mémoire par le système, l'article propose un algorithme d'optimisation. Celui-ci consiste à fusionner les parties des *matchbuffers* qui sont communes à plusieurs séquences, afin d'éviter la duplication des

évènements maintenus en mémoire (et ainsi limiter la consommation mémoire du système CEP de manière générale).

Sase+ propose donc un modèle d'exécution non-déterministe, et une stratégie de sélection programmable par l'utilisateur. La stratégie de consommation est elle non-programmable : *NFAb* ne consomme jamais les évènements sélectionnés, qui peuvent donc être évalués par plusieurs règles concurrentes.

2.3.2 Le modèle *T-Rex*

T-Rex [Cugola et Margara, 2012a] utilise également un modèle d'automate non-déterministe pour évaluer les règles de détection *TESLA*. *Cugola et al.* expliquent le choix de l'automate par sa faible latence de détection, qui est due à sa nature incrémentale : les évènements sont traités au fil de l'eau par l'automate, garantissant une faible latence, ce qui est un aspect crucial pour *T-Rex*. Comme le modèle *Sase+*, le modèle *T-Rex* permet de mémoriser les évènements sélectionnés dans une structure de données dédiée associée à chaque automate. L'automate *T-Rex* possède un état d'initialisation où une séquence en cours d'identification commence, un état final où la séquence est complète (autrement dit où la règle est détectée), et un ensemble d'états et de transitions correspondant aux états intermédiaires de la séquence.

Le processus de compilation d'une règle *TESLA* vers son modèle d'automate correspondant consiste à décomposer la règle en un ensemble de séquences d'évènements indépendantes, qui sont chacune traduites en un modèle de séquence. Les modèles de séquence sont des automates déterministes dont l'ensemble des états et transitions permettent de reconnaître la séquence d'évènements correspondante. C'est ce processus de compilation qui permet d'associer aux transitions des prédicats, qui déterminent les évènements à sélectionner pour former une séquence. Les automates de séquence sont ensuite combinés afin d'obtenir l'automate non-déterministe final, qui représente la règle dans son ensemble.

L'évaluation non-déterministe du modèle *T-Rex* consiste à initialiser une nouvelle séquence au déploiement de la règle, puis à cloner systématiquement les séquences lorsqu'elles sélectionnent un évènement. Ainsi, le nombre de séquences maintenues par le système pour une règle donnée est exponentiel par rapport à la taille de la fenêtre spécifiée par la règle.

Les stratégies de sélection et de consommation employées par le modèle *T-Rex* sont toutes deux programmables par l'utilisateur.

2.3.3 Le modèle *Cayuga*

Cayuga [Demers *et al.*, 2006] modélise également les règles de détection par des automates non-déterministes. Les travaux de *Demers et al.* ont montré la compatibilité des

automates non-déterministes avec l'algèbre de *Cayuga*, sur lequel le langage *CEL* précédemment décrit est basé.

Ainsi, toute règle de détection *CEL* peut être traduite directement vers un automate *Cayuga* correspondant. Un automate *Cayuga* possède un état initial dans lequel l'évaluation de la règle commence, un état final dans lequel la règle est effectivement détectée, ainsi qu'un ensemble d'états et de transitions intermédiaires. Les évènements sélectionnés pendant l'évaluation sont mémorisés au niveau des états de l'automate, qui possèdent chacun une structure de données dédiée.

Plusieurs types de transition sont définis : les transitions permettant d'accéder à l'état suivant, et celles permettant de rester dans l'état courant de l'automate (nommées *transitions-boucles*). Ces dernières sont de deux types : les *transitions-filtres* et les *transitions d'itération*. Les *transitions-filtres* correspondent aux opérateurs `NEXT` et `FOLD`, et permettent de sélectionner via une formule logique les évènements pertinents, ou d'ignorer les évènements non sélectionnés. Les *transitions d'itération* correspondent exclusivement à l'opérateur `FOLD`, et servent à réécrire la formule d'itération à chaque pas, afin de prendre en compte le dernier évènement sélectionné de l'itération.

La stratégie de sélection utilisée est une stratégie de sélection multiple : tous les évènements correspondant à la règle sont systématiquement sélectionnés. Par ailleurs, le modèle *Cayuga* ne consomme jamais les évènements sélectionnés.

2.3.4 Vers un modèle d'évaluation des règles de détection paramétrable

Comme nous venons de le voir, la compilation des règles de détection vers un modèle à automate est l'approche prédominante dans le domaine du CEP. En effet, les automates se prêtent bien à la sémantique de l'évaluation des règles de détection. Notre approche est donc également basée sur un modèle d'automate. AUROS propose néanmoins plusieurs stratégies d'évaluation du modèle d'automate, qui découlent des concepts de stratégie de sélection et de consommation. En fonction de la stratégie spécifiée pour une règle, l'automate AUROS est évalué par un algorithme de détection différent, qui permet à l'utilisateur de configurer l'expressivité et l'efficacité de la règle. Le chapitre 4 présente en détail le modèle d'automate AUROS et les différents algorithmes d'évaluation qui correspondent aux différentes stratégies.

2.4 Architecture du système CEP

L'architecture d'un système CEP, en permettant l'interaction efficace entre les différents composants de l'environnement d'exécution, est un des facteurs majeurs de la performance de détection en temps réel. Selon le type d'architecture employé, les systèmes CEP peuvent être conçus pour traiter des volumes de données plus ou moins importants, et sont plus ou moins compatibles avec le concept de passage à l'échelle. Nous distinguons

essentiellement les architectures centralisées des architectures distribuées, cette dernière catégorie pouvant être divisée en deux sous-catégories : les architectures distribuées dites *éparses*, et les architectures distribuées dites *clusterisées*.

Pour chaque catégorie, nous présentons dans cette section des architectures et environnements d'exécution employés par plusieurs systèmes CEP existants.

2.4.1 Architectures centralisées

Les architectures centralisées, comme leur nom l'indique, dépendent d'un unique nœud de calcul pour faire fonctionner l'ensemble des composants qui constituent le système CEP. Le potentiel de passage à l'échelle de ces architectures est donc limité, car les performances du systèmes sont bornées par rapport aux ressources (essentiellement CPU et *RAM*) disponibles dans le seul nœud central. Ces architectures sont donc plus adaptés aux scénarios à échelle modérée, où la puissance de calcul d'un seul noeud est suffisante, en particulier pour maintenir et mettre à jour les séquences en cours d'identification maintenues par l'ensemble des règles déployées.

De plus, les architectures centralisées dépendent par définition d'un seul point de défaillance, et sont donc susceptibles aux pannes, qui sont un réel problème en particulier dans le monde de l'industrie.

Sase

L'architecture de *Sase* [Gyllstrom *et al.*, 2006] est organisée en couches successives, toutes déployées au sein d'un unique nœud de calcul central. La première couche, située au niveau le plus bas, est constituée par les sources de données à l'origine du flux d'évènements (par exemple, un ensemble d'antennes *RFID*). La deuxième couche a comme fonction le filtrage et le préformatage des évènements. La troisième couche, qui constitue le cœur du système, contient l'*Event Processor*. Celui-ci est chargé d'évaluer les règles de détection, et d'émettre les évènements complexes correspondants dans le flux de sortie. Les évènements complexes peuvent à ce moment être archivés dans une base de données relationnelle intégrée au système, appelée *Event Database*.

Les règles de détection sont évaluées continuellement par l'*Event Processor*, de leur déploiement jusqu'à leur suppression explicite par l'utilisateur. L'*Event Processor* est aussi capable d'intégrer, en plus des évènements provenant du flux d'évènements lui-même, des données supplémentaires provenant de sources statiques externes, comme une base de données d'archivage. Ces données externes peuvent être utilisées pour enrichir les évènements complexes produits. Pour cela, *Sase* requête la base de données externe, récupère les résultats de cette requête, puis effectue une jointure avec les données contenues dans l'évènement complexe pour créer la notification finale émise en sortie du système. Dans ce cas, le processus induit cependant une certaine latence au niveau de la production des évènements complexe.

Sase est implémenté en *Java*, et est conçu comme une architecture client-serveur. La partie serveur est centrée sur l'*Event Processor*, alors que ce sont les sources de données (en entrée du système), les collecteurs de données (en sortie du système) et les utilisateurs qui agissent comme les clients. *Sase* propose d'ailleurs une interface graphique simple destinée aux utilisateurs du système, permettant de spécifier les règles et de requêter directement l'*Event Database*.

Bien que des mécanismes d'optimisation agissent au niveau des différentes structures de données de l'*Event Processor*, notamment pour limiter la consommation de mémoire, l'absence d'architecture distribuée fait de ce composant le goulot d'étranglement du système. En effet, si les ressources qui lui sont allouées ne sont pas suffisantes pour gérer un débit du flux d'évènements trop important, alors le système dans son ensemble n'est plus en mesure de fonctionner.

T-Rex

Dans l'architecture de *T-Rex* [Cugola et Margara, 2012b], le *Rule Manager* maintient l'ensemble des règles déployées, qu'il est chargé de compiler en modèles d'automate correspondants. Les évènements en entrée sont consommés dans une structure de données FIFO (*First In First Out*), et sont évalués par un *Static Index*. Le rôle de celui-ci est d'indexer les états courants de l'ensemble des séquences en cours d'identification, et de déterminer en temps réel celles qui doivent évaluer un évènement donné. Si l'évènement en question ne correspond à aucune séquence, alors il est simplement ignoré par le système. Sinon, l'évènement est envoyé vers les séquences correspondantes, qui le sélectionnent ou non, et sont mises à jour le cas échéant.

Les groupes de séquences maintenus pour chaque règle sont traités en parallèle via l'utilisation de *threads*, qui permettent de mieux répartir les calculs au niveau des CPU disponibles dans l'unique noeud de calcul.

Le composant *Stored Events* permet de mémoriser, en éliminant les duplications, l'ensemble des évènements sélectionnés par les séquences en cours d'identification. Ces séquences pouvant expirer à cause des fenêtres temporelles définies au niveau des règles, il peut arriver qu'un évènement mémorisé dans *Stored Events* ne corresponde plus à aucune séquence en cours. Dans ce cas, l'évènement est supprimé de *Stored Events*.

Lorsqu'une séquence est identifiée, le *Generator* récupère automatiquement l'ensemble des évènements associés à la séquence dans le *dans le Stored Events*, et génère à partir de cet ensemble d'évènements un évènement complexe.

Enfin, les évènements complexes produits sont passés au *Subscription Manager*, qui se charge de distribuer ceux-ci aux clients intéressés via un flux de notifications.

T-Rex est implémenté en *C++*, selon une architecture client-serveur. Les composants du système CEP sont encapsulés dans le serveur, et les sources de données, collecteurs de données et utilisateurs ont le rôle de clients. Ces derniers se connectent au système via des *Adapters*, qui exposent des *API* compatibles avec les langages *C++* et *Java*.

Des mécanismes d'optimisation ont été implémentés dans *T-Rex*, notamment en ce qui concerne le *Stored Events*, afin de limiter la consommation de ressources du système. Mais, comme cela est le cas pour *Sase*, l'absence d'architecture distribuée ne prédispose pas *T-Rex* à des scénarios à grande échelle.

Cayuga

Dans l'architecture *Cayuga* [Demers *et al.*, 2006], les évènements en entrée sont consommés par les *Event Receivers*. Ces derniers peuvent associer une date aux évènements, dans le cas où les dates d'occurrence ne seraient pas déjà spécifiées. Chaque *Event Receiver* est dédié à une source de données, et est géré par un *thread* spécifique. Ils transfèrent les évènements qu'ils reçoivent à la *Priority Queue*, dans laquelle le *Cayuga Query Engine*, au cœur de l'architecture *Cayuga*, récupère les évènements par ordre chronologique. Un algorithme permet à *Cayuga* de corriger les anomalies dues aux éventuels délais réseau et autres désynchronisations pouvant avoir lieu entre les différentes sources de données.

Le rôle du *Cayuga Query Engine* est de maintenir et mettre à jour les séquences en cours d'identification. Pour cela, chaque évènement passe dans une série d'*Evaluators*, dont le rôle est d'évaluer l'évènement par rapport aux formules spécifiées au niveau des transitions de l'automate. L'*Evaluator* est optimisé pour exécuter de manière la plus efficace l'algèbre *Cayuga*. Lorsqu'une séquence est identifiée dans le flux par un automate *Cayuga*, l'évènement complexe produit peut soit être réintroduit au niveau de la *Priority Queue* (récursivité des règles), soit transféré vers les *Client Notifiers*.

Demers et al. ont implémenté des mécanismes complexes d'optimisation des structures de données, qui peuvent notamment être partagées entre plusieurs modèles d'évaluation de règles ayant des parties de spécification communes. Par ailleurs, la gestion de la mémoire de *Cayuga* est optimisée par un *Garbage Collector* spécifique, adapté aux créations et destructions fréquentes d'objets (en particulier pour les séquences en cours d'identification et les évènements, dont les durées de vie sont généralement éphémères). Cette approche permet au système d'être particulièrement efficace dans un environnement non distribué, mais l'absence d'architecture distribuée pose toutefois les mêmes limitations que pour les systèmes précédents.

2.4.2 Architectures distribuées éparses

Les architectures distribuées, contrairement aux architectures centralisées, prennent place sur un ensemble de nœuds de calcul. Elles ont l'avantage de pouvoir utiliser davantage de ressources, leur permettant ainsi de gérer des scénarios à grande échelle. De plus, les architectures distribuées ne dépendent pas d'un unique noeud pour sa disponibilité, et sont donc plus tolérants aux pannes.

Les architectures distribuées éparses sont un type particulier d'architecture distribuée. Leur but est d'optimiser les liens de communication entre différentes sources de données, qui peuvent être éloignées géographiquement les unes des autres, et les noeuds de calcul du système. Pour cela, les noeuds de calcul opèrent à proximité des sources, et donc des flux d'évènements auxquels ils sont assignés. Les liens de communication sont donc rendus efficaces entre un flux d'évènements et son noeud de calcul correspondant, au détriment de la communication entre les différents noeuds de calcul du système.

PADRES

PADRES [Li et Jacobsen, 2005] est un système *publish-subscribe* distribué, permettant de détecter des évènements complexes dans des flux d'évènements épars. Les évènements complexes produits sont ensuite diffusés via des notifications aux souscripteurs intéressés. *PADRES* est donc optimisé pour gérer la dispersion géographique des sources d'évènements et des souscripteurs, et vise à réduire au maximum la latence entre souscripteurs et flux.

Pour cela, le système se base sur un réseau de *brokers*, qui sont les noeuds de calcul dans l'architecture *PADRES*. Ce réseau est décrit dans une table de routage appelée *Overlay Routing Table* (*ORT*), qui permet essentiellement à chaque *broker* de connaître ses *brokers* voisins. Par ailleurs, la *Subscription Routing Table* (*SRT*) permet aux demandes de souscription de parvenir aux *brokers* les plus à même de les gérer. Une troisième table de routage, la *Publication Routing Table* (*PRT*), permet de router les évènements complexes vers les souscripteurs intéressés. Ce réseau permet de mettre efficacement en relation les flux d'évènements et les souscripteurs, qui sont des entités potentiellement éloignées géographiquement.

Individuellement, chaque *broker* est composé de plusieurs structures FIFO : une file en entrée gérant le flux d'évènements à traiter, et un ensemble de files en sortie, permettant chacune de router les évènements complexes vers une destination spécifique. Au coeur d'un *broker* se trouve le *Broker Core*, chargé d'identifier des séquences correspondant aux règles, et basé sur le moteur de règles *Java JESS*.

L'avantage principal de *PADRES* est que l'évaluation des règles est automatiquement distribuée sur plusieurs *brokers*. Pour cela, les règles sont décomposés en sous-règles, qui sont attribuées à différents *brokers* du réseau, choisis vis-à-vis de leur proximité géographique par rapport aux flux d'évènements à l'origine des évènements qui composent les sous-règles. Les sous-règles détectent des séquences partielles sur les différents *brokers*, qui peuvent ainsi être routées vers d'autres *brokers* pour reconstituer la séquence complète. Une fois l'ensemble de la séquence reconstituée, un évènement complexe est notifié aux souscripteurs.

L'architecture de *PADRES* est flexible car elle permet à la fois d'optimiser les liens entre les *brokers* et les flux d'évènements, tout en proposant un algorithme d'évaluation de règles distribué sur un réseau de *brokers*. Toutefois, cette distribution est basée sur la localisation

géographique des flux qui alimentent la règle, et non pas par rapport à la structure de la règle elle-même. En particulier, une règle complexe dont tous les évènements proviennent du même flux ne sera pas distribué par *PADRES*. *PADRES* ne propose donc pas de solution efficace pour gérer un unique flux d'évènements massif, dont le débit est trop important pour être traité par un seul *broker*.

GEM

GEM [Mansouri-Samani et Sloman, 1997] est un système CEP conçu pour le *monitoring* d'entités pouvant être géographiquement éloignées les unes des autres. L'architecture de *GEM* est composé d'un ensemble d'*Event Monitors*, assignés à un ensemble d'*Event Generators*. Le rôle des *Event Generators* est d'observer (*monitorer*) une ou plusieurs entités, et de produire des flux d'évènements décrivant l'activité de ces entités. Un *Event Monitor* analyse les évènements provenant d'un *Event Generator*, et détecte les règles spécifiés dans un *script de monitoring*. L'*Event Monitor* produit en sortie des évènements complexes correspondants aux séquences identifiées, qui sont envoyés à l'*Event Disseminator*. Enfin, ce dernier diffuse les évènements complexes vers l'ensemble des souscripteurs, et éventuellement vers une base de données d'archivage.

GEM supporte les règles composites : les évènements complexes produits par un *Event Monitor* peuvent être évalués par des règles définies par d'autres *Event Monitors*, qui à leur tour peuvent produire de nouveaux évènements complexes, etc.

L'architecture de *GEM* est implémentée avec le framework *REGIS* [Magee *et al.*, 1994], en *C++*.

L'approche du système CEP *GEM* permet l'agrégation de flux d'évènements géographiquement éparses, tout en distribuant la détection au niveau des différents flux d'évènements. Cependant, le traitement d'un même flux reste centralisé au niveau de l'*Event Monitor*, impliquant que ce composant doit disposer d'assez de ressources pour assurer la détection de l'ensemble des règles spécifiées pour un flux donné. Autrement dit, *GEM* est conçu pour l'analyse de multiples flux aux débits modérés, et typiquement éloignés géographiquement les uns des autres. Mais *GEM* n'est pas adapté aux scénarios où le système doit évaluer un seul flux d'évènements massif, dont le débit est trop important pour être géré au niveau d'un seul noeud de calcul.

2.4.3 Architectures distribuées clusterisées

Lorsque les flux d'évènements sont géographiquement proches les unes des autres, et proches du système CEP lui-même (ce qui est fréquemment le cas dans les entreprises), les architectures distribuées *clusterisées* sont les mieux adaptées. Celles-ci sont conçues pour optimiser les liens de communication entre les différents noeuds de calcul, en les organisant en *cluster*, c'est-à-dire au plus proche les uns des autres dans un même réseau informatique. Ainsi, contrairement aux architectures *éparses*, les noeuds de calcul d'une

architecture *clusterisée* peuvent efficacement se distribuer une même charge de travail. En effet, les latences dues aux communications entre les différents noeuds de calcul sont alors minimisées, améliorant les performances globales du système CEP. Notons que la majorité des systèmes commerciaux destinés aux entreprises utilisent ce type d'architecture [Cugola et Margara, 2012b], car il s'applique à un grand nombre d'applications réelles.

NextCEP

L'architecture de *NextCEP* [Schultz-Møller *et al.*, 2009] comprend un *Central Manager* et un ensemble d'*Operator Nodes*. Comme leur nom l'indique, les *Operator Nodes* permettent de déployer les opérateurs qui composent les règles de détection spécifiées. Pour cela, au coeur du *Central Manager* se trouve l'*Engine*, chargé de distribuer les différents opérateurs aux *Operator Nodes* disponibles. Ces derniers sont ensuite directement connectés au flux d'évènements et aux souscripteurs.

Dans un *Operator Node*, chaque opérateur déployé est modélisé par un automate. Les automates disséminés dans les *Operator Nodes* peuvent communiquer entre eux, via une chaîne de FIFO, afin de transférer les évènements sélectionnés par un opérateur vers l'opérateur suivant, et ainsi de suite. Cette chaîne d'automates permet permet ainsi l'évaluation d'une règle complète. Les évènements complexes produits par un *Operator Node* peuvent ensuite directement être routés aux souscripteurs.

NextCEP distribue donc la charge de travail sur le *cluster* par rapport à la structure de la règle elle-même. Lorsque plusieurs règles sont déployées dans le système, chaque noeud de calcul exécute en parallèle des opérateurs provenant de règles différentes.

Cette architecture permet à *NextCEP* de pouvoir passer à l'échelle, notamment dans les cas où le débit du flux d'évènements est trop important pour être traité par un seul noeud de calcul. En particulier, *NextCEP* est conçu pour des scénarios tels que la détection de fraude, où de nombreuses séquences peuvent être engendrées pour une même règle, notamment du fait du nombre important de cartes de crédit différentes, qui chacune correspondent à une séquence distincte.

Siddhi

L'architecture de *Siddhi* [Suhothayan *et al.*, 2011] est composée d'un pipeline de *Processors*, interconnectés par des FIFO. Les files en entrée d'un *Processor* permettent de gérer un flux d'évènements, et les files en sortie sont utilisées pour router les évènements complexes produits vers les souscripteurs. Un *Processor* représente une partie d'une règle de détection que *Siddhi* a préalablement décomposée en opérateurs distincts. Ainsi, comme avec *NextCEP*, l'évaluation des règles peut être parallélisée sur un ensemble de *Processors* déployés sur des noeuds de calcul du *cluster*.

En partant du principe que différentes règles de détection ont fréquemment des parties en commun, *Siddhi* a mis en place un algorithme capable de détecter ces duplications et de

les fusionner, diminuant ainsi le nombre total de *Processors* à maintenir dans le système. En interne, un *Processor* est composé d'un ensemble d'*Executors*, chacun exprimant par un arbre de décision une condition associée à un opérateur particulier. Un *Executor* évalue une condition pour chaque évènement du flux, et relaie l'évènement à l'*Executor* suivant s'il est sélectionné ; sinon, l'évènement est ignoré.

2.4.4 Vers une architecture distribuée orientée Big Data

Notre système CEP doit être conçu pour répondre à un scénario d'entreprise, dans lequel un flux d'évènements massif, produit par le site Cdiscount doit être traité efficacement pour détecter un ensemble de règles. De plus, ce flux étant généré par l'activité des clients sur le site ecommerce, son débit est particulièrement variable, et les ressources du système doivent donc pouvoir s'adapter à la charge de travail requise à un instant donné. Pour ce type de scénario, les architectures centralisées ne sont pas adaptées car les ressources d'un seul noeud de calcul sont insuffisantes. Par ailleurs, les architectures distribuées *éparses* sont adaptées au traitement d'une multitude de flux provenant de sources éparses, ce qui n'est pas le cas à Cdiscount. Une architecture *clusterisée* est donc le choix le plus pertinent, car elle permet de distribuer les traitements du flux d'évènements massif sur un cluster contenant plusieurs noeuds de calcul. Il existe cependant plusieurs façons de distribuer les calculs sur un cluster, et différentes architectures clusterisées apportent chacune des garanties spécifiques.

Pour répondre à nos problématiques décrites précédemment, en particulier le passage à l'échelle, ainsi que de la tolérance aux pannes, nous avons fait le choix, pour l'implémentation d'AUROS, de nous orienter vers le paradigme dit *Big Data*. Le terme *Big Data* fait à l'origine référence à un ensemble de techniques permettant de traiter des volumes de données trop importants pour être traités par des systèmes de gestion des données classiques [Ward et Barker, 2013], notamment les bases de données relationnelles. Pour traiter des volumes de données massifs, le paradigme *Big Data* propose une architecture et un modèle de programmation spécifiques, permettant de garantir à la fois le passage à l'échelle et la tolérance aux pannes du système.

Il existe aujourd'hui de nombreuses technologies dites *Big Data*. Dans cette section, nous analysons les principales contributions du domaine *Big Data*, et expliquons le choix d'une telle architecture pour AUROS.

MapReduce et *Hadoop*

Le paradigme précurseur dans le domaine *Big Data* est *MapReduce*, popularisé notamment par *Dean et al.* de *Google* [Dean et Ghemawat, 2008]. Il s'agit d'un paradigme garantissant un passage à l'échelle horizontal et linéaire, c'est-à-dire que la puissance de traitement d'un cluster dépend linéairement du nombre de noeuds de calcul. La particularité de *MapReduce* est la possibilité, pour un cluster exécutant une application quelconque,

d'augmenter au besoin sa capacité de calcul en intégrant simplement de nouveaux noeuds de calcul au cluster, sans avoir à modifier l'application en question. Dans ce cas, on dit que l'application est *scalable*. Ainsi, une architecture *MapReduce* peut mettre à contribution des centaines, voire des milliers de noeuds de calcul pour traiter un volume très important de données [Thusoo *et al.*, 2010b]. Ce concept contraste avec le concept de passage à l'échelle vertical employé pour les architectures centralisées, et consistant à ajouter de plus en plus de ressources à un unique noeud de calcul central afin d'augmenter son potentiel de parallélisation.

MapReduce est donc une architecture, mais également un modèle de programmation spécifique, auquel l'utilisateur doit agréer pour pouvoir développer des applications compatibles. Avant tout, ce modèle de programmation repose sur le format *clé-valeur*. En effet, tout programme *MapReduce* commence par formater un fichier de données, qui est un ensemble d'enregistrements, de telle manière que chaque enregistrement soit composé d'une clé et d'une valeur correspondante. Ce format permet à *MapReduce* de diviser, en fonction de la clé, un fichier volumineux en plusieurs *blocs d'enregistrements* distincts, qui sont ensuite distribués aux différents noeuds de calcul pour être traités en parallèle.

Pour traiter chaque bloc de données, le modèle de programmation *MapReduce* se base sur seulement deux fonctions : la fonction `map` et la fonction `reduce`. La fonction `map` permet de transformer un enregistrement e en un ensemble de paires (k, v) :

$map : e \rightarrow (k_1, v_1), (k_2, v_2), ..., (k_n, v_n)$

La fonction `reduce` permet d'agréger l'ensemble des valeurs correspondantes à une même clé pour calculer un résultat : $(k, v_1), (k, v_2), ..., (k, v_n) \rightarrow (k, v_r)$.

Un programme *MapReduce* est en fait un *pipeline* composée de fonctions `map` et `reduce` chaînées les unes aux autres. En exécutant un tel pipeline, une application *MapReduce* peut ainsi transformer efficacement, de manière distribuée, un fichier de données volumineux pour produire les résultats escomptés.

La figure 2.2 illustre le processus employé par *MapReduce* pour traiter un fichier de données.

Le principal avantage des programmes spécifiés avec ce modèle est donc leur parallélisation automatique : chaque étape (`map` ou `reduce`) qui constitue le programme est traduite en un ensemble de *tâches*, qui sont distribuées par le système aux différents nœuds de calcul.

Le modèle *MapReduce* a également l'avantage d'être tolérant aux pannes, puisque les noeuds de calcul étant totalement indépendants les uns des autres, la panne de l'un d'entre eux pendant le traitement des données n'a pas d'incidence sur le résultat final. En effet, pour éviter les pertes de données dues aux pannes, *MapReduce* effectue systématiquement une sauvegarde sur disque des données intermédiaires calculées par chaque noeud, et ce après chaque phase du *pipeline* constituant le programme.

D'autre part, les auteurs montrent que l'expressivité des programmes *MapReduce*, même si elle est *a fortiori* limitée par l'emploi des seules fonctions `map` et `reduce`, est tout de même suffisante pour résoudre de nombreux problèmes de la vie réelle.

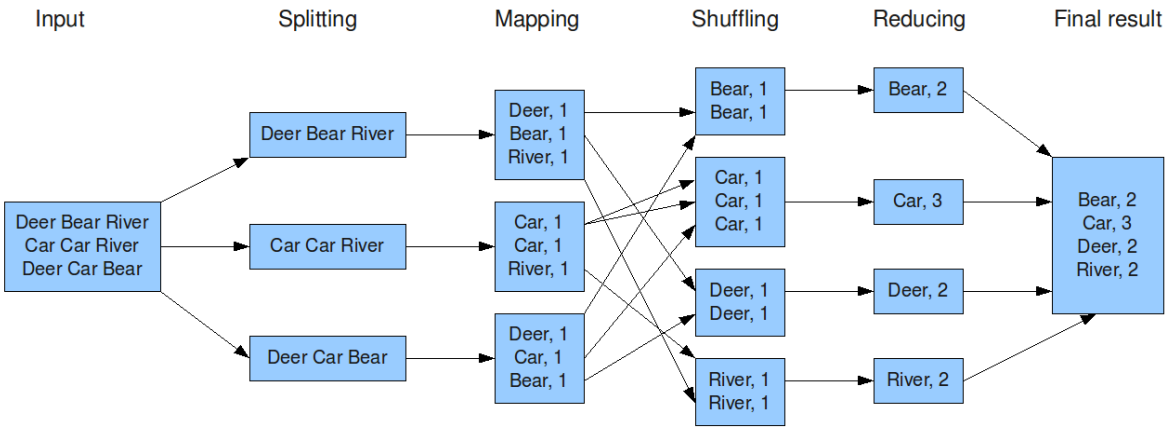

FIGURE 2.2 – Exemple de programme *MapReduce* calculant, pour chaque mot unique, son nombre d'occurrences dans un fichier texte. La première étape consiste à diviser ce fichier en trois blocs, qui sont traités en parallèle par le programme. Celui-ci créé pour chaque enregistrement, via la fonction `map`, un ensemble de paires $(mot, 1)$. Puis, par une opération de `reduce`, le programme calcule la somme des `1` correspondants à une même clé, c'est-à-dire à un même mot. Le résultat final revient à calculer le nombre d'occurrences de chaque mot dans le texte d'origine.

Hadoop [Borthakur *et al.*, 2011] est une implémentation *open-source* du modèle *MapReduce*, qui est aujourd'hui largement utilisée pour traiter des volumes de données importants. *Hadoop* permet de stocker sur disque des fichiers volumineux grâce à système de fichiers distribué nommé *HDFS* [Shvachko *et al.*, 2010], capable de pré-partitionner les fichiers en blocs, et de distribuer ces blocs sur différents noeuds de calcul. Un programme *Hadoop* est un programme *Java*, qui est compilé vers un modèle de graphe directionnel représentant le *pipeline* de fonctions `map` et `reduce` spécifié à exécuter par le système.

Ce modèle, s'il permet le traitement de fichiers volumineux impossibles à traiter par d'autres systèmes et modèles de programmation, a cependant plusieurs inconvénients majeurs :

— Les programmes *MapReduce*, bien que relativement simples si on les compare à d'autres modèles de programmation distribuée (par exemple *MPI*), restent relativement difficiles à spécifier pour l'utilisateur. En particulier, *MapReduce* est difficilement accessible pour des utilisateurs non-techniques tels que l'équipe Marketing de Cdiscount.

— L'implémentation permettant la garantie de tolérance aux pannes a des conséquences non négligeables vis-à-vis de la latence de traitement des données. En effet, le mécanisme de sauvegarde systématique des données intermédiaires sur disque au niveau de chaque phase du programme induit en contrepartie des latences im-

portantes au niveau de chaque phase d'un programme. Ces latences rendent donc toute application interactive, c'est-à-dire exigeant une réponse rapide de la part du système, incompatible avec le modèle.

— L'expressivité du modèle, même si elle est suffisante pour de nombreuses applications, reste limitée car seules les fonctions `map` et `reduce` peuvent être utilisées.

Hive *et* Pig. *Hive* [Thusoo *et al.*, 2010a] et *Pig* [Olston *et al.*, 2008] sont des frameworks proposant notamment des langages de haut-niveau pour spécifier des programmes *MapReduce*. Pour cela, les langages *HiveQL* (sous-ensemble de SQL) et *Pig Latin* permettent à l'utilisateur de spécifier des requêtes de haut-niveau à partir desquelles sont automatiquement générés des pipelines *MapReduce*. Ainsi, il n'est pas nécessaire de programmer manuellement les fonctions `map` et `reduce` au sein d'un programme comme c'est le cas avec *Hadoop*. Cependant, le modèle d'exécution sous-jacent reste *MapReduce*, et les inconvénients du système liés aux latences de traitement ainsi qu'à l'expressivité limitée ne sont donc pas adressés.

Spark

Spark [Zaharia *et al.*, 2010] est un *framework* proposant essentiellement une nouvelle implémentation du paradigme *MapReduce*, adressant l'ensemble des problématiques soulevées précédemment. Tout en restant basé sur le même concept, Spark améliore le modèle *MapReduce* existant en plusieurs points. Tout d'abord, Spark généralise les *pipelines MapReduce* pour supporter toutes sortes de fonctions, et non plus seulement les fonctions `map` et `reduce`, rendant le modèle Spark plus expressif que le modèle *MapReduce*. D'autre part, Spark propose un nouveau modèle de représentation des données appelé *Resilient Distributed Dataset* (*RDD*) [Zaharia *et al.*, 2012]. Le RDD permet notamment de remédier aux problèmes de latences de traitement inhérents à *MapReduce*, car ils peuvent, tout en conservant la garantie de tolérance aux pannes, être traités entièrement en mémoire, supprimant ainsi toute latence induite par des sauvegardes sur disque. Pour cela, Spark maintient de manière fiable un *lineage* pour chaque RDD, qui correspond au pipeline d'opérations exécutées pour ce RDD, et qui peut être à nouveau exécuté en cas de panne pour reconstituer en parallèle les données à partir du RDD de base, conservé en mémoire. Les performances obtenues par Spark lui permettent de supporter le traitement de flux de données, via un modèle appelé *Discretized Stream* (*DStream*) [Zaharia *et al.*, 2013].

L'ensemble de ces fonctionnalités font de Spark un candidat approprié pour les applications interactives [Stonebraker *et al.*, 2005] manipulant de larges volumes de données. Ainsi, bien qu'il ne soit en aucun cas conçu spécifiquement pour le CEP, Spark est une plateforme compatible avec les contraintes que nous avons définies avec Cdiscount pour l'implémentation d'AUROS.

Nous présentons en détail l'architecture et l'environnement d'exécution d'AUROS dans le chapitre 5.

2.5 Synthèse

Nous avons constaté que le domaine du CEP peut être divisé selon trois aspects essentiels : le langage de spécification des règles de détection, le modèle d'exécution de ces règles, et l'architecture sur laquelle se base le système pour fonctionner.

La conception du système CEP AUROS, que nous présentons dans cette thèse, a donc nécessité l'implémentation de chacun de ces trois aspects de façon à répondre aux exigences précises exprimées par l'utilisateur Cdiscount. Dans la suite de ce manuscrit, nous présentons successivement chaque aspect du système AUROS. Tout d'abord, le langage AUROS, présenté dans le chapitre 3, permet d'exprimer aisément des comportements client à identifier dans le flux d'évènements généré par le site Cdiscount. Ensuite, l'automate AUROS présenté dans le chapitre 4 permet d'exécuter efficacement les règles spécifiées, grâce à un mécanisme de compilation permettant de transformer tout règle en un modèle d'automate fini correspondant. Enfin, AUROS fait usage du paradigme Big Data, et plus précisément du modèle de traitement de données Spark, pour garantir le passage à l'échelle du système ainsi que la tolérance aux pannes. Ainsi, comme décrit dans le chapitre 5, AUROS se base sur une architecture et un modèle de calcul distribués pour paralléliser au maximum l'évaluation des règles de détection pour chaque client, sur un flux d'évènements massif.

CHAPITRE

3 Un langage de spécification de règles de détection

Dans ce chapitre nous décrivons le langage AUROS permettant d'exprimer des règles de détection. Nous décrivons les différents opérateurs proposés, qui permettent de sélectionner, de filtrer et de combiner les évènements d'une règle. Pour illustrer chaque opérateur, nous donnons des exemples issus de réels cas d'utilisation Cdiscount.

Sommaire

3.1 Script de spécification des règles de détection

Le langage AUROS permet de définir des règles de détection qui seront évaluées sur un flux d'évènements en entrée. Un *script* AUROS est composé de plusieurs *règles de détection*.

Chaque règle de détection a un nom unique qui doit figurer en tête de la règle. Ce nom peut être référencé par les autres règles du script, afin de pouvoir composer les règles. Le nom d'une règle est suivi de la spécification de la règle, qui comporte deux parties. Dans la première partie de la spécification, appelée *séquence*, on sélectionne les évènements intéressants et on définit leur ordre d'occurrence. Dans la seconde partie de la spécification, qui est optionnelle, on précise un ensemble de *contraintes* de relation entre les évènements (filtres, prédicats, etc.).

Le script 3.1 présente un script AUROS constitué de deux règles.

LISTING 3.1 – Exemple de script AUROS déclarant deux règles `Rule1` et `Rule2`. Une règle est spécifiée par une séquence d'évènements, suivie d'un ensemble de contraintes préfixées du caractère #.

```
Rule1 :
    <sequence>
    # <contrainte 1>
    # <contrainte 2>

Rule2 :
    <sequence>
    # <contrainte>
```

3.2 Sélection des évènements

3.2.1 Atomes

Chaque évènement du flux d'évènements est notamment caractérisé par un type (cf. Chapitre 2). Une règle de détection définit avant tout une combinaison d'évènements de types différents, qu'AUROS doit sélectionner parmi de nombreux évènements présents dans le flux.

Ainsi, le langage AUROS permet de définir le type d'un évènement à sélectionner via un *atome*. Un atome est un opérateur unaire qui est satisfait lorsqu'un évènement d'un certain type est détecté dans le flux. Il est à noter qu'on peut associer à tout atome une variable nommée, permettant d'accéder à l'évènement sélectionné dans le reste de la règle.

Le listing présente une règle nommée `ViewRule`, qui vise à détecter uniquement les évènements de type `View`. Dans le contexte Cdiscount, les évènements de type `View` matérialisent la visualisation par un client d'une page correspondant à un produit quelconque.

LISTING 3.2 – La règle `ViewSimple` détecte les évènements de type `View`.

```
ViewSimple :
    View
```

La règle `ViewSimple` de l'exemple 3.2 détecte tout évènement de type `View` présent dans le flux.

3.2.2 Filtres

Il est possible d'associer à un atome un ou plusieurs *filtres*, qui contraignent la sélection de l'évènement en fonction de ses attributs.

LISTING 3.3 – La règle `ViewTVRule` détecte les évènements de type `View` qui concernent un produit dont la catégorie est "TV".

```
ViewTV :
    View:v
    # v.category = "TV"
```

Dans l'exemple 3.3, l'évènement de type `View` est filtré en fonction de la catégorie du produit, qui est accessible via l'attribut `category` de l'évènement. Dans la règle `ViewTVRule`, l'attribut `category` d'un évènement `View` peut être accédé via la variable `v` associée, afin d'évaluer la condition `v.category = "TV"`. Cette condition spécifie que pour tout évènement `View`, son attribut `category` doit être égal à `"TV"`. Lorsque cette condition est satisfaite pour un évènement de type `View`, alors cet évènement est sélectionné. La règle `ViewTVRule` permet donc de détecter les visionnages associées aux téléviseurs.

3.3 Combiner les évènements

Les règles de détection sont généralement constituées de plusieurs évènements. Le langage AUROS permet d'exprimer des combinaisons et des séquences d'évènements grâce à un ensemble d'opérateurs présentés dans cette section.

3.3.1 Opérateurs logiques

Les opérateurs logiques `or` et `and` sont des opérateurs binaires, permettant respectivement d'exprimer la disjonction et la conjonction d'évènements. Ces opérateurs étant associatifs ils ne prennent pas en considération la date d'occurrence des évènements concernés.

LISTING 3.4 – La règle `ViewOrAdd` détecte la visualisation d'un produit *ou* un ajout d'un produit au panier.

```
ViewOrAdd :
    View or AddToCart
```

L'exemple 3.4 exprime une disjonction entre un atome `View` et un atome `AddToCart`. Ainsi, la règle `ViewOrAdd` permet de détecter soit un évènement de visualisation d'une fiche produit, soit un évènement d'ajout d'un produit au panier. AUROS valide la détection de cette règle au plus tôt, dès l'instant où l'un des deux évènements est détecté.

3.3.2 Opérateurs de séquence

L'opérateur de séquence, appelé `FollowedBy`, permet de spécifier l'ordre d'occurrence des évènements d'une règle. Il existe deux variantes de l'opérateur de séquence : la première variante (symbole `->`) est non-contiguë, signifiant que des évènements quelconques peuvent être présents dans le flux entre les deux évènements spécifiés. La seconde variante (symbole `.`) est contiguë, signifiant que les deux évènements spécifiés doivent impérativement se suivre directement dans le flux, c'est-à-dire que tout évènement non reconnu interrompt alors la règle.

LISTING 3.5 – La règle `SearchThenView` permet de détecter un évènement de recherche suivi d'un évènement de visualisation d'un produit. Des évènements quelconques peuvent survenir entre ces deux évènements.

```
SearchThenView :
    Search -> View
```

Dans l'exemple 3.5, la règle `SearchThenView` permet de détecter une recherche (effectuée via le moteur de recherche du site Cdiscount) suivie de la visualisation d'une fiche produit. Notons que les évènements d'une séquence ne sont pas nécessairement contigus dans le flux : les évènements pouvant survenir entre `Search` et `View` seront ignorés.

LISTING 3.6 – La règle `SearchThenDirectlyView` permet de détecter un évènement de recherche suivi immédiatement d'un évènement de visualisation d'un produit, sans autres évènements entre les deux.

```
SearchThenDirectlyView :
    Search . View
```

Au contraire, dans l'exemple 3.6, la règle interdit la présence d'évènements non spécifiés entre `Search` et `View`.

L'opérateur de séquence `FollowedBy` peut être chaîné pour exprimer des séquences d'évènements. L'ordre d'occurrence des évènements de la séquence est alors déterminé de gauche à droite.

LISTING 3.7 – La règle `SearchThenViewThenAdd` permet de détecter une recherche suivie d'une visualisation, elle-même suivie d'un ajout au panier.

```
SearchThenViewThenAdd :
    Search -> View -> AddToCart
```

Dans l'exemple 3.7, la règle `SearchThenViewThenAdd` exprime une séquence d'évènements commençant par une recherche, puis une visualisation, et enfin un ajout au panier.

3.3.3 Opérateur d'itération

L'opérateur d'itération `KleenePlus` permet de sélectionner une séquence d'évènements composée d'au moins deux évènements *de même type*. Cet opérateur est analogue à l'opérateur du même nom utilisé dans les expressions régulières.

LISTING 3.8 – La règle `SearchSeq` permet de détecter plusieurs évènements de recherche consécutifs.

```
SearchSeq :
    Search+
```

L'exemple 3.8 montre la règle `SearchSeq` qui identifie une itération d'évènements de type `Search`. Autrement dit, cette règle permet de détecter une série d'une ou plusieurs recherches consécutives.

Il est important de noter cependant qu'AUROS considère cette règle comme étant non valide. En effet la taille de la séquence d'évènements détectée par l'opérateur `KleenePlus` n'étant pas bornée par défaut, la règle `SearchSeq` n'est pas détectable. En évaluant cette règle, le système serait indéfiniment en attente d'un prochain évènement de type `Search` (boucle infinie). Pour éviter ce cas de figure, le langage AUROS effectue une simple étape de vérification des règles spécifiées, et interdit l'utilisation de l'opérateur `KleenePlus` à la fin de celles-ci.

LISTING 3.9 – La règle `SearchSeqThenView` permet de détecter plusieurs recherches consécutives suivies d'une visualisation.

```
SearchSeqThenView :
    Search+ -> View
```

En revanche, la règle de l'exemple 3.9 est bien valide, car l'opérateur de séquence permet ici de réaliser une transition entre l'itération d'évènements de type `Search`, et un atome de type `View`. Dans ce cas, l'opérateur `KleenePlus` se termine dès que l'évènement `View` est détecté, quelque soit la taille de la séquence d'évènements `Search`.

Il est aussi possible de définir précisément la condition d'arrêt de `KleenePlus`. Le nombre d'évènements contenus dans la séquence `KleenePlus` peut être borné via la fonction `Size(seq, n)`, `seq` étant une variable de séquence `KleenePlus`, et `n` la taille de la séquence.

LISTING 3.10 – La règle `BoundedSearchSeq` permet de détecter trois évènements de recherche consécutifs.

```
BoundedSearchSeq :
    Search+:s
    # Size(s, 3)
```

L'exemple 3.10 permet de détecter exactement trois évènements `Search` consécutifs. L'utilisation de la fonction `Size` permet ici à cette règle d'être valide car la condition de terminaison de l'opérateur `KleenePlus` est bien définie (taille de la séquence `s` égale à 3).

Une autre façon de définir la condition de terminaison d'une itération `KleenePlus` est de paramétrer la durée pendant laquelle celle-ci doit être active, c'est-à-dire en attente de nouveaux évènements. Pour cela, AUROS propose la fonction `Duration(seq, d)`, `d` étant la durée pendant laquelle s'applique l'itération.

LISTING 3.11 – La règle `TimedSearchSeq` permet de détecter plusieurs recherches consécutives dans un intervalle de temps de 5 minutes.

```
TimedSearchSeq :
    Search+:s
    # Duration(s, 5m)
```

Dans l'exemple du listing 3.11, l'itération `s` est active pendant exactement 5 minutes, à partir du premier évènement `Search` sélectionné. Ce délai passé, l'opérateur `KleenePlus` se termine. Notons que dans ce cas le nombre d'évènements `Search` dans la séquence `s` est indéterminé *a priori*, car il dépend du contenu du flux d'évènements pendant une période donnée.

Pour faciliter la définition des contraintes, les évènements contenus dans une séquence `KleenePlus` peuvent y être accédés via leur index.

LISTING 3.12 – La règle `SearchSeqSameCategory` permet de détecter plusieurs recherches consécutives dans la même catégorie de produits.

```
SearchSeqSameCategory :
    Search+:s
    # s[i].category = s[i-1].category
    # Size(s, 5)
```

La règle présenté dans l'exemple 3.12 montre l'utilisation de la variable spéciale `i`, qui permet à AUROS d'accéder, pendant l'évaluation de la règle, à l'évènement courant dans la séquence `s`. Cette variable spéciale est ici utilisée pour définir une condition d'itération, spécifiant que tout évènement `Search` de la séquence `s` doit appartenir à une même catégorie.

Il est également possible d'utiliser une valeur numérique pour accéder à un index spécifique dans une séquence `KleenePlus`.

LISTING 3.13 – La règle `SearchSeqTVCategory` permet de détecter plusieurs recherches consécutives, dont la première concerne la catégorie `"TV"`.

```
SearchSeqTVCategory :
    Search+:s
    # s[0].category = "TV"
    # s[i].category = s[0].category
    # Size(s, 5)
```

L'exemple 3.13 montre une condition basée sur un index fixe dans une séquence `KleenePlus`. Cette condition précise que la première recherche, située à l'index `0` de la séquence `s`, doit concerner la catégorie "TV".

Pour accéder au dernier élément d'une séquence `KleenePlus`, l'indice spécial `last` peut être utilisé.

Enfin, des fonctions d'agrégation telles que `Count(seq)` ou `Avg(seq, attr)` peuvent être utilisées pour calculer une valeur à partir des évènements contenus dans une séquence `KleenePlus`. `attr` correspond à l'attribut sur lequel est basée l'agrégation.

LISTING 3.14 – La règle `ViewMoreExpensive` permet de détecter plusieurs visualisations de produits, chaque visualisation correspondant à un produit dont le prix est plus élevé que la moyenne des prix des produits précédemment visualisés.

```
ViewMoreExpensive :
    View+:v
    # v[i].price > Avg(Sub(v, 0, i-1), price)
    # Duration(v, 1h)
```

Dans l'exemple 3.14, la règle spécifie que la valeur de l'attribut `price` associé à chaque évènement `View` de la séquence `s` doit être supérieure à la moyenne des valeurs de l'attribut `price` calculée sur les évènements `View` précédents. La fonction `Sub(seq,` i_{start}`,` i_{end}`)` est utilisée pour accéder aux évènements `View` précédant l'évènement `View` courant. De manière générale, cette fonction permet d'extraire une sous-séquence dans une itération : i_{start} étant l'index de début de la sous-séquence et i_{end} son index de fin. La fonction `Avg` permet ici d'agréger cette sous-séquence en calculant une moyenne sur l'attribut `price` des évènements `View` (qui contient le prix des produits visionnés).

3.4 Spécifier des contraintes de relation

Les contraintes globales ne sont pas liées à un opérateur particulier, et peuvent être définies pour tout couple d'évènements appartenant à une même règle, même si ceux-ci sont séparés par un ou plusieurs opérateurs de séquence `FollowedBy`.

3.4.1 Liens

Un *lien* est une condition permettant de combiner deux évènements en fonction des valeurs de leurs attributs.

LISTING 3.15 – La règle `SearchThenView` permet de détecter une visualisation d'un produit qui a été affiché à la suite d'une recherche.

```
SearchThenView :
    Search:s -> View:v
    # v.sid = s.sid
```

La règle présentée dans l'exemple 3.15 permet de combiner un évènement `Search` et un évènement `View` en fonction de leur `sid` (*Search IDentifier*). Le `sid` correspond à un identifiant unique généré lorsqu'un internaute effectue une recherche sur le site Cdiscount. Le lien `v.sid = s.sid` permet de garantir que les évènements `Search` et `View` correspondent à un même `sid`.

Via l'utilisation de liens, les règles peuvent détecter des parcours de navigation spécifiques, en sélectionnant des évènements liés par une condition spécifique.

3.4.2 Contraintes temporelles

Les *contraintes temporelles* permettent de spécifier l'intervalle de temps qui sépare deux évènements. Ceux-ci doivent être séparés au minimum par un opérateur de séquence `FollowedBy` : une contrainte temporelle n'aurait pas de sens pour un opérateur logique car ceux-ci n'ont pas la notion de temps.

Une contrainte temporelle est définie via la fonction $Within(e_1, e_2, i)$, e_1 et e_2 étant deux évènements, et i l'intervalle de temps maximum séparant ces deux évènements.

LISTING 3.16 – La règle `SearchThenViewWithin` permet de détecter une recherche suivie d'une visualisation dans un intervalle de temps de 10 minutes.

```
SearchThenViewWithin :
    Search:s -> View:v
    # Within(s, v, 10m)
```

La règle de l'exemple 3.16 garantit un intervalle maximum de 10 minutes entre les évènements `Search` et `View`. Si ce délai n'est pas respecté, la règle est invalidée.

Les contraintes temporelles peuvent également être spécifiées pour une itération.

LISTING 3.17 – La règle `ViewSeqWithin` permet de détecter plusieurs visualisations de produits pendant une période d'une heure, et séparées les unes des autres d'une période de 5 minutes au plus.

```
ViewSeqWithin :
    View+:v
    # Within(v, 5m)
    # Duration(v, 1h)
```

Dans l'exemple 3.17, la fonction `Within(seq, i)` permet de spécifier l'intervalle maximum de périodicité pour l'évènement `View` dans l'itération `v`. Cette itération est active pendant une durée d'une heure, mais si l'intervalle de périodicité entre chaque évènement `View` n'est pas respecté, la règle est immédiatement invalidée.

3.5 Spécifier des fenêtres temporelles

Les fenêtres temporelles permettent de limiter le champ d'application d'une règle dans le temps. Il existe deux types de fenêtres temporelles dans le langage AUROS: les fenêtres glissantes, et les fenêtres fixes.

3.5.1 Fenêtres fixes

Une fenêtre fixe est un intervalle de temps absolu, qui est initialisé à partir du premier évènement sélectionné par la règle. Elles permettent donc de fixer un intervalle de temps maximum pour chaque séquence correspondant à la règle, qui commence à l'instant où le premier évènement de la règle est identifié. Lorsque la fenêtre expire, la séquence est invalidée. Les *fenêtres fixes* sont déclarées via la fonction `FixedWindow(duration)`.

LISTING 3.18 – La règle `FixedWindowExample` permet de détecter une recherche suivie de plusieurs visualisations, suivies d'un ajout au panier dans une fenêtre glissante d'une journée.

```
    FixedWindowExample :=
    Search:s -> View+:v -> AddToBasket:a
    # FixedWindow(1d)
```

La règle de l'exemple 3.18 définit une fenêtre fixe d'un jour. Autrement dit, l'intervalle de temps entre le premier évènement `Search` et l'évènement `AddToBasket` ne doit pas dépasser 24 heures.

3.5.2 Fenêtres glissantes

Contrairement à une fenêtre fixe, l'intervalle de temps d'une fenêtre glissante est relatif au dernier évènement sélectionné par la règle. La sélection d'un nouvel évènement par la

règle fait effectivement "glisser" cet intervalle de temps pour le fixer sur le nouvel évènement. En d'autres termes, les fenêtres glissantes permettent de spécifier un délai maximum d'inactivité pour chaque séquence correspondant à la règle, au-delà duquel la fenêtre expire, et la séquence est invalidée. Une fenêtre glissante peut être déclarée via la fonction `SlidingWindow(duration)`.

LISTING 3.19 – La règle `WindowExample` permet de détecter une recherche suivie de plusieurs visualisations, suivies d'un ajout au panier dans une fenêtre glissante de 30 minutes à partir du dernier évènement sélectionné.

```
WindowExample :=
    Search:s -> View+:v -> AddToBasket:a
    # SlidingWindow(30m)
```

Dans l'exemple 3.19, la fenêtre glissante spécifiée permet de garder la règle active dans le système pendant un intervalle de 30 minutes, décompté depuis l'instant où le dernier évènement a été sélectionné. Si aucun évènement n'a été sélectionné pendant la dernière demi-heure, alors la règle est invalidée. Les fenêtres glissantes sont un moyen de limiter le nombre de règles simultanément actives dans AUROS.

3.6 Spécifier des négations

Les *négations* sont essentiellement l'inverse des règles de détection : il s'agit de combinaisons d'évènements dont l'occurrence est interdite au sein d'une règle donnée. Le champ d'application d'une négation dans une règle est délimité par deux évènements. Une négation est spécifiée via la fonction $Neg(e_1, e_2, n)$, e_1 et e_2 étant deux évènements de la règle délimitant l'intervalle dans lequel la négation est active, et `n` la spécification de la négation elle-même. Il existe également une variante de cette fonction, `Neg(seq, n)`, permettant d'interdire la règle `n` pendant une itération `seq`.

LISTING 3.20 – La règle `SearchThenAddWithoutView` permet de détecter une recherche suivie d'un ajout au panier, sans évènement `View` entre les deux.

```
SearchThenAddWithoutView :
    Search:s -> AddToBasket:a
    # Neg(s, a, View)
    # a.sid = s.sid
```

Dans l'exemple 3.20, la règle détecte une recherche suivie d'un ajout panier, *sans visualisation* de la fiche produit correspondante (ce qui est un scénario courant sur le site Cdiscount). La négation indique qu'entre les évènements `Search` et `AddToBasket`, l'évènement `View` ne doit pas être présent. Si un évènement `View` est détecté dans cet intervalle, alors la règle est immédiatement invalidée.

3.7 Spécifier une stratégie de sélection

Le langage AUROS permet d'associer une *stratégie de sélection* à toute règle. Pour rappel, la stratégie de sélection permet de déterminer quels évènements doivent être sélectionnés par une règle, dans les cas où plusieurs séquences d'évènements correspondent à la règle à un instant donné. Dans l'implémentation actuelle d'AUROS, plusieurs stratégies de sélection peuvent être spécifiées. La stratégie de sélection à appliquer pour une règle est spécifiée dans ses métadonnées. Les métadonnées suivent la définition de la règle, et chaque paramètre est préfixé par le symbole `@`.

Pour comprendre le fonctionnement des différentes stratégies de sélection que propose AUROS, il est nécessaire d'introduire la notion de *chevauchement* de séquences d'évènements.

Des séquences d'évènements qui se chevauchent correspondent à des périodes de temps qui s'entrelacent.

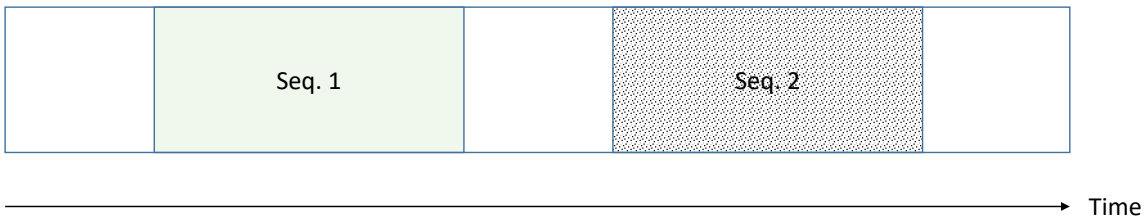

FIGURE 3.1 – Deux séquences d'évènements identifiées par une règle quelconque. Les deux séquences ne se chevauchent pas, c'est-à-dire qu'elles leur intersection temporelle est nulle.

La figure 3.1 illustre deux séquences d'évènements qui ne se chevauchent pas dans un flux. L'intersection temporelle des deux séquences est nulle.

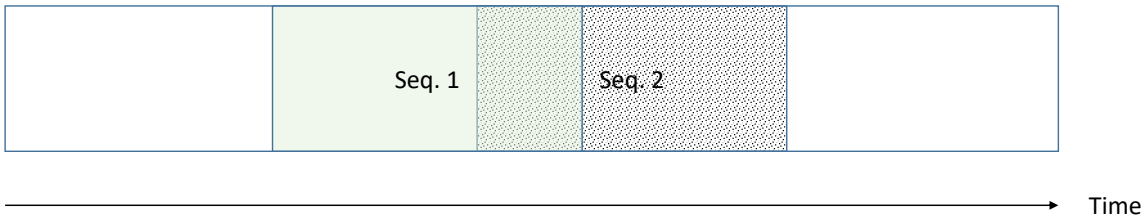

FIGURE 3.2 – Deux séquences d'évènements identifiées par une règle quelconque. Les deux séquences se chevauchent, c'est-à-dire que leur intersection temporelle est non-nulle.

La figure 3.2 illustre deux séquences d'évènements qui se chevauchent dans un flux. L'intersection temporelle des deux séquences est non nulle.

Les stratégies de sélection d'AUROS peuvent être divisées en deux catégories :

— Les stratégies permettant de détecter des séquences qui ne se chevauchent pas. Ces stratégies permettent à la règle d'ouvrir au maximum une séquence à la fois. Dans cette catégorie se trouvent la stratégie *First*.

— Les stratégies permettant de détecter des séquences qui se chevauchent. Dans cette catégorie se trouve la stratégie *Every*, qui permet à la règle d'ouvrir plusieurs séquences en parallèle.

D'un point de vue sémantique, la stratégie définie au niveau d'une règle influe sur sa capacité à détecter des sous-ensembles de séquences différents, parmi l'ensemble des séquences identifiables dans un flux donné. En particulier, la stratégie *Every* permet à la règle de détecter un maximum de séquences dans le flux, alors que la stratégie *First* se concentre sur un sous-ensemble des séquences que la règle peut identifier dans le flux.

L'utilisateur d'AUROS fait le choix d'une stratégie pour une règle donnée en fonction de ses besoins.

LISTING 3.21 – La règle `SearchThenView` détecte une recherche suivie d'une visualisation.

```
SearchThenView :
Search:s -> View:v
# s.sid = v.sid
@Strategy = <strategy>
```

Par exemple, la règle `SearchThenView` du listing 3.21 décrit une séquence comportant un évènement `Search` suivi d'un évènement `View`.

LISTING 3.22 – Flux d'évènements produit par un unique client (d'identifiant c_1) du site marchand

```
[
(c1,1,Search,{sid=1}),
(c1,5,Search,{sid=2}),
(c1,8,View,{sid=1}),
(c1,17,View,{sid=2}),
]
```

Soit le flux du listing 3.22, contenant les évènements produits par un client quelconque. Les dates associées aux évènements sont simplifiées par soucis de clarté. Pour la règle `SearchThenView` du listing 3.21, ce flux contient des chevauchements de séquence. En effet, les séquences `[(c1,1,Search,{sid=1}), (c1,8,View,{sid=1})]` et `[(c1,5,Search,{sid=2}), (c1,17,View,{sid=2})]` identifiées par la règle ont une intersection temporelle non nulle. Selon la stratégie définie (paramètre `@Strategy`), la règle identifie des séquences différentes.

Stratégie *First*

Une première stratégie proposée par AUROS est la stratégie *First*. Cette stratégie permet d'initialiser une nouvelle séquence dès que le premier évènement de la règle est identifié. Tant que cette séquence est ouverte, la règle ne peut pas initialiser d'autres séquences en parallèle. Ainsi, dans le cas de séquences qui se chevauchent, la stratégie *First* n'identifie que la *première* parmi les séquences se chevauchant.

Pour illustrer plus concrètement le fonctionnement de la stratégie *First*, reprenons la règle du listing 3.21 avec en paramètre `@Strategy =First`, et le flux du listing 3.22.

Analysons, étape par étape, comment cette règle interprète le flux :

1. La règle `SearchThenView` commence par identifier l'évènement `[(c1,1,Search,{sid=1})`, et initialise une nouvelle séquence ($t = 1$).
2. L'évènement `Search` suivant `(c1,5,Search,{sid=2})` est ignoré par la règle car il ne correspond à la séquence en cours, qui n'est pas encore terminée ($t = 5$).
3. Puis, la règle identifie l'évènement `(c1,8,View,{sid=1})` qui correspond et termine la séquence en cours ($t = 8$). À cet instant, la règle se remet en attente d'un nouvel évènement `Search` pour ouvrir une nouvelle séquence.
4. Enfin, le second évènement `View` est ignoré car il ne correspond pas au type `Search` ($t = 17$).

Au final, une seule séquence d'évènements a été identifiée dans le flux par la stratégie *First* : `[(c1,1,Search,{sid=1}), (c1,8,View,{sid=1})]`.

Stratégie *Every*

À la différence de la stratégie *First*, la stratégie *Every* a la capacité de manipuler parallèlement plusieurs séquences d'évènements. Cette stratégie initialise une nouvelle séquence *à chaque fois* que le premier évènement de la règle est identifié et sélectionné. La stratégie *Every* permet d'identifier toutes les séquences identifiables, qu'elles se chevauchent ou non.

Soit la règle du listing 3.21 avec en paramètre `@Strategy =Every`, et le flux du listing 3.22.

Analysons étape par étape comment cette règle interprète le flux :

1. La règle `SearchThenView` commence par identifier l'évènement `[(c1, 1, Search, {sid=1})` et initialise une nouvelle séquence ($t = 1$).
2. L'évènement `Search` suivant `(c1,5,Search,{sid=2})` permet d'ouvrir une nouvelle séquence, qui est active parallèlement à la première ($t = 5$).
3. L'évènement `View` suivant `(c1,8,View,{sid=1})` correspond à la première séquence et la termine ($t = 8$).
4. Enfin, le second évènement `View` `(c1,17,View,{sid=2})` correspond et termine la deuxième séquence ($t = 17$). À cet instant, la règle se remet en attente d'un nouvel évènement `Search` pour ouvrir une nouvelle séquence.

Au final, la stratégie *Every* a identifié deux séquences qui se chevauchent dans le flux :

— `[(c1,1,Search,{sid=1}), (c1,8,View,{sid=1})]`

— `[(c1,5,Search,{sid=2}), (c1,17,View,{sid=2})]`

3.8 Synthèse

Le langage AUROS permet d'exprimer des séquences d'évènements spécifiques à identifier dans le flux, via des règles de détection. Mais une fois exprimées, ces règles doivent être évaluées par le système AUROS de manière efficace via un modèle d'exécution des règles. Dans le chapitre 4 suivant, nous montrons comment AUROS permet de compiler une règle en un modèle d'automate correspondant, qui permet de détecter en temps réel les séquences décrites par les règles.

CHAPITRE

4 Un modèle d'exécution des règles de détection

Ce chapitre décrit le modèle d'exécution des règles de détection d'AUROS, basé sur l'automate fini. Il commence par décrire l'automate AUROS, puis explique la façon dont les règles de détection (décrites dans le chapitre 3) sont compilées vers ce type d'automate. Enfin, il décrit comment l'automate exécute une règle de détection pour identifier des séquences d'évènements dans le flux.

Sommaire

4.1 Définition du modèle d'automate

Un automate AUROS est un automate fini permettant d'exécuter une règle de détection spécifiée dans le langage AUROS. Chaque règle est donc compilée en automate, qui est un quintuplet $A = \langle \Sigma, Q, T, i, F \rangle$.

Σ est un alphabet fini représentant l'ensemble possible des types d'évènements. Un flux d'évènements est un mot infini où chaque évènement a un et un seul type défini dans l'alphabet. Par simplification, on dira que les évènements e appartiennent à l'alphabet Σ ($e \in \Sigma$). Ainsi, une séquence d'évènements identifiée par un automate AUROS est un mot fini de Σ.

Un automate AUROS est caractérisé par : (i) Un ensemble fini Q contenant tous les états de l'automate. (ii) Un état initial $i \in Q$. (iii) Un ensemble fini $F \subset Q$ contenant les états finaux de l'automate. (iv) Un ensemble de transitions T, qui est une partie de $Q \times \Sigma \times Q$.

Une transition de l'automate est caractérisé par un état source et un état cible (les deux pouvant être le même état dans le cas de transitions-boucles). Les transitions permettent de reconnaître les évènements à identifier par la règle de détection.

Une transition possède une étiquette, indiquant le(s) type(s) d'évènement qu'elle reconnaît. De plus, il existe deux étiquettes spéciales. L'étiquette ϵ est caractéristique d'une *transition-epsilon*, et l'étiquette $*$ caractérise une *transition-étoile*. Les transitions-epsilon reconnaissent l'évènement vide, et elles peuvent donc être franchies même si aucun évènement n'apparait dans le flux. Les transitions-étoiles reconnaissent n'importe quel évènement qui n'est pas déjà reconnu par une transition (non-étoile) partant du même état source. De fait, il peut n'y avoir qu'une seule transition-étoile sortante pour un état donné.

Optionnellement, une transition possède un ensemble de gardes à évaluer permettant ou non le franchissement de la transition. Ces gardes permettent d'exprimer certaines contraintes définies dans la règle de détection, notamment les filtres et les liens.

Par ailleurs, à chaque transition (sauf les transitions-epsilon) peuvent être associées plusieurs types d'action, qui sont exécutées au moment de leur franchissement :

— L'action de *sélection* permet d'enregistrer (action *Append*) ou d'ignorer (action *Drop*) l'évènement reconnu par la transition. Il est à noter que les transitions-étoiles sont systématiquement de type *Drop*, car leur rôle est d'ignorer les évènements qui ne sont pas reconnus par les autres transitions.

— L'action de *temporisateur* permet de créer ou de détruire un temporisateur. Les temporisateurs permettent de gérer les contraintes temporelles et les fenêtres définies dans la règle (cf. section 3.4.2).

— L'action de *négation* permet la création ou la destruction d'un automate de négation. Les automates de négations servent à évaluer les négations qui peuvent être définies dans la règle (cf. section 3.6).

4.2 Compilation des règles de détection

La compilation d'une règle de détection vers un automate fini se déroule en deux étapes. La première étape consiste à construire un automate non-déterministe qui correspond à la règle, en assemblant plusieurs automates qui chacun correspondent à des parties spécifiques de la règle. La deuxième étape consiste à rendre déterministe cet automate notamment en supprimant les transitions-epsilon, l'objectif étant d'optimiser l'exécution de l'automate.

4.2.1 Construction de l'automate correspondant à une règle de détection

L'algorithme de construction de l'automate correspondant à une règle de détection est inspiré de la construction de Thompson [Thompson, 1968], utilisée pour la génération d'automates d'expressions régulières.

Le principe est de combiner récursivement des automates simples pour former des automates de plus en plus complexes, jusqu'à l'obtention de l'automate final. Dans notre contexte, cet automate final correspond à une règle de détection dans son ensemble.

Dans AUROS, l'automate le plus simple est celui qui correspond à un atome (cf. section 3.2.1). Pour rappel, un atome permet simplement de définir un évènement d'un certain type.

Les automates simples correspondant à des atomes peuvent alors être composés pour représenter des conjonctions, des disjonctions, des séquences (opérateur FollowedBy) et des itérations (opérateur KleenePlus). Ces automates portent également, à travers leurs transitions, les contraintes définies dans la règle. Ces automates peuvent être composés notamment à l'aide de transitions-epsilon. Les sections suivantes montrent les différentes constructions d'automate possibles qui correspondent aux différents éléments du langage AUROS.

Automate de l'atome : l'automate de base

Le listing 4.1 montre une règle composée d'un unique atome et d'un filtre associé. Cette règle détecte les évènements de type `View` dont la catégorie est `"TV"`.

LISTING 4.1 – Règle composée d'un atome `View` et d'un filtre sur la catégorie `"TV"`.

```
ViewTV :=
    View:v
    # v.category = "TV"
```

La figure 4.1 illustre l'automate simple qui correspond à cette règle. Cet automate est constitué d'un état initial I, d'un état final F, et d'une transition reliant I à F. La transition

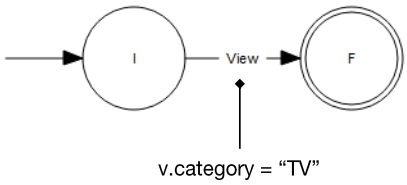

FIGURE 4.1 – Automate généré pour la règle `ViewTV`. La transition de type `View` comporte un filtre permettant de filtrer les évènements de type `View` en fonction de leur attribut `category` (qui ici doit être égal à `"TV"`).

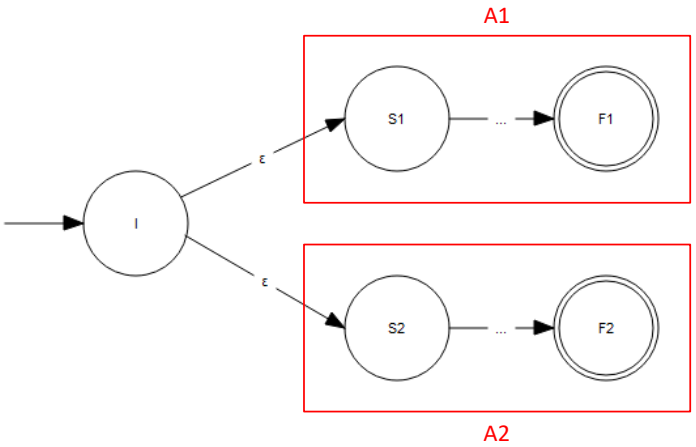

FIGURE 4.2 – Automate généré pour la règle `R1 || R2`. Les automates des sous-règles `A1` et `A2` sont délimités par des rectangles. L'automate final possède deux états finaux.

est étiquetée du type `View`, et possède une garde correspondant au filtre sur la catégorie `"TV"` spécifié dans la règle.

Automate de disjonction

Soit un opérateur de disjonction `R1 || R2` dont les deux opérandes `R1` et `R2` sont deux règles quelconques, représentées par les automates $A1$ et $A2$.

La figure 4.2 montre l'automate correspondant à la règle obtenue en applicant cette disjonction. On voit qu'un état de disjonction (ici l'état initial I) est introduit et relié via des epsilon-transitions aux états initiaux des automates $A1$ et $A2$.

Automate de séquence

Soit un opérateur de séquence `R1 -> R2`, dont les deux opérandes `R1` et `R2` sont deux règles quelconques représentées par deux automates $A1$ et $A2$.

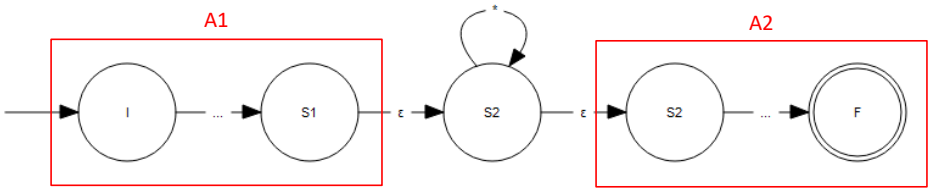

FIGURE 4.3 – Automate généré pour la règle `R1 -> R2`. Les automates des sous-règles `A1` et `A2` sont délimités par des rectangles.

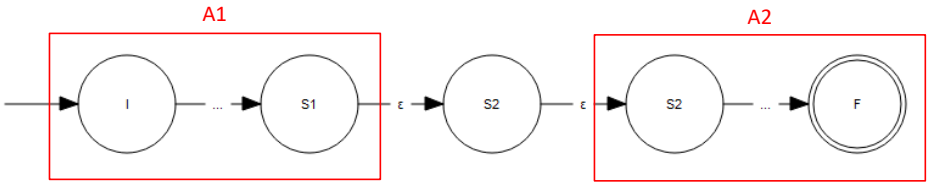

FIGURE 4.4 – Automate généré pour la règle `R1 . R2`. Les automates des sous-règles `A1` et `A2` sont délimités par des rectangles.

La figure 4.3 montre l'automate correspondant à la règle obtenue en applicant cet opérateur de séquence, qui combine $A1$ et $A2$ de manière non contiguë (autrement dit, d'autres évènements non-spécifiés dans la règle peuvent survenir entre `R1` et `R2`). L'automate de séquence réalise la transition entre $A1$ et $A2$ en les reliant par des transitions-epsilon, et en passant par un état de transition ($S2$ dans ce cas) portant une transition-étoile en boucle. Cette dernière transition permet d'ignorer les évènements non-identifiés entre `R1` et `R2`.

La figure 4.4 montre l'automate correspondant à l'opérateur de séquence dans sa variante contigu. Cet automate est équivalent au précédent à l'exception de la transition-étoile qui n'est pas présente, forçant ainsi les évènements à se suivre directement.

Automate KleenePlus

Soit un opérateur d'itération `R+` dont l'opérande est une règle quelconque `R`, représentée par un automate A.

La figure 4.5 montre comment l'automate KleenePlus est obtenu par enrichissement de l'automate A. Cet automate permet de capturer au moins une répétition de `R`. Pour cela, l'automate de KleenePlus introduit un état d'itération ($S2$ dans ce cas) relié par des transitions-epsilon aux états initial et final de A.

Dans le cas d'une itération non-contiguë, cet état d'itération porte une transition-étoile boucle permettant d'ignorer les évènements non-identifiés entre chaque itération de `R`.

Notons que la figure 4.5 montre bien que l'automate KleenePlus ne porte pas d'état final, ce qui explique qu'AUROS détecte la règle `R+` comme étant invalide (cf. section 3.3.3).

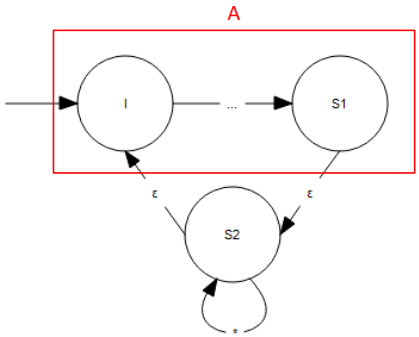

FIGURE 4.5 – Automate généré pour la règle `R+`. L'automate de la sous-règle `A` est délimité par un rectangle.

Application des contraintes de taille et de durée. Pour rappel, le langage AUROS permet de définir la taille (contrainte `Size`) ou la durée maximale (contrainte `Duration`) d'une itération KleenePlus. Le listing 4.2 montre une règle spécifiant une itération de type `View`, à laquelle est associée une contrainte de taille `Size(v) = 3`.

LISTING 4.2 – Règle composée d'une itération `View+` et d'une contrainte `Size` associée.

```
ViewSeq :=
View+:v
# Size(v) = 3
```

Par rapport à l'automate KleenePlus précédent, dans le cas de l'application d'une contrainte de type *Size* ou *Duration*, l'automate A introduit un état supplémentaire F qui est final, et une transition supplémentaire permettant d'atteindre cet état final. Cette transition porte une garde correspondant à la contrainte *Size* ou *Duration* associée à KleenePlus. L'ensemble des transitions de A portent alors la négation de cette garde afin qu'il n'y ait pas d'ambiguïté.

La figure 4.6 montre l'automate correspondant à la règle du listing 4.2.

Application des contraintes de lien

Le listing 4.3 décrit une règle qui comporte une contrainte liant deux évènements (`Search` et `View`) en fonction de leurs attributs (respectivement `url` et `referrer`).

LISTING 4.3 – Règle spécifiant un lien entre un évènement `Search` et un évènement `View`.

```
SearchThenView :=
    Search:s -> View:v
    # v.referrer = s.url
```

L'évènement `Search` est référencé par la variable `s`, et l'évènement `View` par la variable `v`. Pour évaluer la contrainte de lien, il est nécessaire de disposer à la fois de `s` et de `v`, afin

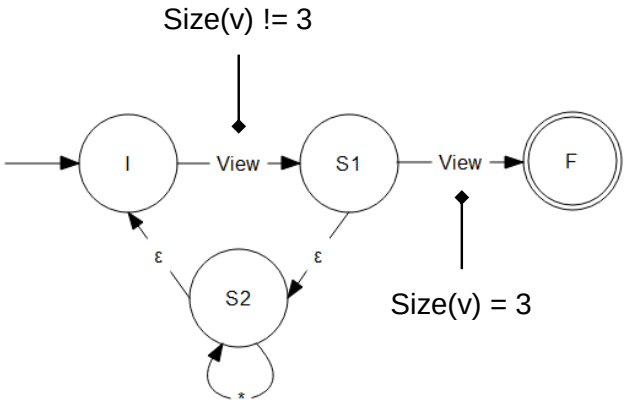

FIGURE 4.6 – Automate gnéré pour la règle `ViewSeq`. Les transitions de type `View` comportent des prédicats permettant de contrôler la taille de la séquence identifiée.

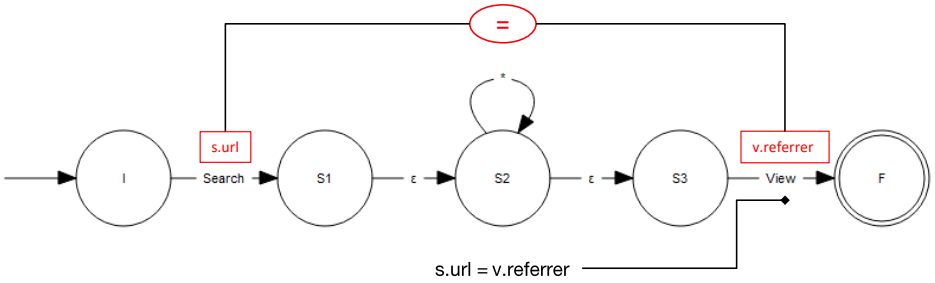

FIGURE 4.7 – Automate généré pour la règle `SearchThenView`. La transition de type `View` comporte un prédicat permettant de lier les évènements `s` et `v`. Ce prédicat de lien permet de vérifier l'égalité entre les attributs `url` de l'évènement `s`, et `referrer` de l'évènement `v`.

de pouvoir extraire et comparer leurs attributs `url` et `referrer`. La variable référançant l'évènement le plus à droite dans la règle étant `v`, le prédicat `v.referrer = s.url` est donc automatiquement attaché à la transition reconnaissant l'évènement de type `View`.

La figure 4.7 montre l'automate correspondant à cette règle. Au moment où la transition reconnaissant l'évènement de type `View` est franchie, les deux attributs sont alors accessibles et la garde de lien peut être évaluée par l'automate.

Nous verrons ultérieurement comment l'automate a recours à une structure de données particulière pour récupérer les évènements `Search` et `View` sélectionnés, et comparer leurs attributs `url` et `referrer`.

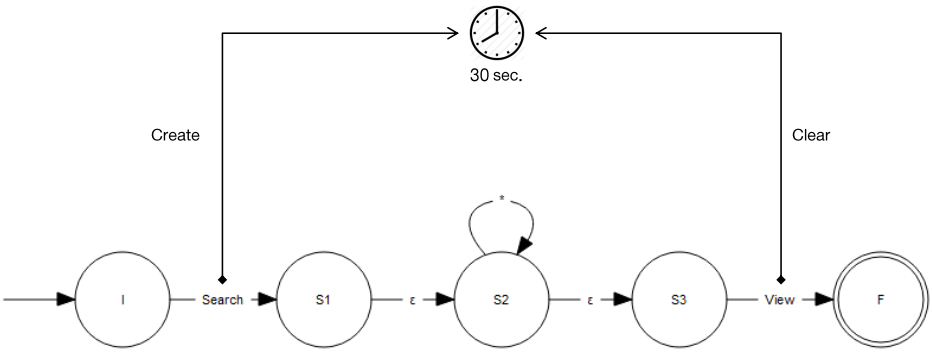

FIGURE 4.8 – Automate généré pour la règle `SearchThenViewWithin30s`. Les transitions de type `Search` et `View` comportent des actions permettant de contrôler le temporisateur vérifiant la contrainte temporelle entre les évènements `s` et `v`.

Application des contraintes temporelles et des fenêtres

LISTING 4.4 – Règle spécifiant une contrainte temporelle entre deux évènements `Search` et `View`

```
SearchThenViewWithin30s :=
    Search:s -> View:v
    # Within(s, v, 30s)
```

Le listing 4.4 présente une règle spécifiant un intervalle maximum de 30 secondes entre deux évènements `Search` et `View` qui se succèdent.

Une contrainte temporelle est traduite dans l'automate à deux endroits. Dans un premier temps, la transition de l'évènement de gauche (dans ce cas de l'évènement `Search` associé à la variable `s`) contient une action qui permet de créer un temporisateur, qui est maintenu par l'automate pour chaque séquence distincte. Puis, la transition de l'évènement de droite (dans ce cas, de l'évènement `View` associé à la variable `v`) contient une action qui supprime le temporisateur.

Si le temporisateur expire avant que la transition de droite ne soit franchie, alors cela signifie que la contrainte temporelle n'a pas été respectée, et la séquence d'évènements courante est alors invalidée par la règle.

La figure 4.8 illustre les actions de temporisateur associées aux transitions `Search` et `View` de l'automate, et qui permettent l'évaluation de la contrainte temporelle `Within(s, v, 30s)`.

Dans le cas d'une fenêtre fixe, le fonctionnement est similaire : le temporisateur est activé à l'initialisation d'une séquence, et perdure jusqu'à la détection de la règle.

Dans le cas d'une fenêtre glissante, la différence est que le temporisateur est remis à sa valeur initiale à chaque fois qu'un nouvel évènement est sélectionné dans la séquence.

Application des négations

Une négation est une règle qui, si elle est évaluée en totalité par la règle principale, invalide la séquence courante (cf. section 3.6). Une négation est donc vue comme une règle à part, qui est compilée comme toute autre règle en un automate correspondant. Cet automate est appelé *automate de négation*, et est co-évalué avec l'automate de la règle principale.

Cela signifie que les évènements du flux évalués par la règle sont à la fois évalués par l'automate principal et l'automate de négation.

Le listing 4.5 montre une règle qui exprime une négation. Elle identifie une recherche (évènement `Search`) suivie d'un ajout au panier (évènement `Add`), sans visualisation (évènement `View`) entre temps.

LISTING 4.5 – Règle avec une négation

```
SearchThenAddWithoutView :=
Search:s -> Add:a
# Neg(s, a, View)
```

Lors de la construction de l'automate AUROS correspondant à cette règle, la négation `Neg(s, a, View)` se reflète au niveau de deux transitions. La transition de gauche (associée à la variable `s` dans l'exemple) porte une action de création de l'automate de négation. Dans l'exemple, cet automate de négation est l'automate correspondant à l'atome `View`. La transition de droite (associée à la variable `a` dans l'exemple) porte quant à elle une action de destruction de cet automate de négation.

Ainsi, lorsque l'évènement `Search` est sélectionné, l'automate de négation est initialisé. Celui-ci est détruit lorsque l'évènement `View` est sélectionné par la suite.

Si, avant que l'automate de négation soit détruit, la co-évaluation amène celui-ci dans son état final (autrement dit, si un évènement `View` est détecté entre `s` et `a`), alors la séquence courante est invalidée.

La figure 4.9 représente l'automate correspondant à la règle `SearchThenAddWithoutView`. Le rectangle délimite l'automate de négation, et les flèches les actions de création et de suppression de cet automate.

4.2.2 Suppression des transitions-epsilon

La deuxième étape de compilation consiste à appliquer l'algorithme *Powerset* sur l'automate généré lors de la première étape. Ici, cet algorithme est utilisé pour supprimer l'ensemble des transitions-epsilon de l'automate, afin d'obtenir un automate déterministe prêt à être exécuté.

Si cet algorithme construit un nouvel automate à partir de l'automate obtenu à la suite de la première étape, il préserve toutes les caractéristiques de ce dernier. L'automate

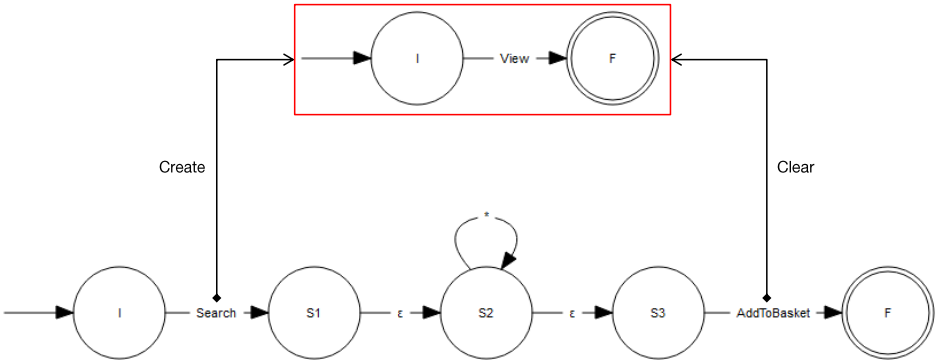

FIGURE 4.9 – Automate généré pour la règle `SearchThenAddWithoutView`. Les transitions de type `Search` et `View` comportent des actions permettant d'activer et désactiver l'automate de négation, délimité par un rectangle.

sans transitions-epsilon reconnait donc bien la même règle, en respectant l'ensemble des contraintes.

Powerset étant un algorithme classique pour déterminiser des automates non-déterministes [Silva *et al.*, 2010], nous illustrons rapidement son effet à travers un exemple, qui nous servira dans la suite de ce chapitre pour expliquer la façon dont nous exécutons les automates AUROS pour identifier des séquences d'évènements correspondant à une règle.

LISTING 4.6 – Règle décrivant plusieurs recherches successives suivies d'un ajout au panier.

```
SearchSeqThenAdd :=
Search+:s -> Add:a
```

Prenons la règle du listing 4.6. La règle `SearchSeqThenAdd` cherche à identifier une itération d'évènements `Search` suivie d'un évènement `Add`.

La figure 4.10 présente l'automate AUROS généré pour la règle `SearchSeqThenAdd`, à la suite de la première étape de compilation. Cet automate contient donc un ensemble de transitions-epsilon.

La figure 4.11 montre quant à elle l'automate généré à la suite de la deuxième étape de compilation. L'automate ne contient plus de transitions-epsilon, et est donc plus compact. Cet automate est prêt à être évalué par AUROS.

4.3 Exécution de l'automate

L'évaluation d'une règle de détection revient à exécuter l'automate AUROS obtenu par compilation de cette règle. L'exécution de l'automate consiste à lire continuellement des évènements provenant du flux en entrée. En lisant un évènement, l'automate applique cer-

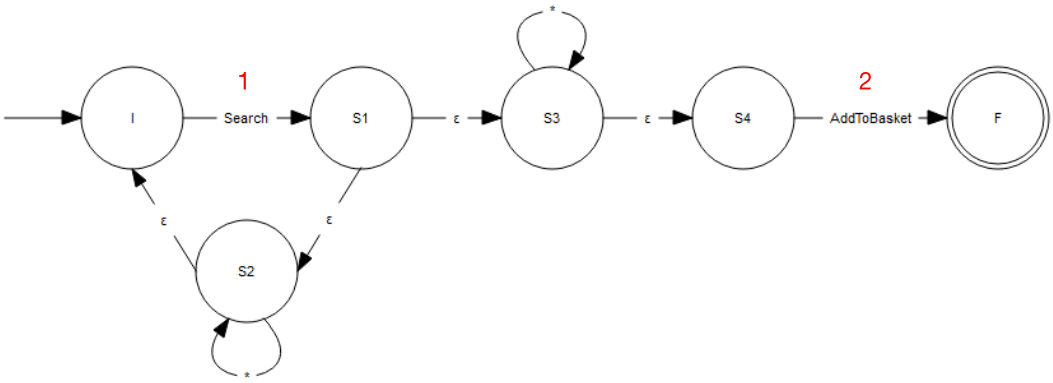

FIGURE 4.10 – Automate intermédiaire généré par la première étape de compilation de la règle `SearchSeqThenAdd`.

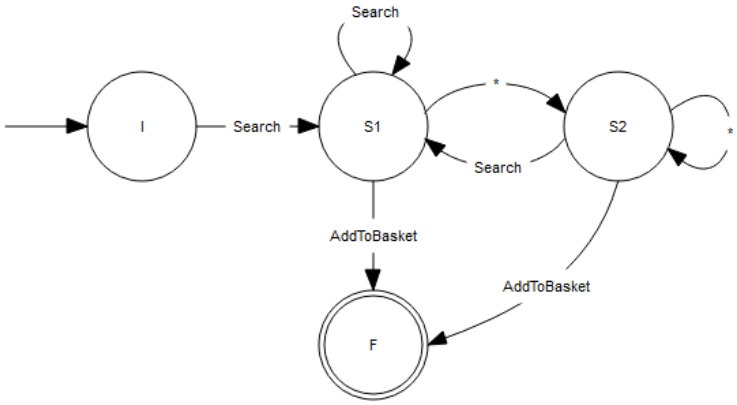

FIGURE 4.11 – Automate final de la règle `SearchSeqThenAdd`, après suppression des transitions-epsilon par l'algorithme Powerset.

taines règles dictées par l'algorithme de détection, qui dépend de la stratégie de sélection définie pour la règle correspondante.

4.3.1 Le concept de jeton

Pour expliquer en détail l'algorithme de détection qui permet d'exécuter l'automate, nous introduisons le concept de *jeton*. Un jeton représente l'état courant de l'automate, et maintient via une structure de données qui lui est propre une séquence d'évènements en cours d'identification.

Un automate peut avoir un ou plusieurs jetons simultanément actifs selon la stratégie de sélection définie dans la règle correspondante.

L'exécution d'un automate AUROS est précisément définie par les règles suivantes :

1. Lorsqu'une règle est activée, quelque soit sa stratégie d'évaluation, un jeton est placé dans l'état initial de l'automate.
2. Lorsqu'un nouvel évènement arrive dans le flux, l'automate met à jour son ou ses jeton(s) de la manière suivante :
   — Si la stratégie est *First*, il n'y a à tout instant qu'un unique jeton dans l'automate. Depuis l'état où se trouve le jeton, on cherche en priorité une transition dont l'étiquette est conforme au type de l'évènement. S'il n'en existe pas, on cherche ensuite une transition-étoile. Si une transition est trouvée et peut être franchie (pour rappel, cf. section 4.1), le jeton avance dans l'état suivant. Dans le cas contraire, le jeton reste dans l'état courant. Si aucune transition ne correspond, le jeton est supprimé et l'automate est réinitialisé (retour à l'étape 1).
   — Si la stratégie est *Every*, il peut y avoir n jetons simultanément actifs dans l'automate : le premier jeton est toujours sur l'état initial, et les autres jetons peuvent progresser d'état en état dans l'automate. Pour chaque jeton, depuis l'état où il se trouve, on recherche une transition comme décrit dans l'étape 2. Contrairement à la stratégie *First*, lorsqu'une transition est trouvée par un jeton, celui-ci ne change pas d'état. Au lieu de cela, le jeton est dupliqué, et on applique au nouveau jeton le comportement de la stratégie *First* décrit dans l'étape 2.
3. Si un jeton arrive sur un état final de l'automate, la séquence identifiée est retournée et le jeton est supprimé. Si la stratégie est *First*, l'automate est réinitialisé (retour à l'étape 1).
4. Si la règle correspondant à l'automate est manuellement désactivée, alors l'automate, ainsi que tous ses jetons, sont supprimés.

4.3.2 Exemples

Pour visualiser le fonctionnement de l'automate AUROS, reprenons l'automate de la figure 4.11 et le flux d'évènements représenté dans le listing 4.7. Ce flux contient les actions réalisées sur le site Cdiscount par un unique client (d'identifiant `c1`).

LISTING 4.7 – Flux d'évènements produit par un unique client du site Cdiscount

```
[
e1 = (c1,1,Search,{query="TV"}),
e2 = (c1,2,View,{productID="TV_344"}),
e3 = (c1,3,View,{productID="TV_531"}),
e4 = (c1,4,Search,{query="TV 4K"}),
e5 = (c1,5,View,{productID="TV_789"}),
e6 = (c1,6,Add,{productID="TV_789"})
]
```

TABLEAU 4.1 – Exécution de l'automate de la règle `SearchSeqThenAdd` avec la stratégie *First*

t	Jetons dans l'état `I`	Jetons dans l'état `S1`	Jetons dans l'état `S2`	Jetons dans l'état `F`
0	J1 = []			
1		J1 = [e1]		
2			J1 = [e1]	
3			J1 = [e1]	
4		J1 = [e1, e4]		
5			J1 = [e1, e4]	
6				J1 = [e1, e4, e6]

Comme vu précédemment (cf. section 3.7), selon la stratégie de sélection spécifiée pour la règle (*First* ou *Every*), l'exécution de l'automate identifie des séquences d'évènements différentes dans le flux.

Dans les sections suivantes nous analysons l'exécution de l'automate avec chacune des trois stratégies possibles.

Stratégie *First*

L'automate 4.1 montre illustre l'exécution de l'automate de la figure 4.11.

La première colonne contient les dates simplifiées (que l'on retrouve dans les évènements du flux), et les autres colonnes contiennent les jetons se trouvant dans un état spécifique de l'automate, en l'occurrence `I`, `S1`, `S2` ou `F`. Le jeton arrivant à l'état `F` signifie qu'une séquence conforme à la règle a été identifiée.

Avec la stratégie *First*, l'automate commence par créer un jeton `J1` dans l'état initial `I`. Cet unique jeton sélectionne successivement l'évènement `e1`, puis l'évènement `e4`, avant d'identifier une séquence complète correspondant à la règle avec le dernier évènement `e6`. AU final, une seule séquence est donc identifiée par la règle via l'unique jeton `J1` : `[e1, e4, e6]`

Stratégie *Every*

L'automate 4.2 montre illustre l'exécution de l'automate de la figure 4.11.

Avec la stratégie *Every*, l'automate commence par créer un jeton `J1` dans l'état initial `I`. Ce jeton sélectionne d'abord l'évènement `e1`, ce qui déclenche la création d'un nouveau jeton `J2`. Puis, la sélection de l'évènement `e4` par le jeton `J1` resté dans l'état initial déclenche la création d'un autre jeton `J3`. Mais l'évènement `e4` est au même instant sélectionné par le jeton `J2`, qui déclenche la création de `J4`, et ainsi de suite. Au final, trois séquences sont identifiées via les jetons `J5`, `J6` et `J7` : `[e1, e4, e6]`, `[e1, e6]` et `[e4, e6]`. Notons que la stratégie *Every* ne tient pas compte des transitions-étoiles qui génèrerait de nombreuses

TABLEAU 4.2 – Exécution de l'automate de la règle `SearchSeqThenAdd` avec la stratégie *Every*

t	Jetons dans l'état `I`	Jetons dans l'état `S1`	Jetons dans l'état `S2`	Jetons dans l'état `F`
0	J1 = []			
1	J1 = []	J2 = [e1]		
2	J1 = []	J2 = [e1]		
3	J1 = []	J2 = [e1]		
4	J1 = []	J2 = [e1] J3 = [e4] J4 = [e1, e4]		
5	J1 = []	J2 = [e1] J3 = [e4] J4 = [e1, e4]		
6	J1 = []	J2 = [e1] J3 = [e4] J4 = [e1, e4]		J5 = [e1, e6] J6 = [e4, e6] J7 = [e1, e4, e6]
7	J1 = []	J2 = [e1] J3 = [e4] J4 = [e1, e4]		

séquences inutiles. Cela est le cas car un évènement d'un type non reconnu n'a pas d'effet avec la stratégie *Every* : il est simplement ignoré, et par conséquent n'amènera pas l'automate à créer un nouveau jeton. En revanche, la transition-étoile est important pour la stratégie *First*, car elle permet effectivement d'ignorer les évènements non reconnus entre deux évènements séparés par un opérateur de séquence non contigu.

4.3.3 Matchbuffer

Le matchbuffer est un tableau dont chaque entrée correspond à un évènement de la séquence à identifier. Il est rempli à chaque fois que le jeton franchit une transition. Le matchbuffer est aussi exploité pour vérifier les gardes des transitions. En plus des évènements de la séquence à identifier, il embarque donc aussi les variables et gère leur affectation vers les attributs des évènements.

Le listing 4.8 reprend notre exemple de règle en y ajoutant une contrainte pour illustrer le rôle du matchbuffer. La contrainte de cette règle utilise deux variables (s et a). Le matchbuffer doit réserver deux emplacements mémoires pour ces deux variables.

LISTING 4.8 – La règle `SearchSeqThenAdd2` identifie les recherches successives mais dont la dernière recherche aboutit à un ajout au panier.

```
SearchSeqThenAdd2 :=
Search+:s -> Add:a
# a.sid = s[last].sid
```

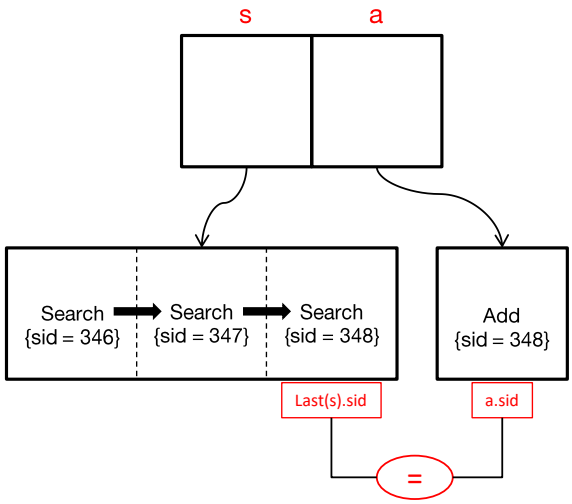

FIGURE 4.12 – Schéma d'un matchbuffer correspondant à la règle `SearchSeqThenAdd2`. La structure de données comporte deux emplacements, un réservé à l'itération `s`, et un réservé à l'atome `a`. Le système est en mesure de vérifier le lien entre `Last(s)` et `a` en récupérant la valeur de l'attribut `sid` de `Last(s)` dans le matchbuffer. Si cette valeur correspond à la valeur de l'évènement `Add` courant, alors celui-ci est sélectionné et enregistré dans l'emplacement `a` du matchbuffer.

Par soucis d'efficacité, nous pré-réservons ces deux emplacements lors de la compilation de la règle. Plus précisément, la structure du matchbuffer est définie à la compilation. Cette structuration statique permet ainsi de pouvoir accéder rapidement aux variables ($O(1)$) en lecture et en écriture lors de l'exécution de l'automate.

La figure 4.12 illustre le matchbuffer correspondant à la règle `SearchSeqThenAdd2`.

4.4 Synthèse

Jusqu'à présent nous avons vu comme le système CEP AUROS identifie des séquences correspondant à des règles de détection dans un flux d'évènements, grâce à un modèle d'automate. Cependant, les exemples que nous avons vu ne considèrent qu'un seul client. En réalité, le flux d'évènements traité par AUROS est massif et est composé des évènements de millions de clients. Pour traiter ce volume important d'évènements, AUROS s'appuie sur un paradigme permettant de traiter le flux d'évènements de manière massivement parallélisée, en distribuant les calculs sur un ensemble de noeuds de calcul. Dans le chapitre 5 suivant, nous décrivons comment le système AUROS passe à l'échelle en s'appuyant sur une plateforme et un modèle de traitement distribués pour traiter un grand nombre de clients et de règles simultanément.

CHAPITRE

5 Une plateforme distribuée pour le système CEP

Ce chapitre décrit l'environnement et le modèle de traitement distribués permettant au système AUROS de passer à l'échelle et d'être tolérant aux pannes. Dans la première section, nous présentons le modèle de traitement distribué des données sur lequel AUROS s'appuie, et justifions les choix technologiques. Puis, nous expliquons la façon dont AUROS utilise ce modèle de traitement distribué, en l'occurrence Spark [1], pour évaluer les règles de détection, qui sont exécutées via un modèle d'automate.

Sommaire

1. Site web du projet Apache Spark : http://spark.apache.org/

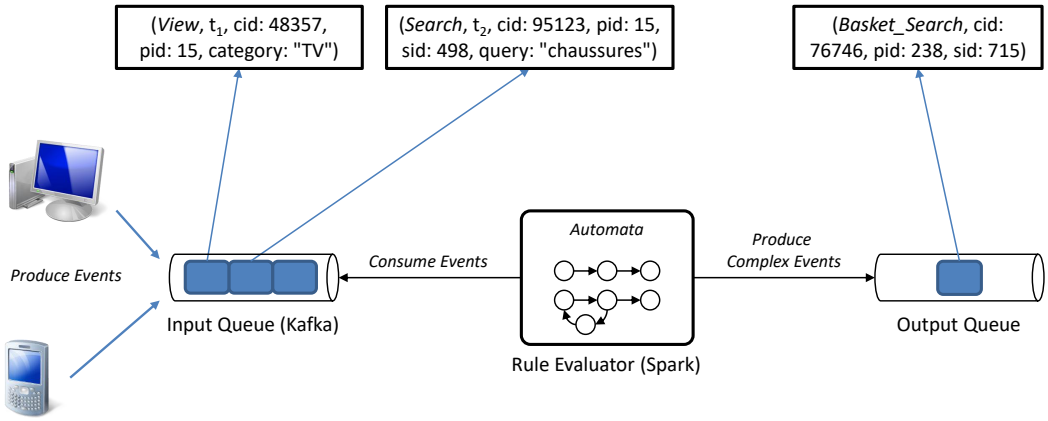

FIGURE 5.1 – Architecture du système AUROS mise en place à Cdiscount. Les clients du site produisent en continu des évènements qui sont récupérés dans une file de traitement distribuée (ici *Kafka*). Ces évènements sont consommés par le système AUROS, déployé sur une plateforme distribuée de traitement des données (Spark). Le *Rule Evaluator* est le composant principal du système permettant la détection de séquences en évaluant les automates obtenus par la compilation des règles de détection spécifiées. L'identification de séquences correspondantes aux règles de détection déclenche la production d'évènements complexes, qui sont mis à disposition dans une file de sortie.

5.1 Architecture distribuée et choix technologiques

La figure 5.1 présente l'architecture sur laquelle repose le fonctionnement du système AUROS, telle qu'elle a été mise en place à Cdiscount.

Cette architecture est composée de trois principaux composants essentiels au fonctionnement du système AUROS:

— L'*Input Queue* est la file de traitement qui regroupe tous les évènements produits par le système de traçage du site Cdiscount. Il constitue le flux d'évènements qui est continuellement consommé en entrée du système. Ce composant s'appuie sur la technologie *Kafka*[2], que nous présenterons plus en détail par la suite.

— Le *Rule Evaluator* est le composant central au système AUROS. Son rôle est d'évaluer, pour chaque client individuel, les règles de détection qui ont été auparavant compilées en modèles d'automate correspondants. Le système AUROS s'appuie sur la plateforme distribuée Spark, dont nous détaillerons le fonctionnement dans la suite de ce chapitre.

— L'*Output Queue* contient les évènements complexes produits par le *Rule Evaluator* pour chaque séquence identifiée. C'est via ce composant que les évènements

2. Site web du projet Apache Kafka : https://kafka.apache.org/

complexes sont rendus disponibles aux utilisateurs du système (en l'occurrence, l'équipe Marketing de Cdiscount). Concrètement, ce composant peut prendre des formes multiples en fonction de l'application. Nous considérons ici qu'il s'agit d'une deuxième file de traitement *Kafka*, dans laquelle les évènements complexes peuvent être consommés en sortie du système.

Cette architecture ainsi que les choix technologiques sous-jacents découlent des exigences formulées par Cdiscount, que nous rappelons ici :

— Les scénarios spécifiés doivent être identifiés en temps réel, c'est-à-dire avec une latence inférieure à une seconde. Cette latence est définie par la différence entre l'instant où un évènement est consommé par AUROS dans l'*Input Queue*, et l'instant où AUROS produit l'évènement complexe correspondant à l'identification d'un scénario, déclenchée par ce même évènement.

— Le système doit être capable de traiter un flux d'évènements dont le débit peut atteindre 10 000 évènements à la seconde (ce chiffre correspont à une projection de Cdiscount sur les 2 années à venir). Pour cela, le système doit être capable de passer à l'échelle, autrement dit d'augmenter ses capacités de traitement en fonction de la charge de travail demandée, qui dépend du débit du flux.

Les sections suivantes détaillent chaque composant de l'architecture mise en place pour le fonctionnement du système AUROS à Cdiscount, en montrant en quoi ces composants répondent aux exigences, et en justifiant les choix technologiques.

5.1.1 Input Queue

Dans le contexte de Cdiscount, les clients du site web sont considérés comme les émetteurs des évènements qui constituent le flux. Le système de traçage du site produit les évènements correspondants aux actions réalisées par les clients, et les écrit dans l'Input Queue. Les évènements écrits dans l'Input Queue sont immédiatement mis à disposition du système AUROS.

L'Input Queue hérite d'une partie des exigences posées pour le système AUROS. En effet, comme ce dernier, il doit être capable de supporter un flux massif, et donc de passer à l'échelle. Autrement dit, ce composant doit pouvoir s'adapter à la charge de travail induite par le nombre et la fréquence des écritures.

Pour cela, nous avons fait le choix de mettre à contribution la plateforme *Kafka*. *Kafka* est un système *publish-subscribe* distribué permettant la parallélisation des écritures et des lectures de messages dans des files de traitement. L'avantage principal de *Kafka* est qu'il permet de partitionner chaque file de traitement en fonction d'une clé. La clé de partitionnement est typiquement calculée à partir d'un ensemble de caractéristiques extraites de chaque message. Les partitions résultantes peuvent ensuite être distribuées sur les différents nœuds d'un cluster *Kafka*, permettant le passage à l'échelle de l'Input Queue. D'autre

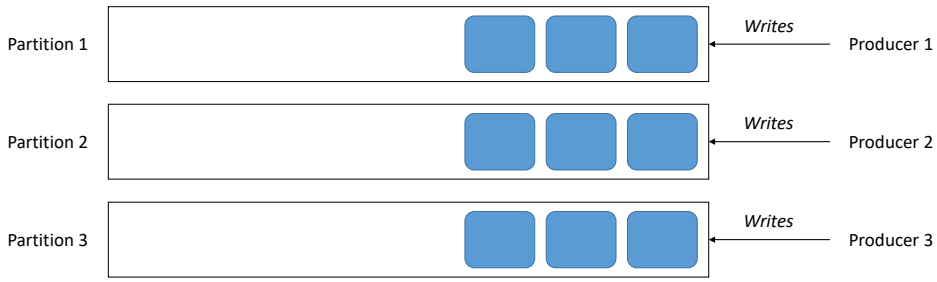

FIGURE 5.2 – Une file de messages composée de 3 partitions, répliquées sur plusieurs serveurs *Kafka*. Les producteurs peuvent écrire des évènements en parallèle dans les différentes partitions. Ainsi, augmenter le nombre de partitions permet d'augmenter le nombre maximum d'écritures en parallèle. Au sein de chaque partition, l'ordre d'écriture des évènements est garanti d'être maintenu.

part, chaque partition est répliquée sur plusieurs noeuds, garantissant la tolérance aux pannes au niveau du flux d'évènements.

La figure 5.2 illustre une file de traitement *Kafka* composée de trois partitions.

Le partitionnement de cette file de traitement est un aspect essentiel pour le système AUROS, car il lui permet effectivement de paralléliser les traitements des évènements se trouvant dans des partitions différentes. Dans le cas de Cdiscount, c'est l'identifiant du client, présent dans tout évènement du flux, qui est utilisé pour générer la clé de partitionnement correspondant à chaque évènement. Cette clé permet à *Kafka*, étant donné un nombre de partitions p, de savoir dans quelle partition parmi les p écrire un évènement donné. Le nombre de clients étant largement supérieur à p, on retrouve donc plusieurs identifiants client différents au sein d'une même partition. Cependant, *Kafka* garantit par ce mécanisme de partitionnement que tous les évènements correspondant à un même identifiant client se retrouvent dans la même partition. De plus, la fonction de hachage utilisée pour calculer la clé de partitionnement permet de garantir que chaque partition contient un nombre d'évènements similaire. Nous verrons que cette propriété est essentielle, car elle contribue à équilibrer la charge au niveau du traitement des évènements par le système AUROS. Enfin, *Kafka* garantit (i) que les évènements d'une partition sont toujours rangés dans l'ordre dans lequel ils ont été écrits, et (ii) que symétriquement, les consommateurs lisent les évènements d'une partition dans l'ordre dans lequel ils ont été écrits. Le Rule Evaluator n'a donc jamais à réordonner les évènements, ce qui aurait induit un coût supplémentaire au niveau du traitement du flux, dont nous parlons dans la section suivante.

5.1.2 Rule Evaluator

Le Rule Evaluator est le composant principal du système AUROS, responsable du traitement du flux d'évènements et de l'identification de séquences pour chaque client. Pour supporter l'exécution de plusieurs règles de détection pouvant chacune engendrer un nombre important de séquences à maintenir par le système (*cf.* section 4.3), il est essentiel que le Rule Evaluator puisse passer à l'échelle. Aussi, il est important que ce composant repose sur un modèle de traitement des données distribué afin de traiter efficacement le flux d'évènements massif de Cdiscount.

Pour faire face à ces enjeux, AUROS se base sur la plateforme open-source Spark pour l'exécution du Rule Evaluator. Spark est un modèle de traitement et un environnement d'exécution distribués permettant de répartir les traitements des données sur un cluster de noeuds de calcul. Plus précisément, Spark permet de déployer des applications sur ces nœuds en créant des unités logiques de traitement, appélées conteneurs, dédiées à chaque application. Il existe deux types de conteneur Spark:

— Les *exécuteurs* sont les unités basiques de traitement. Leur rôle est d'exécuter les tâches de calcul qui leur sont attribuées par le conteneur maître. Plusieurs exécuteurs peuvent être déployés sur chaque nœud, et à chaque exécuteur est alloué un sous-ensemble des ressources disponibles sur le nœud (CPU et mémoire).

— Le conteneur *maître* est central dans le fonctionnement d'une application Spark. Il permet d'ordonnancer et de coordonner les tâches de calcul en les distribuant aux différents exécuteurs.

Les conteneurs Spark sont déployés via un *négociateur* de ressources, chargé d'allouer sur les différents nœuds les ressources nécessaires au fonctionnement de chaque conteneur. Une fois les ressources requises allouées, le négociateur déploie les conteneurs sur les nœuds, puis contrôle leur fonctionnement jusqu'à la fin de l'application.

La figure 5.3 illustre le processus de déploiement d'une application sur la plateforme Spark, qui peut être résumé en 3 étapes :

— L'utilisateur commence par soumettre une application au négociateur de ressources, en demandant un certain nombre d'exécuteurs, et une certaine quantité de ressources (CPU et mémoire) par exécuteur.

— Le négociateur de ressources vérifie que le cluster dispose de suffisamment de ressources. Si cela est le cas, il initialise les exécuteurs demandés, ainsi qu'un conteneur maître pour superviser l'application.

— Le conteneur maître se coordonne avec les exécuteurs et déclenche l'exécution de l'application distribuée.

Le Rule Evaluator est une application Spark permettant de distribuer le traitement du flux d'évènements sur un cluster de noeuds de calcul. Spark permet de distribuer le flux d'évènements consommé dans l'Input Queue afin de paralléliser le traitement des

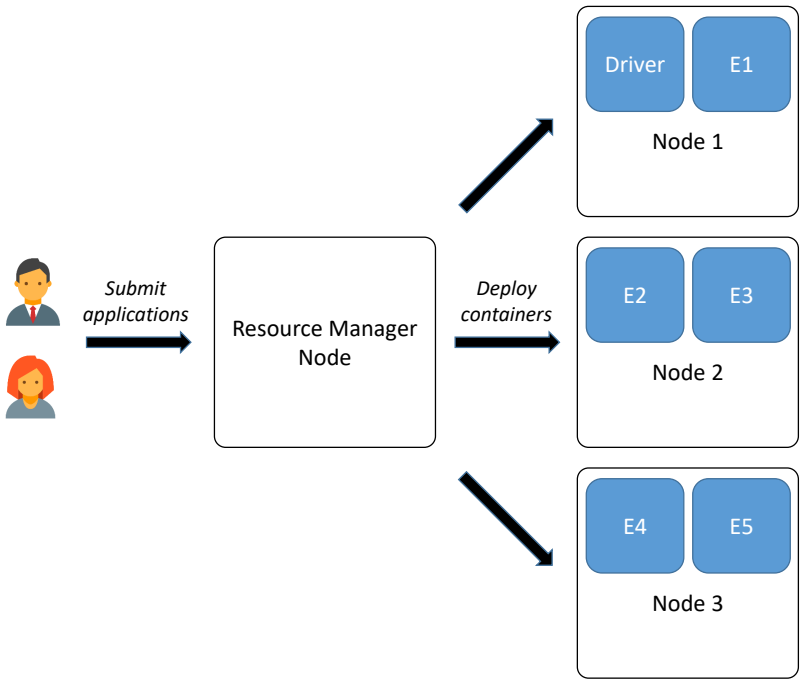

FIGURE 5.3 – Processus de soumission d'une application au négociateur de ressources. L'utilisateur demande l'allocation de 5 exécuteurs sur les trois nœuds de calcul qui composent le cluster. Le négociateur de ressources déploie un conteneur maître et un exécuteur sur le nœud 1, et deux exécuteurs sur les autres nœuds. L'ajout de noeuds au cluster permet de déployer davantage d'exécuteurs, permettant ainsi le passage à l'échelle de l'application, sans modifier celle-ci.

données. Pour cela, le flux d'évènements est découpé en lots d'évènements périodiques, qui sont chacun partitionnés et traités par un ensemble d'exécuteurs. Cependant, en se connectant à une file de traitement *Kafka*, Spark conserve le partitionnement et l'ordre des évènements déjà effectués en amont dans l'Input Queue. Comme déjà mentionné auparavant, le partitionnement des évènements par identifiant client dont le Rule Executor hérite de *Kafka* est judicieux, car il permet à Spark de distribuer équitablement les évènements aux différents exécuteurs. Cela signifie que les exécuteurs ont chacun environ la même quantité d'évènements à traiter, équilibrant ainsi la charge de travail globale. Pour passer à l'échelle, le Rule Executor peut être exécuté sur plus ou moins d'exécuteurs Spark, sans que le composant n'ait besoin d'être modifié à aucun moment.

5.1.3 Output Queue

L'Output Queue est le dernier composant de la chaîne dans l'architecture du système CEP AUROS. Il s'agit d'un module dans lequel les évènements complexes produits par le système sont mis à disposition des utilisateurs. Dans le cas de Cdiscount, ce composant est typiquement une deuxième file de traitement *Kafka*, dans laquelle des applications tierces peuvent consommer les évènements complexes en temps réel. Un critère essentiel pour l'Output Queue est sa capacité, comme l'Input Queue, à supporter un certain nombre d'écritures parallèles effectuées par le Rule Executor. En effet, chaque séquence identifiée par le Rule Executor déclenche la production d'un évènement complexe, qui est immédiatement écrit dans l'Output Queue. Les exécuteurs Spark se connectent chacun directement à l'Output Queue pour y écrire.

5.2 Un modèle de traitement distribué du flux d'évènements

Une application exécutée par la plateforme Spark, telle que le Rule Executor, doit suivre un modèle de traitement spécifique permettant essentiellement de manipuler et de transformer les données, via une chaîne d'opérations pouvant être traduite en un ensemble de tâches parallélisables et distribuées sur les différents exécuteurs. Le Rule Evaluator est donc encodé selon ce modèle, et exécute continuellement une chaîne d'opérations lui permettant de distribuer le processus d'exécution des règles de détection pour l'ensemble des clients du site Cdiscount. Nous détaillons les différentes étapes de cette chaîne d'opérations dans la suite de cette section.

5.2.1 Consommation du flux d'évènements

La première étape réalisée par le Rule Evaluator consiste à consommer les évènements du flux mis à disposition par l'Input Queue. Pour cela, le module de la plateforme Spark dédié au traitement de flux, appelé Spark Streaming, propose le modèle *DStream*. Le Rule Evaluator utilise le modèle DStream pour traiter en continu les évènements du flux, qui sont périodiquement regroupés dans des lots et traités entièrement en mémoire.

La figure 5.4 illustre le modèle DStream de Spark Streaming. Le module Spark Streaming permet de voir le flux d'évènements comme un DStream infini. En réalité, le DStream est une séquence de lots d'évènements représentant chacun une période de 1 seconde du flux d'évènements (cette période est un paramètre de Spark Streaming, fixé ici à 1 seconde conformément à nos objectifs). Spark associe ensuite à chaque lot d'évènements une structure de données appelée *Resilient Distributed Dataset* (RDD). Un RDD est un ensemble de données immuable et partitionné, résidant dans la mémoire de Spark. Cette propriété d'immuabilité implique qu'un lot d'évènements ne supporte que des opérations

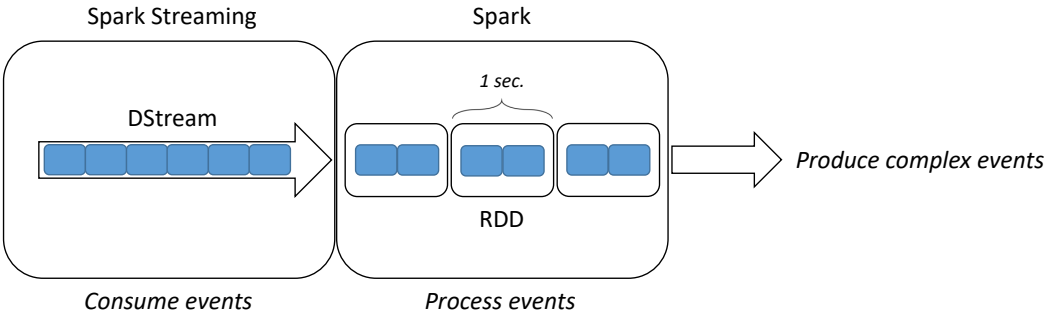

FIGURE 5.4 – Modèle DStream proposé par Spark Streaming et utilisé par le Rule Evaluator. Le flux d'évènements en entrée est représenté par un DStream. Un DStream est en fait une séquence de lots d'évènements partitionnés, appelés RDD. Chaque RDD correspond à une portion du flux d'évènements d'une période de 1 seconde. La plateforme Spark permet de traiter les RDD en appliquant des opérations de transformation succéssives qui sont distribuées sur le cluster. Le Rule Evaluator traite les lots d'évènements pour identifier des séquences pour chaque client. Les évènements complexes produits sont directement écrits dans l'Output Queue.

de transformation, dans le cadre du paradigme Spark. Nous détaillerons ces opérations permettant au Rule Evaluator d'identifier des séquences dans la suite de cette section.

Pour le Rule Evaluator, le DStream créé par Spark Streaming est la représentation de la file de traitement *Kafka* de l'Input Queue. Cette file de traitement étant déjà partitionnée par client, Spark Streaming conserve ce partitionnement pour le DStream correspondant, et mécaniquement, pour les RDD sous-jacents. Autrement dit, il y a une correspondance 1:1 entre les partitions des RDD Spark et les partitions de la file *Kafka*. Cette approche est utile car elle signifie que les garanties proposées par *Kafka* sont héritées par le Rule Evaluator : les partitions Spark contiennent approximativement le même nombre d'évènements, et sont donc traitées par les tâches Spark en un temps similaire. Il y a ainsi un minimum de tâches retardaires, ce qui contribue à assurer la fluidité et la stabilité du système dans son ensemble.

Rappelons également que l'odre des évènements garanti au sein de chaque partition *Kafka* est maintenu à l'identique dans les partitions Spark, évitant au Rule Evaluator d'avoir à effectuer un opératiion de tri coûteuse. Le nombre d'évènements contenus dans chaque RDD dépend quant à lui uniquement du débit du flux, étant donnée la période pour chaque lot d'évènements (fixée à 1 seconde pour le Rule Evaluator).

5.2.2 Chaîne de traitements des lots d'évènements

Le paradigme Spark impose aux applications telles que le Rule Evaluator de construire et structurer une chaîne de traitements à l'aide de fonctions de transformation, s'appli-

quant l'une après l'autre aux RDD à traiter. L'objectif de la chaîne de traitements pour le Rule Evaluator est double. D'une part, elle a bien sûr un objectif fonctionnel, qui est de permettre d'obtenir un résultat, à savoir l'identification de séquences pour chaque client, en transformant successivement un RDD contenant un ensemble d'évènements. D'autre part, elle a un objectif non-fonctionnel, qui est de garantir l'efficacité et la stabilité du système sur le long terme. En effet, la chaîne de traitements doit être la plus efficace possible, afin de réduire les temps de traitement de chaque RDD, et par conséquent la latence d'identification des séquences. De plus, les RDD étant traités les uns à la suite des autres par Spark, les temps de traitement des lots doivent être en moyenne inférieurs à une seconde, conformément à la période de RDD paramétrée, sous peine d'accumuler les lots d'évènements à traiter en entrée du système. L'accumulation de RDD dans la file de traitement de Spark peut compromettre la stabilité du système dans son ensemble, car ces RDD sont gardés en mémoire, qui est en quantité finie.

La chaîne de traitements d'une application est représentée dans Spark par un graphe orienté acyclique (*Directed Acyclic Graph* en anglais, ou *DAG*) d'opérations. Le DAG précise l'ordre dans lequel les différents opérations doivent être successivement exécutées, ainsi que les dépendances entre ces opérations. Il est maintenu et utilisé par le nœud maître pour piloter l'exécution en planifiant et en coordonnant les tâches extraites du DAG sur l'ensemble des exécuteurs.

La chaîne de transformation spécifiée pour le Rule Evaluator est, à haut niveau, composée de deux fonctions principales : `Parse` et `UpdateState`.

La fonction `Parse` consiste à extraire des évènements contenus dans les RDD les données nécessaires et suffisantes à l'identification des règles spécifiées, pour créer des évènements au format AUROS. Elle prend en entrée un RDD d'évènements "bruts", c'est-à-dire directement issus du flux d'évènements, et génère en sortie un RDD d'évènements au format AUROS. Dans le cas de Cdiscount, les évènements bruts sont des documents *JSON*, desquels la fonction `Parse` extrait l'ensemble des attributs nécessaires par rapport aux règles de détection spécifiées.

Puis, la fonction `UpdateState` exécute les automates obtenus par compilation des règles de détection (*cf.* chapitre 4), pour calculer les séquences correspondantes à chaque client à partir des évènements AUROS contenus dans le RDD précédemment calculé. Le résultat de cette fonction est un RDD contenant l'ensemble des états de tous les clients, autrement dit les séquences associées à chaque client, ainsi que leur état courant dans chaque automate actif.

La figure 5.5 décrit le DAG correspondant à cette chaîne de traitements.

Plus spécifiquement, la première fonction `Parse` transforme tout RDD d'évènements bruts, contenant des documents JSON, en RDD contenant des paires clé-valeur de la forme (cid, e), cid étant l'identifiant client correspondant à l'évènement, et e une entité correspondant à l'évènement lui-même. En positionnant cid comme clé de chaque enregistrement dans le RDD, le Rule Evaluator peut efficacement associer un évènement e à son client émetteur. Précisons que les évènements consommés dans le flux par le Rule Eva-

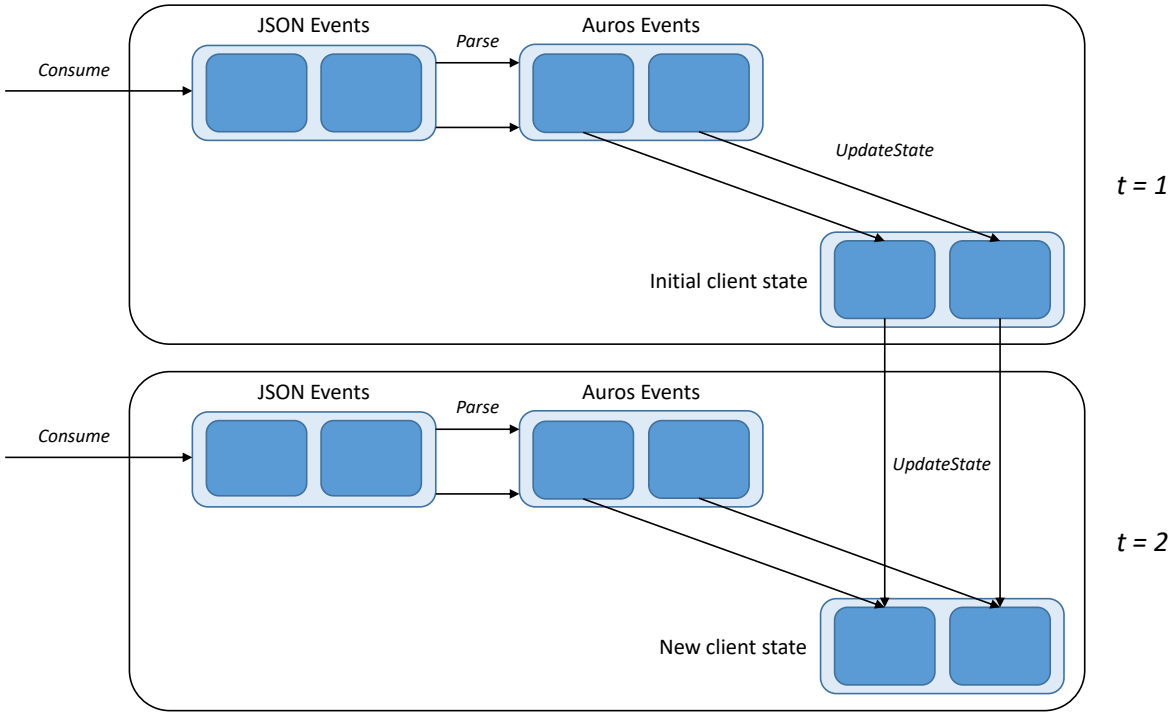

FIGURE 5.5 – DAG d'opérations simplifié généré par Spark à partir de la chaîne de traitements spécifiée pour le Rule Evaluator. Chaque étage représente un RDD associé à une période donnée. À la période $t = 1$, le Rule Evaluator reçoit un premier RDD d'évènements à partir duquel sont calculés les états initiaux des clients, qui sont maintenus dans un RDD spécifique. À $t = 2$, le Rule Evaluator reçoit un second RDD d'évènements. Le RDD d'états client calculé lors de l'étape précédente est réutilisé pour calculer un nouveau RDD contenant les états client mis à jour. La chaîne de traitements continue ainsi indéfiniment pour traiter le flux d'évènements.

luator sont, dans notre cas, considérés valides syntaxiquement et sémantiquement. C'est pourquoi il n'est pas nécessaire d'ajouter d'étapes de filtrage ou de validation des évènements, car ces tâches sont effectuées en amont. La fonction `Parse` est, comme toute opération exécutée par Spark, distribuée sur l'ensemble des exécuteurs dédiés au Rule Evaluator. Pour chaque partition de chaque RDD est créée une tâche qui est attribuée à un exécuteur de telle façon que la charge de travail soit équilibrée au niveau de chaque noeud de calcul. Une fois l'ensemble des tâches correspondantes à la fonction `Parse` terminées, le conteneur maître initialise les tâches correspondantes à la fonction suivante dans le DAG (`UpdateState`).

La deuxième fonction `UpdateState` est centrale au Rule Evaluator. Elle permet d'exécuter les automates obtenus par compilation des règles de détection pour maintenir

et calculer les séquences correspondantes à chaque client. Le RDD d'évènements calculé par la fonction précédente (`Parse`) est passé en entrée de la fonction `UpdateState`, qui transforme tout RDD de paires (cid, e) en RDD contenant l'ensemble des états des clients. Ce RDD particulier s'appelle le `StateRDD`, et est constitué de paires clé-valeur de la forme (cid, s). s est un état client est maintenu dans une structure de données appelée `ClientState`, qui est composée de l'état courant dans chaque automate de règle déployé, ainsi que de l'ensemble des séquences associées à chaque règle (les séquences en question sont les équivalents des jetons décrits dans la section 4.3). Plus précisément, `UpdateState` effectue les opérations suivantes pour toute paire (cid, e) :

— Si un `ClientState` correspondant au `cid` existe déjà dans le `StateRDD` courant, ce `ClientState` est récupéré : il s'agit de l'état actuel du client. Si aucun `ClientState` correspondant au `cid` n'existe (ce qui est lorsqu'un client génère un évènement dans le flux pour la première fois), un nouveau `ClientState` est initialisé pour le client

— Puis, chaque séquence du `ClientState` est mise à jour en fonction des nouveaux évènements : on obtient un nouvel état client s'

— Si des séquences complètes correspondant à une règle de détection sont identifiées, un évènement complexe correspondant est produit et immédiatement diffusé dans l'Output Queue

— Enfin, le `ClientState` mis à jour est enregistré dans le nouveau `StateRDD`.

5.2.3 Tolérance aux pannes

Un aspect important découlant du paradigme Spark est la tolérance aux pannes. En effet, les RDD, qui sont les structures de données de base représentant les ensembles de données manipulées par une application Spark telle que le Rule Evaluator, ont certaines propriétés (notamment l'immuabilité) qui permettent de garantir la tolérance aux pannes pour l'ensemble de la chaîne de traitements.

Dans le cas du Rule Evaluator, la tolérance aux pannes est garantie par trois mécanismes principaux : (i) le maintien dans un RDD spécifiques des positions des têtes de lecture permettant de récupérer les évènements dans chaque partition *Kafka*, (ii) le maintien en mémoire de la chaîne de traitements permettant de calculer un RDD donné, et (iii) la sauvegarde périodique, sur un système de fichiers tolérant aux pannes (tel que *HDFS* [Shvachko *et al.*, 2010]), des états clients maintenus dans le `StateRDD`.

Ainsi, ces mécanismes permettent la récupération du système dans les différents cas de panne pouvant se présenter :

— Dans le cas de la panne d'un noeud de calcul contenant tout ou partie d'un RDD contenant les évènements bruts, le système récupère en redemandant les évènements perdus grâce aux tête des lectures dont la dernière position dans la file de traitement *Kafka* a été sauvegardée.

— Dans le cas de la panne d'un noeud de calcul contenant tout ou partie d'un RDD intermédiaire, tel que celui calculé par la fonction `Parse`, le système peut récupérer en réexécutant la chaîne de traitements ayant permis de calculer ce RDD. Cela implique que chaque RDD d'évènements bruts soit conservé en mémoire Spark pendant une certaine période.

— Enfin, dans le cas de la panne d'un noeud de calcul contenant tout ou partie du `StateRDD`, le système peut récupérer les évènements clients sur *HDFS*, qui est un système de fichiers fiable.

Le troisième mécanisme est le plus coûteux pour le Rule Evaluator, car la sauvegarde périodique des états clients sur un système de fichiers implique des accès disques relativement fréquents (tous les 10 RDD). Ces accès disque augmentent mécaniquement les temps de traitements des RDD déclenchant la sauvegarde, et ce surcoût doit donc impérativement être compensé par le traitement plus rapide des RDD suivants, afin de préserver la stabilité du système sur le long terme.

5.3 Synthèse

Dans ce chapitre, nous avons vu comment le système AUROS s'intègre dans une architecture distribuée pour traiter le flux d'évènements massif de Cdiscount. Le chapitre 6 suivant, nous évaluons cette architecture dans un cas d'utilisation réel, et montrons que les exigences définies par Cdiscount, à savoir l'identification des scénarios en moins d'une seconde, et le passage à l'échelle du système, sont respectées. Nous verrons également que certaines garanties de la plateforme Spark, notamment la tolérance aux pannes, ont un coût significatif qu'il faut prendre en considération pour assurer la fluidité et la stabilité du système AUROS.

CHAPITRE

6 Évaluation d'un cas d'usage réel

Dans cette section nous évaluons le système CEP AUROS par rapport à l'ensemble des exigences formulées par Cdiscount. Pour cela, nous nous appuyons sur un cas d'utilisation réel établi par l'équipe Marketing, ainsi que sur un véritable flux d'évènements également fourni par Cdiscount. Nous clarifions tout d'abord les objectifs et le contexte de l'évaluation, puis les différentes expérimentations que nous avons effectué pour répondre à ces objectifs. Puis, nous discutons des résultats obtenus, et en tirons les leçons en synthèse de ce chapitre.

Sommaire

Pour cette évaluation d'un cas d'usage réel, nous posons les objectifs suivants :

— Mesurer la performance et la capacité de passage à l'échelle d'AUROS en montrant que des scénarios réels peuvent être identifiés par AUROS en temps réel (c'est-à-dire avec un délai inférieur à la seconde), et dans un flux d'évènements massif (c'est-à-dire dont le débit est supérieur à 10 000 évènements par seconde).

— Montrer à travers un cas d'usage réel comment les différentes caractéristiques des règles de détection, telles que la taille de la fenêtre et la stratégie de sélection, influent sur le comportement du système dans son ensemble. Pour cela, nous nous appuyons sur deux mesures clés : le temps de traitement par le système des lots d'évènements qui composent le flux, et la quantité de mémoire globale utilisée par le système.

— Montrer que l'analyse préalable du flux d'évènements est primordiale dans un cas d'utilisation réel tel que celui présenté ici. Connaître le flux permet d'anticiper les besoins en ressources du système et les aléas du flux, tels que les robots pouvant générer une quantité anormale d'évènements par rapport aux clients normaux. Cette analyse du flux s'appuie sur des mesures statistiques effectuées sur le flux d'évènements réel fourni par Cdiscount.

Le cas d'utilisation réel sur lequel se base cette évaluation est composé de deux scénarios typiques de Cdiscount, exprimés sous la forme de règles de détection, via le langage AUROS. Pour évaluer le système AUROS, nous le soumettons à l'identification de ces scénarios dans un flux d'évènements réel fourni par Cdiscount, et observons deux types de mesure :

— La *cadence de production* du système est définie par le temps de traitement d'une portion du flux d'évènements (que nous appelons *lot d'évènements*) par unité de temps.

— La *consommation mémoire globale* du système correspond à la quantité de mémoire totale utilisée par le système, sur l'ensemble des noeuds de calcul sur lequel il est distibué.

Combinées, ces deux métriques permettent d'évaluer les performances d'AUROS étant donnés un ensemble de paramètres précis. Les paramètres que nous faisons varier dans les différentes expérimentations sont les suivants : (i) le débit du flux d'évènements, (ii) la règle de détection elle-même, (iii) l'application d'une fenêtre temporelle sur une règle donnée (*cf.* chapitre 3), (iv) la stratégie définie pour une règle donnée (*cf.* chapitre 4), et (v) des ressources allouées au système (*i.e.* CPU et RAM, *cf.* chapitre 5).

Dans la suite de ce chapitre, nous commençons par décrire la plateforme matérielle et logicielle mise à contribution pour la réalisation de cette évaluation, dans la section 6.1. Puis, nous détaillons le protocole d'évaluation dans la section 6.2, ainsi que la nature du flux d'évènements fourni par Cdiscount dans la section 6.3. Nous discutons ensuite les résultats obtenus pour l'ensemble des expérimentations dans la section 6.4. Enfin, nous synthétisons les résultats et discutons les limites du système AUROS dans la section 6.5.

6.1 Environnement d'exécution d'AUROS

Pour réaliser cette évaluation, nous disposons d'un cluster constitué de 12 serveurs *DELL C6220*, possédant chacun 24 CPU, 64 gigaoctets de RAM, et 3 téraoctets dédiés au stockage.

Les ressources du cluster sont allouées au système AUROS par le négociateur de ressources YARN (Yet Another Resource Negociator) [1]. YARN permet de paramétrer à la demande, au moment du déploiement du système AUROS, l'ensemble des ressources qui lui sont dédiées (CPU et RAM).

L'exécution d'AUROS repose sur une plateforme Spark Streaming (version 2.0). La durée d'un lot d'évènements est paramétré à un intervalle de 1 seconde, et Spark Streaming sauvegarde périodiquement les données maintenues du système sur le système de fichier distribué *HDFS* (*cf.*). Au cours de l'exécution, l'état global est maintenu en mémoire par Spark sous la forme d'objets sérialisés au format *Kryo* [2].

À chaque exécuteur Spark sont attribués (cf. section 5.1.2) 5 CPU et 10 gigaoctets de RAM. Le nombre maximum de partitions Spark est configuré en fonction du nombre total de CPU alloués à AUROS, et dépend donc du nombre d'exécuteurs. Ainsi, si n est le nombre d'exécuteurs dédiés à AUROS, alors le nombre de partitions est $p = n \times 5 \times 2$ (*i.e.* chaque CPU alloué au système traite en moyenne 2 partitions : c'est la configuration recommandée par Spark pour équilibrer plus efficacement la charge sur les différentes unités de calcul).

Le flux d'évènements est géré par un unique serveur *Kafka*, sans réplication des données. Le nombre de partitions de la file *Kafka* doit correspondre au nombre de partitions Spark (*cf.* section 5.1.1), et est donc égal à p.

6.2 Protocole d'expérimentation

Dans le cadre de cette évaluation, nous déployons dans le système AUROS deux règles de détection exprimées par Cdiscount.

LISTING 6.1 – La première règle permet d'identifier les recherches menant indirectement (c'est-à-dire en passant par une fiche produit correspondante) à un ajout au panier.

```
SearchViewAdd :
    Search:s -> View:v -> Add:a
    # s.page_url = v.referrer
    # v.page_url = a.referrer
```

La première règle, décrite dans le listing 6.1, correspond à un scénario classique de navigation décrit par l'équipe Marketing de Cdiscount. Dans ce scénario, le client commence

1. Site web du projet Apache YARN : http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
2. Site web du projet Kryo : https://github.com/EsotericSoftware/kryo

par effectuer une recherche via le moteur de recherche du site, puis clique sur un des résultats affichés, l'amenant sur la fiche produit correspondante. Enfin, le client ajoute le produit au panier.

Grâce à cet règle, AUROS peut identifier l'ensemble des ajouts panier *indirects*, c'est-à-dire passant par une fiche produit (contrairement aux ajouts panier *directs*, qui sont effectués directement depuis la page de résultats de recherche) qui sont liés à une recherche. Ce type de règle est typiquement utilisé par l'équipe Marketing pour mesurer en temps réel le taux d'efficacité du moteur de recherche proposé aux clients du site ecommerce.

LISTING 6.2 – La deuxième règle permet de détecter une succéssion de trois visionnages d'une même fiche produit, sans ajout au panier, ce qui correspond à un comportement d'hésitation.

```
ViewSeq :
    View+:v
    # v[i].url = v[i-1].url
    # Size(v, 3)
    # Neg(v, Add:a # a.ref = v[0].url)
```

La deuxième règle, spécifiée dans le listing 6.2, décrit un comportement d'hésitation d'achat du client. Dans ce scénario, le client revient au moins trois fois de suite sur une même fiche produit, sans ajouter le produit en question au panier.

Via cette règle, AUROS est capable de proposer automatiquement une aide interactive au client, l'aidant ainsi à mieux évaluer le produit sur lequel il hésite.

Ces deux règles font usage de différents éléments du langage AUROS dont l'impact au niveau de la performance du système est plus ou moins important.

Comme dit précédemment, nous faisons varier plusieurs paramètres différents de ces règles pour évaluer AUROS dans des contextes différents :

— Le débit du flux d'évènements à analyser,

— La règle de détection à évaluer,

— La stratégie de sélection associée à la règle de détection,

— La taille de la fenêtre sur laquelle s'applique la règle de détection.

Pour chaque expérimentation, le système AUROS exécute via Spark la même chaîne d'opérations de transformation, décrite dans la section 5.2.2. Cette chaîne d'opérations consiste dans un premier temps à créer un DStream représentant le flux d'évènements Cdiscount, consommé dans la file de traitement *Kafka*. Chaque RDD du DStream représente un lot d'évènements, dont la taille dépend du débit du flux. Les RDD sont d'abord transformés par la fonction `Parse`, qui extrait, formatte et filtre les évènements consommés. Puis, la fonction `UpdateState` permet, en mettant à contribution les automates AUROS représentant les règles de détection spécifiées, de calculer un nouveau RDD contenant l'ensemble des séquences pour tous les clients, formant ainsi l'*état global* du système, qui

est maintenu en mémoire. Par ailleurs, Spark dispose d'un mécanisme permettant de sauvegarder périodiquement cet état global à intervalles réguliers, afin de garantir la tolérance aux pannes du système.

6.3 Flux d'évènements

Simulation du flux à partir d'une trace réelle

Le flux d'évènements utilisé dans le cadre de cette évaluation est issu d'une véritable trace, générée par l'activité des clients du site Cdiscount pendant le mois de janvier 2016. Cette activité a été enregistrée dans un fichier d'une taille d'environ 200 gigaoctets, qui est la source pour l'ensemble des expérimentations qui seront présentées dans cette évaluation. Notons que le mois de janvier est un mois d'activité particulièrement élevée pour Cdiscount car il s'agit d'une période de soldes. D'autre part, afin de refléter le plus possible un cas d'utilisation réel, le flux d'évènements traité par AUROS est reproduit tel qu'il a été fourni par Cdiscount, c'est-à-dire sans étapes préalables de filtrage ou de modification quelconque des données.

LISTING 6.3 – Un évènement extrait de la trace fournie par Cdiscount. Il s'agit d'un document JSON comportant de nombreux attributs. Seuls quelques attributs utiles aux règles de détection spécifiées sont montrés ici.

```
{
    "type" : "view",
    "page_url" : "http://www.cdiscount.com/electromenager/four/
sauter-sfp945x-four-multifonction/f-110230401-sau3660767560550.html,
    "referrer" : "http://www.cdiscount.com/",
    "visid_low" : "4611687166257563908",
    "visid_high" : "3044855440395536440",
    "date_time" : "2016-01-01 18:14:45"
}
```

Le listing 6.3 présente un évènement tiré du fichier d'évènements fourni par Cdiscount. Chaque évènement est un document *JSON*, généré par le système de traçage intégré à la plateforme web ecommerce. Pour rappel, les actions effectuées par les clients du site en naviguant sur les différentes pages web du site déclenchent continuellement la production de tels évènements, qui sont envoyés directement dans une file de traitement.

Les évènements du flux comportent chacun plus de 100 attributs, qui décrivent en détail le contexte web du client au moment de la réalisation de l'action. Par soucis de clarté, seuls les attributs utiles aux règles de détection soumises pour l'évaluation sont ici représentés. Parmi ceux-ci, on retrouve (comme décrit dans la définition 2.1) (i) le type de l'évènement (attribut `type`, ici `view`, signifiant le visionage d'un produit), (ii) la date correspondant à l'action (attribut `date_time`), (iii) l'identifiant client, qui est ici composite (attributs `visid_low` et `visid_high`), et (iv) des données de contexte, par exemple l'URL de la page

TABLEAU 6.1 – Statistiques présentant le nombre d'évènements par client dans le flux d'évènements.

	Cumul	Maximum	Moyenne	Médiane	99e centile
Nb. év. par client	105 521 662	9 966	8	3	89
Nb. type `search` par client	22 818 957	3 282	1	0	25
Nb. type `view` par client	72 648 015	9 949	5	2	59
Nb. type `add` par client	10 054 690	2 294	0	0	13

visitée (attribut `page_url`), et l'URL de la page précédemment visitée par le client (attribut `referrer`).

L'évaluation diffère cependant d'un scénario réel sur un aspect en particulier : le flux étant simulé à partir d'un fichier, son débit ne correspond pas au débit véritable. Le véritable débit du flux de Cdiscount est en moyenne d'environ 1 000 évènements par seconde, et varie au cours de la journée selon l'intensité de l'activité des clients sur le site. En particulier, le flux d'activité de Cdiscount est fortement réduit en période nocturne, et est le plus important en fin d'après-midi. Hors, dans le cadre de cette évaluation, nous considérons que le débit est constant pendant toute la durée de chaque expérimentation. Nous faisons varier ce débit selon l'expérimentation, de 10 000 évènements par seconde (ce chiffre correspondant à la projection maximale à 2 ans estimée par Cdiscount) jusqu'à 50 000 évènements par seconde. Ce débit est contrôlé au niveau des producteurs *Kafka*, qui sont chargés de lire les évènements dans le fichier fourni par Cdiscount, et de les écrire dans la file de traitement *Kafka* dédiée, simulant ainsi un flux. De cette façon, AUROS peut continuellement récupérer dans cette file les évènements à traiter.

Analyse descriptive du flux

Afin d'avoir une vision globale de la constitution du flux en termes d'évènements et de clients, nous en présentons dans le tableau 6.1 une analyse descriptive. Ce flux représente 12 509 022 clients, et est composé de 105 521 662 évènements de type `search`, `view`, ou `add` (les évènements d'autres types ne sont pas considérés dans la suite de cette évaluation, car ils seront ignorés par le système).

En plus du résumé descriptif du contenu de l'ensemble du flux, il est utile d'étudier l'impact des règles de détection exprimées par rapport au flux d'évènements analysé.

Le tableau 6.2 présente une analyse statistique concernant la règle `SearchViewAdd` décrite dans le listing 6.1. Le nombre total de séquences d'évènements créées par AUROS pour cette règle en analysant le flux d'évènements est 48 085 093.

De même, le tableau 6.3 présente une analyse statistique concernant la règle `ViewSeq`, décrite dans le listing 6.2. En évaluant le flux d'évènements, le système a engendré un total de 109 697 813 séquences correspondant à cette seconde règle.

TABLEAU 6.2 – Statistiques présentant pour la règle `SearchViewAdd` le nombre de séquences par client et les tailles des séquences engendrées par le système AUROS en analysant le flux d'évènements.

	Cumul	Maximum	Moyenne	Médiane	99e centile
Nb. séq. par client	48 085 093	288	1,4	1	5
Taille séquence	64 937 519	2	1,4	1	2

TABLEAU 6.3 – Statistiques présentant pour la règle `ViewSeq` le nombre de séquences par client et les tailles des séquences engendrées par le système AUROS en analysant le flux d'évènements.

	Cumul	Maximum	Moyenne	Médiane	99e centile
Nb. séq. par client	109 697 813	16 191	2,3	2	8
Taille séquence	129 732 556	2	1,2	1	2

Cet ensemble de statistiques permet à l'utilisateur d'AUROS, en l'occurrence l'équipe Marketing de Cdiscount, de mettre en évidence l'impact d'une règle donnée par rapport au flux d'évènements donné, en permettant notamment d'estimer leur emprunte mémoire. Étudier l'impact des règles de détection déployées est surtout crucial pour garantir la robustesse et la stabilité d'AUROS dans un scénario réel. En effet, ces statistiques permettent de mettre en relief certaines phénomènes aberrants, en particulier en ce qui concerne la nombre de séquences par client. Par exemple, on peut observer que le nombre maximum de séquences par client pour la règle `ViewSeq` est 16 191, à comparer avec un 99ème centile à 8. Ce nombre est la manifestation d'un *robot* effectuant des actions automatisés sur le site, et générant ainsi un nombre anormal d'évènements, ce qui amène le système à générer exponentiellement de nouvelles séquences. Étant en très faible proportion par rapport à l'ensemble des clients, les robots peuvent être difficilement observables sans analyse préalable du flux d'évènements. Les robots sont problématiques pour le système AUROS, car ils sont susceptibles de provoquer des pics au niveau des temps de traitement des lots d'évènements, pouvant ralentir, voire rendre totalement indisponible le système.

Afin de rendre le système robuste aux clients automatisés pouvant mettre à mal son fonctionnement, nous faisons le choix de limiter le nombre maximal de séquences qu'un client peut engendrer, en fonction de la règle. Ce seuil est déterminé indépendamment pour chaque règle, sur la base des statistiques correspondantes, en particulier celle du 99ème centile. Concrètement, cela signifie qu'AUROS garantit la couverture de la totalité des séquences identifiées pour 99% des clients, les 1% des clients restants n'étant donc pas couverts en totalité (notons que ces 1% sont tout de même partiellement couverts, pour un sous-ensemble des séquences qu'ils génèrent).

6.4 Expérimentations et résultats

Les expérimentations que nous présentons dans cette section sont divisées en deux catégories : les expérimentations permettant d'observer les temps de traitement des lots Spark Streaming, et les expérimentations permettant d'observer la consommation mémoire globale. Pour rappel, un lot d'évènements correspond à un RDD, contenant un ensemble d'évènements consommés dans la file de traitement *Kafka*. Pour AUROS, chaque RDD correspond à une période de 1 seconde. La consommation mémoire globale quant à elle correspond à la quantité de mémoire consommée par la totalité du système, c'est-à-dire la quantité obtenue en additionnant celle de tous les exécuteurs Spark à un instant donné.

6.4.1 Mesures des temps de traitement des lots d'évènements

Les temps de traitement des lots d'évènements correspondent au temps mesuré par Spark pour la complétion de l'ensemble des tâches liées au lot d'évènements. Pour rappel, les RDD sont partitionnés, et chaque partition subit une série de transformations qui sont traduites en tâches. Ainsi, chaque mesure d'un temps de traitement inclut essentiellement (i) les temps d'initialisation et d'ordonnancement des tâches sur les différents exécuteurs, (ii) les temps de calcul, qui constituent l'essentiel des temps d'exécution des tâches (et donc du lot d'évènements), (iii) les temps des éventuels transferts réseau de données entre les noeuds, lorsque les tâches qui s'enchaînent le requièrent, et (iv) le temps de sauvegarde sur disque des données de l'état global, qui a lieu ponctuellement (1 lot sur 10 déclenche la sauvegarde de l'état global par Spark).

La figure 6.1 présente les résultats obtenus pour l'ensemble des expérimentations visant à mesurer les temps de traitement des lots d'évènements, en fonction de plusieurs critères, à savoir (de haut en bas et de gauche à droite) (i) le débit du flux d'évènements analysé, (ii) la règle de détection évaluée, (iii) la stratégie associée à la règle de détection, et (iv) la fenêtre temporelle sur laquelle s'applique la règle de détection.

Le graphe (1) de la figure 6.1 présente un ensemble de cinq boîtes à moustaches, calculées pour les débits (de bas en haut, en évènements par seconde) [10 000, 20 000, 30 000, 40 000, 50 000]. Chaque boîte à moustaches représente, pour un débit donné, l'ensemble des temps de traitement des lots mesurés durant 5 minutes d'expérimentation. Dans ce premier graphe, les lots sauvegardés périodiquement sur disque par Spark (et dont les temps de traitement sont supérieurs) ne sont pas représentés, car il s'agit ici de mesurer les temps de traitement "purs", c'est-à-dire ceux qui ne dépendent pas de la taille de l'état global à sauvegarder (et qui sont largement majoritaires, puisque pour rappel seul 1 lot sur 10 déclenche la sauvegarde de l'état global par Spark). On observe que les temps de traitement des lots d'évènements augmentent en fonction du débit du flux. Cela s'explique naturellement par le fait que plus le lot contient d'évènements, plus Spark passe de temps à le traiter. Ainsi, lorque le débit est de 10 000 évènements par seconde, chaque lot contient

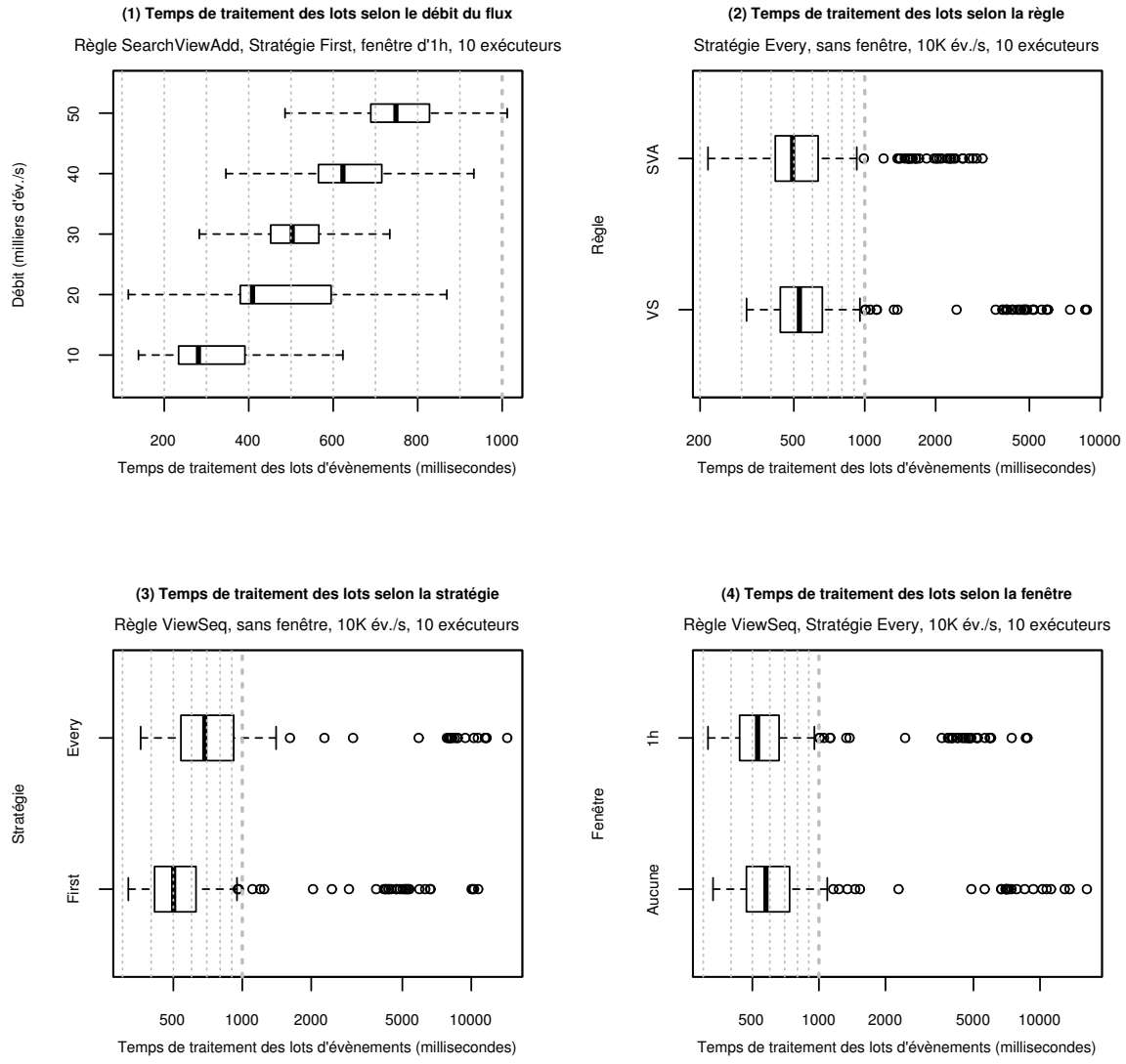

FIGURE 6.1 – Mesures des temps de traitement des lots d'évènements en fonction (1) du débit du flux (2) de la règle de détection (3) de la stratégie associée à la règle et (4) de la fenêtre temporelle. (1) On observe que l'augmentation du débit entraîne une augmentation des temps de traitement des lots. (2) La règle évaluée a un effet marqué sur les temps de traitement des lots déclenchant la sauvegarde de l'état global (points aberrants). (3) La stratégie *Every* entraîne une augmentation des temps de sauvegarde de l'état global par rapport à la stratégie *First*. (4) L'application d'une fenêtre temporelle permet de réduire voire de borner les temps de sauvegarde de l'état global.

10 000 évènements pour lesquels Spark doit exécuter les deux fonctions principales `Parse` et `UpdateState` (*cf.* section 5.2.2) : la médiane des temps de traitement des lots pour la règle `SearchViewAdd` (avec la stratégie *First*) est de 281 millisecondes. Lorsque le débit du flux est de 50 000 évènements par seconde, la médiane des temps de traitement des lots (qui contiennent donc chacun 50 000 évènements) est de 748 millisecondes. En disséquant chaque de temps de traitement mesuré, on observe que c'est le temps de calcul qui est le plus impacté par l'exécution des fonctions `Parse` et `UpdateState`, suivi du temps de transferts réseau des partitions (les partitions comportant plus d'évènements mettent plus de temps à être transférées). Le temps d'initialisation des tâches Spark qui permettent le traitement du lot est lui fixe, quelque soit le débit du flux (car il dépend strictement d'un ensemble de paramètres Spark qui sont indépendants de la charge de travail). Pour aller plus loin dans l'analyse de l'évolution du temps de traitement des lots en fonction du débit du flux d'évènements, nous posons l'hypothèse qu'il existe une relation de corrélation entre ces deux variables. Pour vérifier cette hypothèse, nous utilisons un outil statistique appelé *coefficient de Spearman* [Spearman, 1904]. Le coefficient de Spearman permet de mesurer la dépendance statistique entre deux variables, ici entre le temps de traitement des lots et le débit du flux, en se basant sur les rangs des valeurs de chaque variable. Ce test spécifique permet d'obtenir un coefficient ρ compris entre -1 et 1 : une valeur positive indique que les deux variables évoluent dans la même direction (lorsque l'une augmente, l'autre augmente également) ; une valeur négative indique que les deux variables évoluent dans une direction contraire (lorsque l'une augmente, l'autre diminue) ; et une valeur égale à 0 signifie qu'il n'y a aucune relation de corrélation entre les deux variables. Le coefficient de Spearman obtenu pour les 5 expérimentations représentées sur le graphe (1) est $\rho = 0.6118409$, indiquant une relation de corrélation positive entre temps de traitement des lots et débit du flux, ce qui confirme notre hypothèse de départ.

Le graphe (2) de la figure 6.1 présente deux boîtes à moustaches, calculées pour les deux règles de détection exprimées par l'équipe Marketing de Cdiscount: `SearchViewAdd` (SVA) et `ViewSeq` (VS). Contrairement au graphe (1), les boîtes à moustaches incluent ici les temps de traitement des lots déclenchant la sauvegarde périodique de l'état global sur disque. Ces temps de traitement particuliers sont ici considérés comme des points aberrants (représentés par des cercles à l'extérieur des boîtes à moustaches) car les sauvegardes de l'état global sont périodiques, ont un temps de traitement plus élevé que les lots "normaux", et concernent une minorité des lots traités. On constate que pour un débit (10 000 évènements par seconde) et une stratégie (*Every*) donnés, les temps de traitement des lots varient en fonction de la règle de détection. En particulier, les temps de traitement des lots qui déclenchent une sauvegarde de l'état global sont plus élevés pour la règle `ViewSeq` que pour la règle `SearchViewAdd`. En revanche, les temps de traitement sur l'ensemble des lots sont similaires pour les deux règles, même si l'on observe une légère augmentation pour la règle `ViewSeq`. Cela s'explique essentiellement par le fait que la règle `ViewSeq` génère un nombre de séquences plus élevé que la règle `SearchViewAdd`, comme mis en évidence par l'analyse descriptive du flux (*cf.* section 6.3). En effet, le nombre de séquences maintenues

par AUROS pour la règle `ViewSeq` est plus de deux fois supérieur au nombre de séquences maintenues pour la règle `SearchViewAdd`, amenant à une taille d'état global supérieure, et donc à des temps de sauvegarde globalement plus importants. Ainsi, les temps de sauvegarde pour la règle `ViewSeq` sont en moyenne de 4 230 millisecondes avec un maximum de 8 771 millisecondes, alors que ceux de la règle `SearchViewAdd` sont en moyenne de 1 995 millisecondes avec un maximum de 3 163 millisecondes. En ce qui concerne l'ensemble des temps de traitement des lots, les médianes se situent à 494 millisecondes pour la règle `SearchViewAdd` et 528 millisecondes pour la règle `ViewSeq`, ce qui montre que les coûts de l'opérateur `Kleene` et de la négation spécifiés dans la règle `ViewSeq` sont ici relativement marginaux. Ceci est un résultat attendu, puisque la création de nouvelles séquences ainsi que le calcul des nouveaux états des clients sont des opérations peu coûteuses pour les automates AUROS. Globalement, les temps de traitement des lots sont donc davantage influencés par la taille de l'état global (qui dépend des types des évènements à identifier dans la règle) que par la complexité de la règle de détection et des opérateurs qui la composent. En l'occurrence, comme indiqué lors de l'analyse descriptive du flux, le nombre d'évènements `view` est plus élevé que le nombre d'évènements `search`, et la règle `ViewSeq` amène donc à maintenir davantage de séquences par client (2,3 en moyenne, contre 1,4 pour la règle `SearchViewAdd`).

Le graphe (3) de la figure 6.1 présente à nouveau deux boîtes à moustaches, chacune représentant une stratégie différente pour une règle de détection donnée, à savoir la règle `ViewSeq`. Comme pour le graphe précédent, tous les temps de traitement des lots sont inclus, y compris ceux incluant les temps de sauvegarde de l'état global par Spark. On observe que la stratégie a un effet sur les temps de traitement des lots, en particulier en ce qui concerne les temps de sauvegarde de l'état global. Cela s'explique essentiellement par le fait que la stratégie *Every* amène le système à maintenir un nombre plus important de séquences que la stratégie *First*, car cette dernière est limitée à une unique séquence par client (*cf.* section 3.7 et section 4.3). Comme déjà mis en évidence dans l'expérimentation précédente, l'augmentation du nombre de séquences maintenues contribue à l'augmentation de la taille de l'état global, et donc à des temps de sauvegarde plus longs. D'autre part, la stratégie *Every* nécessite que toutes les séquences d'un client soient vérifiées pour chaque nouvel évènement, afin d'identifier toutes les séquences possibles (*cf.* exécution de l'automate avec la stratégie *Every*, section 4.3). Le coût de ces itérations est répercuté au niveau de la fonction `UpdateState`, et donc au niveau de l'ensemble des temps de traitement des lots d'évènements. Ainsi, pour un débit de 10 000 évènements par seconde, la médiane sur l'ensemble des temps de traitement des lots est de 502 millisecondes avec la stratégie *First*, et de 686 millisecondes avec la stratégie *Every*. Les temps de sauvegarde sont eux mesurés en moyenne à 5 093 millisecondes pour la stratégie *First*, et 8 320 millisecondes pour la stratégie *Every*.

Le graphe (4) de la figure 6.1 présente également deux boîtes à moustaches, représentant l'ensemble des temps de traitement des lots (sauvegardes de l'état global y compris). L'objectif est ici de mesurer l'impact de la fenêtre temporelle associée à une règle de dé-

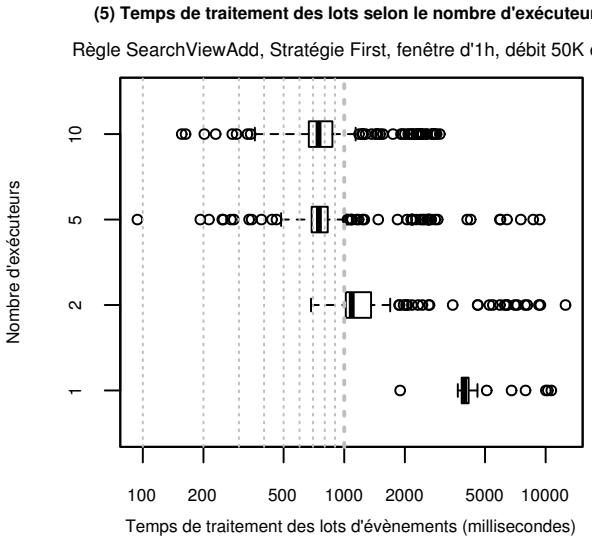

FIGURE 6.2 – Mesures des temps de traitement des lots d'évènements en fonction du nombre d'exécuteurs Spark alloués au système AUROS, et déployés sur le cluster par le négociateur de ressources. Les temps de traitement diminuent clairement dans un premier temps, puis stagnent à partir du moment où suffisamment de CPU sont mis à contribution (à partir de 5 exécuteurs, soit 25 CPU).

tection donnée, en l'occurrence la règle `ViewSeq` (stratégie *Every*). On observe que la définition d'une fenêtre temporelle limitant l'application de la règle dans le temps a un impact considérable sur les temps de traitement des lots d'évènements. En effet, l'application d'une fenêtre temporelle sur la règle a pour effet de limiter la taille de l'état global, car AUROS est alors en mesure de continuellement supprimer les séquences expirées (*cf.* section 3.5 et section 4.2.1). Par conséquent, les temps de sauvegarde de l'état global sont moins importants avec une fenêtre : ils sont en moyenne de 4 230 millisecondes et plafonnent à 8 771 millisecondes au maximum. Sans fenêtre, la taille de l'état global n'est pas limité et augmente continuellement, amenant les temps de sauvegarde à être en moyenne de 7 819 millisecondes, et d'atteindre jusqu'à 16 290 millisecondes pour la même durée d'expérimentation. En ce qui concerne l'ensemble des temps de traitement des lots, les médianes se situent à 528 millisecondes avec une fenêtre, et 576 millisecondes sans fenêtre. Ce léger écart, relativement insignifiant, pourrait s'expliquer par le fait que la fenêtre réduit le nombre moyen de séquences à itérer en évaluant les automates AUROS, pour chaque lot (contenant ici 10 000 évènements chacun).

Enfin, le graphe (5) de la figure 6.2 présente un ensemble de boîtes à moustaches permettant de visualiser les temps de traitement des lots pour 4 expérimentations, chacune mettant à contribution un nombre d'exécuteurs Spark différent pour le fonctionnement du système AUROS. On observe que pour un débit de 50 000 évènements par seconde, le

système est incapable de traiter les lots suffisamment rapidement (c'est-à-dire en moins d'une seconde) lorsque seul(s) 1 ou 2 exécuteur(s) lui sont dédiés. En effet, avec 1 exécuteur, les médianes des temps de traitement des lots se situe à 3 939 millisecondes, et à 1 090 millisecondes avec 2 exécuteurs. Si l'on ne considère que les lots déclenchant une sauvegarde de l'état global, les moyennes se situent à 6 876 millisecondes avec 1 exécuteur et 4 963 millisecondes avec 2 exécuteurs. En revanche, lorsque 5 ou 10 exécuteurs sont alloués à AUROS, le système est stable avec des médianes à 748 et 745 millisecondes, respectivement. Si la différence entre 5 et 10 exécuteurs est insignifiante en ce qui concerne les temps de traitement des lots "normaux", elle est plus significative lorsque l'on considère uniquement les lots déclenchant une sauvegarde de l'état global. Les moyennes se situent alors à 2 914 millisecondes avec 5 exécuteurs, et à 1 606 millisecondes avec 10 exécuteurs, avec des maximums à 9 337 millisecondes et 2 982 millisecondes, respectivement. Ces chiffres montrent la stabilité et la performance améliorées du système lorsque 10 exécuteurs sont mis à contribution, par rapport à 5, 2 ou 1 exécuteur(s). C'est pour cette raison que l'ensemble des expérimentations présentées dans cette évaluation sont effectuées sur une plateforme Spark composée de 10 exécuteurs. Cela permet au système de disposer d'une quantité de mémoire confortable pour le maintien de l'état global, tout en obtenant des performances proches de l'optimal en ce qui concerne le traitement des lots et les sauvegardes de l'état global. D'autres expérimentations, qui ne sont pas représentées ici, montrent une légère dégradation des performances lorsque le nombre d'exécuteurs est supérieur à 10. Cela s'explique par le fait que les opérations exécutées par Spark dans le cadre de cette opération (`Parse` et `UpdateState`) sont relativement peu consommatrices en termes de temps de calcul CPU. L'attribution de davantage de CPU n'améliore donc pas les performances, puisqu'il existe un seuil incompréssible au-delà duquel le temps d'exécution des tâches Spark ne peut plus être réduit. Au contraire, les performances de Spark diminuent lorsque trop de tâches lui sont attribuées en parallèle, car les coûts supplémentaires liés à l'ordonnancement de ces tâches et aux transferts réseaux surpassent alors les gains en termes de temps de calcul. L'utilisation de plus de 10 exécuteurs est donc dans le cas présent inutile voire nuisible au système, du point de vue des temps de traitement des lots. Pour allouer de la mémoire supplémentaire au système, il est donc préférable d'ajouter de la mémoire aux exécuteurs déjà déclarés. Enfin, notons que dans le cas d'une augmentation du débit du flux, ou d'ajout d'étapes de pré-traitement supplémentaires dans la chaîne d'opérations d'AUROS (*e.g.* enrichissement des évènements, filtrage des évènements), le système pourrait alors nécessiter davantage de puissance de calcul (CPU), qu'il est possible d'allouer au système en ajustant le nombre d'exécuteurs qui lui sont dédiés.

6.4.2 Mesures de la consommation mémoire globale

Dans cette section, nous nous intéressons à l'évolution de la consommation mémoire globale du système au cours du temps. Pour cela, nous mesurons la quantité de mémoire utilisée par chaque exécuteur Spark mis à contribution pour le fonctionnement d'AUROS,

et agrégeons les valeurs mesurées à chaque instant. Pour rappel, l'état global d'AUROS permet de maintenir l'ensemble des séquences créées, pour tous les clients et pour l'ensemble des règles de détection. L'état global est un RDD partitionné dont les données sont stockées en mémoire, sous forme sérialisée, et distribuées sur les différents exécuteurs. Comme expliqué lors des expérimentations précédentes, c'est le RDD représentant l'état global qui est sauvegardé périodiquement par Spark sur disque dur afin de garantir la tolérance aux pannes. Les expérimentations présentées ici visent à montrer comment la taille de l'état global évolue au cours du temps en fonction de la règle de détection évaluée, de la stratégie (*First* ou *Every*) associée à la règle, ou la taille de la fenêtre temporelle spécifiée pour la règle.

Le graphe (1) de la figure 6.3 présente deux courbes permettant de comparer la consommation mémoire globale pour les deux règles évaluées : `SearchViewAdd` et `ViewSeq`. La courbe en pointillés longs représente la consommation mémoire globale pour la règle `ViewSeq` (VS), et la courbe en trait continu représente celle de la règle `SearchViewAdd` (SVA). Le débit du flux d'évènements est paramétré à 10 000 évènements par seconde, et la stratégie pour les deux règles est *Every*. On observe qu'au cours du temps, la consommation mémoire globale augmente progressivement au fur et à mesure que de nouvelles séquences sont créées par la stratégie *Every*. Pour les deux règles, la consommation mémoire augmente de manière relativement linéaire car aucune fenêtre temporelle n'est spécifiée : les séquences actives restent donc dans l'état global pendant toute la durée de l'expérimentation. Cependant, la consommation mémoire globale pour la règle `ViewSeq` augmente plus rapidement que pour la règle `SearchViewAdd` : à la fin de l'expérimentation, elle atteint 10,4 gigaoctets pour la première règle, contre 7,7 gigaoctets pour la seconde. Cela s'explique par le nombre plus élevé d'évènements `view` dans le flux Cdiscount par rapport au nombre d'évènements `search` et `add`, comme mis en évidence lors de l'analyse descriptive (*cf.* section 6.3). De plus, l'opérateur `KleenePlus` de la règle `ViewSeq` implique qu'en théorie, le nombre de séquences pour chaque client augmente exponentiellement par rapport à la taille de la fenêtre. Comme également montré lors de l'analyse descriptive du flux, cette croissance n'est généralement pas problématique, puisque la vaste majorité des clients du site Cdiscount effectuent relativement peu d'actions, et engendrent donc peu de séquences associées (pour la règle `ViewSeq`, 99% des clients ont au plus 8 séquences associées, 5 pour la règle `SearchViewAdd`). Cependant, dans certains cas particuliers tels que les clients automatisés (ou robots), le nombre de séquences engendrées peut être problématique pour les performances d'AUROS (le maximum enregistré lors des études préalables du flux Cdiscount est de 16 191 séquences pour la règle `ViewSeq`, et 288 pour la règle `SearchViewAdd`).

Le graphe (2) de la figure 6.3 présente deux courbes mettant en évidence la différence de consommation mémoire globale selon la stratégie spécifiée pour la règle `ViewSeq`. La courbe en pointillés longs représente la consommation mémoire globale pour la stratégie *Every*, et la courbe en trait continu représente celle de la stratégie *First*. Comme attendu, on remarque que la consommation mémoire globale augmente plus rapidement pour la

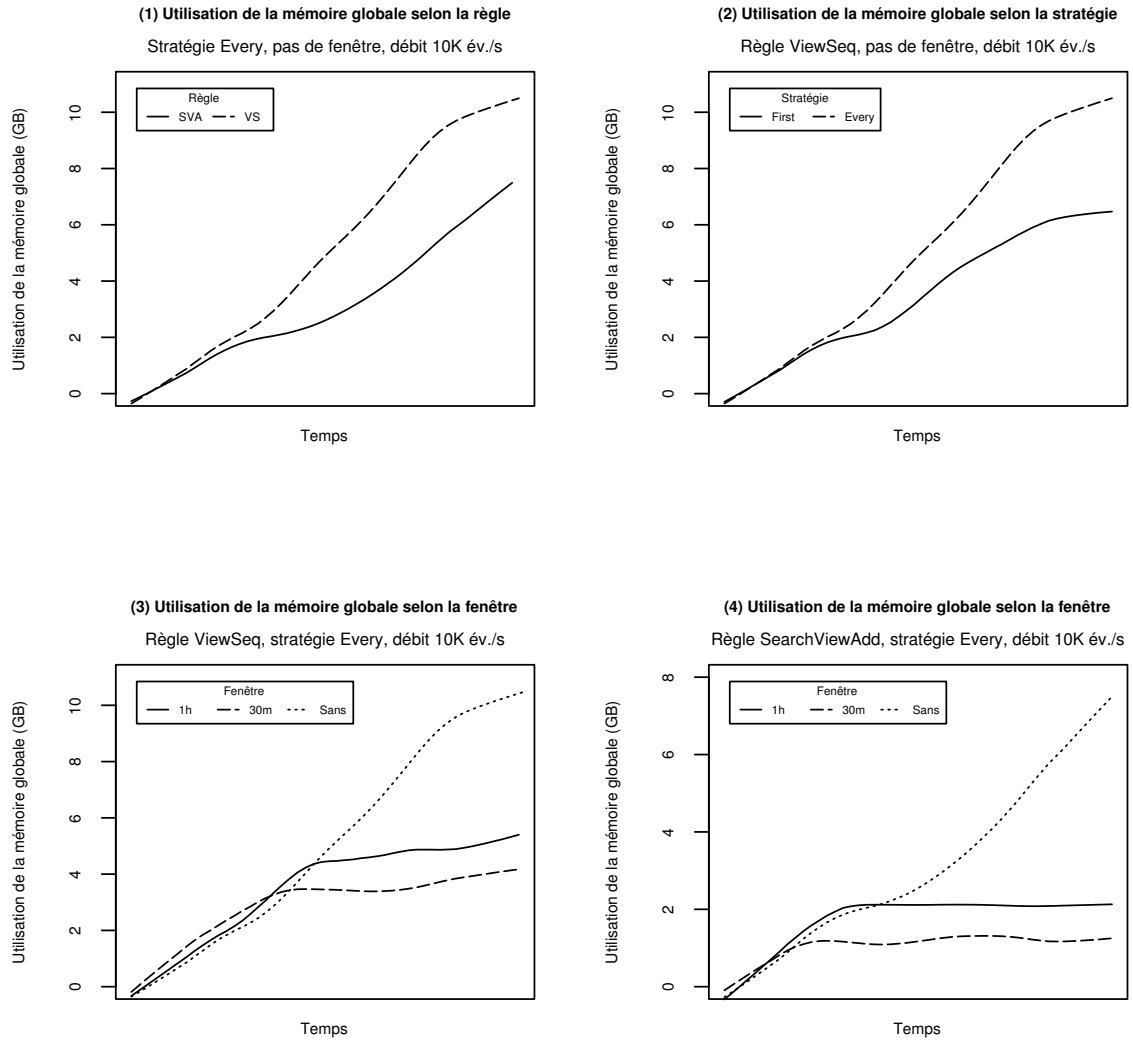

FIGURE 6.3 – Mesures de l'évolution de la consommation mémoire globale du cluster au cours du temps, en fonction (1) de la règle de détection (2) de la stratégie associée à la règle et (3) (4) de la fenêtre temporelle. (1) On mesure une consommation mémoire globalement plus élevée pour la règle `ViewSeq`, par rapport à la règle `SearchViewAdd`. (2) La stratégie *Every* entraîne une consommation mémoire globale plus élevée que la stratégie *First*. (3) (4) Pour les deux règles étudiées, l'application d'une fenêtre temporelle permet de considérablement et durablement réduire la consommation mémoire globale.

stratégie *Every* que pour la stratégie *First* : sans application d'une fenêtre temporelle, avec un débit de 10 000 évènements par seconde, la consommation mémoire globale atteint 10,4 gigaoctets pour la première stratégie, contre 6,1 gigaoctets pour la seconde. Cela s'explique par le fait que la stratégie *First* limite le nombre de séquences par client à 1, alors que la stratégie *Every* ne présente pas cette limite (cf. section 4.3). Cependant, il est intéressant de noter que dans le cas de Cdiscount, le rapport entre la consommation globale des deux stratégies est relativement faible (inférieur à 2). Comme expliqué dans la précédente expérimentation, cela est du au fait que la vaste majorité des clients Cdiscount effectuent relativement peu d'actions, et génèrent donc assez peu de séquences. Plus spécifiquement, pour la règle `ViewSeq` ici évaluée, chaque client génère en moyenne 2,3 séquences avec la stratégie *Every* (contre 1 pour la stratégie *First*). Le fait que le ratio observé soit inférieur à 2 peut s'expliquer d'une part par le fait que l'expérimentation présentée à une durée relativement courte, alors que les statistiques calculées concernent une période d'un mois de données ; et d'autre part, par le fait que la consommation mémoire globale n'inclut pas uniquement la taille de l'état globale, mais également une partie variable réservée à l'exécution des fonctions `Parse` et `UpdateState`, ainsi qu'à d'autres données internes à Spark.

Le graphe (3) de la figure 6.3 présente trois courbes, représentant la consommation mémoire globale pour une taille de fenêtre donnée. La règle de détection étudiée est la règle `ViewSeq`, avec la stratégie *Every*. La courbe en pointillés courts illustre l'évolution de la consommation mémoire globale sans application d'une fenêtre temporelle sur la règle. La courbe en trait continu représente la consommation mémoire globale étant donné l'application d'une fenêtre temporelle d'une heure, et la courbe en pointillés longs avec une fenêtre de 30 minutes. Les fenêtres en question sont des fenêtres glissantes, dont la durée est exprimée sur la base du débit moyen réel du flux d'évènements Cdiscount, qui est ici accéléré pour des raisons pratiques (le débit du flux pour l'expérimentation est paramétré à 10 000 évènements par seconde, contre 1 000 évènements par seconde en moyenne pour le flux réel). On constate que l'application d'une fenêtre temporelle sur la règle de détection a un impact nettement observable sur l'évolution de la consommation mémoire globale. En effet, la fenêtre temporelle a pour effet de déclencher la suppression de toute séquence arrivant à expiration (cf. section 4.2.1). Par conséquent, l'application d'une fenêtre temporelle contribue à contrôler la taille de l'état global, et donc la consommation mémoire globale. On observe que plus la fenêtre est réduite, plus la taille de l'état global est faible. La consommation mémoire globale atteint un maximum de 5,1 gigaoctets avec la fenêtre d'une heure, et seulement 3,9 gigaoctets avec la fenêtre de 30 minutes, contre 10,4 gigaoctets sans fenêtre. Initialement, l'ensemble des courbes croissent à la même vitesse, jusqu'à l'expiration effective des premières séquences, que l'on peut observer en premier pour la courbe en pointillés longs (fenêtre de 30 minutes), et ensuite pour la courbe en trait continu (fenêtre d'1 heure).

Enfin, le graphe (4) de la figure 6.3 présente la même expérimentation que le graphe (3) pour la règle de détection `SearchViewAdd`. Comparé à la règle `ViewSeq`, on observe que les tailles d'état globales mesurées sont nettement inférieures. Cela s'explique, comme pré-

cédemment évoqué, par le fait que le nombre d'évènements "view" dans le flux est plus important que le nombre d'évènements "search" et "add", ce qui implique un nombre de séquences maintenues plus réduit pour la règle `SearchViewAdd`. En effet, la consommation mémoire globale atteint un maximum de 2,1 gigaoctets avec la fenêtre d'une heure, et seulement 1,2 gigaoctets avec la fenêtre de 30 minutes, contre 7,6 gigaoctets sans fenêtre.

6.5 Conclusions et limitations

Dans cette évaluation, nous avons montré que l'utilisation d'un système CEP tel qu'AUROS dans le cadre d'un cas d'utilisation réel est un processus complexe.

Dans un premier temps, nous avons montré, en mesurant les temps de traitement des lots d'évènements par Spark, que la performance du système AUROS dépend de plusieurs variables : le débit du flux d'évènements, la règle de détection évaluée, ainsi que la stratégie et la fenêtre temporelle associées à cette règle. Par exemple, l'utilisation de la stratégie *Every* (*cf.* section 3.7), en particulier dans le cas de règles telles que `ViewSeq` qui comportent des itérations (*cf.* section 3.3.3), amène le système à maintenir de nombreuses séquences en parallèle que le système doit garder en mémoire d'une part, et mettre à jour en continu d'autre part. Aussi, nous avons montré que la stratégie *First* (*cf.* section 3.7) est moins contraignante pour le système que la stratégie *Every*, puisque seule une séquence est maintenue pour chaque client. De ce fait, la consommation mémoire pour la stratégie *First* est proportionnelle au nombre de clients, et dépend donc directement de la taille de la fenêtre. En revanche, la consommation mémoire pour la stratégie *Every* est *a priori* plus chaotique, car elle dépend de l'activité de chaque client. Cette activité peut être intense pendant une période donnée, générant ainsi de nombreuses séquences à maintenir, puis plus calme pendant la période suivante. Cette hétérogénéité rend le comportement du système difficile à prédire pour une règle donnée. L'utilisation de la stratégie *Every* n'est donc pas sans risques pour la stabilité du système, et il est donc nécessaire pour l'utilisateur d'AUROS de prendre certaines précautions avant le déploiement de chaque règle de détection.

Dans la première partie de l'évaluation, nous avons montré que si les temps de traitement des lots sont stables étant donné un débit de flux, les temps de sauvegarde de l'état global ne le sont pas, car ils dépendent directement de la taille de l'état global. La garantie de tolérance aux pannes a donc un coût, ici matérialisé par des temps de traitement importants pour les lots qui déclenchent périodiquement la sauvegarde de l'état global sur disque dur. Afin de limiter ce coût, il est donc primordial de maîtriser la taille de l'état global au cours du temps, afin de borner les temps de sauvegarde à des valeurs raisonnables permettant d'assurer la stabilité du système dans le temps. En particulier, nous avons observé que la spécification de fenêtres temporelles et la mise au point d'un seuil du nombre maximum de séquences par client permettent de considérablement réduire les temps de sauvegarde, permettant au système d'être stable pour des débits supérieurs à 10 000 évè-

nements par seconde, ce qui représente 10 fois le débit actuel du réel flux d'évènements de Cdiscount.

Dans la deuxième partie de l'évaluation, nous avons montré sous un autre angle, par la mesure de la consommation mémoire globale, que la maîtrise de la taille de l'état global est un facteur primordial pour assurer la stabilité et la performance du système AUROS dans le temps, étant donné un environnement d'exécution dont les ressources sont bornées. En effet, le flux d'évènements étant par définition infini et donc imprédictible, il est nécessaire de calibrer le système de telle façon de garantir sa stabilité dans le temps, quelque soit les règles de détection à évaluer.

Pour pouvoir garantir la stabilité du système dans le temps, installer toute règle de détection dans le système requiert au préalable une bonne connaissance de la nature du flux d'évènements. Cette connaissance du flux est acquise notamment en effectuant une analyse descriptive statistique telle que présentée dans la section 6.3. La règle de détection `ViewSeq` est un exemple de règle *a priori* particulièrement difficile à maintenir, car l'opérateur `KleenePlus` combiné à la stratégie *Every* peut en théorie amener à une explosion du nombre de séquences par client. Cependant, en pratique cette croissance exponentielle du nombre de séquences est un phénomène rare dans le flux Cdiscount, car nombre de clients effectuent en réalité un nombre d'actions limité. Il est toutefois nécessaire de prendre en compte ces phénomènes rares, en particulier les robots générant de très nombreux évènements, afin de garantir la stabilité du système dans le temps.

Dans un cas d'utilisation réel comme celui de Cdiscount où les ressources du cluster sont limitées, il est donc primordial de déterminer au préalable, pour toute règle de détection, une taille de fenêtre adéquate permettant au système de consommer une quantité de mémoire limitée, tout en respectant les exigences formulées par l'utilisateur (en l'occurrence l'équipe Marketing). Pour toute règle, la calibration de la fenêtre temporelle, combinée avec le seuil maximum de séquences par client, doit permettre au système de borner l'état global de telle façon que la consommation mémoire globale reste stable. Dans cette évaluation, la règle `ViewSeq` représente le pire des cas pour le flux d'évènements Cdiscount (car le type d'évènement le plus présent est le type `view`, et que la règle contient à la fois un opérateur *KleenePlus* et une négation). Toutefois, la calibration du système est nécessaire pour toute règle à soumettre au système.

Enfin, nous avons montré que les performances d'AUROS dépendent, jusqu'à un certain point, de la quantité de ressources qui lui sont attribuées. En effet, en fonction des quantités de CPU et de mémoire disponibles, le système peut traiter un volume plus ou moins important d'évènements par unité de temps, et maintenir un certain nombre de séquences parallèles en mémoire. Cela contribue à assurer la stabilité du système quelque soit la charge de travail induite par le nombre de règles de détection actives, ou par le débit du flux d'évènements. Cependant, cette flexibilité n'est réellement exploitable qu'à condition que les différentes règles soient correctement calibrées, par les méthodes précédemment expliquées, à savoir la définition d'une fenêtre temporelle et d'un seuil maximum du nombre de séquences par client.

CHAPITRE

7 Conclusion

Dans ce dernier chapitre, nous tirons les conclusions de la thèse en rappelant ce qu'est le système AUROS, et pour quels besoins spécifiques il a été conçu. Puis, nous rappelons les principales contributions qui découlent de la conception de ce système, ainsi que ses limites identifiées. Enfin, nous montrons comment AUROS est utilisé chez notre partenaire industriel Cdiscount, et entamons une discussion concernant les perspectives du projet AUROS, aussi bien à court qu'à long termes.

7.1 Le système AUROS: exigences, contributions et limites

AUROS est un système CEP, c'est-à-dire un système visant à permettre l'identification en temps réel de scénarios précis et complexes dans un flux d'évènements. Le flux d'évènements analysé par AUROS est particulier car il est massif : il est constitué de l'ensemble des évènements générés par les actions de navigation des clients du site Cdiscount (consultations de produits, recherches, clics, etc). Jusqu'à deux millions de visiteurs uniques utilisent la plateforme ecommerce quotidiennement.

Ainsi, AUROS a été créé pour répondre à des exigences bien spécifiques : (i) l'expression de comportements ou parcours de navigation variés à identifier, (ii) l'identification des scénarios spécifiés en temps réel, c'est-à-dire avec un délai inférieur à une seconde entre leurs occurrences réelles et leur identification par le système, et (iii) le passage à l'échelle du système afin de traiter efficacement un flux d'évènements massif, dont le débit peut atteindre 10 000 évènements par seconde.

Pour répondre à cet ensemble d'exigences, le système AUROS a été conçu selon trois axes principaux :

— Le langage AUROS rend possible l'expression de règles de détection spécifiant les scénarios à identifier,

— L'automate AUROS est un automate fini déterministe représentant le modèle d'exécution d'une règle de détection,

— Le système AUROS s'appuie sur une plateforme distribuée permettant l'exécution continue et à grande échelle du système.

Le langage AUROS est un nouveau langage CEP conçu pour Cdiscount, dont les objectifs sont multiples. Tout d'abord, le langage est l'interface du système permettant de faciliter l'expression des scénarios de navigation à identifier via la définition de séquences d'évènements et de contraintes, qui constituent des règles de détection (cf. chapitre 3). Les contraintes peuvent s'appliquer soit individuellement au niveau d'un évènement (*e.g.* filtres), soit au niveau des relations entre plusieurs évènements différents (*e.g.* liens, contraintes temporelles). Les règles de détection spécifient également une fenêtre temporelle qui limite leur champ d'application dans le temps. Par ailleurs, à chaque règle de détection est associée une stratégie (*First* ou *Every*) qui permet de paramétrer l'algorithme de détection pour la règle en question. Une règle de détection écrite avec le langage AUROS peut ensuite être traduite en modèle d'automate correspondant, par un procédé de compilation décrit dans le chapitre 4.

Un automate AUROS est le résultat de ce processus de compilation d'une règle de détection. Il s'agit d'un automate fini déterministe sur lequel se base le système pour évaluer de façon incrémentale une règle de détection. L'état final de cet automate déclenche l'identification effective d'une séquence d'évènements correspondant à la règle de détection. Selon la stratégie associée à la règle, une ou plusieurs séquences d'évènements peuvent être simultanément attachées à un automate. Chaque séquence correspond à une instance de la règle en cours d'identification, et est composée d'un état courant dans l'automate, et de la séquence partielle des évènements de la règle précédemment identifiés dans le flux. Dans le système AUROS, à chaque client du site (représenté par un identifiant client) correspond son propre ensemble de séquences actives, et les évènements du flux sont rattachés aux séquences du client correspondant.

Pour maintenir, évaluer et mettre à jour efficacement la multitude de séquences engendrées par l'ensemble des clients du site, et pour chaque règle de détection active, le système AUROS s'appuie sur Spark, qui est un modèle et une plateforme de traitement de données massives (cf. chapitre 5). Spark possède la particularité d'effectuer les calculs en mémoire, et de proposer un modèle de traitement de flux. Il permet de partitionner le flux d'évènements pour le traiter en parallèle sur plusieurs de noeuds de calcul. Ce partitionnement s'effectue sur la base des identifiants des clients, garantissant ainsi que tous les évènements d'un client puissent être traités localement et dans l'ordre souhaité. Le système AUROS s'appuie sur ce modèle pour calculer et mettre à jour en parallèle les séquences associées aux clients, et ce pour chaque règle de détection active. L'état du système, constitué de l'ensemble des séquences actives pour tous les clients, est maintenu en mémoire

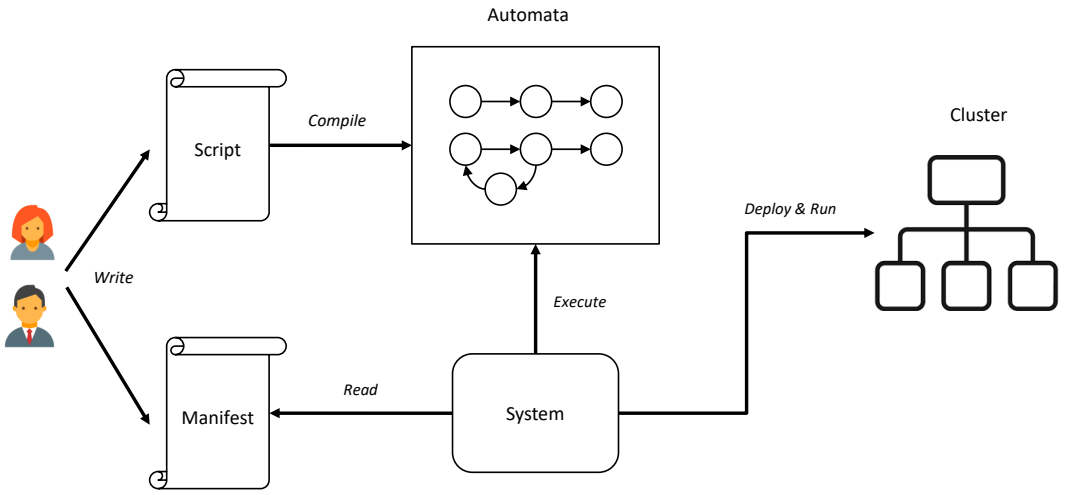

FIGURE 7.1 – Utilisation du système AUROS à Cdiscount: les utilisateurs spécifient les règles de détection au sein d'un script, qui est compilé pour obtenir un ensemble d'automates AUROS. Puis, ces automates sont passés au système au moment du déploiement pour être exécutés par la plateforme Spark. Le manifeste permet de paramétrer certaines aspects du système, notamment ses entrées/sorties (typiquement le flux d'évènements à consommer, et le flux de sortie contenant les évènements complexes produits).

et sauvegardé à intervalles réguliers sur disque dur via un système de fichiers distribué, ce qui permet de garantir la tolérance aux pannes du système. Enfin, le nombre de nœuds de calcul dédiés au système peut être paramétré pour lui allouer plus ou moins de ressources, permettant ainsi à AUROS de s'adapter à la charge de travail demandée, qui est susceptible d'augmenter dans le futur. AUROS garantit donc le passage à l'échelle, les performances, et la tolérance aux pannes du système CEP. Ainsi, notre approche montre que certains modèles du paradigme Big Data tels que Spark peuvent constituer une plateforme intéressante pour un système CEP à grande échelle.

Pour utiliser le système CEP AUROS, les utilisateurs membres de l'équipe Marketing de Cdiscount suivent le processus illustré en figure 7.1, constitué des étapes suivantes :

1. Écriture d'un script contenant un ensemble de règles de détection à évaluer, en utilisant le langage AUROS
2. Compilation de ce script pour obtenir un ensemble d'automates correspondants aux règles spécifiées
3. Écriture d'un manifeste permettant de paramétrer le système
4. Déploiement et exécution du système sur la plateforme Spark.

Après la réalisation de ces 4 étapes, le système AUROS détecte des séquences dans le flux d'évènements en entrée, et produit des évènements complexes. Ces évènements complexes peuvent être exploités en tant que notifications permettant au site web de réagir

aux actions individuelles des clients. Ils peuvent également être utilisés pour mettre à jour des tableaux de bord permettant de suivre en temps réel l'activité des clients sur le site, ou simplement enregistrés en base de données pour des traitements différés.

Dans le chapitre dédié à l'évaluation du système AUROS (*cf.* chapitre 6), nous nous sommes attachés à montrer que celui-ci est fonctionnel dans un cas d'usage réel tel que celui de Cdiscount. Nous avons également montré que la composition de la règle, la stratégie, et la taille de la fenêtre temporelle ont chacun des effets importants sur les performances du système, que ce soit au niveau des temps de traitement des évènements ou de la consommation mémoire. La principale variable résultante de l'ensemble de ces paramètres est le nombre de séquences que le système est amené à maintenir simultanément à un instant donné. Le nombre de séquences à maintenir a un impact mesurable sur le système, observable par la mesure des temps de traitement des lots d'évènements, et par la mesure de la consommation mémoire résultant essentiellement de l'accumulation des séquences générées au cours du temps.

Par ailleurs, nous avons montré lors de l'évaluation que la propriété de tolérance aux pannes, essentielle pour une utilisation prolongée du système en conditions réelles, est une garantie qui a un coût. En sauvegardant l'état du système sur disque à intervalles réguliers, Spark introduit des latences ponctuelles qui, si elles sont excessives, peuvent à terme congestionner la file de traitement des lots d'évènements issus du flux, et par conséquent rendre le système instable voire inutilisable. Lorsque le nombre de séquences à maintenir par le système est trop important, la propriété de passage à l'échelle inhérente au paradigme Big Data n'est pas toujours suffisante pour contrebalancer le coût des sauvegardes périodiques. Par conséquent, le mécanisme de sauvegarde permettant à Spark de garantir la propriété de tolérance aux pannes pour le système peut effectivement être vu comme un goulet d'engorgement pour le système.

Étant donnés ces constats, nous avons montré que la mise en place de mécanismes particuliers, à savoir la fenêtre temporelle et le seuil maximum du nombre de séquences par client, s'avère être cruciale pour maîtriser les performances du système, en permettant de contrôler le nombre de séquences à maintenir. La mise en place de ces mécanismes s'appuie sur une connaissance du flux d'évènements à analyser, en l'occurrence de celui fourni par Cdiscount. D'une part, déterminer la taille de la fenêtre temporelle associée à chaque règle de détection s'avère critique pour limiter le nombre de séquences total. D'autre part, l'analyse du flux d'évènements a permis de mettre en évidence la présence de robots, dont le comportement aberrant est à prendre en compte afin de préserver le fonctionnement d'AUROS. Ces robots, même en très faible nombre, peuvent aisément entraîner le dysfonctionnement du système en faisant croître exponentiellement le nombre de séquences à maintenir. En proposant une méthode permettant de neutraliser l'impact de ces robots en établissant un seuil maximum de séquences par client pour chaque règle, nous avons montré que le système AUROS peut faire face aux aléas d'un flux réel pour préserver les performances du système, sans en remettre en cause les fonctionnalités.

Toutefois, ces mécanismes de contournement, s'ils sont efficaces, peuvent être fasti-

dieux à mettre en place. En effet, déterminer les valeurs appropriées pour la taille de la fenêtre temporelle associée à une règle, ou pour le seuil maximum du nombre de séquences par client nécessite une analyse relativement poussée du flux d'évènements, et de l'effet de chaque règle de détection par rapport à ce flux. Dans un cas réel comme celui de Cdiscount, borner la taille de l'état à maintenir nécessite donc un effort significatif de la part de l'utilisateur.

7.2 Perspectives

À relativement court terme, le système AUROS pourrait bénéficier de certaines optimisations permettant notamment de réduire l'utilisation de la mémoire globale. Par exemple, il pourrait être intéressant d'utiliser des techniques de partage de sous-séquences communes à plusieurs clients (similairement à l'algorithme d'optimisation présenté par *Agrawal et al.* [Agrawal *et al.*, 2008]) pour réduire de manière significative la taille de l'état global, et donc la consommation mémoire du système dans son ensemble. En ce qui concerne spécifiquement l'utilisation de la plateforme Spark, il semblerait qu'il soit possible de réduire davantage les temps de traitement des lots en réduisant, voire en supprimant les coûts liés à des échanges réseaux et à des repartitionnements inutiles de données. D'autre part, il paraitrait judicieux de disposer d'une stratégie plus efficace que *Every*, notamment en ce qui concerne les règles pour l'opérateur d'itération `Kleene`. En effet, cet opérateur peut théoriquement générer un nombre de séquences qui croît exponentiellement par rapport à la taille de la fenêtre [Agrawal *et al.*, 2008]. Hors, il n'est pas nécessairement utile pour une règle donnée de détecter toutes les combinaisons possibles d'évènements pouvant être capturées par l'opérateur `Kleene`, ce que fait la stratégie *Every*. Pour faciliter l'utilisation du système, il pourrait être utile de proposer des outils permettant de déterminer à la volée, pour chaque règle de détection, des tailles de fenêtre et seuils du nombre de séquences par client appropriés.

À plus long terme, il serait possible d'étendre les fonctionnalités d'AUROS afin de rendre le système CEP plus polyvalent. Récemment, les domaines du CEP et celui du traitement de flux (*Stream Processing*) ont tendu à se rapprocher [Cugola et Margara, 2012b], notamment dans le milieu industriel. On y trouve ainsi des systèmes plus universels permettant à la fois (i) la gestion et la manipulation d'une multitude de flux de données pouvant être interrogés, filtrés, transformés, joints (*cf.* Borealis [Abadi *et al.*, 2005] ou CQL [Arasu *et al.*, 2006]), etc, et (ii) l'analyse complexe de flux d'évènements, comme le font les systèmes CEP dédiés (*cf.* chapitre 2).

Il est utile de rappeler que la plateforme Big Data Spark sur laquelle AUROS s'appuie est à la base conçue pour faciliter la transformation d'ensembles de données massifs. Par conséquent, ce paradigme est tout à fait adapté au type d'opérations que l'on attribue au domaine du *Stream Processing*, et le système AUROS est donc extensible en ce sens.

Plus spécifiquement, la chaîne d'opérations que nous présentons dans le chapitre 5

(constituée pour rappel des deux fonctions essentielles `Parse` et `UpdateState`) pourrait être rendue extensible au gré de l'utilisateur (par exemple via le manifeste de déploiement, voire directement par le langage AUROS), afin d'y ajouter notamment :

— Des étapes de pré-traitement du flux d'évènements permettant notamment de filtrer, de formater ou d'enrichir les évènements. Ces opérations de transformation peuvent être appliquées avant l'exécution de la fonction `Parse` dans la chaîne d'opérations Spark.

— Des étapes de post-traitement permettant de construire sur-mesure les évènements complexes produits lors de l'identification de séquences correspondant à une règle de détection donnée, avant leur intégration dans le flux de sortie du système. Il s'agirait d'opérations de transformation s'appliquant au sein de la fonction `UpdateState` de la chaîne d'opérations Spark, et permettant de calculer un évènement complexe à partir d'une séquence identifiée. Il pourrait également être possible d'enrichir les évènements complexes à la volée en réalisant une jointure avec une table en mémoire cache (par exemple, pour récupérer la catégorie d'un produit à partir d'un attribut contenant un identifiant de catégorie).

Bien que l'ajout de ces opérations implique des calculs supplémentaires pour chaque lot d'évènements consommés, ces opérations-là sont entièrement parallélisables et distribuables sur un cluster Spark. Les coûts de ces calculs pourraient donc être effectivement absorbés par le cluster en allouant davantage de ressources (CPU et RAM) au système AUROS.

Bibliographie

Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A. S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y. et Zdonik, S. (2005). The Design of the Borealis Stream Processing Engine. *In Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA. Cité page 97.

Agrawal, J., Diao, Y., Gyllstrom, D. et Immerman, N. (2008). Efficient Pattern Matching over Event Streams. *In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 147–160, New York, NY, USA. ACM. Cité pages 13, 17, et 97.

Arasu, A., Babu, S. et Widom, J. (2006). The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2):121–142. Cité page 97.

Borthakur, D., Gray, J., Sarma, J. S., Muthukkaruppan, K., Spiegelberg, N., Kuang, H., Ranganathan, K., Molkov, D., Menon, A., Rash, S., Schmidt, R. et Aiyer, A. (2011). Apache Hadoop Goes Realtime at Facebook. *In Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1071–1080, New York, NY, USA. ACM. Cité page 29.

Cugola, G. et Margara, A. (2010). TESLA: A Formally Defined Event Specification Language. *In Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 50–61, New York, NY, USA. ACM. Cité page 13.

Cugola, G. et Margara, A. (2012a). Complex Event Processing with T-REX. *J. Syst. Softw.*, 85(8):1709–1728. Cité pages 13 et 19.

Cugola, G. et Margara, A. (2012b). Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surv.*, 44(3):15:1–15:62. Cité pages 2, 8, 16, 22, 26, et 97.

Dean, J. et Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113. Cité pages 4 et 27.

Demers, A., Gehrke, J., Hong, M., Riedewald, M. et White, W. (2006). Towards Expressive Publish/Subscribe Systems. *In* Ioannidis, Y., Scholl, M. H., Schmidt, J. W., Matthes, F., Hatzopoulos, M., Boehm, K., Kemper, A., Grust, T. et Boehm, C., éditeurs : *Advances in Database Technology - EDBT 2006*, numéro 3896 de Lecture Notes in Computer Science, pages 627–644. Springer Berlin Heidelberg. DOI: 10.1007/11687238_38. Cité pages 14, 19, et 23.

Farahbod, R., Glässer, U. et Vajihollahi, M. (2004). Specification and Validation of the Business Process Execution Language for Web Services. *In* Zimmermann, W. et Thalheim, B., éditeurs : *Abstract State Machines 2004. Advances in Theory and Practice*, numéro 3052 de Lecture Notes in Computer Science, pages 78–94. Springer Berlin Heidelberg. DOI: 10.1007/978-3-540-24773-9_7. Cité page 12.

Gyllstrom, D., Wu, E., Chae, H.-J., Diao, Y., Stahlberg, P. et Anderson, G. (2006). SASE: Complex Event Processing over Streams. *arXiv:cs/0612128*. arXiv: cs/0612128. Cité pages 12 et 21.

Li, G. et Jacobsen, H.-A. (2005). Composite Subscriptions in Content-Based Publish/-Subscribe Systems. *In* Alonso, G., éditeur : *Middleware 2005*, numéro 3790 de Lecture Notes in Computer Science, pages 249–269. Springer Berlin Heidelberg. DOI: 10.1007/11587552_13. Cité pages 11 et 24.

Magee, J., Dulay, N. et Kramer, J. (1994). Regis: a constructive development environment for distributed programs. *Distributed Systems Engineering*, 1(5):304. Cité page 25.

Mansouri-Samani, M. et Sloman, M. (1997). GEM: a generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96. Cité page 25.

Olston, C., Reed, B., Srivastava, U., Kumar, R. et Tomkins, A. (2008). Pig Latin: A Not-so-foreign Language for Data Processing. *In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA. ACM. Cité page 30.

Schultz-Møller, N. P., Migliavacca, M. et Pietzuch, P. (2009). Distributed Complex Event Processing with Query Rewriting. *In Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 4:1–4:12, New York, NY, USA. ACM. Cité page 26.

Shvachko, K., Kuang, H., Radia, S. et Chansler, R. (2010). The Hadoop Distributed File System. *In 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. Cité pages 29 et 73.

Silva, A., Bonchi, F., Bonsangue, M. M. et Rutten, J. J. M. M. (2010). Generalizing the powerset construction, coalgebraically. *In* Lodaya, K. et Mahajan, M., éditeurs : *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010)*, volume 8 de *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 272–283, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. Cité page 56.

Spearman, C. (1904). The Proof and Measurement of Association between Two Things. *The American Journal of Psychology*, 15(1):72–101. Cité page 84.

Stonebraker, M., Çetintemel, U. et Zdonik, S. (2005). The 8 Requirements of Real-time Stream Processing. *SIGMOD Rec.*, 34(4):42–47. Cité page 30.

Suhothayan, S., Gajasinghe, K., Loku Narangoda, I., Chaturanga, S., Perera, S. et Nanayakkara, V. (2011). Siddhi: A Second Look at Complex Event Processing Architectures. *In Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*, GCE '11, pages 43–50, New York, NY, USA. ACM. Cité pages 14 et 26.

Thompson, K. (1968). Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM*, 11(6):419–422. Cité page 49.

Thusoo, A., Sarma, J., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H. et Murthy, R. (2010a). Hive - a petabyte scale data warehouse using Hadoop. *In 2010 IEEE 26th International Conference on Data Engineering (ICDE)*, pages 996–1005. Cité page 30.

Thusoo, A., Shao, Z., Anthony, S., Borthakur, D., Jain, N., Sen Sarma, J., Murthy, R. et Liu, H. (2010b). Data Warehousing and Analytics Infrastructure at Facebook. *In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1013–1020, New York, NY, USA. ACM. Cité page 28.

Ward, J. S. et Barker, A. (2013). Undefined By Data: A Survey of Big Data Definitions. *arXiv:1309.5821 [cs]*. arXiv: 1309.5821. Cité page 27.

Wu, X., Zhu, X., Wu, G. Q. et Ding, W. (2014). Data mining with big data. *IEEE Transactions on Knowledge and Data Engineering*, 26(1):97–107. Cité page 2.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S. et Stoica, I. (2012). Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. *In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA. USENIX Association. Cité pages 4 et 30.

Zaharia, M., Chowdhury, N. M. M., Franklin, M., Shenker, S. et Stoica, I. (2010). Spark: Cluster Computing with Working Sets. Rapport technique, EECS Department, University of California, University of California at Berkeley, Berkeley, California. Cité page 30.

Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S. et Stoica, I. (2013). Discretized Streams: Fault-tolerant Streaming Computation at Scale. *In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA. ACM. Cité pages 4 et 30.

Table des figures

Liste des tableaux

Listings

Abstract

Pattern detection over streams of events is gaining more and more attention, especially in the field of eCommerce. Our industrial partner Cdiscount, which is one of the largest eCommerce companies in France, aims to use pattern detection for real-time customer behavior analysis. The main challenges to consider are efficiency and scalability, as the detection of customer behaviors must be achieved within a few seconds, while millions of unique customers visit the website every day, thus producing a large event stream. In this thesis, we present Auros, a system for large-scale and efficient pattern detection for eCommerce. It relies on a domain-specific language to define behavior patterns. Patterns are then compiled into deterministic finite automata, which are run on a Big Data streaming platform. Our evaluation shows that our approach is efficient and scalable, and fits the requirements of Cdiscount.

Keywords: *Complex Event Processing, Events, Big Data, Stream Processing, Distributed System*

Résumé

La détection d'évènements complexes dans les flux d'évènements est un domaine qui a récemment fait surface dans le ecommerce. Notre partenaire industriel Cdiscount, parmi les sites ecommerce les plus importants en France, vise à identifier en temps réel des scénarios de navigation afin d'analyser le comportement des clients. Les objectifs principaux sont la performance et la mise à l'échelle : les scénarios de navigation doivent être détectés en moins de quelques secondes, alors que des millions de clients visitent le site chaque jour, générant ainsi un flux d'évènements massif. Dans cette thèse, nous présentons Auros, un système permettant l'identification efficace et à grande échelle de scénarios de navigation conçu pour le eCommerce. Ce système s'appuie sur un langage dédié pour l'expression des scénarios à identifier. Les règles de détection définies sont ensuite compilées en automates déterministes, qui sont exécutés au sein d'une plateforme Big Data adaptée au traitement de flux. Notre évaluation montre qu'Auros répond aux exigences formulées par Cdiscount, en étant capable de traiter plus de 10,000 évènements par seconde, avec une latence de détection inférieure à une seconde.

Mots clés : *Détection d'évènements complexes, Évènements, Big Data, Traitement de flux, Système distribué*

LaBRI — 351, Avenue de la Libération — 33400 Talence — France