Approche préventive pour une gestion élastique du traitement parallèle et distribué de flux de données

Roland Kotto-Kombi, Nicolas Lumineau, Philippe Lamarre

Approche préventive pour une gestion élastique du traitement parallèle et distribué de flux de données

Roland Kotto-Kombi*,Nicolas Lumineau**, Philippe Lamarre*

* Univ Lyon, INSA de Lyon, LIRIS UMR5205, F-69621 Villeurbanne, France
affil2 Univ Lyon, Université Claude Bernard Lyon 1, LIRIS UMR5205,
F-69622 Villeurbanne, France
http://liris.cnrs.fr/
prénom.nom@liris.cnrs.fr

Résumé. Dans un contexte de traitement de flux de données, il est important de garantir à l'utilisateur des propriétés de performance, qualité des résultats et passage à l'échelle. Mettre en adéquation ressources et besoins, pour n'allouer que les ressources nécessaires au traitement efficace des flux, est un défi d'actualité majeur au croisement des problématiques du Big Data et du Green IT. L'approche que nous suggérons permet d'adapter dynamiquement et automatiquement le degré de parallélisme des différents opérateurs composant une requête continue selon l'évolution du débit des flux traités. Nous proposons i) une métrique permettant d'estimer l'activité future des opérateurs selon l'évolution des flux en entrée, ii) l'approche *AUTOSCALE* évaluant a priori l'intérêt d'une modification du degré de parallélisme des opérateurs en prenant en compte l'impact sur le traitement des données dans sa globalité iii) grâce à une intégration de notre proposition à *Apache Storm*, nous exposons des tests de performance comparant notre approche par rapport à la solution native de cet outil.

1 Introduction

Avec la multiplication des sources de flux de données (capteurs, objets connectés...), les méthodes d'acquisition, stockage et traitement de ces données ont évolué pour en gérer la masse et la vélocité. Ces flux sont des séquences de n-uplets dont le débit et la distribution des valeurs peuvent varier au cours du temps. L'interrogation de ces flux via des requêtes, dites *continues* (Sattler et Beier (2013)), soulèvent des défis majeurs en terme de performance et passage à l'échelle. En terme de performance, les systèmes de gestion de flux de données doivent pouvoir traiter à la volée les données issues de flux. De la capacité de ces systèmes à absorber ces flux pour les traiter, dépend également la qualité des résultats qui seront produits. En ce qui concerne le passage à l'échelle, ces systèmes doivent être en mesure d'absorber des débits de données potentiellement très variables et élevés.

Travaux partiellement financés par le projet Socioplug ANR-13-INFR-0003, http://socioplug.univ-nantes.fr/index.php/SocioPlug_Project

Afin de répondre à ces enjeux, des systèmes de gestion de flux de données (Gedik et al. (2014); Neumeyer et al. (2010); Peng et al. (2015); Schneider et al. (2009); Zaharia et al. (2012)) ont été développés. Nous nous concentrerons ici sur les systèmes représentant les requêtes continues comme un graphe d'opérateurs, dit *workflow* et gérant un support d'exécution distribué car ces solutions répondent au mieux aux défis ciblés. Nous nous plaçons dans un cadre où des techniques, dont celles de réécriture, ont permis d'identifier les différentes factorisations possibles entre un ensemble de requêtes continues et donc les workflows à traiter. Notre problème est donc de traiter au mieux ces workflows face aux évolutions des flux d'entrée et en accord avec les ressources disponibles.

Dans un contexte parallèle et distribué, deux aspects distincts jouent des rôles majeurs sur l'usage des ressources : la gestion du degré de parallélisme des opérateurs et la stratégie d'allocation des ressources. La dispersion globale des traitements dépend du degré de parallélisme de chaque opérateur. Modifier dynamiquement le degré de parallélisme d'un opérateur permet d'adapter sa capacité d'absorption en fonction des variations des flux de données en entrée. La stratégie adoptée par le mécanisme d'allocation a un impact évident sur les ressources utilisées. Par exemple, une stratégie de répartition de charge utilisera au maximum l'ensemble des ressources disponibles. A contrario, une stratégie centrée sur la diminution du trafic réseau aura l'effet de concentrer davantage les opérateurs sur un sous-ensemble de ressources. La qualité de l'adaptation dynamique, tant du point de vue de la qualité des résultats que de l'usage des ressources est le résultat des solutions apportées à ces deux aspects ainsi qu'à leur interaction.

Dans cet article, nous focaliserons sur l'adaptation du degré de parallélisme de chaque opérateur face à des flux à débits variants. En effet, une augmentation du débit en entrée d'un opérateur peut conduire à sa *congestion* (Gedik et al. (2014); Xu et Peng (2016)). Cela se traduit par une augmentation rédhibitoire de la latence pouvant conduire à une défaillance du système. Pour éviter ce problème, le mécanisme d'allocation peut déplacer les opérateurs vers des ressources ayant plus de puissance disponible. Lorsque cela n'est plus possible, seul le changement de degré de parallélisme constitue une solution.

Des travaux récents (Gedik et al. (2014); Peng et al. (2015); Xu et al. (2014)) s'intéressent à la notion d'élasticité pour optimiser les performances et l'usage des ressources. Par ressources, nous entendons CPU et RAM des machines ainsi que la bande passante du réseau. À notre connaissance, les solutions adaptant dynamiquement le degré de parallélisme des opérateurs (Gedik et al. (2014); Schneider et al. (2009)) ne permettent pas d'anticiper l'activité des opérateurs. D'autre part, des solutions (Neumeyer et al. (2010); Xu et Peng (2016)) nécessitent l'intervention de l'utilisateur.

Nous avons choisi d'intégrer notre système de gestion dynamique du degré de parallélisme de chaque opérateur à *Apache Storm* [2]. Le nombre de répliques d'un opérateur est calculé à partir d'une métrique permettant d'anticiper l'activité à court terme. Cette métrique se base sur l'évolution des flux d'entrée et sur l'activité récente d'un opérateur. Les reconfigurations passant par une augmentation (*scale-out*) ou une diminution (*scale-in*) du degré de parallélisme sont évaluées par notre approche *AUTOSCALE* à la fois individuellement et globalement afin d'identifier celles qui sont cohérentes et favorables à la performance et la stabilité du système.

Dans la suite, nous présentons les limites des solutions existantes (Section 2). Les métriques caractérisant l'activité d'un opérateur sont introduites dans la Section 3. L'approche *AUTOSCALE* adaptant dynamiquement le degré de parallélisme des opérateurs est décrit dans

2. Apache Storm : https://storm.apache.org/

la Section 4. Enfin, nous exposons les résultats de l'évaluation expérimentale de l'approche que nous proposons dans la Section 5.

2 Motivation

2.1 Contexte d'exécution

Afin de préciser le problème que nous traitons et de nous positionner par rapport à l'existant, nous considérons trois requêtes continues. Comme illustré sur la Figure 1, ces requêtes sont représentées par les topologies R1, R2 et R3 qui correspondent à leurs plans d'exécution logiques respectifs.

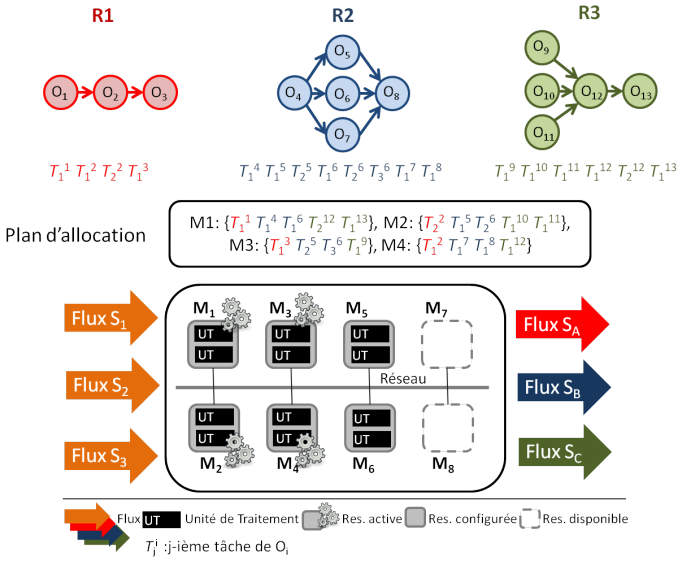

FIG. 1 – *Principe du traitement parallèle et distribué multi-requêtes sur des flux de données.*

Chaque requête porte sur les flux en entrée S_1, S_2 et S_3. La topologie R1 est séquentielle, R2 est organisée en diamant et R3 est en étoile. Chacune de ces topologies représente un motif élémentaire car toute topologie peut être considérée comme une composition de ces topologies. À chaque opérateur est associé un ensemble de *tâches*. Le nombre de tâches associées à un opérateur correspond à son *degré de parallélisme*. L'opérateur O_2 est, par exemple, associé aux tâches T_1^2 et T_2^2 et a donc un degré de parallélisme de 2. Ces tâches sont allouées selon le plan d'allocation sur les unités de traitement des machines M_1, M_2,..., M_8 disponibles pour y être exécutées. Sur la Figure 1, les quatre tâches de la topologie R1 sont distribuées sur les machines M_1 à M_4. Nous distinguons trois types de machines et donc trois types de ressources. Les machines de M_1 à M_4 sont des ressources *actives*, car elles traitent des tâches qui leurs ont été affectées. Les machines M_5 et M_6 sont des ressources *configurées* mais non actives car aucune de leurs unités de traitement ne traitent de tâches. Enfin, les machines M_7 et M_8

sont des ressources *disponibles* mais non configurées et donc non utilisable en l'état par le mécanisme d'allocation.

Nous posons les hypothèses suivantes sur le contexte d'exécution. Premièrement, les ressources et latences réseaux sont homogènes (H1). De plus, ces ressources sont suffisantes pour traiter les flux d'entrée (H2). Nous nous plaçons dans le cas où plusieurs requêtes continues peuvent être traitées simultanément (H3). Ensuite, la stratégie d'allocation est gérée par le système de gestion de flux de données (H4). Enfin nous considérons un ensemble de flux à débits variant mais dont la distribution des valeurs à un écart type faible (H5).

La problématique d'adaptation dynamique face à une distribution variante des valeurs est abordée dans Rivetti et al. (2015) et est hors du scope de cet article.

2.2 Approches existantes

La performance d'une solution de traitement de flux et la qualité des résultats produits sont fortement dépendants de la réactivité du système aux variations de l'environnement d'exécution. Le problème est cependant complexe car tout en réagissant aux variations du contexte d'exécution, il faut éviter l'instabilité du système qui affecterait la performance et la qualité des résultats.

Certaines solutions (Aniello et al. (2013); Xu et al. (2014)) se basent sur l'état des ressources et le trafic réseau afin de déterminer l'ensemble quasi-optimal des affectations des tâches sur les unités de traitement. Cela permet de réduire la latence globale de la topologie en évitant des échanges réseaux coûteux entre les machines. Toutefois, ces solutions n'ont qu'un impact limité sur la capacité de traitement de chaque opérateur. En effet, une fois que toutes les tâches associées à un opérateur sont réparties sur un maximum de ressources, la capacité de traitement ne peut plus être augmentée.

Les solutions se basant systématiquement sur l'usage maximal de l'ensemble des ressources (Neumeyer et al. (2010); Zaharia et al. (2012)) permettent de garantir, selon l'hypothèse H2, le traitement d'une requête continue sans dégradation de la qualité des résultats. Par exemple, sur un support d'exécution donné, cela revient à paralléliser au maximum les opérateurs et les placer sur les unités de traitement selon une stratégie de répartition de charge. Malheureusement, cette solution s'avère inappropriée dans un contexte d'exécution multi-requêtes. De plus, d'un point de vue énergétique et économique cela n'est pas souhaitable.

Dans Xu et Peng (2016), les auteurs nous présentent un algorithme permettant de faire des *scale-in* et *scale-out* à la demande. Cette approche repose soit sur l'utilisateur soit sur la définition d'un script lié à des règles métier, si elles existent. Enfin, cette approche est uniquement curative car n'augmente que le degré de parallélisme des opérateurs déjà congestionnés sans possibilité d'anticipation.

Des solutions (Gedik et al. (2014); Schneider et al. (2009)) permettent d'adapter dynamiquement et automatiquement le degré de parallélisme des opérateurs toutefois elles reposent sur la détection de congestions effectives. Bien que ces solutions puissent réduire la durée de congestion des opérateurs, elles ne peuvent les prévenir. De même, dans Heinze et al. (2014), les auteurs nous proposent une solution basée sur un algorithme d'apprentissage. De cette manière, le système peut adapter le degré de parallélisme des opérateurs en apprenant au fur et à mesure les gains réalisés en fonction des reconfigurations effectuées. Toutefois, cette détection se base sur la consommation des ressources (CPU et RAM) et donc en aval des congestions du système.

Ces différentes approches sont donc toutes curatives puisqu'elles interviennent pour résoudre un problème de congestion qui s'est déjà produit avec des conséquences sur la qualité des résultats. De plus, la plupart nécessite la présence et l'expertise de l'utilisateur. Une solution anticipant les congestions afin de les limiter, voire de les éviter, est donc naturellement souhaitable.

3 Anticipation de l'activité d'un opérateur

Nous proposons ici une formalisation des différentes notions que nous manipulons pour caractériser l'exécution d'une topologie de requête continue. De plus, nous introduisons les métriques nécessaires à l'estimation de l'adéquation entre degré de parallélisme et niveau d'activité des opérateurs.

Soit $\mathcal{T} = (\mathcal{O}, \mathcal{V})$ la topologie d'une requête continue représentée par un graphe orienté où l'ensemble $\mathcal{O}$ des nœuds représente les opérateurs et l'ensemble $\mathcal{V}$ des arcs représente le sens de transmission des données. Nous considérons chaque opérateur $\mathcal{O}_i$ comme un opérateur physique pouvant être exécuté en parallèle par un ensemble de tâches. Le nombre de ces tâches, noté $degree(\mathcal{O}_i)$, définit le degré de parallélisme de l'opérateur.

Soit $\mathcal{F}$ un ensemble de fenêtres d'analyse $\mathcal{F}_i = \{(F_j^i)\}_{j \in \mathbb{N}^+}$ chacune composée d'un ensemble d'itérations F_j^i. Chaque fenêtre d'analyse $\mathcal{F}_i$ est associée à l'opérateur O_i. Chaque F_j^i est définit par une durée Δ et regroupe les mesures effectuées durant cet intervalle de temps. Ces mesures sont effectuées selon un ensemble prédéfini de *timestamps* $\mathcal{M}_{i,j} = \{m_1^{i,j}, m_2^{i,j}, ..., m_n^{i,j}\}_{n \in \mathbb{N}^+}$. Pour chaque opérateur O_i, nous effectuons, à chaque *timestamp* $m_k^{i,j}$, des mesures prenant en compte les n-uplets reçus et traités sur l'intervalle $[m_{k-1}^{i,j}, m_k^{i,j}[$ avec k=1...n. Il est important de préciser que les mesures effectuées sur des opérateurs appartenant à une même topologie sont synchrones.

Soit $\mathcal{R}^i$ l'ensemble potentiellement infini des n-uplets émis en entrée de l'opérateur O_i. Nous définissons $\mathcal{R}^{i,j}$ l'ensemble des n-uplets reçus par l'opérateur O_i durant la fenêtre F_j^i et $\mathcal{R}_k^{i,j}$ l'ensemble des n-uplets reçus durant l'intervalle $[m_{k-1}^{i,j}, m_k^{i,j}[$.

3.1 Estimation de la charge à traiter par l'opérateur

Nous proposons d'estimer, en fin d'une itération, le nombre de n-uplets que le système devra traiter durant l'itération suivante et d'estimer si la capacité de traitement est compatible avec cette charge. Si ce n'est pas le cas l'opérateur sera considéré comme une source potentielle de congestion du système.

La charge à traiter d'un opérateur durant une itération correspond au nombre de n-uplets nouvellement reçus qui s'ajoutent aux n-uplets qui n'ont pas pu être traités à l'itération précédente. La charge effective de l'itération F_j^i correspond à la valeur :

$$Charge_j^i = |\mathcal{R}^{i,j}| + nbEnAttente_{F_{j-1}^i} \quad (1)$$

où $nbEnAttente(F_j^i)$ correspond au nombre de n-uplets en attente de traitement durant l'itération F_j^i et dont le traitement sera achevé durant une itération ultérieure. Du fait de la valeur $|\mathcal{R}^{i,j}|$ dans la formule (1), le calcul de cette charge effective ne peut se faire qu'à la fin de F_j^i.

Pour pouvoir anticiper une congestion, nous avons besoin d'estimer cette charge dès la fin de F^i_{j-1}. Nous estimons le nombre de nouveaux n-uplets reçus durant l'itération F^i_j par régression linéaire [3] en se basant sur le nombre de n-uplets observés durant l'itération F^i_{j-1}. Soit f^i_{j-1} la fonction affine calculée par régression linéaire à partir des couples $(m^{i,j-1}_k, \mathcal{R}^{i,j-1}_k)$. L'estimation du nombre total des n-uplets attendus durant l'itération F^i_j à la fin de l'itération F^i_{j-1} est définie par :

$$|Estim\mathcal{R}^{i,j}| = \sum_{m^{i,j-1}_k \in \mathcal{M}_{i,j-1}} \lceil f^i_{j-1}(m^{i,j-1}_k) \rceil \tag{2}$$

L'estimation de la charge attendue durant l'itération F^i_j est donc définie par :

$$EstimCharge_{F^i_j} = |Estim\mathcal{R}^{i,j}| + nbEnAttente_{F^i_{j-1}} \tag{3}$$

3.2 Estimation de la capacité de traitement de l'opérateur

Maintenant que nous avons une estimation de la quantité de n-uplets que devrait avoir à traiter l'opérateur O_i sur l'itération F^i_j, il nous reste à estimer la capacité de traitement de l'opérateur sur cette même itération.

Nous définissons la capacité de traitement d'un opérateur comme étant le nombre moyen de n-uplets que l'opérateur est capable de traiter durant une itération.

$$Capacite_{F^i_j} = \frac{1}{Lat_{F^i_j}} \times degree(\mathcal{O}_i) \times \Delta \tag{4}$$

où $Lat_{F^i_j}$ correspond à la latence intra-opérateur, hors file d'attente, moyenne observée sur les n-uplets traités durant l'itération F^i_j. Sous l'hypothèse H5 portant sur le caractère uniforme de la distribution des données dans les flux en entrée des opérateurs, nous utilisons simplement la covariance pour estimer la capacité moyenne de traitement de l'opérateur O_i durant la fenêtre F^i_j :

$$EstimCapacite_{F^i_j} = Capacite_{F^i_{j-1}} + \epsilon_i \tag{5}$$

où ϵ_i correspond à la covariance entre la capacité de traitement estimée pour l'itération F^i_j et les capacités de traitement observées sur les itérations précédentes.

3.3 Estimation du niveau d'activité et de la congestion d'un opérateur

Ces différentes observations et estimations nous permettent à présent de définir une métrique de contrôle de l'activité d'un opérateur. La notion de 'Niveau d'Activité', noté NdA, représente intuitivement l'adéquation entre degré de parallélisme et débit des flux en entrée. Elle est définie par :

$$NdA_{F^i_j} = \frac{EstimCharge_{F^i_j}}{EstimCapacite_{F^i_j}} \tag{6}$$

3. Choix arbitraire des auteurs d'une technique de régression non remise en cause par les résultats expérimentaux.

Soit θ_{min} et θ_{max}, deux seuils paramétrables définissant respectivement un niveau d'activité faible et un niveau d'activité fort. L'interprétation du NdA d'un opérateur est la suivante :

- Si $NdA_{F_j^i} \leq \theta_{min}$, l'activité de l'opérateur est dite 'faible' car la capacité de l'opérateur est considérée comme trop importante par rapport au nombre de n-uplets en attente de traitement durant l'itération F_j^i.
- Si $\theta_{min} < NdA_{F_j^i} \leq \theta_{max}$, l'activité de l'opérateur est dite 'normale' car l'opérateur est en capacité de traiter tous les n-uplets en attente de traitement durant l'itération F_j^i.
- Si $\theta_{max} < NdA_{F_j^i} \leq 1$, l'activité de l'opérateur est dite 'forte' car l'opérateur arrive en limite de capacité pour traiter tous les n-uplets en attente de traitement durant l'itération F_j^i.
- Si $NdA_{F_j^i} > 1$, l'activité de l'opérateur est dite 'critique' car l'opérateur n'est pas en mesure de traiter tous les n-uplets en attente de traitement durant l'itération F_j^i.

4 Approche *AUTOSCALE*

L'approche *AUTOSCALE* détermine pour chaque opérateur une modification de son degré de parallélisme qui peut être une augmentation (*scale-out*), une diminution (*scale-in*) ou une conservation en l'état (*nothing*). Notre approche prend en compte à la fois l'estimation du niveau d'activité de chaque opérateur mais aussi le contexte global. En effet, les reconfigurations ont des effets en cascade prévisibles. Par exemple, l'augmentation de la capacité d'un opérateur dont l'activité est forte ou critique va augmenter son débit en sortie et donc avoir un impact sur le débit en entrée des opérateurs en aval.

4.1 Initialisation du Graphe d'Actions Possibles

Dans un premier temps, nous allons définir l'ensemble des reconfigurations à effectuer pour chaque opérateur en fonction de son activité. Cet ensemble est représenté sous la forme d'un graphe disposant de la même structure que la topologie $\mathcal{T}$. Les sommets sont étiquetés par une proposition d'action sur l'opérateur correspondant dans $\mathcal{T}$. Les actions possibles sont : *scale-in* , *scale-out* ou *nothing*.

Soit $\mathcal{G}_j = (\mathcal{A}_j, \mathcal{V})$ le graphe des actions possibles ($\mathcal{A}_j$) pour la fenêtre F_j^i. Pour rappel, la fonction affine f_j^i calculée par régression linéaire pour estimer la charge d'un opérateur (voir formule 2). Cette fonction permet d'estimer la tendance d'évolution de la charge via sa dérivée. Si cette dérivée est strictement positive, la charge est considérée comme croissante, sinon elle est considérée comme décroissante ou constante.

Tous les opérateurs de la topologie sont parcourus les uns après les autres. Pour la fenêtre courante et pour l'opérateur courant, selon le niveau d'activité de l'opérateur (faible, normal, fort ou critique) et la tendance d'évolution de la charge, il est possible de proposer une action de modification du degré de parallélisme selon la matrice de décision locale définie dans le Tableau 1.

Activité de l'opérateur / Tendance d'évolution de la charge	activité faible	activité normale	activité forte	activité critique
Décroissante ou constante	*scale-in*	*nothing*	*nothing*	*scale-out*
Croissante	*nothing*	*nothing*	*scale-out*	*scale-out*

TAB. 1 – *Matrice de décision locale pour la construction du graphe d'actions possibles*

4.2 Prise en compte du contexte global

Une fois le graphe des actions possibles construit, l'approche *AUTOSCALE* vérifie la cohérence globale du graphe d'actions possibles ainsi obtenu. Pour cela nous introduisons une relation d'ordre sur l'ensemble des actions possibles. Ainsi, une action *scale-out* prédomine sur une action *scale-in* qui elle même prédomine sur une action *nothing*. Afin de déterminer si une action prise localement est cohérente globalement, nous définissons la matrice de décision globale (voir Tableau 2).

Pour le déclenchement des actions de reconfiguration, nous parcourons le graphe des actions possibles à partir des opérateurs d'entrée. En utilisant la matrice de décision globale, nous identifions l'ensemble des reconfigurations cohérentes à effectuer sur les opérateurs d'une topologie.

décision locale pour l'opérateur / action prédominante en amont de l'opérateur	*nothing*	*scale-in*	*scale-out*
nothing	*nothing*	*scale-in*	*scale-out*
scale-in	*nothing*	*scale-in*	*scale-out*
scale-out	*scale-out*	*nothing*	*scale-out*

TAB. 2 – *Matrice de décision globale*

Grâce à l'ordre sur l'ensemble des actions possibles, nous pouvons déterminer l'action prédominante observée sur les nœuds précédant le nœud courant dans la topologie. Ainsi, d'après le Tableau 2, un *scale-in* est validé si en amont du nœud courant, l'action prédominante est au plus un *scale-in*. Dans le cas contraire, le *scale-in* est remplacé par une action nulle afin d'éviter une contradiction à court terme.

4.3 Quantification des reconfigurations

Les étapes précédentes nous permettent de savoir s'il faut reconfigurer un opérateur ainsi que le type de reconfiguration Il nous reste donc à quantifier le degré de parallélisme de chaque opérateur à reconfigurer. Soit $degree_{j-1}(O_i)$, le nombre de tâches associées à l'opérateur O_i

durant l'itération F^i_{j-1}. Soit $maxP_{O_i}$ le nombre de maximal de tâches pouvant être initialisées pour l'opérateur O_i. Si une action *scale-out* ou *scale-in* est validée pour O_i alors le degré de parallélisme $degree_j(O_i)$ pour F^i_j sera défini par :

$$degree_j(O_i) = min(maxP_{O_i}, \lceil degree_{j-1}(O_i) \times \text{NdA}(F^i_j) \rceil)$$

Dans le cas particulier où un *scale-out* est préconisé à cause d'une activité forte mais non critique, le degré de parallélisme courant de l'opérateur est incrémenté de 1.

5 Expérimentations

Apache Storm permet de définir une requête continue sous la forme d'une *topologie* d'opérateurs dans un langage de programmation haut niveau (Java, Python, Clojure...). Cet outil permet à l'utilisateur de configurer la parallélisation des opérateurs. De plus, contrairement à Neumeyer et al. (2010); Zaharia et al. (2012), Storm n'impose pas un paradigme de représentation des données comme le modèle clé-valeur, fondement des approches MapReduce (Zaharia et al. (2012)). Cela offre une plus grande flexibilité pour la définition des opérateurs. De plus, *Apache Storm* garantit que chaque n-uplet soit suivi individuellement jusqu'à sa sortie d'une topologie. Nous avons développé notre approche en l'intégrant à *Apache Storm* 0.10.1. Nous présentons nos protocoles et résultats expérimentaux.

5.1 Protocole expérimental

Notre cluster de test se compose de 7 machines virtuelles disposant chacune de deux CPU Intel(R) Xeon(R) E5-2620 cadencés à 2.00GHz, 4Go de mémoire RAM et 40Go de disque dur. Une machine gère la coordination des 6 autres qui sont dédiées à l'exécution de tâches qui leurs sont affectées. Chacune de ces 6 machines (*supervisors*) gère 4 unités de traitement (*workers*). Notre module de gestion dynamique du degré de parallélisme implémente l'interface *IScheduler* de l'API Storm. Sur l'hôte du coordinateur (*Nimbus*), nous avons également déployé une base de données MySQL afin de stocker l'historique des différentes mesures détaillées dans la Section 3.

Afin de valider notre approche, nous avons choisi de mettre en avant son impact sur 3 topologies caractéristiques : une topologie linéaire, une topologie en étoile et une en diamant. Chaque topologie caractéristique est composée de deux types d'opérateurs : les opérateurs *intermediate* ayant une faible latence intra-opérateur et les opérateurs *sink* possédant une forte latence intra-opérateur.

Nous avons construit un flux synthétique possédant plusieurs caractéristiques : 1) une distribution uniforme des valeurs 2) des variations caractéristiques. En effet, comme présenté sur la figure 2, le flux d'entrée est stable en émettant peu de n-uplets puis il augmente exponentiellement en se stabilisant à un débit important. Enfin, le débit du flux synthétique diminue par paliers jusqu'à se stabiliser sur un débit faible. Nous avons également fait en sorte à ce que ce flux soit irrégulier afin nous approcher d'un flux réel. Cela nous permet d'observer l'adaptation automatique de Storm avec l'approche *AUTOSCALE* .

Nous nous comparons à la solution native dans deux configurations *Storm* pour lesquelles nous paramétrons le nombre de tâches par opérateur. Dû à notre support d'exécution, au plus

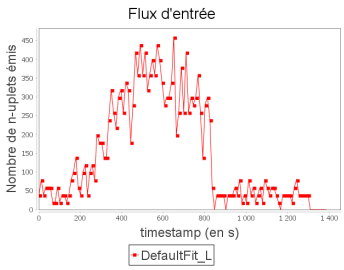

FIG. 2 – *Flux d'entrée des topologies*

24 *workers* peuvent être alloués pour une topologie. Dans la configuration *ConfMin*, le nombre initial de tâches par opérateur est équivalent au nombre minimal de tâches soit 1. Intuitivement, la configuration *ConfMin* est adaptée à de faibles débits en entrée mais ne peut absorber les débits maximaux. Dans la configuration *ConfExpt*, le nombre initial de tâches pour l'opérateur *intermediate* est de 1. Pour l'opérateur *sink*, le nombre initial de tâches est de 4 pour la topologie en étoile et 12 pour les topologies linéaire et diamant. Les choix faits pour la configuration *ConfExpt* se justifie par une expertise sur le flux d'entrée. En effet, les paramètres de la configuration *ConfExpt* permettent d'absorber les débits maximaux du flux d'entrée sans gaspillage de ressources.

Pour chaque configuration, nous avons mesuré la latence globale de la topologie (performance de calcul) et le nombre de n-uplets déphasés (qualité des résultats). En ce qui concerne la réactivité du système et la consommation des ressources, nous nous sommes intéressés au nombre de de tâches affectées pour chaque opérateur.

5.2 Résultats

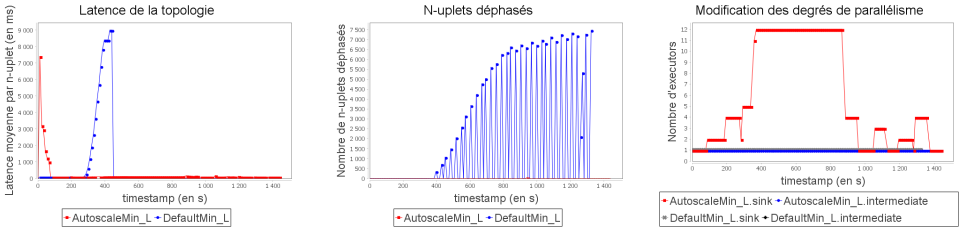

FIG. 3 – *Comparatif entre Storm (Default) et AUTOSCALE avec ConfMin.*

Nous avons choisi d'exposer uniquement les résultats pour la topologie linéaire qui sont représentatifs des résultats pour les autres topologies élémentaires. L'intégralité des résultats de nos expérimentations peuvent être consultés en annexe [4]. Afin de respecter les pratiques de programmation d'*Apache Storm*, nous avons implémenté le rejeu de n-uplets lorsque ceux-ci

4. Annexe disponible sur : https://liris.cnrs.fr/~rkottoko/autoscale/v1/

sont déphasés. Avec *ConfMin*, nous observons que le flux ne peut plus être absorbé et conduit à une congestion totale de la topologie. En effet, la topologie n'est plus en mesure d'émettre des n-uplets. Dès lors, tous les n-uplets émis par la source finissent par être déphasés et sont rejoués indéfiniment par la source jusqu'à intervention d'un utilisateur. À l'inverse, notre approche *AUTOSCALE* augmente dynamiquement et automatiquement le degré de parallélisme de l'opérateur critique afin d'absorber les variations du flux d'entrée. Lorsque le flux diminue, le degré de parallélisme est diminué en conséquence afin d'éviter la surconsommation de ressources.

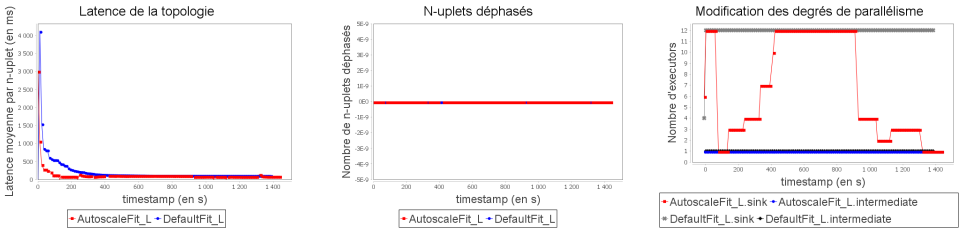

FIG. 4 – *Comparatif entre Storm (Default) et AUTOSCALE avec ConfExpt.*

Avec *ConfExpt* (voir Figure 4), nous partons d'une configuration capable d'absorber le débit maximal du flux d'entrée cependant cette configuration est surévaluée en début et fin de notre flux synthétique. Notre approche *AUTOSCALE* diminue donc le degré de parallélisme lorsque les opérateurs ne nécessitent pas une telle capacité de traitement au vue de la charge. Puis, à l'instar de la *ConfMin*, le degré de parallélisme est adapté progressivement. *AUTOSCALE* parvient ainsi à diminuer dynamiquement et automatiquement la quantité de ressources allouées de près de 30% et à maintenir une latence équivalente.

6 Conclusion

Nous avons proposé une approche permettant d'adapter dynamiquement et automatiquement le degré de parallélisme des opérateurs d'une topologie *Apache Storm* en fonction de l'évolution du flux d'entrée. Les expérimentations nous ont montré d'une part que cette approche permet de limiter les risques de congestion des opérateurs mais également qu'elle converge vers une configuration utilisant uniquement les ressources nécessaires. Cette approche offre un traitement élastique sous l'hypothèse que le support d'exécution fournisse suffisamment de ressources. Nos travaux en cours portent sur une expérimentation large échelle ainsi que sur les problématiques liées à des configurations où les ressources sont insuffisantes.

Références

Aniello, L., R. Baldoni, et L. Querzoni (2013). Adaptive online scheduling in storm. In *The 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13, Arlington, TX, USA - June 29 - July 03, 2013*, pp. 207–218.

Gedik, B., S. Schneider, M. Hirzel, et K.-L. Wu (2014). Elastic scaling for data stream processing. *IEEE Trans. Parallel Distrib. Syst. 25*(6), 1447–1463.

Heinze, T., V. Pappalardo, Z. Jerzak, et C. Fetzer (2014). Auto-scaling techniques for elastic data stream processing. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, New York, NY, USA, pp. 318–321. ACM.

Neumeyer, L., B. Robbins, A. Nair, et A. Kesari (2010). S4 : Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pp. 170–177.

Peng, B., M. Hosseini, Z. Hong, R. Farivar, et R. H. Campbell (2015). R-storm : Resource-aware scheduling in storm. In *Proceedings of the 16th Annual Middleware Conference, Vancouver, BC, Canada, December 07 - 11, 2015*, pp. 149–161.

Rivetti, N., L. Querzoni, E. Anceaume, Y. Busnel, et B. Sericola (2015). Efficient key grouping for near-optimal load balancing in stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015*, pp. 80–91.

Sattler, K.-U. et F. Beier (2013). Towards elastic stream processing : Patterns and infrastructure. In G. Cormode, K. Yi, A. Deligiannakis, et M. N. Garofalakis (Eds.), *BD3@VLDB*, Volume 1018 of *CEUR Workshop Proceedings*, pp. 49–54. CEUR-WS.org.

Schneider, S., H. Andrade, B. Gedik, A. Biem, et K.-L. Wu (2009). Elastic scaling of data parallel operators in stream processing. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–12.

Xu et G. Peng (2016). Stela : Enabling stream processing systems to scale-in and scale-out on-demand. In *Proc. IEEE International Conference on Cloud Engineering (IC2E), 2016*.

Xu, J., Z. Chen, J. Tang, et S. Su (2014). T-storm : Traffic-aware online scheduling in storm. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pp. 535–544.

Zaharia, M., T. Das, H. Li, T. Hunter, S. Shenker, et I. Stoica (2012). Discretized streams : A fault-tolerant model for scalable stream processing. Technical Report UCB/EECS-2012-259, EECS Department, University of California, Berkeley.

Summary

In a context of stream processing, it is important to guarantee some properties of performance, quality of results and scalability to final users. Adjusting resource usage to processing requirements in order to consume only necessary resources, is a major challenge dealing with Big Data and Green IT. The approach suggested in this article, adapts dynamically and automatically the parallelism degree of operators belonging to a same continuous query. It takes into account the evolution of input stream rates. We suggest i) a metric estimating the activity level of operators in a near future ii) the approach *AUTOSCALE* which evaluates the gain brought by a set of the parallelism degree modifications at local and global scope iii) thanks to an integration to the solution *Apache Storm*, we show performance tests comparing our approach to the native solution of this stream processing engine.