Distributed query processing over fluctuating streams

Roland Kotto-Kombi

N°d'ordre NNT : 2018LYSEI050

THESE de DOCTORAT DE L'UNIVERSITE DE LYON
opérée au sein de
L'INSA DE LYON

Ecole Doctorale N° 532
INFOMATHS

Spécialité/ discipline de doctorat :
INFORMATIQUE

Soutenue publiquement le 29/06/2018, par :
Roland Olivier Kotto Kombi

Distributed query processing over fluctuating streams

Devant le jury composé de :

Defude, Bruno	Professeur	Télécom Sud Paris	Président
Amann, Bernd	Professeur	Sorbonne Université	Rapporteur
Morvan, Franck	Professeur	Université Paul Sabatier	Rapporteur
Skaf-Molli, Hala	Maître de Conférences	Université de Nantes	Examinatrice
Lamarre, Philippe	Professeur	INSA de Lyon	Directeur de thèse
Lumineau, Nicolas	Maître de Conférences	Université Claude Bernard Lyon 1	Co-directeur de thèse

Département FEDORA – INSA Lyon - Ecoles Doctorales – Quinquennal 2016-2020

SIGLE	ECOLE DOCTORALE	NOM ET COORDONNEES DU RESPONSABLE
CHIMIE	**CHIMIE DE LYON** **http://www.edchimie-lyon.fr** Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage **secretariat@edchimie-lyon.fr** INSA : R. GOURDON	M. Stéphane DANIELE Institut de recherches sur la catalyse et l'environnement de Lyon IRCELYON-UMR 5256 Équipe CDFA 2 Avenue Albert EINSTEIN 69 626 Villeurbanne CEDEX **directeur@edchimie-lyon.fr**
E.E.A.	**ÉLECTRONIQUE, ÉLECTROTECHNIQUE, AUTOMATIQUE** **http://edeea.ec-lyon.fr** Sec. : M.C. HAVGOUDOUKIAN **ecole-doctorale.eea@ec-lyon.fr**	M. Gérard SCORLETTI École Centrale de Lyon 36 Avenue Guy DE COLLONGUE 69 134 Écully Tél : 04.72.18.60.97 Fax 04.78.43.37.17 **gerard.scorletti@ec-lyon.fr**
E2M2	**ÉVOLUTION, ÉCOSYSTÈME, MICROBIOLOGIE, MODÉLISATION** **http://e2m2.universite-lyon.fr** Sec. : Sylvie ROBERJOT Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 INSA : H. CHARLES **secretariat.e2m2@univ-lyon1.fr**	M. Philippe NORMAND UMR 5557 Lab. d'Ecologie Microbienne Université Claude Bernard Lyon 1 Bâtiment Mendel 43, boulevard du 11 Novembre 1918 69 622 Villeurbanne CEDEX **philippe.normand@univ-lyon1.fr**
EDISS	**INTERDISCIPLINAIRE SCIENCES-SANTÉ** **http://www.ediss-lyon.fr** Sec. : Sylvie ROBERJOT Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 INSA : M. LAGARDE **secretariat.ediss@univ-lyon1.fr**	Mme Emmanuelle CANET-SOULAS INSERM U1060, CarMeN lab, Univ. Lyon 1 Bâtiment IMBL 11 Avenue Jean CAPELLE INSA de Lyon 69 621 Villeurbanne Tél : 04.72.68.49.09 Fax : 04.72.68.49.16 **emmanuelle.canet@univ-lyon1.fr**
INFOMATHS	**INFORMATIQUE ET MATHÉMATIQUES** **http://edinfomaths.universite-lyon.fr** Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage Tél : 04.72.43.80.46 Fax : 04.72.43.16.87 **infomaths@univ-lyon1.fr**	M. Luca ZAMBONI Bât. Braconnier 43 Boulevard du 11 novembre 1918 69 622 Villeurbanne CEDEX Tél : 04.26.23.45.52 **zamboni@maths.univ-lyon1.fr**
Matériaux	**MATÉRIAUX DE LYON** **http://ed34.universite-lyon.fr** Sec. : Marion COMBE Tél : 04.72.43.71.70 Fax : 04.72.43.87.12 Bât. Direction **ed.materiaux@insa-lyon.fr**	M. Jean-Yves BUFFIÈRE INSA de Lyon MATEIS - Bât. Saint-Exupéry 7 Avenue Jean CAPELLE 69 621 Villeurbanne CEDEX Tél : 04.72.43.71.70 Fax : 04.72.43.85.28 **jean-yves.buffiere@insa-lyon.fr**
MEGA	**MÉCANIQUE, ÉNERGÉTIQUE, GÉNIE CIVIL, ACOUSTIQUE** **http://edmega.universite-lyon.fr** Sec. : Marion COMBE Tél : 04.72.43.71.70 Fax : 04.72.43.87.12 Bât. Direction **mega@insa-lyon.fr**	M. Jocelyn BONJOUR INSA de Lyon Laboratoire CETHIL Bâtiment Sadi-Carnot 9, rue de la Physique 69 621 Villeurbanne CEDEX **jocelyn.bonjour@insa-lyon.fr**
ScSo	**ScSo*** **http://ed483.univ-lyon2.fr** Sec. : Viviane POLSINELLI Brigitte DUBOIS INSA : J.Y. TOUSSAINT Tél : 04.78.69.72.76 **viviane.polsinelli@univ-lyon2.fr**	M. Christian MONTES Université Lyon 2 86 Rue Pasteur 69 365 Lyon CEDEX 07 **christian.montes@univ-lyon2.fr**

*ScSo : Histoire, Géographie, Aménagement, Urbanisme, Archéologie, Science politique, Sociologie, Anthropologie

Résumé étendu en français de la thèse intitulée Distributed query processing over fluctuating data streams

Roland Kotto Kombi

1 Introduction et motivation

Avec la démocratisation d'applications consommant des données en temps réel (géolocalisation, recommandations ciblées, appareils intelligents, etc.), de larges volumes de données, dits *flux de données*, transitent entre les sources ayant produites ces données et des services de traitement. Un flux peut être représenté comme une séquence potentiellement infinie de données estampillées et émises en temps réel. Les utilisateurs peuvent interrogés un ensemble de flux via un langage de requête dédié [1, 2, 3]. Cependant, à l'inverse des requêtes sur des données statiques, les requêtes sur des flux de données génèrent de nouveaux résultats dès que de nouveaux n-uplets sont reçus en entrée et sont alors dites *continues*. Ces requêtes continues sont traitées par des systèmes de gestion de flux de données (SGFD) académiques [4, 5, 6], industriels [2, 7, 8] et libres [9, 10, 11] reposant sur une infrastructure distribuée. Une infrastructure distribuée est composée d'un ensemble d'unités de traitement définies par des ressources initiales (processeur et mémoire) potentiellement partagées.

Afin de répartir l'utilisation des ressources nécessaires à l'exécution d'une requête continue, un SGFD distribué transforme chaque requête soumise par un utilisateur en un graphe d'opérateurs [3, 9, 10]. En fonction de son type, un opérateur peut être répliqué en plusieurs tâches. Ce nombre de tâches est appelé le *degré de parallélisme* de l'opérateur. Les tâches sont ensuite affectées sur des unités de traitement. Il est important de noter que le flux en entrée de chaque tâche a potentiellement un débit variant en terme de volume et de distribution des valeurs au cours du temps. La qualité des résultats générés par les requêtes continues émises par l'utilisateur dépend de la capacité de l'infrastructure à absorber la charge induite par les flux d'entrée.

1.1 Traitement distribué et élastique de flux de données

Ces travaux ont été réalisés dans le cadre du projet ANR Socioplug [1] (ANR-13-INFR-0003), nous nous intéressons à une plateforme de traitement de flux distribuée où chaque unité de traitement est un nano-ordinateur connecté à Internet ayant des capacités de traitement et de stockage limitées. Chaque utilisateur dispose d'une unité de traitement et peut interroger un ensemble de flux via des requêtes continues. Afin de garantir le traitement de ces requêtes indépendamment du débit des flux d'entrée, la plateforme regroupe tous les utilisateurs intéressés par les mêmes résultats en une *communauté*. La requête continue caractérisant une communauté est alors distribuée et exécutée une seule fois sur l'ensemble des unités de traitement associées à la communauté. Cela permet aux utilisateurs d'interroger des flux potentiellement massifs tout en leur donnant le contrôle sur le traitement de leurs données et leurs requêtes.

1.2 Problématique

En considérant des flux variants, deux phénomènes peuvent être observés ; phénomènes pouvant être problématique du point de vue de l'utilisateur, si cela implique une dégradation de la qualité des résultats et du point de vue du fournisseur de ressources si des ressources allouées ne sont pas exploitées.

— Le premier phénomène dit de *sur-allocation* apparait lorsque le débit des flux d'entrée est largement inférieur au débit de traitement des tâches d'un opérateur. Cela implique que le degré de parallélisme est trop important et donc que des ressources superflues sont sollicités par le SGFD. Ces ressources peuvent être libérés afin de bénéficier à d'autres tâches.

— Le second phénomène dit de *risque de congestion* apparait lorsque des tâches ne peuvent plus traiter les n-uplets aussi vite qu'ils arrivent. En effet, le débit de traitement d'un opérateur est borné par la complexité de la fonction qu'il implémente. Dans ce cas, les n-uplets en entrée de la tâche s'accumulent en attendant d'être traité ce qui rallonge la latence de traitement globale (*i.e.*, le temps mis par un n-uplet pour traverser le graphe d'opérateurs). Lorsque le volume de n-uplets accumulés en entrée d'une tâche dépasse la quantité de mémoire prévue au stockage temporaire des données, cela entraine la *congestion* de l'opérateur. Dès lors, l'opérateur congestionné entraine la perte irréversible de n-uplets et requiert davantage de ressources disponibles pour traiter l'intégralité de ses entrées.

Or, le SGFD n'a pas de contrôle sur l'évolution des flux d'entrée du point de vue du volume et de la distribution des valeurs. Il est donc nécessaire d'adapter l'exécution d'une requête continue sur l'infrastructure distribué

1. `http://socioplug.univ-nantes.fr/index.php/SocioPlug_Project`

lorsque l'un des phénomènes est observé. Le problème est donc de savoir comment ajuster dynamiquement et automatiquement les ressources utilisables par chaque opérateur aux besoins de traitement évoluant au cours du temps. Pour ce faire, nous nous intéressons dans le cadre de ces travaux de thèse à la définition d'une stratégie d'auto-parallélisation des opérateurs qui possèdent les caractéristiques suivantes :

- Augmenter le degré de parallélisme d'un opérateur lorsqu'il présente un risque de congestion et ce avant que la congestion ne soit effective.
- Réduire le degré de parallélisme d'un opérateur lorsque ses besoins de traitement sont significativement inférieurs à son débit de traitement global.
- Effectuer les modifications de degré de parallélisme uniquement lorsque cela présente un bénéfice en terme de performance sur la durée afin d'éviter d'introduire des coûts de reconfiguration trop importants.

1.3 Contributions

Afin de répondre aux différents défis, nous proposons les contributions suivantes :

- Une étude bibliographique présentant les différents mécanismes permettant un traitement élastique des flux de données. Nous expliquons également comment ces méthodes sont utilisées dans les principaux SGFD et nous suggérons une classification des méthodes de détection d'opérateurs congestionnés.
- Une classification d'une large sélection de SGFD représentatifs des solutions existantes. Cette classification est basée sur des critères allant de la représentation des requêtes à la gestion d'opérateurs congestionnés.
- Une architecture générique, appelée *ORACL*, distinguant les différents étapes pour la reconfiguration dynamique de requêtes continues.
- Une approche originale d'auto-parallélisation des opérateurs baptisée AUTOSCALE. Cette approche repose sur une observation régulière de l'activité des opérateurs afin de déduire la charge à traiter dans un futur proche. Cela permet d'ajuster le degré de parallélisme des opérateurs de manière proactive. De plus, nous proposons un algorithme permettant de vérifier la cohérence d'un ensemble de reconfigurations à l'échelle du graphe d'opérateurs.
- Une extension d'AUTOSCALE, nommée AUTOSCALE+ prenant en compte la consommation réelle du processeur pour chaque tâche.
- Une association d'AUTOSCALE+ avec une stratégie d'équilibrage de charge compatible afin de proposer une approche combinée nommée DABS. Cette approche permet d'identifier avec précision les besoins en ressources d'opérateurs sensibles, en terme de latence de traitement, à la distribution des valeurs des flux d'entrée.

2 État de l'art

Durant ces travaux de thèse, nous nous sommes intéressés aux différents aspects du traitement de flux de données sur des infrastructures distribuées. Cela nous a conduit a étudié dans un premier temps les notions fondamentales nécessaires à l'interrogation de flux de données via des requêtes continues.

2.1 Traitement de flux de données

Les flux de données sont des séquences potentiellement infinies de données éphémères arrivant à des débits variants au cours du temps. Nous identifions 4 classes de flux en fonction de la variation de leurs débits : des flux constants, bornés, par motifs réguliers et enfin erratiques. Pour chacune de ces classes nous considérons deux types : les flux *équilibrés* et *déséquilibrés*. Le type d'un flux dépend à la fois de la distribution des valeurs et de la sensibilité d'une requête continue aux valeurs en entrée en terme de latence de traitement.

Afin de définir ces requêtes continues plusieurs types de langages existent :

— Des langages déclaratifs permettant de définir une requête continue sous forme d'une expression composée d'opérateurs prédéfinis.

— Des langages graphiques permettant de définir une requête continue directement comme un graphe d'opérateurs prédéfinis.

— Des langages impératifs permettant de définir une requête continue comme une succession d'opérateurs définis par l'utilisateur dans un langage de programmation haut niveau tel que Java ou Python.

Quelque soit le type de langage utilisé, un SGFD distribué transforme la requête en un graphe d'opérateurs. Chaque opérateur peut être divisé en un ensemble de tâches qui sont allouées sur des unités de traitement du support d'exécution. Chaque tâche reçoit en entrée un flux au débit variant au cours du temps.

2.2 Gestion d'opérateurs congestionnés en contexte flux

Le flux d'entrée de chaque tâche pouvant atteindre des débits critiques à n'importe quel instant durant le traitement, il apparait alors nécessaire que les SGFD intègrent des mécanismes permettant d'adapter les traitements au débit en entrée. Ces mécanismes sont alors dits *élastiques* car ils permettent d'augmenter ou de réduire le débit de traitement global d'un graphe d'opérateurs. Ces différents mécanismes peuvent modifier le graphe d'opérateurs à différents niveaux :

— Au niveau du graphe d'opérateurs dans son intégralité en modifiant l'ordre d'exécution des opérateurs sur les données entrantes.

— Au niveau des opérateurs en modifiant le nombre de tâches associées à chaque opérateur. L'enjeu majeur de la parallélisation consiste à

ajuster le degré de parallélisme des opérateurs pour fournir les ressources processeur et mémoire nécessaires aux traitements. De plus, le placement des tâches sur les unités de traitement peut impacter le débit entre tâches successives.

— Au niveau de l'implémentation, le choix d'une implémentation au détriment d'une autre peut avoir un impact important sur la latence de traitement et donc sur le débit de traitement.

— En fonction de la distribution des valeurs et de la fonction appliquée par l'opérateur, il peut être nécessaire d'équilibrer la charge de données à traiter entre les différentes tâches. Cela consiste à partitionner les données entrantes entre toutes les tâches associées à un même opérateur. Enfin, lorsque le volume de données est trop important, des stratégies d'échantillonnage permettent de réduire le débit d'entrée tout en maitrisant la dégradation de la qualité des résultats.

À chaque niveau, les mécanismes élastiques peuvent être déclenchés soit sur demande de l'utilisateur, soit de manière automatique. Le déclenchement manuel requiert une expertise et une observation constante de la part de l'utilisateur afin d'améliorer les performances d'une requête continue. Le déclenchement automatique permet d'ajuster les ressources allouées aux besoins des traitements, toutefois, la réactivité de la méthode dépend des métriques considérées pour identifier des opérateurs congestionnés. Afin de mettre en évidence les différences entre les méthodes de détection d'opérateurs congestionnés, nous proposons de les regrouper en catégories. Ces catégories sont basées sur la capacité d'anticipation de ces méthodes et sur l'algorithme de prise de décision, *i.e.* quand une augmentation ou une diminution du degré de parallélisme doit être effectuée.

2.3 Classification des SGFD

Suite à l'étude bibliographique, nous avons identifié une grande variété de langages de requêtes et de méthodes de gestion d'opérateurs congestionnés. Nous suggérons alors une classification de SGFD basée sur la représentation des requêtes continues (graphes d'opérateurs ou jobs *MapReduce*), le type de langage de requêtes supporté et la gestion des opérateurs congestionnés. Pour chacun des systèmes sélectionnés, nous analysons chaque critère et expliquons comment une requête est gérée de sa déclaration à la gestion d'opérateurs congestionnés pendant l'exécution. Les SGFD analysés et intégrés dans cette classification ont été sélectionné pour leur performance et leur représentativité des solutions existantes. Cette classification met en évidence la grande diversité des solutions mais également l'absence de solutions prenant en compte tous les aspects nécessaires à la gestion dynamique d'opérateurs congestionnés.

3 Traitement élastique de flux de données

Différents facteurs interviennent dans l'apparition d'une congestion. En effet, entre la requête continue définie par l'utilisateur et les tâches distribuées sur un ensemble d'unités de traitement interconnectées, plusieurs phases d'optimisation sont appliquées. Les choix effectués par le SGFD durant chacune de ces phases peut causer l'apparition d'un goulot d'étranglement au niveau du réseau ou des traitements.

3.1 Optimisation algébrique

À l'instar des SGBD, certains SGFD [12, 3, 5] permettent aux utilisateurs de définir des requêtes à l'aide d'un langage déclaratif. La requête correspond alors à une expression traduite par le SGFD en une topologie équivalente. En fonction des propriétés algébriques des opérateurs, le SGFD peut chercher l'ordonnancement optimal des opérateurs afin de minimiser le volume de données transitant entre les opérateurs. Cette phase d'optimisation se base sur les solutions éprouvées depuis de nombreuses années dans les systèmes de gestion de base de données (SGBD).

3.2 Parallélisation des opérateurs

Une fois un graphe d'opérateurs choisi, un SGFD distribué peut fixer le degré de parallélisme de chaque opérateur. Ce choix a un impact direct sur le débit de traitement théorique de l'opérateur et donc le débit maximal que l'opérateur peut recevoir en entrée sans accumuler de n-uplets sur sa file d'attente. Il apparait donc évident qu'un degré de parallélisme sous-évalué par rapport au débit réel du flux entrainera à terme une congestion. À l'inverse, un degré de parallélisme significativement surévalué implique une réservation de ressources superflues pour les besoins de traitement.

3.3 Allocation des tâches

Lorsque le degré de parallélisme de tous les opérateurs est fixé, le SGFD doit décider du plan d'affectation des tâches sur les unités de traitement. Ce plan d'allocation est défini selon une stratégie d'allocation. Certaines stratégies [13, 14] ont pour objectif de minimiser les échanges réseaux entre les unités de traitement tandis que d'autres stratégies [9] ont pour objectif de concentrer toutes les tâches sur un sous-ensemble minimal d'unités de traitement. Pour chacune de ces stratégies, des contraintes sur les ressources processeur et mémoire nécessaires sont considérées. En fonction des opérateurs et de la structure du graphe d'opérateurs, l'adoption d'une stratégie peut amener à la formation de goulots d'étranglement réseaux ou à des surcharges fréquentes des unités de traitement actives. Dans les deux cas, cela

entrainera des déplacements de tâches, réduisant la stabilité du système et augmentant la latence de traitement globale.

3.4 Optimisation locale

Certains SGFD proposent un ensemble d'opérateurs logiques pour la définition de requêtes continues. Pour chaque opérateur logique, plusieurs implémentations peuvent être disponibles afin d'adapter localement les traitements en fonction du contexte d'exécution. Par exemple, pour un opérateur de jointure, un SGFD peut disposer d'une implémentation basée sur un algorithme de hachage et une autre implémentation sur des boucles liées. Selon la taille des entrées une implémentation est plus performante que l'autre.

Grâce à l'étude approfondie de différents SGFD, nous avons identifié que les choix de stratégies pour la parallélisation des opérateurs, l'équilibrage de charge intra-opérateur et l'allocation des tâches sont primordiaux pour maintenir un équilibre entre besoins des traitement et ressources allouées. Bien que des solutions automatiques et quasi-optimales existent pour l'équilibrage de charge intra-opérateur et l'allocation des tâches, les stratégies de parallélisation des opérateurs présentent des limites importantes en terme de réactivité et d'automaticité.

4 AUTOSCALE : Une approche préventive pour le traitement élastique de flux de données

Dans le cadre de cette thèse, nous proposons une approche préventive d'auto-parallélisation des opérateurs, nommée AUTOSCALE, identifiant les opérateurs présentant un risque de congestion dans un futur proche et analysant le contexte d'exécution pour estimer le degré de parallélisme adapté. Pour ce faire, AUTOSCALE repose sur les deux grandes étapes suivantes : i) l'estimation de l'activité de chaque opérateur dans un futur proche grâce à une métrique d'activité ii) une analyse à l'échelle du graphe d'opérateurs permettant d'identifier le sous-ensemble cohérent d'opérateurs à reconfigurer pour éviter l'apparition d'opérateurs congestionnées.

4.1 Estimation de l'activité des opérateurs

Afin d'estimer les risques de congestion dans un futur proche, nous observons chaque opérateur individuellement sur une fenêtre d'observation. À intervalle de temps régulier, le nombre de n-uplets en entrée est mesuré. À partir de toutes les mesures effectuées durant la fenêtre d'observation, nous estimons, par régression linéaire, le volume global de n-uplets que l'opérateur aura à traiter durant la prochaine itération de la fenêtre d'observation. Dans

le même temps, nous mesurons la latence moyenne par n-uplet de l'opérateur afin de dériver le nombre théorique de n-uplets qu'il pourra traiter durant la prochaine itération de la fenêtre d'observation en tenant compte également du nombre de tâches traitant les données en parallèle. En comparant ces deux volumes, nous pouvons alors estimer si l'opérateur est capable, avec son degré de parallélisme courant, de traiter le volume estimé de n-uplets dans un futur proche.

4.2 Analyse des reconfigurations à l'échelle du graphe d'opérateurs

Grâce à la métrique d'activité, il est alors possible de déterminer si un opérateur nécessite une augmentation, une diminution ou la conservation de son degré de parallélisme courant. Toutefois, une prise de décision basée sur des estimations locales peut s'avérer inappropriée du point de vue de l'état global d'une topologie.

En effet, la modification du degré de parallélisme d'un opérateur peut impacter fortement la quantité de données en entrée des opérateurs en aval de celui-ci. Il serait donc importun de prendre la décision locale de réduire le degré de parallélisme d'un opérateur suite à l'observation d'une faible activité si le degré de parallélisme de l'opérateur en amont est fortement modifié.

Cette analyse globale favorise la stabilité du système en évitant de déclencher des reconfigurations qui seront contredites à court terme mais également en évitant des reconfigurations antagonistes qui dégradant les performances du SGFD au lieu de les améliorer. Enfin, cette analyse globale permet au SGFD de disposer de deux estimations de l'activité d'un opérateur et de choisir soit une stabilité accrue en ne modifiant le degré de parallélisme que lorsque les deux estimations le suggèrent soit une réactivité accrue en modifiant le degré de parallélisme dès la détection d'un risque de congestion.

4.3 Évaluation expérimentale d'AUTOSCALE

Afin d'évaluer l'approche AUTOSCALE, nous avons choisi de la comparer avec le comportement natif de la solution Apache Storm selon différentes configurations. Apache Storm permet aux utilisateurs de définir le degré de parallélisme de chaque opérateur à l'initialisation de la requête continue. Par défaut, ces degrés initiaux ne sont pas modifiés par Storm.

Nous avons proposé un micro-benchmark composé de quatre requêtes continues ou *topologies* :

— Trois topologies simples : linéaire, diamant et en étoile. Ces topologies sont dites élémentaires car la majorité des topologies peuvent être découpées selon ces motifs. Chacune de ces topologies est composée d'une ou plusieurs sources, d'au moins un opérateur à forte latence de traitement et d'au moins un opérateur à faible latence de traitement.

— Une topologie complexe inspirée par un cas d'usage réel. Cette topologie possède plusieurs opérateurs à forte et faible latence de traitement.

Pour chacune de ces topologies, nous pouvons appliquer 2 types de flux : un flux ayant une évolution progressive du débit en entrée et un flux ayant des augmentations et des diminutions soudaines du débit en entrée.

Nous définissons également deux configurations pour chaque topologie : une configuration minimale définissant un degré de parallélisme pour chaque opérateur adapté à des débits faibles en entrée et une configuration experte définissant des degrés de parallélisme adaptés au débits maximaux des flux.

Sur l'ensemble des évaluations, nous mettons en évidence l'intérêt d'AUTOSCALE pour éviter la congestion d'opérateurs en configuration minimale au démarrage. En effet, lors de l'augmentation du débit en entrée, AUTOSCALE augmente le degré de parallélisme des opérateurs à forte latence de traitement afin de maintenir un débit de traitement supérieur au débit d'entrée. Dans le cas de Storm, la congestion entraîne un arrêt complet des traitement et des pertes irréversibles de données. Lorsque la configuration experte est choisie, AUTOSCALE parvient à maintenir une latence de traitement globale suffisamment faible pour éviter l'apparition de congestions tout en réduisant lorsque cela est possible les degrés de parallélisme des opérateurs. Au plus, AUTOSCALE parvient à réduire la consommation de ressources de 37% pour des performances équivalentes.

5 Vers un modèle d'auto-parallélisation prenant en compte le contexte d'exécution

AUTOSCALE propose une approche d'auto-parallélisation entièrement guidée par les données. En effet, l'adéquation entre ressources nécessaires pour les traitements et ressources disponibles avec le degré de parallélisme courant est estimée sous réserve que les deux conditions suivantes soient remplies :

— Chaque tâche dispose d'autant de ressources que nécessaire sur l'unité de traitement sur laquelle elle est assignée.

— La charge de traitement est équitablement répartie entre les tâches associée à un même opérateur indépendamment des propriétés du flux.

La première condition peut ne pas être vérifiée lorsque, sur une unité de traitement donnée, la somme des ressources requises par les tâches dépasse les ressources disponibles. Dans ce cas, une concurrence pour l'utilisation du processeur et de la mémoire apparait, causant potentiellement un décalage entre débit de traitement estimé et débit de traitement réel.

Concernant la seconde condition, il est important de noter que certaines applications sont composées d'opérateurs naturellement sensibles aux valeurs passées en entrée. Par exemple, une jointure génère un volume de données en sortie fonction du volume de données en entrée et de la répartition des

valeurs dans l'ensemble des données en entrée. Si le résultat de cette jointure est calculé en parallèle grâce à un partitionnement des entrées, il n'est pas garantie que chaque partition nécessite un temps de traitement équivalent aux autres du fait de distributions de valeurs potentiellement différentes.

Il apparait alors nécessaire de prendre en considération les ressources réellement utilisables pour chaque tâche afin d'estimer un débit de traitement plus proche de la valeur réelle. De plus, la combinaison d'AUTOSCALE avec une stratégie d'équilibrage de charge adaptée est nécessaire afin de maintenir une charge équivalente entre les tâches associées à même un opérateur.

5.1 Prise en compte des ressources pour l'évaluation de l'activité

Afin d'estimer la capacité de chaque opérateur tout en prenant en compte les ressources réellement utilisables, nous nous focalisons sur le temps processeur utilisable pour chaque tâche. À la différence de certains systèmes [9] considérant uniquement des réservations de temps processeur statiques, nous suggérons de mesurer régulièrement, l'usage réel du processeur par chaque tâche. Ainsi, pour un opérateur donné, nous pouvons estimer le temps processeur moyen réellement utilisable par chacune de ses tâches et donc, déduire la capacité globale de l'opérateur avec une précision accrue par rapport à l'approche AUTOSCALE.

5.2 Évaluation expérimentale d'AUTOSCALE+

Lors de l'évaluation d'AUTOSCALE+, nous avons choisi de nous concentrer sur l'apport d'une approche préventive par rapport à des approches courantes de la littérature. Nous avons ainsi implémenté deux stratégies d'auto-parallélisation : une approche par exploration incrémentale des degrés de parallélisme et une approche par apprentissage avec renforcement.

Nous avons modifié le micro-benchmark en considérant une topologie linéaire possédant un opérateur à forte latence et insensible aux valeurs en entrée, une topologie linéaire équivalente mais avec un opérateur à latence sensible aux valeurs en entrée et enfin une topologie complexe possédant un opérateur sensible aux valeurs en entrées.

Nous avons considéré cette fois-ci 4 flux :

— 2 flux ayant respectivement des variations progressives et soudaines du débit et une distribution uniforme des valeurs.

— 2 flux ayant les mêmes variations en terme de débit mais ayant une distribution des valeurs biaisée selon une loi de distribution prédéfinie.

Étant donné que nous comparons uniquement des stratégies de parallélisation automatique des opérateurs, toutes les topologies sont initialisées avec une configuration minimale. Nous observons qu'AUTOSCALE+ adapte au moins aussi bien le degré de parallélisme d'une topologie qu'une straté-

gie basée sur de l'apprentissage par renforcement déjà entraînée. De plus, AUTOSCALE+ parvient à réduire la latence de traitement en ne réduisant le degré de parallélisme que lorsque cela présente un intérêt significatif en terme d'économie de ressources.

6 DABS : Association d'AUTOSCALE+ avec une stratégie d'équilibrage de charge

6.1 Principe

Pour pallier le déséquilibre de charge entre les tâches associées à un même opérateur, plusieurs stratégies existent. Certaines stratégies établissent des partitions en fonction de la valeur de chaque n-uplet présent dans le flux. D'autres stratégies tendent à répartir équitablement le nombre de n-uplets dans chaque partition indépendamment des valeurs des n-uplets. Dans les deux cas, la charge entre les tâches n'est équilibrée que sous l'hypothèse que le flux d'entrée a une distribution des valeurs ne variant pas significativement tout au long des traitements. Nous suggérons alors de combiner AUTOSCALE+ avec une stratégie d'équilibrage de charge prenant en compte l'évolution de la distribution des valeurs dans le flux. Cette stratégie, baptisée OSG, est basée sur une évaluation de temps de traitement de chaque valeur présente dans le flux d'entrée. Grâce à une structure de données compressée, OSG peut évaluer la charge de chaque tâche en terme de temps de traitement, afin d'équilibrer les files d'attentes de chaque tâche.

6.2 Évaluation expérimentale de DABS

Afin d'évaluer la performance de l'approche DABS, nous avons étudié la performance combinée de stratégies d'auto-parallélisation avec différentes solutions pour l'équilibrage de charge entre tâches d'un même opérateur. En effet, certains opérateurs étant naturellement sensible aux valeurs en entrée, en terme de latence de traitement, le choix de la méthode d'équilibrage de charge peut avoir un impact majeur sur les performances.

Nous avons testé DABS face à AUTOSCALE+ sur une topologie complexe face à des flux ayant un biais important dans la distribution des valeurs. Il apparait que DABS permet de réduire considérablement le déséquilibre de charge entre les tâches d'un même opérateur. Cela a pour conséquence de réduire le taux de remplissage des files d'attente jusqu'à 11% sur un même flux et donc de réduire la latence de traitement globale. De plus, DABS peut délivrer de meilleures performances qu'AUTOSCALE+ avec un degré de parallélisme moins élevé. En effet, grâce à une meilleure répartition de la charge, le débit de traitement global d'un opérateur est plus important pour un degré de parallélisme donné.

Références

[1] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language : Semantic foundations and query execution," *The VLDB Journal*, vol. 15, pp. 121–142, June 2006.

[2] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams : A fault-tolerant model for scalable stream processing," Tech. Rep. UCB/EECS-2012-259, EECS Department, University of California, Berkeley, Dec 2012.

[3] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora : A new model and architecture for data stream management," *The VLDB Journal*, vol. 12, pp. 120–139, Aug. 2003.

[4] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik, "Scalable Distributed Stream Processing," in *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, (Asilomar, CA), January 2003.

[5] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The design of the borealis stream processing engine," in *In CIDR*, pp. 277–289, 2005.

[6] F. Yang, Z. Qian, X. Chen, I. Beschastnikh, L. Zhuang, L. Zhou, and J. Shen, "Sonora : A platform for continuous mobile-cloud computing," *Technical Report. Microsoft Research Asia, Tech. Rep*, 2012.

[7] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran, "Ibm infosphere streams for scalable, real-time, intelligent transportation services," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 1093–1104, ACM, 2010.

[8] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel : fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[9] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. H. Campbell, "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, 2015.

[10] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink : Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[11] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza : stateful scalable stream processing at linkedin," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.

[12] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "Stream : The stanford data stream management system," Springer, 2004.

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

[14] L. Aniello, R. Baldoni, and L. Querzoni, "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, 2013.

Le manuscrit en anglais

Acknowledgments

I would like to express my sincere gratitude to my supervisors Philippe Lamarre and Nicolas Lumineau. During these last years, they have pushed me, with great benevolence, to develop and exploit my own abilities. They devoted more time than I could expect to teach me the rigor of scientific research. Working with them was one of the most valuable experience for my own progression as a scientist and a person.

I want to thank warmly all members of the BD team. Mohand-Saïd Hacid, Jean-Marc Petit, Emmanuel Coquery and Romuald Thion for their precious advises and support during my Master degree and my thesis. I also thank all other members for welcoming me into this research team. I would also like to thank all members of the Socioplug project, especially Yann Busnel, for their valuable feedback on my works. Moreover, I want to thank Yves Caniou for our enriching discussions and Marc Plantevit for initiating me to research and supervising my first works.

I would like to express my gratitude to Franck Morvan and Bernd Amann for devoting time to review this thesis and make valuable comments about my works. I also want to thank Hala Skaf-Molli and Bruno Defude for their participation to the presentation.

I thank my parents for their constant confidence and support during my studies and to push me to do what I like. I thank my sisters and my brothers-in-law for their help and advises during those years. I address a special thank to my girlfriend, for her immeasurable thoughtfulness and support even when I was working several sunday nights in a row. I thank all my precious friends for their kindness and to make me explain my works regularly. It would have never be possible without all of them around me.

Finally, I thank all members of the University Claude Bernard and of the INSA de Lyon I have met for making those places such great workplaces.

Publications

International conferences (refered full papers)

Roland Kotto Kombi, Nicolas Lumineau, Philippe Lamarre, *"A preventive auto-parallelization approach for elastic stream processing"*, 37th IEEE International Conference on Distributed Computing Systems, Juin 2017, Atlanta

National conferences (refered full papers)

Roland Kotto Kombi, Nicolas Lumineau, Philippe Lamarre, *"Approche préventive pour une gestion élastique du traitement parallèle et distribué de flux de données"*, 17ième Conférence Extraction et Gestion des Connaissances, Janvier 2017, Grenoble

National journal (refered full papers)

Cécile Favre, Chloé Artaud, Clément Duffau, Ophélie Fraisier, Roland Kotto Kombi, *"Forum Jeunes Chercheurs à Inforsid 2016"*, Ingénierie des systèmes d'information Volume 22

Posters

Roland Kotto Kombi, *"Stream and Resource-Aware Elastic Stream Processing"*, Forum Jeunes Chercheurs au 34ième Conférence Informatique des Organisations et Systèmes d'Information et de Décision, Juin 2016, Grenoble

Technical reports

Roland Kotto Kombi, Nicolas Lumineau, Philippe Lamarre, Yves Caniou,*"Parallel and Distributed Stream Processing: Systems Classification and Specific Issues"*, HAL: `https://hal.archives-ouvertes.fr/hal-01215287,Octobre2015`

Contents

List of Figures

1

Introduction

Contents

1 Stream Processing

During the last decade, stream processing has become a very active research domain as presented in [Babcock et al., 2002, Cherniack et al., 2003a, Ishii and Suzumura, 2011, Hummer et al., 2013, Lohrmann et al., 2015, Hochreiner et al., 2015]. These researches are motivated by the growing number of domains using stream-based applications. For instance, in the e-marketing domain, advertising applications analyze in real time logs of users browsing online in order to suggest relevant advertisements. Suggesting promotional offers in real time is more efficient than delayed offers because it relies on current interests and activity of users. In digital entertainment, online servers collects massive amount of data describing actions performed by players through sensors like gamepads or keyboards. It allows game providers to synchronize multiplayer sessions with low latency, which is necessary to offer a satisfying gaming experience. Another use case is geolocalization. Stream-based applications take in input the localization of users at high frequency and return their positions on a synthetic map. In addition, some sponsorized places (restaurants, shops, etc.) are also suggested to users. To guide users and promote some places, it is crucial that applications approximate positions of users and cross them with geographic data as soon as new GPS signals are received.

This proliferation and diversification of sources emitting data in real-time has led to the development of specific techniques for data stream management [Heinze et al., 2014b]. In opposition to persistent data stored on disk, data streams are potentially infinite sequences of transient data emitted by sources as soon as they are produced.

Users can process these data streams by submitting specific queries, denoted *continuous queries*, as introduced in [Babcock et al., 2002, Arasu et al., 2006, Arasu et al., 2004]. Contrary to one-shot queries on persistent data which are submitted each time a new result is expected, continuous queries are submitted once and potentially never terminate. A continuous query generates updates of the result as soon as new data arrive through input streams.

So processing data streams continuously requires to deal with Big Data issues linked to velocity and volume [Babcock et al., 2002, Cherniack et al., 2003a, Abadi et al., 2005]. Indeed, stream-based applications require that new data are processed on the fly to return results before they become obsolete for end users. Moreover, most stream-based applications requires to keep the recent history of data streams in memory to perform aggregate or join operations. Nevertheless, the volume of data sharing a common timestamp may significantly vary and critically increase the resources needed to store and process the recent history.

Initially, some centralized solutions [Schreier et al., 1991] have been suggested to compute continuous queries over database management systems (DBMS). These solutions store both data and queries to compute updates of results at regular interval. Continuous queries are turned into equivalent direct acyclic graphs (DAGs) of operators, denoted *workflows*. An operator is an atomic sequence of instructions applying a predefined, eventually user-defined, function on inputs. So, operators are applied sequentially on new data to generate result updates at regular interval. Nevertheless, it presents several limitations in terms of velocity and volume. First, as data arrive continuously, it requires frequent disk accesses which involve important latency to store and retrieve data. While the frequency of data arrival exceeds the frequency of disk access, it causes an accumulation of data in main memory until overflow. Likewise, when a continuous query contains one or many operators with high processing latency (e.g., cartesian product) applied sequentially, the data input rate may exceed the throughput of the continuous query. Then, as data streams are potentially infinite they cannot be stored fully in memory or disk, so relying on finite memory and disk spaces require to manage data lifecycle to discard obsolete data. It involves additional management overheads having an impact on the performance of the system.

Then, some centralized solutions [Arasu et al., 2004, Abadi et al., 2003, Chandrasekaran et al., 2003] exploiting the parallelism of a multicore architecture have been suggested. They are intrinsically different from DBMS derived solutions because they keep data on main memory for faster accesses and they push new data in a pipeline of operators running in parallel. Moreover, an operator with high latency may be split in equivalent *tasks* running in parallel. It assumes that it is feasible according to operator properties (*e.g.*, algebraic properties). These architectural modifications significantly improve the scalability of systems and define a new class of systems called data streams management systems (DSMS). Nevertheless, at the era of Big Data, volumes of data transferred in real time exceed significantly the resources (CPU, memory and bandwidth) of a single multicore machine. Even if operators can be replicated to scale treatments, the number of threads a machine can handle is limited and it prevents computing complex queries over multiple input streams.

To tackle these issues, some industrial [Peng et al., 2015, Akidau et al., 2013, Gedik et al., 2008, Biem et al., 2010], academic [Abadi et al., 2005, Chandrasekaran et al., 2003, Gulisano et al., 2012, Balazinska et al., 2004], and open-source [Peng et al., 2015, Noghabi et al., 2017, Carbone et al., 2015] distributed DSMS have been developed. They aim at exploiting the inherent of a distributed infrastructure without requiring specific knowledge to end users. They allow users to define continuous queries over sets of data streams through stream-oriented languages [Arasu et al., 2006, Jain et al., 2008] or API [Peng et al., 2015, Akidau et al., 2013, Yang et al., 2012, Zaharia et al., 2012b] in a programmation language like Java, C or Python. Distributed DSMS use parallelization frameworks as MapReduce [Dean and Ghemawat, 2004] to improve the scalability of stream-based applications. So, contrary to centralized DSMS [Arasu et al., 2004, Abadi et al., 2003, Backman et al., 2012], distributed DSMS are not bounded in terms of resource by the features

of a single machine but can exploit an extensible cluster. It allows users to process larger volumes of data with low latency. To distribute continuous queries over a cluster of machines, operators are potentially replicated into many tasks. Then, all tasks are assigned for a parallel execution.

In the context of the ANR project Socioplug[1] (ANR-13-INFR-0003), we consider a sets of users, denoted *communities*, interested by results of some continuous queries. Each user has at its disposal a processing unit with limited resources in terms of CPU and memory. The objective of the project Socioplug is to provide a platform allowing each user to compute results over large volumes of data arriving over time. Users should be able to submit multiple continuous queries simultaneously and so, belong to many communities. The problem is that users have their disposal limited resources but also want to get results with low latency independently of stream rates and the number of continuous queries running in parallel. So, an architectural adaptation of data management systems is required to satisfy users requirements. It appears that the exploitation of a distributed infrastructure is the key feature to suggest a reliable stream processing platform as expected in the project Socioplug. However, one objective of this project is to rely on resources owned by users instead of processing continuous queries on a third cluster. To answer this problem, processing units of users belonging to a same community are interconnnected to build a cluster. This way, each community has at its disposal a cluster to compute once results of the continuous query.

As mentioned above, each user may belong to many communities. Thus, for the provider managing the cluster, the general problem is to balance resource usage of each processing unit between all continuous queries it should process. In consequence, it becomes crucial for the provider to fit resource usage of continuous queries to their respective processing requirements. In a stream processing context, it is a major challenge as processing requirements vary as stream rates fluctuate over time. The dynamic adaptation of workflows representing continuous queries, called *elastic stream processing* [Ishii and Suzumura, 2011, Hochreiner et al., 2015, Heinze et al., 2014b], is then necessary to maintain a balance between processing requirements and resource usage at runtime.

2 Problem Statement

On one side, the elastic treatment of continuous queries satisfy user requirements independently of variations happening in the execution environment. On the other side, it allows DSMS to take advantage dynamically of available resources without needing an oversized cluster. Nevertheless, reconfiguring workflows representing queries implies important reconfiguration overheads. Indeed, as operators process continuously transient data stored in main memory, modifying the configuration of some operators requires to pause treatments, apply some transformations on a workflow and spread them over the cluster. Moreover, it may involve code and data transfers through network. All those operations are time and resource consuming so they degrade momentarily the performance of the DSMS.

To take fully advantage of elastic stream processing, a DSMS must identify when a reconfiguration is needed or recommended to improve processing latency and result quality of continuous queries. To ensure that a reconfiguration fits effectively processing requirements to resource usage, it is crucial that the type of reconfiguration corresponds to the problem caused by a variation in the execution environment. Three types of reconfiguration are usually considered to adapt the configuration of a DSMS:

[1] http://socioplug.univ-nantes.fr/index.php/SocioPlug_Project

- Operator scheduling aims at assigning operators such as the global scheduling plan satisfy some conditions. It can be achieved by spreading evenly the processing load over all available machines [Zaharia et al., 2012b] or minimize network traffic to avoid network bottlenecks [Xu et al., 2014]. So, operator scheduling can prevent congestion due to massive network transmissions but it cannot increase the global processing rate of an operator. If computations are bounded by the time complexity of the operator, reassigning it on another machine is inefficient.

- Load balancing should serve as guarantee that the load is evenly distributed over tasks associated to a given operator [Rivetti et al., 2015]. If it is not the case, it significantly reduces benefits brought by parallel execution. To improve the global processing rate, a load balancing strategy assumes there is a natural imbalance in data that a specific routing plan may compensate.

- Parallelization of operators [Lohrmann et al., 2015, Schneider et al., 2009, Gedik et al., 2014, Shukla and Simmhan, 2017] consists in increasing (scale-out) or decreasing (scale-in) the number of tasks, also denoted *parallelism degree*, of an operator. It has a direct impact on the processing rate of an operator and bounded theoretically the input rate an operator is able to handle without going to congestion. So, adapting the number of tasks associated to an operator defines mainly the input rate an operator can process.

Most distributed DSMS integrate parallelization mechanism, scheduling and load balancing strategies. All reconfiguration types can be triggered offline [Aniello et al., 2013, Zaharia et al., 2012b, Neumeyer et al., 2010] to define a static configuration which is not modified unless they are triggered manually. Some DSMS [Xu et al., 2014, Gedik et al., 2014, Lei and Rundensteiner, 2014] can also adapt execution of workflows at runtime according to a continuous monitoring of the execution environment. Nevertheless, to our knowledge, most elastic solutions focus on one reconfiguration type and neglect the synergy between the different mechanisms adapting the execution of workflows at runtime.

From these observations, we look after a solution which integrates both dynamic mechanisms for parallelization, scheduling and load balancing of operators in a distributed stream processing context. Many DSMS integrate near optimal scheduling [Peng et al., 2015, Aniello et al., 2013] and load balancing strategies [Rivetti et al., 2015, Lei and Rundensteiner, 2014] but the automatic parallelization, or *auto-parallelization*, of operators raise several challenges: (i) when a modification of parallelism degree should be triggered? (ii) which metrics should be considered to adjust the parallelism degree? (iii) how to avoid massive reconfiguration overheads system but prevent congestion of operators? (iv) how to avoid reconfigurations which do not adapt effectively resource usage?

Thus, we focus on auto-parallelization of operators to perform elastic stream processing. The first challenge (i) is linked to the reactivity of the system. Actually, as presented above, the global rate in input of an operator may vary at any time. So, if the system reacts with an important delay to a significant increase in input rate, it may only correct an effective congestion and let result quality degrades for a certain period of time. Nevertheless, if the system performs a scale-out or a scale-in each time the input varies significantly, it may reconfigure itself continuously, bringing important overheads and causing a major degradation of performance and result quality.

The second aspect (ii) deals with the accuracy of each reconfiguration. While considering recent fluctuations in input rate, which value or aggregated value should be considered as the expected input rate? Several approaches have been proposed to forecast values in near future under some assumptions on the evolution of stream rate.

Then, the stability of the system (iii) can be evaluated at the scope of each task (*i.e.*, controlling the frequency of reconfiguration) but it can also be evaluated at the scope of the entire continuous query. Indeed, limiting the occurrence of scale-in/out for each operator does not guarantee that the system does not reconfigure operators belonging to the same continuous query with a domino effect. For instance, if a workflow performing a join between two input streams receives larger volumes in input, it is beneficial for the system to reconfigure simultaneously all downstream operators as the system can anticipate that the join operator will generate large volumes of outputs.

Finally, in a distributed multicore context, some aspects like concurrency for resource usage may create a gap between expected and effective resource usage(iv). Catching such restrictions imposed by the execution support is necessary to adapt accurately resource usage for each continuous query.

3 Contributions and Organization

In a first time, we analyze which aspects have an impact on distributed stream processing. We suggest an abstract architecture for elastic stream processing. This architecture highlight the different levels of query execution from logical layer to physical layer. For each layer, we detail its role, how a reconfiguration needed is detected and which impact it has on global execution of continuous queries. We also give an overview of common adaptation strategies used at each level of query execution. In addition, we expose dependencies between adaptations and discuss about the compatibility of different adaptation strategies.

From this generic architecture, we highlight lacks in elasticity while facing critical fluctuations in input rate and distribution in terms of reactivity and automaticity. To tackle this issue, we suggest an original auto-parallelization strategy which allow to perform *preventive* elastic stream processing. Indeed, our auto-parallelization analyzes both stream and operator properties like the selectivity factor to estimate accurately processing requirements at runtime. Compared to other solutions suggested in the literature, our solution requires neither a learning phase nor user expertise or intervention to size available resources to treatments.

The contributions presented in this manuscript are the following:

- A survey on congestion management presenting common techniques used to enable elastic stream processing over a distributed infrastructure. In addition, we suggest a classification of detection methods for congestion management extending the classification of auto-scaling methods presented in [Lorido-Botran et al., 2014].

- The review of a selection of DSMS covering the variety of solutions existing in the literature. For each DSMS, we present its specific features in terms of query definition and congestion management.

- The abstract architecture for elastic stream processing, called ORACL loop, which identify the different steps of dynamic reconfiguration for congestion management.

- The original auto-parallelization AUTOSCALE which relies on a monitoring module observes the recent history of operator activity at regular interval of time. Through this module, AUTOSCALE can analyze the behavior of the operator at local scope and compute an activity metric. This metric estimates the gap between input and processing rates at operator scope

and in near future. It allows to adjust the parallelism degree of operators in a proactive way and prevent congestion.

- An algorithm checking the consistency at workflow scope of reconfigurations detected at operator scope and integrated to AUTOSCALE. This algorithm takes as input local reconfiguration requirements and the structure of the workflow and computes the set of reconfigurations to trigger simultaneously in order to improve the performance and the stability of system. It aims at avoiding reconfigurations with domino effect and antagonist modifications of parallelism degrees. We highlight the effectiveness of this algorithm through an experimental study against different strategies.

Then, we extend this auto-parallelization strategy to the approach AUTOSCALE+ with the support of concurrency and load imbalance between tasks. To do so, the following improvements have been integrated:

- The consideration of effective resource usage for each task of operator. We focus on CPU usage as it is the most crucial resource for computation in a stream processing context.
- A resource-aware variation of the activity metric which takes effective resource usage into account to define the gap between input and processing rate.

We integrate AUTOSCALE+ within a solution, named DABS, which adapts resource usage to processing needs. The solution has the following properties:

- The auto-parallelization strategy AUTOSCALE+ associated to a complementary load balancing strategy. The association of compatible approaches for auto-parallelization of operators and load balancing ensures that each operator has the capacity to process input streams independently of data volume and distribution.
- Configurable parameters allowing users to adapt treatments to their needs.

We implemented these auto-parallelization strategies over the DSMS Apache Storm[2] and suggest a comparative evaluation with the native behavior of this solution.

In this research, we focus our efforts on preventive auto-parallelization of operators with consideration for the trigger of scale-in/out, the consistency of reconfiguration and the stability of the system. All aspects linked to data partitioning and state management [Shukla and Simmhan, 2017, Castro Fernandez et al., 2013, Gedik, 2014, Wu and Tan, 2015, Ding et al., 2015, Nasir, 2016, Cardellini et al., 2016] are out of the scope of this research.

Organization of the manuscript

In chapter 2, we introduce the background about data stream representation and stream processing. In chapter 3, we suggest a survey of elastic stream processing oriented on congestion management. It is composed of a catalog of techniques for congestion detection and management and also a classification of DSMS illustrated by representative academic, industrial and open-source solutions. The generic architecture ORACL for elastic stream processing is detailed in chapter 4. Then, in chapter 5, we present the auto-parallelization strategy AUTOSCALE and explain how it adapts dynamically and automatically parallelism degrees of operators. We expose an experimental study of optimizations integrated in the algorithm. The extension, called

[2] http://storm.apache.org/

AUTOSCALE+, is detailed in chapter 6 and we test its compatibility with a load balancing strategy of the literature. Experiments on microbenchmark and complex topologies are commented in chapter 7. Finally, we summarize conclusions of our works and present future works in chapter 8.

2

Preliminaries

Contents

In this chapter, we briefly remind basic concepts about data streams and data stream processing. We present the execution context that we consider in the remainder of this work. In addition, we give an overview of main methods used to detect congestion in a stream processing context. Finally, we expose issues linked to operator congestion and concepts related to system elasticity.

1 Data Streams

1.1 Modeling and features

Intuitively, data streams are sequences of *stream element* sets [Arasu et al., 2006, Tucker et al., 2003] arriving continuously over time. More formally, a stream is defined as follow:

Definition 1. (Stream) [Arasu et al., 2006] *Let consider a stream S described by a schema and an ordered timestamp set τ. A stream is a potentially infinite multiset of elements $< s_i , \tau_i >$ where s_i is a tuple of the stream respecting the schema of S and $\tau_i \in \tau$ the associated timestamp.*

A timestamp τ_i belongs to a *timespace* τ which defines the chronological order over stream elements.

Definition 2. (Timespace) [Petit et al., 2010] *A timespace τ is an isomorphism of $\mathbb{R}$ naturally and totally ordered. Any timestamp τ_i belongs to τ.*

It is worth noting that a timestamp τ_i may be assigned explicitly by the source which emit the stream element or implicitly by the processing system upon arrival according to wall-clock time. For example, let consider a stream consumed by a heatwave monitoring application. Stream elements belonging to this stream are described by attributes corresponding to climatic

properties like temperature, humidity and CO_2 concentration. According to such schema, a stream element can be $<$ [23, 60.0, 320], 1508154964$>$ where the first value is the temperature in celsius degrees, the second the humidity in percentage and the third the CO_2 concentration in parts-per million. The timestamp is the time elapsed since an initial timestamp (*e.g.*, January, 1st 1970) in seconds.

Data streams are not only collections of timestamped data, they are emitted as soon as they are produced and this ephemeral nature raises several issues while processing streams:

- Most data stream sources cannot delay emission of stream elements or store the recent history of emitted elements so any loss of stream elements due to computation or network failure is definitive. So, processing systems must remain available to receive new data at anytime.
- The number of stream elements arriving at each timestamp is finite but potentially huge implying that processing systems must be able to receive large stream element sets.
- Data streams may be described by simple attributes (*e.g.*, numerical attributes for meteorological sensors) but also complex attributes (*e.g.*, large matrices for high definition video streams). The variety of data a system is able to handle defines the perimeter of applications it is able to support.

1.2 Classification of data streams

After this formalization of data streams and the presentation of their inherent properties, we suggest a classification of data streams and illustrate each category with some typical examples. It highlights specific features of streams to take into account while processing each category of streams.

1.2.1 Criteria of classification

As illustrated on Figure 1, streams can be described according to the evolution of two properties over time: rate and distribution of values.

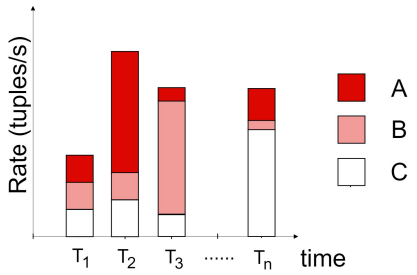


Figure 1: A stream with variations in input rate and distribution of values

Stream rate

Stream rate refers to the quantitative aspect of streams where we only consider the number of stream elements arriving at each timestamp. Considering an infinite stream, the distribution of values defines the relative occurrence of each possible value.

We distinguish four types of streams: steady, bounded, per-pattern and erratic streams.

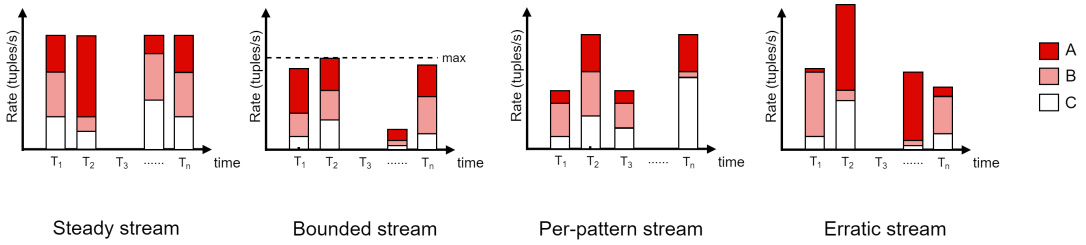


Figure 2: Stream types according to rate

Steady streams are emitted by sources sending stream elements of predefined size and at regular time interval. Typically, it corresponds to measurements generated by sensors or monitoring systems. There are several application domains manipulating such streams: in a context of home automation, sensors send at configurable time intervals measurements about the temperature, the humidity and the luminosity. In military logistic context[3], sensors send latitude, longitude and speed of vehicles for GPS tracking and coordination. We also find steady streams in stock analysis context like Nasdaq[4] updating values of stock prices with a frequency up to the minute.

Bounded streams are variable but have absolute bounds in input rate as illustrated on Figure 2. For example, let consider a fixed number of motion sensors connected to a single wireless receptor and generating a stream element each time they detect a movement. This receptor sends the union of all sensor streams to an application analyzing this merged stream. Strictly in term of input rate, the maximal workload the application has to process corresponds to all sensors sending a stream element at the same timestamp. As the number of sensors is fixed, the input rate cannot be greater than this case. At the opposite, if no sensor detects a movement the stream is empty.

Per-pattern streams fluctuate according to patterns more or less long over time. According to the complete history of stream variations on a significant time period, fluctuations in near future are predictable with high probability. In a real-world context, many domains use per-pattern streams. For example, streams generated by road traffic monitoring fluctuate periodically in terms of volume at different granularities. At day granularity, the volume is important around working hours and low during them. At year granularity, the volume increases significantly around summer holidays. Another example of per-pattern streams are click logs generated by websites especially online stores. Actually, at year granularity, users visit massively such websites around holiday season compared to the rest of year.

As shown on Figure 2, rate of *erratic streams* at a given timestamp are independent of previous variations making those streams unpredictable. An example of erratic stream is online auctions supported by platform such as eBay where new items can be added for auction at anytime and bids arrive depending on the popularity of items, their prices and several other parameters.

[3] http://infolab.stanford.edu/stream/sqr/
[4] http://www.nasdaq.com/quotes/real-time.aspx

Stream distribution

As mentioned above, for all these types of stream, the distribution of values may change over time too but contrary to stream rate, the impact of this property on computations depends completely of the consuming application. Streams can be grouped in two categories independently of the four types presented above: *even* and *uneven* streams.

If an application has a fixed overall latency no matter which value is passed in input, the stream is said *even*. According to a given application, the distribution of values of an even stream could be ignored as it has no impact on computation. A stream may be even for several reasons:

- Some values with similar impacts on processing latency are extremely frequent (i.e., the sum of their relative occurrence is close to 1).
- All computations performed within a given application are independent of values in input (*e.g*, projection of attributes).
- Computations which have a time complexity depending on the value in input have a marginal impact on the overall latency of the considered application.

Contrary to even streams, variations in data distribution of *uneven* streams may have a significant impact on processing latency. It is due to an application containing treatments with time complexity depending of input values and consuming streams with varying distribution of values over time. It may cause important variations of processing latency for a given application and with fixed resources.

To sum up, streams are intrinsically different from static data. They are potentially infinite and arrive continuously over time. Moreover, they may or may not significantly fluctuate in terms of volume and value distribution over time as shown on Figure 2. Nevertheless, if fluctuations in stream rate have a direct impact on computations, fluctuations in distribution of values have only an impact under assumptions mentioned above. In all cases, processing streams raises several issues due to their ephemeral nature and their variety in a Big data context.

2 Data Stream Processing

To tackle issues linked to velocity and volume of data streams, some solutions based on traditional *Database Management Systems* (DBMS) [Schreier et al., 1991, Rosenthal et al., 1989] have been suggested but they show some serious limitations [Abadi et al., 2003, Stonebraker et al., 2005] essentially due to data availability. Indeed, processing data streams involves two main features: on-the-fly computation of ephemeral data and an *active* processing model to submit query. So, it appears that DBMS cannot satisfy high velocity requirement [Stonebraker et al., 2005] because of the important latency due to I/O accesses on disk. Thus, reversing active and passive protagonists is necessary to compute data on-the-fly. This processing paradigm allow to keep ephemeral data in movement without requiring a costly storage. Moreover, it favors availability and velocity of systems [Stonebraker et al., 2005] as data remain in memory for all computations.

DBMS suggest a *human active database passive* (HADP) paradigm [Schreier et al., 1991, Abadi et al., 2003] for data processing. Each time a user wants an updated result, he has to submit a query over static data stored on disk. It means that every time there is an update on data, the user must submit again his query to update the result. While processing streams, a user wants to submit a query once and to receive result updates as soon as the system receives

new stream elements. So, it corresponds to a paradigm *database active human passive* (DAHP). To turn from HADP to DAHP paradigms, queries must support operators turning substreams into finite relations. That means dividing the potentially infinite sequence of stream elements into finite subsequences which can be processed within a finite time.

2.1 Continuous queries

To enable DAHP paradigm, a new class of queries, called *continuous queries* [Babu and Widom, 2001], appeared.

Definition 3. (Continuous query) *A continuous query is an endless and deterministic application taking as input a set of streams and generating one or many output streams. A result may be generated as soon as a new stream element arrives or at regular interval.*

Contrary to queries on static data, continuous queries may never terminate as input streams are potentially infinite. They can be applied on a single stream element (*e.g.*, filtering or transformation) or on sets of stream elements (*e.g.*, aggregation). Nevertheless to perform an aggregation, it requires theoretically to store the entire stream on memory but it is impossible as streams may be infinite. So a common trade-off between completeness and storage space consists in considering only the recent history of data streams to perform aggregation.

2.1.1 Computation windows

As data are produced and arrive in real time, recent data have a greater impact in most application domains. For instance, geolocalization applications rely on GPS signals received within the last seconds to approximate the position and the direction of the end user. Likewise, most stream-based applications generate results which loose quickly interest for end users over time. Thus, using recent history is a common approximation while processing data streams. It ensures that results are relevant for users and reduce the volume of data to consider at each computation.

Window semantic

Recent history, or *window* [Babcock et al., 2002], defines the finite portion of the stream to consider.

Definition 4. (Computation Window) *A computation window is a logic stream discretization [Zaharia et al., 2012b] which is defined by a size and a slide (see Figure 3). Considering the front of a window and the chronological order, the size defines the subset of elements to consider. The slide defines the step between two consecutive window fronts.*

When the slide is smaller than the size, the window is called a *sliding window* [Arasu et al., 2006, Arasu et al., 2004, Golab et al., 2004]. Sliding windows are characterized by an overlap between one or many consecutive iterations. Several works [Golab et al., 2004, Kang et al., 2003, Qiao et al., 2003] exploit this property to optimize query execution over data streams. When the slide equals the size, the window is called *tumbling*. In this case, the intersection between two consecutive iterations is empty.

A window is an operator turning a finite subset of stream elements into an instantaneous, or temporal [Petit et al., 2010], relation $R(\tau_i)$ [Arasu et al., 2006]. As discussed in section 1.1 and considering stream elements timestamped explicitly or implictly, the problem is to define which stream elements are relevant for computation. Indeed, there are two ways to define window size and step: *time-based* and *count-based* windows [Golab et al., 2004, Kang et al., 2003, Qiao et al., 2003].

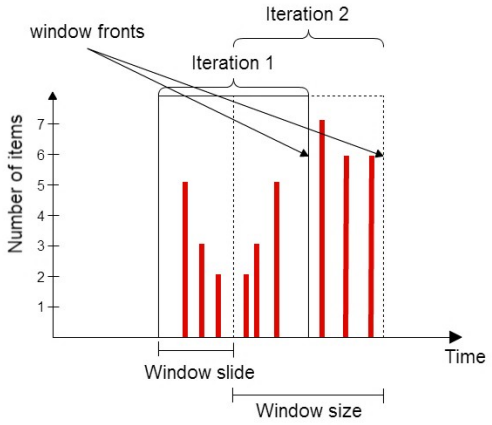


Figure 3: Two iterations of a computation window

Time-based windows

Considering the current timestamp τ_{now}, a time-based window [Arasu et al., 2006, Golab et al., 2004] defines as relevant all stream elements associated to a timestamp τ_i with $\tau_i \in]\tau_{now} - \Delta; \tau_{now}]$. The duration Δ is the *size* of the window. Formally, it produces an instantaneous relation $R(\tau_{now})$ from a stream S defined in formula (1) (see [Arasu et al., 2006]):

$$R(\tau_{now}) = \{s | \langle s, \tau_i \rangle \in S \wedge (\tau_i \leq \tau_{now}) \wedge (\tau_i \geq max\{\tau_{now} - \Delta, 0\}\} \quad (1)$$

From this definition, it appears that a time-based window contains a finite but potentially variable number of stream elements.

Count-based windows

A count-based window, or tuple-based window [Arasu et al., 2006, Arasu et al., 2004], is defined by a size N defining how many stream elements are relevant for computation. The slide defines the number of new elements to receive before computing a new result on the updated window content. As zero, one or many stream elements may be associated to a timestamp, there is no guarantee about acquisition time. Contrary to time-based, count-based windows are not deterministic. Indeed, stream elements associated to a same timestamp cannot be ordered explicitly so considering last N stream elements may not produce a similar result depending on stream sources.

A variation of count-based windows are partition-based windows [Arasu et al., 2006]. A partition-based window is defined by a size N and a finite set of attribute/value pairs defining each partition. A partition-based window is complete when there are N stream elements satisfying each partition.

Window implementation

There are two main mechanisms to implement windows:

- *Punctuations* [Tucker et al., 2003] to consider stream as the union of finite substreams. They are specific elements in a stream which define the end of a substream. This mechanism is particularly useful because it allows elements to be processed on the fly [Backman et al., 2012]

without meeting issues with blocking operators. Nevertheless, punctuations rely on a *a priori* knowledge on data. To insert punctuation correctly in a stream, it is required to know when the last element of the last timestamp belonging to a given window is arrived in the system.

- *Buffer-based stream discretization* [Gedik et al., 2008] turns a stream fragment into an in-memory relation. Buffers gather stream elements according to window definition. For time-based windows, timestamps are used to determine when a buffer is full and ready to be processed.

To sum up, the use of computation windows allows users to specify which data elements to consider. It enables in-memory processing through the reduction of a stream to an instantaneous relation. Depending on operator semantic, an instantaneous relation may be processed as a batch of static data or on-the-fly as new stream elements arrive to the system. Indeed a distinction between *stateless* and *stateful* operators [Zaharia et al., 2012b] is made.

2.1.2 Operator types

Stateless operators

Stateless operators, for example filters based on an attribute value, process data streams element by element. They return a new result with an unpredictable frequency an input may not generate an output depending on its value (e.g., if a filtering predicate is not statisfied). Moreover, these operators do not have information about previous and current computation windows when a data stream element is processed. Nevertheless, a stateless operator may used historic data stored on local memory or disk. It allows to compute joins with a static dataset within a stateless operator [Abadi et al., 2005, Abadi et al., 2003, Cherniack et al., 2003b, Balazinska et al., 2004].

Stateful operators

In opposition, stateful operators take as input a set of elements grouped on a window to compute a single result. When stateless operators can be applied on-the-fly on each new stream element, stateful operators can only return a result after the completion of a window iteration. In addition, these operators keep information like the identifier of the current window or any intermediate result. Information generated during a stateful operator runtime is denoted its *state* [Zaharia et al., 2012b]. For example, a window-based stateful operator computes the sum of values associated to an attribute. Its state contains the identifier of the current window, the attribute to group and the current sum value.

Thus, stateless and stateful operators are defined with different parameters. When stateless operators require only a function to apply on each input, a stateful operator relies on a computation window to complete its definition. This semantic gap has to be taken into account by the query language.

2.1.3 Query languages

To define continuous queries, we distinguish three main categories of query languages: declarative, imperative and graphical languages.

Declarative definition

Declarative approaches [Arasu et al., 2006, Abadi et al., 2005, Chandrasekaran et al., 2003] suggest algebraic extensions of traditional declarative query languages like SQL to support stream management. As any declarative language, they allow users to use a predefined set of operators to describe expected results. The system takes charge of turning the declarative expression into an ordered set of operators.

CQL [Arasu et al., 2006, Arasu et al., 2004] is declarative language for continuous query definition derived from SQL-99 and supporting both traditional relation and data streams. There are three classes of operators in CQL:

- *Stream-To-Relation* operators turn a data stream into an instantaneous relation. It corresponds to window declaration through a clause RANGE taking as parameter window size. It is worth noting that window slide is not explicitly define in CQL queries and is considered as a global parameter of the system. CQL supports timed-based, count-based and partition-based windows.

- *Relation-To-Relation* operators take as input a set of instantaneous relation and produce an instantaneous relation. It correspond to standard SQL operators (Filter, Projection, Join...) and can be applied on instantaneous relations like SQL operators on static data.

- *Relation-To-Stream* operators produce a stream from a set of instantaneous relations. There are three Relation-To-Stream operators: *IStream* returns only updates between the current and the previous iteration of the window. *DStream* returns results produced during the previous iteration of the window and which not belong to results of the current iteration. Finally, *RStream* returns all results.

There are several declarative languages to process data streams. In most cases, they include a subset of SQL relation-to-relation operators and provide specific stream-to-relation and relation-to-stream operators.

SQLStream[5] is an extension of SQL which support complex window definition through a clause WINDOW. Compared to CQL, SQLStream has a greater expressiveness as time-based and count-based windows are defined with explicit window size and slide. It allow to fix the offset between the reception of the last stream element included in the iteration of the window and the arrival of updated results to end users.

SparkSQL[6] offers a support of common SQL operators over *Apache Spark*[7]. This language restricts SQL expressiveness and is not especially dedicated to stream processing even if it supports the WINDOW clause for aggregation.

There are several declarative languages for stream processing built around SQL. If they induce low efforts for maintenance and reusability due to short definition of continuous queries, some applications are intrinsically difficult to turn into a SQL expression (*e.g.*, matrix multiplication). To increase expressiveness of the definition language, some solutions [Peng et al., 2015, Zaharia et al., 2012b, Neumeyer et al., 2010] support high-level imperative languages.

[5] http://sqlstream.com
[6] https://spark.apache.org/docs/latest/sql-programming-guide.html
[7] https://spark.apache.org/

Imperative definition

Some solutions [Peng et al., 2015, Zaharia et al., 2012b, Neumeyer et al., 2010] support continuous queries defined through an imperative high-level language like Java, Python or C++. From user side, query definition is significantly different. Instead of defining the entire query through a single expression, users have to define the query operator by operator through some operator patterns and the implementation of business logic. These patterns specify the form of input and, optionally, output data and manage data lifecycle automatically [Peng et al., 2015, Neumeyer et al., 2010]. Some patterns replace explicit declaration of *stream-to-relation* operators. *Relation-to-relation* operators are entirely part of business logic implemented by users and *relation-to-stream* operators are managed implicitly by patterns.

If imperative definition requires important development and maintenance efforts, it offers more expressiveness for applications managing complex data (*e.g.*, video stream or motion sensors analysis). At the edge of these two categories of query languages, some approaches [Abadi et al., 2005, Abadi et al., 2003] suggest languages more user-friendly which require less development effort than imperative definition but more flexibility than declarative languages.

Graphical definition

Box and arrow paradigm [Abadi et al., 2003, Cherniack et al., 2003b, Balazinska et al., 2004] represents an application as a direct acyclic graph (DAG) of *boxes* connected by *arrows*. A box corresponds to an operator, stateless or stateful, taking stream elements as input. Arrows indicate how stream elements are routed between boxes. Authors in [Abadi et al., 2003] suggest a query language named *SQuAl* composed of some standard SQL operators like *Join* and *Filter* enriched with stream-oriented operators like *Tumble* and *WSort* [Abadi et al., 2003]. Operators can be connected to define the DAG through a graphical interface. SQuAl has a limited expressiveness compared to CQL because it offers only a subset of standard SQL operators. Moreover, stream-based operators support only treatments over tumbling windows restricting possible applications. The main difference between SQuAl and CQL is that continuous queries cannot benefit from automatic query optimization based on algebraic properties. Performances of box and arrow DAGs depend more on user implementation than an equivalent CQL query. In [Pruett, 2007], authors present the interface Yahoo Pipes allowing users to define graphically continuous queries over RSS streams. Yahoo Pipes suggests mainly filtering operators.

As traditional DBMS, a DSMS has to define an execution plan for each submitted continuous query. It means turning a user definition into a set of connected operators. This operator set is the internal representation of any continuous query for DSMSs. Actually, if users define continuous queries through an imperative language or a graphical interface, the DSMS can use directly queries as defined by users but if queries are defined through a declarative language, the DSMS takes charge of turning the declarative expression into a set of predefined operators.

2.1.4 Execution plans and paradigms

As performed in DBMS, query plans are defined at two levels: logical and physical. Query plans at logical level corresponds to a direct acyclic graph (DAG) of logical operators (*e.g.*, Select-Project-Join for SQL-derived languages). According to user definition, it organizes the sequence of operators to apply on operators. If the query is defined through a declarative language the definition of the query plan is based on algebraic properties to find the sequence of operators having the lowest cost [Garcia-Molina, 2008] (e.g., transferring the lowest number of stream

elements between operators). If the query has been defined through an imperative or a graphical language, the execution plan corresponds directly to user definition.

Then, logical operators are mapped to physical operators [Garcia-Molina, 2008, Graefe, 1993] to define physical query plans. If a continuous query has been defined using a declarative language, predefined implementations of operators are used such as users do not have to implement them. Otherwise, users must implement explicitly the logic of operators through a high level programming language.

So, each continuous query is equivalent to an execution plan within a DSMS. However, all query plans do not share a same structure. We distinguish two structures of query plans: *workflows* and *MapReduce jobs* [Dean and Ghemawat, 2004]. These structures impact applicable forms of parallelism between operators. Moreover, they affect data representation and lifecycle.

Workflow paradigm

Workflows are most generic structures that we will use by default to represent query plans.

Definition 5. (Workflow) *A workflow is a DAG where vertices are operators and edges define data transmission between operators.*

Let consider a workflow W_1 composed of some stateless operators. As operators process each stream element as soon as it arrives in input, the workflow exploits naturally stream pipelining [Sattler and Beier, 2013], also denoted task parallelism [Hirzel et al., 2014].

Definition 6. (Stream pipelining) *Let W be a workflow which can be divided into k consecutive operators. Each $\mathcal{O}_i$, $i \in [1;k]$, is denoted the i-th stage of W and is executed on an exclusive process.*

According to Definition 6, stream elements are routed through all stages sequentially. Of course, stream pipelining is limited if a stateful operator belongs to the workflow. Indeed, as stateful operators group stream elements by windows before computing a result, they limit the benefit of parallel treatments. However, stateful and stateless operators may benefit from data parallelism [Hirzel et al., 2014], or stream partitioning, depending on their respective semantics.

Definition 7. (Data parallelism) *Let $\mathcal{T}_i^1$, $\mathcal{T}_i^2$,..., $\mathcal{T}_i^k$ be k equivalent tasks. We suppose that all tasks $\mathcal{T}_i^j$ take as input, outputs produced by an operator $\mathcal{O}_0$. In order to process $\mathcal{T}_i^1$, $\mathcal{T}_i^2$,..., $\mathcal{T}_i^k$ in parallel, $\mathcal{O}_0$ can split its outputs in k partitions and distribute a partition to each tasks $\mathcal{T}_i^j$.*

According to Definition 7, data parallelism can be exploited efficiently only under the assumption that tasks $\mathcal{T}_i^1$, $\mathcal{T}_i^2$,..., $\mathcal{T}_i^k$ are semantically equivalent. Moreover, the global result of operators $\mathcal{T}_i^1$, $\mathcal{T}_i^2$,..., $\mathcal{T}_i^k$ must be independent of the partitioning method.

MapReduce paradigm

MapReduce [Dean and Ghemawat, 2004] is a well-known framework developed initially to process huge amount of disk-based data on large clusters. The strength of this framework is to exploit efficiently data parallelism (see Definition 7) with a simple programming paradigm. Actually, the core of any MapReduce application relies on two functions: Map and Reduce. These generic functions are defined as follow according to [Dean and Ghemawat, 2004]:

- **Map** (k1, v1) $\longrightarrow$ list(k2, v2)
- **Reduce** (k2, list(v2)) $\longrightarrow$ list(v3)

As mentioned above, MapReduce framework aims disk-based data processing. Contrary to DBMS, MapReduce-based systems do not rely on data model to optimize treatments. In order to distribute great amount of data on a large cluster, data are partitioned with regards to cluster configuration (*e.g.* number of nodes executing Map and Reduce functions). Each partition is identified with a key used to affect the partition to a Map node. The scheduling between partitions and Map nodes follows distribution strategies like *Round-Robin* in order to balance computation load.

Each Map node applies the user-defined Map function on one or many partitions. The function produces a list of intermediate key/value list pairs depending on partition contents. Then, outputs of Map nodes are shuffled and sorted in order to perform Reduce phase more easily.

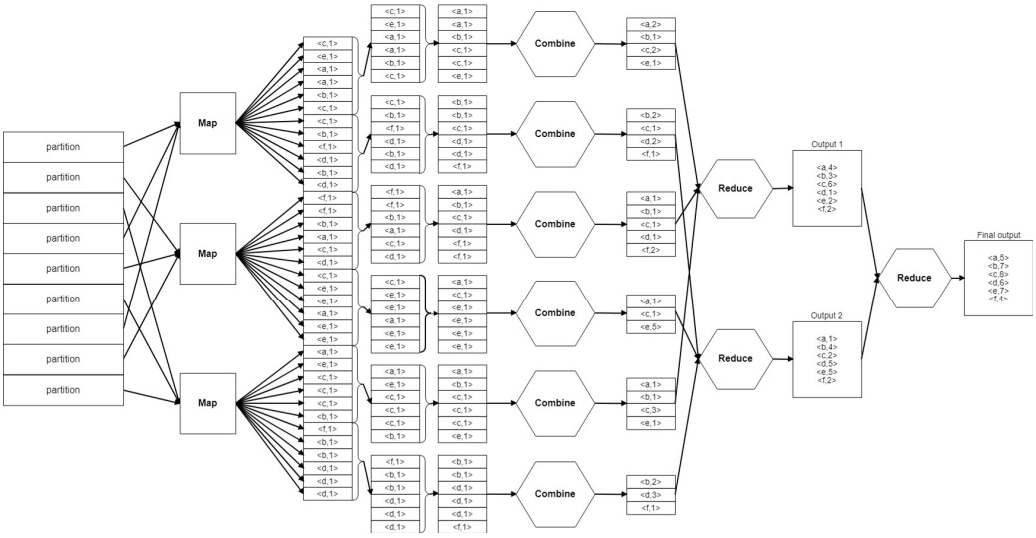


Figure 4: A MapReduce job

As illustrated on Figure 4, an optional phase, called *Combine*, can be performed on each Map node. This phase consists in applying the Reduce function on Map outputs in order to have results for each partition. It may be useful while having potentially several redundant computation like presented in [Backman et al., 2012]. Each Reduce node gathers intermediate key/value list pairs and computes a list of value which are final results.

To sum up, we have introduced a formal representation of data streams. They differ from static data in the way that they are potentially infinite multiset of elements arriving to the system with fluctuating rates. From this, traditional DBMSs cannot process these data streams with low latency and a finite amount of memory. To tackle this issue, some DSMSs have been developed and use computation windows to query data streams without restricting the expressiveness of query languages. Even if DSMSs allow processing streams without storing huge amount of data,

they need computation resources to process data on the fly. As data streams may have high input rates, it is necessary that DSMSs rely on scalable infrastructures. To do so, many DSMSs have been designed such as they can exploit efficiently a distributed infrastructure.

2.2 Distributed Stream Processing

To meet processing requirements involved by stream processing, DSMSs must take advantage of a distributed infrastructure [Stonebraker et al., 2005] able to evolve as streams do. Some processing units can be added to a cluster statically or dynamically to extend the overall processing capacity. In order to facilitate the reading, we refer to execution plans (see section 2.1.4) as workflows without distinction between workflows or MapReduce jobs in the remainder of this chapter.

2.2.1 Execution context

We present here the execution context of distributed stream processing. We consider users interested by services generating streams. To query streams, users can submit continuous queries on a distributed infrastructure[8].

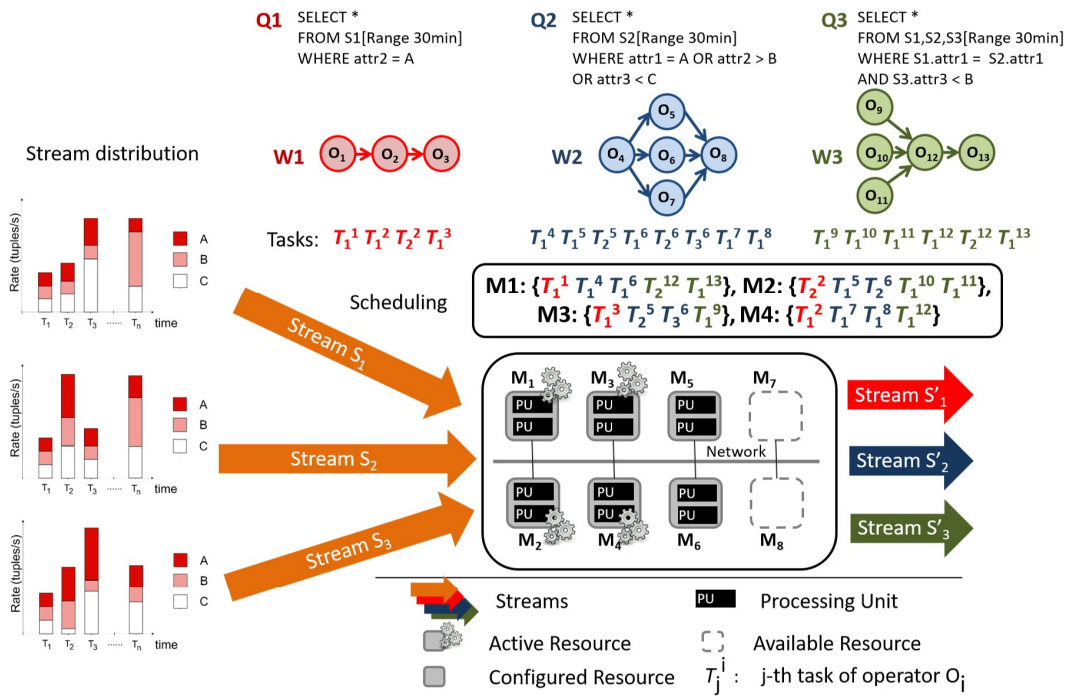


Figure 5: Distributed stream processing

Let us consider three continuous queries Q1, Q2 and Q3 represented respectively by workflows W1, W2 and W3. Q1, Q2 and Q3 are submitted on the distributed infrastructure by some

[8]Execution context motivated by ANR project Socioplug (ANR-13-INFR-0003), http://socioplug.univ-nantes.fr/index.php/SocioPlug_Project

users as illustrated on Figure 5. Each continuous query takes as input one or many streams in the stream set $\{S_1, S_2, S_3\}$. An input stream may have fluctuations in input rate and value distribution as shown on the left of Figure 5. Workflow W1 is linear, W2 is organized in diamond and W3 is a star workflow. Most complex workflows can be decomposed in such elementary patterns [Peng et al., 2015].

As illustrated on Figure 5, tasks T_1^2 and T_2^2 are devoted to operator $\mathcal{O}_2$ so it means that $\mathcal{O}_2$ has a parallelism degree of two. Tasks are assigned on processing units of machines M_1 to M_8 according to a *scheduling plan*. It corresponds to a mapping from tasks to processing units setting resources available for each operator composing a workflow. On Figure 5, the four tasks of workflow W1 are assigned on processing units of machines M_1 to M_4.

We distinguish three states of machines: machines M_1 to M_4 are actives because some tasks are assigned on their processing units. Machines M_5 and M_6 are configured but inactive because there is no task assignment on their processing units. Finally, machines M_7 and M_8 are available but not configured so no task can be assigned on their processing units.

Result streams S'_1, S'_2 and S'_3 are respectively associated to queries Q_1, Q_2 and Q_3. Each output may result from the treatment of a single stream element or an element set if workflows W1, W2 or W3 include some aggregative operators.

Each task T_i^j applies the operator $\mathcal{O}_i$ and receives stream elements on an input queue, also called *pending queue*, which has a fixed and finite size.

With each continuous query, users provide a policy, denoted *quality-of-service* (QoS), specifying expected performance and result quality. Commonly, performance corresponds to a constraint on overall latency. From this specification, latency constraints can be inferred for each operator [Abadi et al., 2005, Cherniack et al., 2003b] or each processing unit [Balazinska et al., 2004]. Result quality defines acceptable losses on a computing windows in order to discriminate relevant results from irrelevant ones. The objective is to guarantee results satisfying QoS constraints to users. These QoS constraints are considered as fixed for the complete lifetime of each query.

The global processing capacity of a distributed infrastructure is extensible through the add of configured machines. When the all resources of the cluster are exploited, some machines should be added to the configured machine set in order to limit the violation of QoS constraints. Indeed, on a cluster like Amazon EC2[9], starting a new VM may take several minutes before new resources are available. Thus, it is important that DSMSs integrate some mechanisms to maintain the global capacity of the cluster over processing requirements. Moreover, as streams may decrease significantly in terms of volume (*e.g.*, a road monitoring stream when the traffic is low), some configured machines may be idle for a significant period. With the generalization of *pay-as-you-consume* solutions (Amazon EC2, Microsoft Azure, Google Cloud Dataflow) and the emergence of *GreenIT* [Murugesan and Gangadharan, 2012], it represents an important waste of financial and energetic resources. So, it appears necessary that DSMSs fit resource usage to processing requirements.

2.2.2 Congestion issue

According to the execution context presented above, tasks are assigned on processing units. Each task processes a substream which has a varying input rate. In some cases, this input rate may causes the *congestion* [Heinze et al., 2014b, Peng et al., 2015, Schneider et al., 2009, Heinze et al., 2014a, Xu and Peng, 2016] of the task and by consequence the congestion of the workflow.

[9] https://aws.amazon.com/fr/ec2/

Definition 8. (Congestion) *The congestion is an execution state defined by irreversible losses of stream elements due to an overload in input of a task, an operator or a workflow.*

According to Definition 8, some query results may be lost definitively when a congestion appears. In a stream processing context, managing congestion of operators consists in solving some *Big Data* issues. Contrary to query processing over static data, an overload may happen anytime during treatments and implies an irreversible degradation of result quality. In addition, the apparition of a congestion involves a degradation of the processing latency at workflow scope.

Detecting a congestion can be performed at different scopes (task, operator, workflow). However, in order to identify the exact cause of a congestion, it is necessary to observe the co-evolution of some metrics. We identify two characteristic situations, or critical cases, causing the apparition of congestion:

- Critical case 1 (**CC1**): Considering a task $\mathcal{T}_i^j$ with a fixed processing throughput TP_i^j, if its input rate is greater than its processing throughput, the task starts to accumulate pending elements on its input queue [Stonebraker et al., 2005, Babcock et al., 2002]. It delays execution of stream elements and emission of new results on output implying a degradation of the global processing latency.

- Critical case 2 (**CC2**): Let consider a task $\mathcal{T}_i^1$ sending its outputs to another task $\mathcal{T}_j^1$ through network interface. If the bandwidth linking those two tasks is smaller than the throughput of $\mathcal{T}_i^1$, the bandwidth limits the input rate of $\mathcal{T}_j^1$ and slows down the global processing latency.

Managing congestion is difficult because it depends of several parameters from the complexity of operators to capacities of available processing units. To remove congestion, some adaptations of treatments and resource usage must be performed statically and dynamically to respect QoS constraints. To do so, some techniques adapting treatments and available resources to stream variations have been developed and integrated into DSMSs.

2.2.3 Resource management

Considering a distributed infrastructure managed as a service (IaaS) [Nikolov et al., 2014], there are two methods to adapt configured resources to processing requirements: substituting or adding some processing units. The substitution consists in a replacement of processing units with others having different capacities. For example, on a cloud provider like Amazon EC2, it consists in replacing some virtual machines (VM) with a new set of VMs having greater (*scale-up*) or smaller (*scale-down*) CPU and RAM resources. Adding processing units consists in increasing (*physical scale-out*) or decreasing (*physical scale-in*) the number of processing units. According to the example on Amazon EC2, it consists in adding or deleting VMs from configured machines.

Substitution of processing units

In order to substitute processing units at runtime, some cloud-based solutions have been proposed in [Maurer et al., 2011, Sedaghat et al., 2013]. These solutions are based on *service level agreement* (SLA) of cloud applications to define optimal VM configuration in order to prevent SLA violations. These approaches are rule-based [Lorido-Botran et al., 2014] and when a potential SLA violation is detected according to CPU, memory and bandwith usage, applications are moved to other VMs with more available resources. However, it has been highlighted

that many operating systems do not support migration of running applications without rebooting [Lorido-Botran et al., 2014, Nikolov et al., 2014] which is time and resource consuming. Another approach integrated in VMWare ESXI[10] suggests a scale-up and scale-down approach by shared resources. It consists in overprovisioning VMs and let some CPU, memory and bandwidth resources inactive when they are not necessary to respect SLA. When a potential violation is detected, these resources are used to simulate scale-up without rebooting. The main problem of this approach is that it requires a constant overprovisioning of resources to perform the substitution at runtime. So, most cloud providers (Amazon EC2, Microsoft Azure...) only suggest to add processing units at runtime.

Add of processing units

For distributed and low latency applications, adding processing units appears more natural to adapt dynamically available resources. It is important to notice that approaches supporting the add of processing units [Nikolov et al., 2014, Andrzejak et al., 2010, Zhu and Agrawal, 2012] consider a unique VM configuration in terms of CPU, memory and bandwidth for all processing units. So, scale-up and scale-down cannot be performed at all. In [Andrzejak et al., 2010], authors suggest a probabilistic model taking SLA constraints as input and defining how many VMs are necessary to prevent violation. This approach is limited by reconfiguration costs. Indeed, while modeling the system to predict how many VMs are necessary, applications with short-term bursts imply heavy reconfiguration. Moreover, it does not take into account unexpected system failures and may removes useful resources. In [Zhu and Agrawal, 2012], an approach relying on *control-based theory* is presented. This solution offers *controllable parameters* defining resource requirements and a feedback loop on application execution after physical scale-out and scale-in to guide manually users to optimal settings.

In this chapter, we have presented some formal concepts and definitions for data stream processing. As data streams are ephemeral sequences of data arriving at fluctuating rates, it is necessary to process them on the fly. To scale available resources to data volumes, it is necessary that DSMSs exploit distributed infrastructures. Nevertheless, it is difficult to maintain consistently enough resources to process all incoming data. To tackle this issue, it is possible to substitute or add processing units at runtime. Nevertheless, these adaptations between processing needs and resource usage are costly in terms of reconfiguration time and energy. Instead of adding the global capacity of the cluster, it is possible to adjust resources consumed by each continuous query with regards to their respective processing needs. To do so, it is necessary to modify workflows according to their definitions, the parallelism degree of their operators, the load balance between their tasks and the assignment of their tasks on processing units.

[10] https://www.vmware.com/fr/products/esxi-and-esx.html

3

State of the art

Contents

In this chapter, we suggest a survey of congestion management in DSMSs. This survey is composed of three parts. In a first time, we present a catalog of patterns used in the literature to manage congestion of operators at different scopes. Then, we explain how DSMSs trigger these patterns through an analysis of methods used to detect congestion of operators. Moreover, we highlight advantages and limits of each class of methods. Finally, we suggest an classification of distinctive DSMSs selected for their performance and popularity. This classification is based on query representation, congestion management and the expressiveness of the query language.

1 Congestion management

1.1 Principle

Congestion management is a major challenge while processing continuous queries over streams with fluctuating input rates. As a congestion (see Definition 8 in chapter 2 section 2.2.2) may happen anytime, it is necessary to maintain a balance between processing needs and resource usage for each operator composing a workflow. Some elastic mechanisms have been developed in order to tackle congestion issues. We group some elastic mechanisms, or *patterns*, into elastic

mechanisms at logical layer. They have in common to modify workflows such as they exploit more or less available resources instead of modifying the execution support. It allows to perform adaptation of resource usage at runtime with short reconfiguration times compared to substitution and add of processing units at runtime.

We present some common patterns used by DSMSs to tackle congestion issue at logical layer. We distinguish four types of patterns:

- Patterns at query level rely on algebraic properties to organize efficiently logical operators according to features like the selectivity factor of operators.
- Patterns at operator level modify the the way a logical operator is executed via one or more physical operator without changing the query plan.
- Patterns at implementation level map operators to an implementation runnable on the execution support.
- Patterns at data level adapt the transmission of streams between operators according to execution parameters.

For each of them, we motivate its interest through an example highlighting a specific case. Then, we explicit assumptions made on operators (algebraic properties, definition, potential parallelism) to specify the applicability of patterns. After this, we expose the principle of each pattern. In addition, we present benefits and overheads induced by each pattern. We sum up patterns with main characteristics in Table 1. More patterns and details can be found in [Hirzel et al., 2014].

Level	Pattern	Required parallelism
Workflow	Operator reordering	task parallelism
Operator	Operator parallelization	data parallelism
Operator	Task scheduling	none
Implementation	Algorithm selection	none
Data	Load balancing	data parallelism
Data	Load shedding	none

Table 1: Patterns for congestion management

As presented in Table 1, each pattern apply modifications which change the entire workflow, a single operator, the implementation used for an operator or data lifecycle. It is also worth noting that patterns may require that operators support a type of parallelism (see Definitions 6 and 7 in chapter 2) to be effective.

1.2 Adaptation at workflow level

1.2.1 Operator reordering

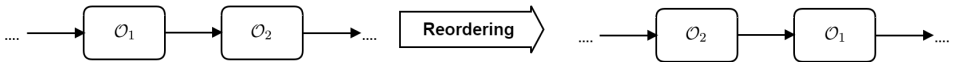


Figure 1: Operator reordering

Motivation

In many continuous queries, operators have different processing cost and produce different amount of data. For example, considering an application detecting heatwaves, temperature and air pollution sensors, at distinct locations send their identifier with average temperature and variation of CO_2 concentration. A first operator $\mathcal{O}_1$ enriches this data with information about sensors like the model and the GPS location. A second operator $\mathcal{O}_2$ filters temperature and CO_2 concentration values and forwards critical measures to an alert system. It appears that $\mathcal{O}_1$ enriches stream elements discarded by $\mathcal{O}_2$ downstream. As $\mathcal{O}_2$ does not filter attributes modified by $\mathcal{O}_1$, some stream elements could be discarded upstream.

Assumptions

Reordering operators can only be performed under some assumptions:

- Before reordering, the upstream operator must be have a fixed selectivity factor. Indeed, the probability that the downstream operator filters a stream element must remain independent from the selectivity factor of the upstream operator.
- Operators must be commutative. Moving an operator $\mathcal{O}_j$ upstream of an operator $\mathcal{O}_i$ requires that executing $\mathcal{O}_j$ before $\mathcal{O}_i$ and executing $\mathcal{O}_i$ before $\mathcal{O}_j$ generates same results. Indeed, let consider the attribute set $\mathcal{A}_i$ composed of attributes added or deleted by an operator $\mathcal{O}_i$ and $\mathcal{A}_j$ the set of attributes used by $\mathcal{O}_j$. If the intersection between $\mathcal{A}_i$ and $\mathcal{A}_j$ is empty, then moving $\mathcal{O}_j$ upstream of $\mathcal{O}_i$ is safe. Indeed, attributes used by $\mathcal{O}_j$ are available before the execution of $\mathcal{O}_i$.

Principle

Definition 9. (Selectivity factor) *The selectivity factor of an operator is the ratio between the number of stream elements in input and output.*

For example, an operator which forwards 25% of received data has a selectivity factor of 0.25. For two consecutive operators, reordering is defined as follow:

Definition 10. (Operator reordering) [Hirzel et al., 2014] *Let $\mathcal{O}_i$ and $\mathcal{O}_j$ be two consecutive operators respectively with a fixed selectivity factor Sel_i independently of value distribution in input and a fluctuating selectivity factor Sel_j. If Sel_j becomes smaller than Sel_i, moving $\mathcal{O}_j$ upstream of $\mathcal{O}_i$ eliminates unnecessary data exchanged between $\mathcal{O}_i$ and $\mathcal{O}_j$.*

Operator reordering is performed statically before the execution of the continuous query or dynamically at runtime. Thus, static reordering of operators is equivalent to logical query plan selection in traditional DBMS [Chaudhuri, 1998]. In this case, selectivity factors of operators are estimated on a subset of representative data and an optimal query plan minimizing amount of exchanged data is defined accordingly. In the other hand, dynamic reordering of operators requires to monitor selectivity factors at runtime and a support of data re-routing [Arasu et al., 2004, Abadi et al., 2005, Chandrasekaran et al., 2003]. In [Chandrasekaran et al., 2003], authors suggest a module, named *Eddy* [Madden et al., 2002], for adaptive routing of stream elements. An *Eddy* receives some streams and routes each stream element to connected operators. Each output produced by operators goes to the *Eddy* before being forwarded to next operator. This approach allows to reorder operators through a light modification of the routing policy of the *Eddy*. Overheads involved by such dynamic approach have been analyzed in [Deshpande, 2004].

Benefits for congestion management

Operator reordering reduces amount of data exchanged between operators. It avoids risk of network bottlenecks due to massive volumes of data to route from a processing unit to another one. In addition, if a query plan contains operators with high selectivity factors and great processing latency (*e.g.*, joins) and operators with low selectivity factors, reordering ensures that costly operators will be moved downstream and receive as less data as possible.

1.3 Adaptation at operator level

1.3.1 Operator parallelization

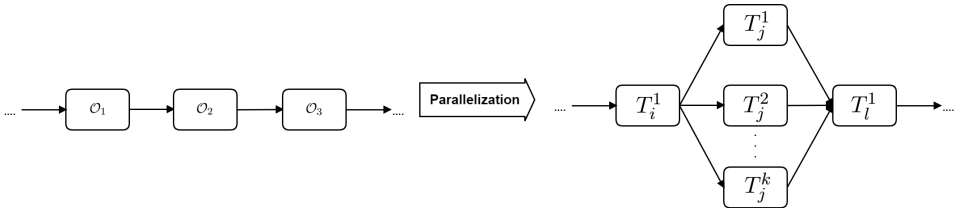


Figure 2: Operator parallelization

Motivation

With the proliferation of multicore machines and the development of distributed computing, many data-intensive applications are executed on infrastructures managing several threads in parallel. Stream-based applications should take advantage of this potential to adapt the throughput of a sequential operator to its input rate. For example, let consider a traffic jam application consuming data from radars. The input rate depends on the number of vehicles observed at each time unit. An operator taking as input registration plates and extracting information about the driver from a remote database is stateless (see chapter 2 section 2.1.2) but has an important processing latency. If the operator receives important volumes in input, it requires more CPU and memory resources to avoid congestion.

Assumptions

Operator parallelization does not require modification of the continuous query only under the following assumptions:

- Operators should be naturally parallel. An operator is naturally parallel if the union of results computed on disjoint data partitions is equal to the result computed on the union of these data partitions. Operator parallelization can be performed on both stateless and stateful operators if they belong to the class of operators naturally parallel [Karp, 1988].

- For stateful operators, it is worth noting that they should keep their states disjoint or synchronized. Parallelizing a stateful operator raises the problem of state management introduced in [Sattler and Beier, 2013, Gedik et al., 2014]. While tasks of stateless operators process stream elements independently from each other, tasks of stateful operators should maintain a global state [Castro Fernandez et al., 2013] or keep independent states on stream partitions. For example, a stock-price analysis application computes average

price values for each stock label. Splitting such stateful operator consists in partitioning stock labels among some tasks so states are completely disjoint.

Principle

Definition 11. (Operator parallelization) *An operator $\mathcal{O}_i$ is parallelization if it is replaced by a set of equivalent tasks $\{T_i^1,...,T_i^k\}$ processing stream elements in parallel. Each task receives in input a partition of $\mathcal{O}_i$ inputs.*

Like reordering, parallelization can be static or dynamic. Static parallelization consists in selecting a number of tasks, or parallelism degree [Mehta and DeWitt, 1995], according to an expected workload. The parallelism degree is set to improve throughput with light overheads. In a stream processing context, the workload is not known before execution so different approaches have been developed. Some approaches [Xu and Peng, 2016, Heinze et al., 2014a] monitors input rates and processing latencies of operators and adapt parallelism degrees accordingly. It suits to parallel operators under the assumption that a greater parallelism degree improves throughput. To eliminate this assumption, some approaches based on *trial-and-error* learning algorithms [Schneider et al., 2009, Gedik et al., 2008, Gedik et al., 2014] aim to map throughput to parallelism degree after an exploration phase.

Benefits for congestion management

Operator parallelization spreads input data of a single operator among many equivalent tasks in order to exploit data parallelism (see Definition 7). It may reduces the input size of each task depending on the number of tasks and the partitioning method. So, it limits overflow in input and improves throughput as long as distribution [Deshpande, 2004] and network [Xu et al., 2014] overheads are compensated.

1.3.2 Task scheduling

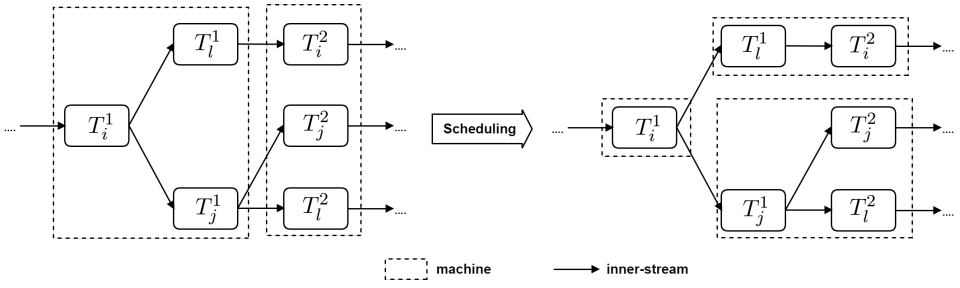


Figure 3: Task scheduling

Motivation

Processing streams on a distributed infrastructure involves communication costs. Indeed, a continuous query is represented as a DAG of operators (see chapter 2 section 2.1) where edges are inner-streams. These inner-streams are supported by the infrastructure through:

- Main memory if tasks are assigned on the same machine and can share common memory space.
- Network interface if tasks are assigned on different machines.

Depending on the volume of inner-streams and available resources (memory and bandwidth), it is better to assign some tasks on the same machine or on distinct machines. For example, let consider an application following the celestial bodies from telescopic images. An operator $\mathcal{O}_1$ turning raw images into matrices sent to an operator $\mathcal{O}_2$ compressing matrices. Then, an operator $\mathcal{O}_3$ receives compressed matrices and analyzes orbits of celestial bodies. As operator $\mathcal{O}_1$ and $\mathcal{O}_2$ manage important volumes of data and reduce significantly data volume through compression, it is interesting to assign tasks associated to these operators on the same machine to avoid massive network communications.

Assumptions

Task scheduling can be performed on a distributed infrastructure assuming the following conditions:

- Processing cost of operators must not exceed available resource on each processing unit as explained in [Aniello et al., 2013, Peng et al., 2015]. To avoid failure of processing units, a scheduling plan should not assign operators requiring more resources than there are on a processing unit. To avoid this situation, scheduling plans can be defined according to provided resource requirements [Peng et al., 2015] or they can be updated through operation re-assignment, or box sliding [Abadi et al., 2003, Cherniack et al., 2003b], according to metrics monitored continuously.
- Each processing unit must have an access to all hardware and software resources needed for computation. Commonly, stream-based applications communicate with remote and disk-based systems (*e.g.*, databases and HDFS file systems). These communications are established within some operators which can be assigned on any processing unit, so it is crucial that all processing units have an access to remote and disk-based systems.
- States of operators must be transferred without loss. It is necessary that operator states can be moved to perform dynamic operator scheduling. To move states, some coordination systems[11] are used. They rely on checkpointing mechanisms to update operator states. So, when an operator is moved and restarted on another processing unit, its state is initialized according to the last checkpoint.

Principle

Definition 12. (Scheduling plan) *Let consider a finite set of tasks* $T = \{T_i^1, T_i^2, ..., T_i^k\}$ *and a finite set of processing units* $\mathcal{M} = \{\mathcal{M}_1, \mathcal{M}_2, ..., \mathcal{M}_q\}$. *A scheduling plan* $\mathcal{SP} : T \rightarrow \mathcal{M}$ *is an application assigning each task* T_i^j *on a processing unit* $\mathcal{M}_n$ *with* $T_i^j \in T$, $\mathcal{M}_n \in \mathcal{M}$.

A scheduling plan describes the global layout [Hirzel et al., 2014] of continuous queries on processing units and is defined statically or dynamically. The static definition relies on operator properties in order to reach an objective [Aniello et al., 2013] (e.g., minimizing network

[11] https://zookeeper.apache.org/

traffic [Xu et al., 2014]). It requires to have algebraic properties on operators and eventually to train the scheduler with representative data to extract selectivity factors. So, the static version is only suitable for solutions based on a query language associated to a predefined algebra. Operator scheduling can be performed dynamically if a monitoring mechanism observes operator properties at runtime to compute updated scheduling plans.

Benefits for congestion management

The benefit of operator scheduling depends on the compromise made between resource usage and communication costs. If two operators assigned on a same processing unit have concurrent usages of some resources (CPU, memory and disk) and they receive critical data volumes in input, it is interesting to move at least one of these operators on another processing unit. The problem is that the performance improvement induced by the re-assignment should compensate communication overheads caused by the migration. On the contrary, if two tasks assigned on different processing units exchange important volume of data but there is a processing unit able to provide available resources without starving one of the tasks, it is preferable to collocate these tasks.

1.4 Adaptation at implementation level

1.4.1 Algorithm selection

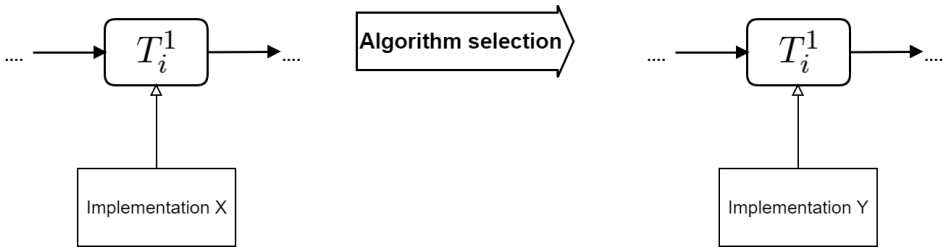


Figure 4: Algorithm selection

Motivation

Let consider an application defined through a declarative language or a graphical interface (see chapter 2 section 2.1). This application computes a join between two streams S_1 and S_2 over sliding windows (see chapter 2 section 2.1.1). As streams have independent fluctuations in input rate, instantaneous relations $R_1(\tau_i)$ and $R_2(\tau_i)$ generated at timestamp τ_i respectively from S_1 and S_2 may have different cardinalities. Let consider that the system supports two implementations of join operation: nested-loop and sort-merge algorithms. If $R_1(\tau_i)$ has a higher cardinality in comparison to $R_2(\tau_i)$, nested-loop implementation delivers good performance. But if $R_1(\tau_i)$ and $R_2(\tau_i)$ have approximately same cardinalities, the sort-merge implementation is more efficient as it scans both relations only once [Mishra and Eich, 1992]. So, the choice of implementation may have an important impact on performance.

Assumptions

Algorithm selection assumes that continuous queries are defined through a declarative lan-

guage. To ensure that algorithm selection does not degrade results or performance of the system, the following assumptions must be valid:

- All implementations of a same operator must be strictly equivalent. Indeed, if there are some specific cases where an implementation $\mathcal{I}_1$ delivers different results than an implementation $\mathcal{I}_2$ supposed equivalent, algorithm selection may affect the semantic of the operator and the continuous query may return wrong results. It is specifically important after operator parallelization because some operators cannot keep their original semantic (*e.g.*, standard deviation).
- Implementations of operators defined through a declarative expression must be available on all processing units. If operator scheduling is performed only statically [Aniello et al., 2013], a site-aware scheduling of operators may ensure that all operators have available implementations where they are assigned. But if operator scheduling is performed dynamically to avoid congestion or recover from a failure, the availability of implementations is not guaranteed anymore. Of course, this issue cannot happen when continuous queries are defined through an imperative language as the implementation is moved with the assignment of the operator on a processing unit.

Principle Algorithm selection is common in database systems. It corresponds to *physical query plan* definition [Garcia-Molina, 2008, Graefe, 1993] mapping operators from *logical algebra* (*e.g.*, sort or join) to the physical algebra (*e.g.*, heapsort, mergesort, hash join or nested loop join).

Definition 13. (Physical query plan) *Let consider the set of tasks* $T = \{T_i^1, T_i^2, ..., T_i^k\}$. *Let* $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_m\}$ *be the set of available implementations. A physical query plan* $\mathcal{QP} : T \rightarrow \mathcal{I}$ *is an injective application which associates an implementation* $\mathcal{I}_j \in \mathcal{I}$ *to each task* $T_i^j \in T$.

According to Definition 13, the cardinality m of implementation set is equal or greater than than the cardinality n of operator set if we do not count equivalent tasks generated after operator parallelization as presented above. Moreover, it is important to note that in the case of operator parallelization, some implementations cannot be used in parallel or involve important overheads to merge results on partitions. Algorithm selection can be performed dynamically to adapt the implementation to input properties. In [Abadi et al., 2005], authors suggests a runtime mechanism having a static set of implementations for each operator and picking the fastest one according to input properties.

Benefits for congestion management

Algorithm selection may increase processing latency of operators if there are a faster implementation than the current one available locally. The speedup may be absolute (*e.g.*, quicksort over bubblesort) or relative (*e.g.*, sort-merge join over nested-loop join) according to input properties.

1.5 Adaptation at data level

1.5.1 Load balancing

Motivation

Let consider an application receiving vehicle speeds observed by radars. An operator $\mathcal{O}_1$ takes as input speeds violating the limit with the associate registration license. $\mathcal{O}_1$ queries a remote database depending on registration license to extract driver information. Each remote database has a specific communication latency. If the load in input of $\mathcal{O}_1$ becomes critical, it can be

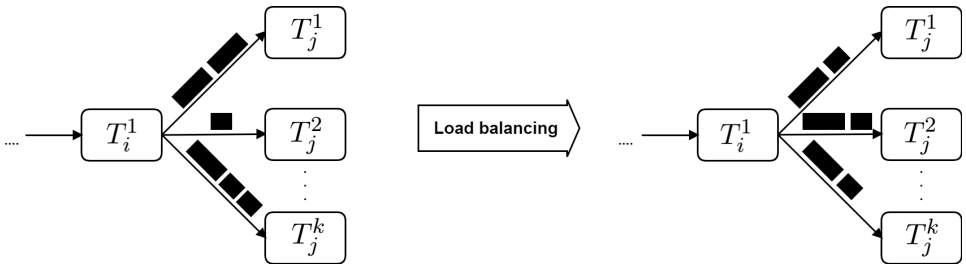


Figure 5: Load balancing

parallelized as presented above. If a task mostly receives registration licenses linked to remote databases with high communication latency, its processing latency is greater than other tasks. It involves an imbalance between tasks and decreases throughput of the parallel region composed of all tasks. So, adapting the distribution of stream elements to loads of tasks may improve the global throughput.

Assumptions

As load balancing becomes relevant after operator parallelization, all assumptions mentioned for operator parallelization are valid for load balancing. In addition, load-aware strategies require a strict equivalence between tasks. Indeed, let consider a stateful operator computing average values of stock prices for multiple stock label. The operator computing average values can be naturally parallelized if disjoint subsets of stock labels are associated to each task. In such case, each task is only able to process a partition of stream elements so load imbalance cannot be corrected.

Principle

Definition 14. (Load balancing) *Let consider a set of equivalent tasks* $T = \{T_i^1, ..., T_i^k\}$ *and the set of their respective loads* $\mathcal{L} = \{\mathcal{L}_1, ..., \mathcal{L}_k\}$ *according to a load metric. For any finite subset of stream elements* $\mathcal{E} = \{e_1, ..., e_n\}$*, balancing load consists in finding k partitions of* $\mathcal{E}$ *such as the standard deviation of loads in* $\mathcal{L}$ *is minimized at any time.*

The choice of the strategy for load balancing and the metric load depends mainly on assumptions on stream properties. Load balancing strategies can be classified in two main categories [Pearce et al., 2012]:

- Value-aware strategies [Neumeyer et al., 2010] route stream elements according to key values without consideration for balance between tasks.
- Load-aware strategies [Rivetti et al., 2016] tend to balance load between threads. They consider load balance in term of execution time or resource consumption (CPU, RAM).

Benefits for congestion management

Load balancing reduces imbalance between equivalent tasks processing stream elements in parallel according to data parallelism. As the global throughput of such parallel region is directly impacted by the slowest operator, load balancing improves global throughput proportionally to the skew in data distribution. Without load balancing, operator parallelization does not improve significantly the global throughput of an operator.

1.5.2 Load shedding

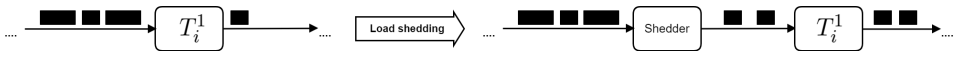


Figure 6: Load shedding

Motivation

Let consider a website suggesting articles to buy online. To suggest in priority interesting products according to user behavior, the website monitors clicks to identify points of interest. Monitoring logs are consumed in real time by a stream-based application identifying behavior of users and suggesting relevant products. The distributed infrastructure executing this application is calibrated to analyze logs generated by a standard traffic on the website. When the festive season starts, the traffic increases by orders of magnitude causing an overflow in input of the stream-based application. In such case, the owner of the website is no more interested in suggesting products very relevant products to each users and accepts to degrade result quality. It avoids to add resources for this specific period through a limitation of the maximal input rate.

Assumptions

As mentioned in motivation of load shedding, this pattern degrades result quality so there is no assumption guaranteeing the preservation of result quality. The single assumption for load shedding concerns the control of the degradation of result quality. Actually, load shedding is enabled to avoid uncontrollable loss of stream elements. The load shedding method must provide a feedback on dropped elements to estimate the impact on final results.

Principle

Definition 15. (Load shedding) *Let consider a task T_i^j with an average throughput TP defined in stream elements processed per time unit and a potentially infinite multiset $\mathcal{S} = \{S_1, S_2, ..., \}$ composed of finite sets arriving at each time unit. Load shedding consists in finding for each set $S_i \in \mathcal{S}$ a subset S_i' such as the cardinality of S_i' remains lower or equal to TP.*

Load shedding inverts the adaptation compared to patterns presented previously. Indeed, while other patterns adapt the DAG of operators and its execution on an infrastructure, load shedding adapts streams to available processing rates [Tatbul et al., 2007]. In [Tatbul et al., 2003], authors suggest load shedders which drop stream elements to satisfy some QoS metrics about overall latency and precision of final results. Aurora* analyzes impact of operators on those metrics through their processing latencies and selectivity factors [Abadi et al., 2003, Cherniack et al., 2003b]. In [Arasu et al., 2004], multiple techniques for load shedding are suggested including load shhedding through statistical approximation of aggregative queries. The distributed solution Borealis [Abadi et al., 2005, Tatbul et al., 2007, Ahmad et al., 2005] suggests several techniques to control quality degradation caused by load shedding including window-aware load shedding [Tatbul and Zdonik, 2006].

Benefits for congestion management

Load shedding avoids overflow in input of tasks. So, it helps to maintain a maximal throughput when initial input rates exceed processing throughput. Load shedding can also be used before

operators generating multiple outputs per stream element in input like joins. It allows to control output rate and have an approximation of result quality.

To sum up, we presented a collection of common patterns adapting continuous query execution or streams in order to avoid congestion. They can tackle issues involved by some specific cases under certain assumptions on the execution context. Nevertheless, finding the right moment to trigger a reconfiguration is complex. On one hand, as adaptation patterns have overheads during and potentially after reconfiguration, it is not advisable to use them if they do not effectively compensate a degradation in the execution. On the other hand, waiting that effective congestion occurs has important effects on system stability and the recovering time (i.e., time elapsed between reconfiguration and an optimal throughput). To trigger reconfiguration with a high probability that it compensates effectively a degradation of the execution, some detection methods have been developed and integrated to DSMSs.

2 Detection methods for congestion management

As presented in section 1, some specific cases can be encountered while processing streams. The effectiveness of patterns presented above relies on the reactivity of the system. We distinguish two main categories of detection methods. On one hand, on user-demand methods [Xu and Peng, 2016] rely on users to trigger a reconfiguration when a degradation of performance or result quality is observed. When user alerts the system, recent history of the behavior of operators, which may be an instantaneous snapshot, is considered to perform appropriate modifications. On the other hand, some automatic methods [Xu et al., 2014, Gedik et al., 2014, Schneider et al., 2009] suggest monitoring mechanisms observing at runtime some metrics about behavior of operators and triggering automatically adequate modifications to respect some QoS constraints.

In this section, we aim at giving an overview of detection techniques to manage congestion of operators. Each method is presented with its principle, advantages and limits illustrated by main variants.

2.1 On user-demand methods

Methods relying on user demand [Aniello et al., 2013, Xu and Peng, 2016] performs operator parallelization and scheduling when user triggers a reconfiguration. The aim is to find an adapted configuration according to a recent history of operator behaviors. For each operator, metrics like input rate, throughput and processing latency are monitored at regular interval. When the user triggers a reconfiguration of a continuous query, the system takes these measures into account to decide which critical cases are encountered and which patterns are appropriate to satisfy QoS constraints.

In [Aniello et al., 2013], authors suggest an offline scheduler aiming at generating a low network traffic between processing units. To do so, the scheduling algorithm has as inupt the workflow representing a continuous query and returns a scheduling plan minimizing the number of edges between processing units. As the scheduler is offline, it is only when users submit or re-submit continuous queries on the system. In [Xu and Peng, 2016], authors present an online algorithm enabling operator parallelization at runtime. This algorithm, named Stela, takes as input user action like add 2 processing units and a threshold discriminating critical operators from normal ones. This threshold is given by user and specifies the upper bound of the ratio

input rate on processing rate. Stela returns the set of operators to parallelize in order to avoid congestion of critical operators.

Benefits brought by approaches on user-demand mainly depend from the master user reactivity and expertise. By, master user, we refer to the user having credentials to reconfigure a given continuous query. As data streams may have fluctuations in input rate and value distribution over time, a fixed configuration may not deliver optimal performance over time. Moreover, a given configuration may lead a workflow to congestion (*e.g.*, a scheduling plan generating network bottlenecks).

Logically, limits of approaches on user demand are users. In [Aniello et al., 2013], as a scheduling plan is generated only on user demand, users have to re-submit continuous queries when there are some execution bottlenecks which could be eliminated through scheduling of operators. In addition, the offline scheduler does not have information about operator properties (selectivity factor or processing latency) so it is assumed that operators do not amplify or downplay a variation in input rate. For example, a cartesian product amplifies significantly a variation in input rate. This assumption is irrelevant for most stream-based applications filtering stream elements. In [Xu and Peng, 2016], if a user adds only few processing units when there are massive overflow, performance is not significantly improved while reconfiguration involved important overheads. Moreover, users may trigger reconfiguration with an important offset after some operators start to accumulate stream elements on their input queues.

2.2 Automatic methods

Stream processing requires high reactivity to maintain treatments while facing critical fluctuations in input rate and value distribution. To ease the management of continuous queries from user point of view, most elastic solutions [Abadi et al., 2003, Zaharia et al., 2012b, Arasu et al., 2004, Abadi et al., 2005, Schneider et al., 2009, Gedik et al., 2014, Xu et al., 2014] integrate automatic and dynamic mechanisms to detect critical cases and trigger reconfiguration of the system according to metrics monitored continuously. These metrics are observed at different scopes of the execution:

- At infrastructure scope, some global metrics are observed like the network traffic [Xu et al., 2014], CPU load and memory usage on each processing unit [Zaharia et al., 2012b].

- At operator scope, metrics describing the execution are observed. It concerns essentially input and output rate [Schneider et al., 2009, Gedik et al., 2014], processing latency [Abadi et al., 2005] and selectivity factors [Abadi et al., 2003, Arasu et al., 2004] of operators.

Automatic methods considering these metrics in input can be grouped two main classes [Das et al., 2005, Lorido-Botran et al., 2014]: *reactive* and *proactive*.

2.2.1 Reactive methods

Like on user demand methods, reactive methods rely on user expertise but not on its reactivity. User expertise is required before treatments to define what is a normal state of the system. By state, we consider a set of metrics describing the execution of some continuous queries. There are two types of reactive methods: threshold-based and reinforcement learning-based approaches.

Threshold-based

Threshold-based algorithms takes as input maximal and minimal thresholds for monitored metrics at infrastructure [Zaharia et al., 2012b, Xu et al., 2014] and/or operator scope [Abadi et al., 2003, Arasu et al., 2004, Abadi et al., 2005]. These thresholds compose a policy defining conditions for the stability of the system. When the value of a monitored metric is under the minimal threshold or over the maximal threshold, some patterns for congestion management are applied. The objective is to maintain a stable state of the system.

Threshold-based algorithms ensure that a given policy is respected but they are strictly *reactive*. Actually, only an effective violation of the policy triggers a reconfiguration of the system. Depending on user expertise, the policy may allow temporary congestion of operators having an important impact on performance of the system and result quality. Moreover, some thresholds like input/process rate may be more critical for some operators like join which generate potentially multiple output per input element and have intrinsically a greater impact on result quality. Adjusting thresholds may be laborious according to the complexity of a continuous query and the execution support.

Reinforcement learning-based

Reinforcement learning-based algorithms (RL-based) aim at reaching a performance goal through interactions with the system but without *a priori* knowledge. In RL model, an *agent* receives as input a *state* describing the execution context (*e.g.*, input rate and throughput of an operator) and applies an action a_i from the set of all possible actions $\mathcal{A} = \{a_1, a_2, ..., a_n\}$. According to the evolution of scalar metric, the agent receives a *reward* as feedback. Through a learning phase, the agent maps rewards to actions for each encountered states. The state-action-reward map is denoted the *policy* of the agent. Once the agent meets a known state, it applies the best action according to its policy and updated it according to the new reward.

In [Heinze et al., 2014b, Heinze et al., 2014a], authors present a distributed reinforcement learning solution, named FUGU, which maintains independent policies on each processing unit. This solution presents the advantage to be independent from workload as it is able to converge to a appropriate configuration. A problem is the convergence time which could be important if the workload varies often. An other limit of this approach is that it considers exclusively variations in input rate to associate parallelism degrees without taking distribution of values into account. Thus, if the distribution of values has an important impact on the processing latency and changes significantly over time, the knowledge base built through training may be obsolete quickly.

Control-based

Control systems [De Matteis and Mencagli, 2016] aim at automating the management of online environments. There are three classes of reactive control-based systems [Lorido-Botran et al., 2014]: open loop and feedback.

Open-loop approaches relies on a model of the system and a controllable metric which should not exceed some thresholds according to the model and the current state of the system. The controller does not receive feedback about the effectiveness of reconfiguration making this solution limited if the system differs from the model over time. For feedback controllers, the model of the system is replaced by an output metric associated to a desired value. Any significant deviation from this value triggers a reconfiguration. For example, considering that the output metric is the throughput of an operator, the controller may increase the parallelism degree when the throughput decreases at constant input rate.

In cite [Gedik et al., 2014], authors suggest a feedback controller relying on a congestion

metric which indicates if an operator is potentially critical. When the value of the congestion metric degrades (increase of input queue sizes), the algorithm triggers operator parallelization for the overloaded operator. To control if the reconfiguration is beneficial to the execution, the evolution the throughput is observed. If the control metric evolves positively, the reconfiguration is repeated until the congestion metric reaches a satisfying value according to QoS constraints.

Control-based algorithms offer more flexibility than threshold-based approaches as it only require that users define congestion and control metrics. The main limit of control-based approach is the accuracy of reconfiguration [Gedik et al., 2014]. Indeed, if the reconfiguration has a minor impact of the congestion metric, it will require several reconfiguration before stabilization.

2.2.2 Proactive methods

Proactive methods aim at anticipating the behavior of system to avoid congestion before a critical workload appear. Contrary to reactive methods, these methods might trigger unnecessary reconfiguration because of sudden variations of workload (*e.g.*, an input rate increasing continuously for a significant time and decreasing massively within a short period) but they react faster to specific cases leading to congestion. On one side, control-based and queuing-based approaches use a model to predict how the system reacts to modifications of the workload. On the other side, time-series based approaches estimate the behavior of the system without model but from recent history.

Control-based

A third class of control-based systems, denoted *feed-forward* [Lorido-Botran et al., 2014], try to anticipate deviation from the desired value in output. According to the model, they predict the behavior of the system with current inputs and reacts before a deviation appears. In most cases, feed-forward controllers are combined with feedback controllers to compensate prediction errors.

Queuing-based

Queuing-based algorithms use mathematical methods associated to te queuing theory as presented in [Kalashnikov, 2013] to model the execution of operators having pending queues as presented in section 1. To model an operator through queuing theory, several parameters are required:

- Stream elements are supposed to arrive at a mean rate λ and are enqueued before being processing.
- The operator has an average throughput μ for all possible values of stream elements.
- The inter-arrival time distribution A which corresponds to the distribution law describing the evolution of λ over time.
- The service time distribution B which corresponds to the distribution law describing the evolution of μ according the arrival rate λ.
- The number of operators C in the case multiple operators share a common pending queue (*e.g.*, after parallelization of operators).
- The system capacity K which corresponds to queue length.

- The calling population N which is the size of the stream. In the case of infinite streams, the queuing model is said *open*.

- The service discipline D which matches with the priority order in the execution context. In most systems, D is FIFO (First In First Out).

According to these parameters, a system is modeled according to the Kendall notation A/B/C/K/N/D or just A/B/C as K, N and D are optional as presented in [Kendall, 1953]. Actually, when not precised K and N are considered as infinite and D is FIFO. A and B are generally equal to M, D or G. M refers to Poisson distribution, D for uniform distribution and G corresponds to any general distribution with known mean and variance.

The aim is to approximate the response time R for an incoming stream element according to the queuing model in Kendall notation. For example, the response time for a M/M/1 is R= $\frac{1}{\mu-\lambda}$.

In [Jiang and Chakravarthy, 2003], authors present models for Select-Project-Join operators according to queuing theory in a stream processing context. It allows to estimate response time of these operators to apply dynamically operator reordering and parallelization to respect some QoS constraints.

Time-series based

Time-series refer to sequence of values generally measured at regular time interval. Time-series analysis can be used for two goals: forecasting future values of a sequence or find repeating patterns to extrapolate future values. According to an estimation of future values, a decision-making policy defines if the system should be reconfigured and how.

Let focus first on forecasting problem. From a recent history of monitored metrics, the aim is to predict resource usage or workload. Common forecasting techniques are Moving Average, Auto-regression, the combination of the two, denoted *ARMA* [Li and McLeod, 1981] and machine-learning techniques.

Moving Average assumes the next value y_{t+1} is the weighted average of the last n observed values x_i as follow:

$$y_{t+1} = a_1x_1 + a_2x_2 + ... + a_nx_n \tag{1}$$

such as the sum of weights a_i is equal to 1. From this general definition, three variants exist:

- Simple Moving Average associates a weight $\frac{1}{n}$ to each measure x_i. It corresponds to the arithmetic average.

- Weighted Moving Average associates different weights to measures. Commonly, the older is the stream element, the lower is the associated weight. So, it gives more weight to recent measures.

- Exponential smoothing assigns decreasing weights over time to measures according to an exponential distribution.

Auto-regression of order n relies on the same principle than Weighted Moving Average but weights are computed as auto-correlation coefficients. The correlation between former forecasted values y_{t_j} with $j < t + 1$ is used to calculate weights for each timestamp.

Auto-regressive Moving Average or ARMA combines Weighted Moving Average with Auto-regression. The next value is a weighted average of previous forecasted values and values of the recent history.

Machine learning techniques include statistical and neural network-based methods. Statistical methods corresponds to regression techniques. Regression aims at determining the polynomial function such that it minimizes the distance to each point composing the recent history. The particular case of polynomial function of order is denoted *linear regression*. This function is used to predict future values. Methods based on neural networks consider a group of neurons interconnected on several layers from input layer having some measures to output layer returning a result. In a stream processing context, the input layer contains one neuron for each measure in the recent history and the output layer contains one neuron for the expected value. The network is trained with initial weights chosen randomly. These weights are adapted through the neural network in order to find a polynomial function which returns the expected value from measures.

To sum up, relying on user demand to manage congestion involves that it is assumed that the user monitors continuously states of the cluster and continuous queries. In addition, users should have an expertise to operate appropriate modifications of the execution context at physical and logical layers. As many applications composed of several operators consume streams varying frequently, some automatic solutions have been proposed. Some of them react to effective situations leading to congestion while other anticipate the behavior of systems to identify needs of reconfiguration to respect QoS constraints.

3 Classification of distributed DSMSs

We have suggested a formalization of streams and continuous queries. In addition, we presented challenges involved by stream processing and described elastic mechanisms for congestion management. In this section, we aim at giving an overview of principal distributed DSMSs developed on last years. This collection is neither exhaustive nor a strict top-tier ranking but it covers most DSMSs in terms of variety according to classification criteria. Centralized DSMSs like STREAM [Arasu et al., 2004], Aurora [Abadi et al., 2003], Medusa [Cetintemel, 2003] and NiagaraCQ [Chen et al., 2000] are detailed in former surveys [Babcock et al., 2002, Stephens, 1997]

3.1 Criteria of classification

As illustrated on Figure 8, we suggest a classification of DSMSs based on the paradigm used to represent continuous queries. It has a major impact on the expressiveness of the definition language and the effectiveness of data and task parallelisms affecting congestion management. As presented in chapter 2 section 2.1.4, two paradigms are commonly used to represent continuous queries: workflow and MapReduce. In addition, a secondary criterion is the expressiveness of the language. Indeed, while processing streams, it is necessary that query languages support specific operators like windows to process streams with stateless and stateful operators. Then, detection method for congestion management defines the reactivity of the system and the configuration effort required to maintain respect QoS constraints at runtime. Finally, according to the stream classification suggested in chapter 2 section 1.2, we highlight which types of streams each DSMS is able to process efficiently and detail which properties determine that choice.

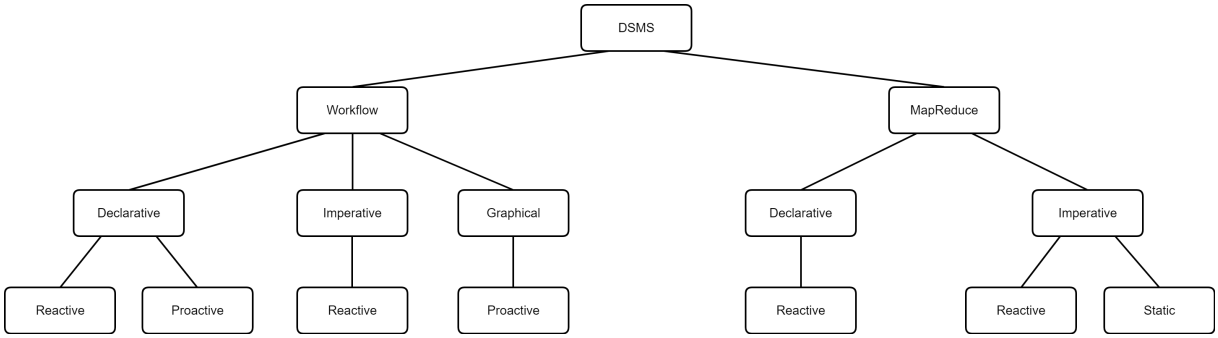


Figure 7: Classification of distributed DSMSs

3.2 Workflow-based solutions

3.2.1 RStorm

The open-source DSMS Apache Storm[12] integrates a resource-aware scheduler since its version 1.x.x and is also called RStorm [Peng et al., 2015]. RStorm offers a high flexibility for query definition and performance tuning. It allows users to specify processing requirements of each operator in terms of CPU and memory and guarantee minimal waste of resources according to these specifications.

Query definition: RStorm relies on a high level language (Java, Scala, Python) for query definition. A continuous query is defined as a sequence of user-defined operators subscribing to sets of streams and generating zero, one or many streams. Each operator follows a template which corresponds to stateless or stateful operators. RStorm integrates a growing collection of templates corresponding to complex relational operators like Joins.

Query representation: A continuous query is represented as a workflow of operators. RStorm considers two types of operators:

- *Spouts* are connectors to stream generating services like sensors, databases or a distributed messaging service like Apache Kafka[13]. Spouts are responsible to manage the lifecycle of stream elements through a workflow. Each spout produces one or many streams.
- *Bolts* is an operator consuming one or many streams and producing a set of streams which may be empty. Bolts follow templates for stateless or stateful operators.

Each operator may or may not support data parallelism. However, for stateful operators, users take charge to declare and update states. RStorm serves as guarantee that states are moved when the operator is moved or restarted after node failure.

Window support: RStorm supports time-based and count-based sliding and tumbling windows through specific templates. Windowing templates take as parameters window size, an optional slide and a function describing business logic.

[12] http://storm.apache.org/
[13] https://kafka.apache.org/

Congestion management: RStorm is able to adapt continuous queries at operator level exclusively. Indeed, operator scheduling is performed at the initialization of a continuous query according to processing requirements specified by users. RStorm relies on a greedy algorithm to find a near-optimal scheduling plan which maximize resource usage on a minimal subset of available machines. Operator parallelization are triggered on user demand with the constraint that users assume that parallelized operators support data parallelism without changing the semantic of results (*e.g.*, parallelization of a mean operator).

Target streams: As RStorm relies on user expertise and reactivity, it is basically designed to handle even steady streams involving rare modifications at runtime. The absence of load shedding mechanism and the guarantee that each stream element is processed exactly once suppose that users set adequate configuration in terms of parallelism degrees and processing requirements according to input volumes.

3.2.2 Sonora

Sonora [Yang et al., 2012] is platform for mobile cloud computing. Sonora is designed to support the execution of distributed cloud services on mobile devices. Sonora integrates a stream engine able to perform fast failover after node failure as mobile devices supporting computations come in and out frequently from the cloud. In addition, Sonora has to be energy efficient to respect battery constraints of mobile devices.

Query definition: Sonora provides interfaces for continuous query definition in a high level language. These interfaces allow users to define functions on streams without restriction. A continuous query is defined as a succession of operators subscribing to some streams and generating an output stream.

Query representation: Continuous queries are represented as workflows of user-defined operators. Even if users define stateless operators, Sonora may group and send stream elements into batches in order to save battery. Indeed, it allows to send a batch of stream elements and turn off the radio instead of keeping the radio active and send stream elements continuously as they arrive.

Window support: Sonora supports time-based and count-based sliding and tumbling windows. Windows are explicitly defined by users through the definition of *timespans* on streams. Sonora computes incrementally results on sliding windows thanks to overlaps between consecutive iterations as discussed in chapter 2 section 2.1.1. In addition, Sonora computes incrementally mean, variance and Discrete Fast Fourier Transformation (DFFT) operators over sliding windows. Incremental computations for these operators remove redundant computations in order to save energy.

Congestion management: Sonora performs adaptation of continuous queries at operator and data level. Operator parallelization is managed through the storage system PacificA which splits dynamically streams into partitions as presented in [Lin et al., 2008]. It allows to perform operator parallelization while PacificA takes charge of splitting input streams and balancing the load at runtime. Sonora can also perform load shedding through a sampling mechanism. It allows Sonora to maintain a fixed rate in input that the execution support (i.e., a mobile cloud) can handle.

Target streams: Sonora is suitable to process even and uneven steady streams like sensors streams. As Sonora does not integrate mechanisms to detect congestion at operator scope, it is not adapted for fluctuating streams. Adaptation mechanisms integrated in Sonora aims at recovering after failure more than facing fluctuations in input streams. Moreover, the load shedding mechanism consists in turning a massive stream (steady or fluctuating) into a steady stream with a maximal input rate.

3.2.3 Millwheel

Google Millwheel [Akidau et al., 2013] is a distributed DSMS designed to analyze data streams at Internet scale. Millwheel is fault-tolerant as the failure of any active machine does not affect results delivered to users. To guarantee fault-tolerance, operators communicate through the following pattern: An operator $\mathcal{O}_1$ applies a function on a stream element, creates a checkpoint locally and returns a result to an operator $\mathcal{O}_2$. As soon as $\mathcal{O}_2$ as processed the result, the checkpointing mechanism saves the state of $\mathcal{O}_2$ and sends a acknowledgement message to $\mathcal{O}_1$ to update its current state.

Query definition: Continuous queries are defined in Millwheel according to a programming model hiding parallelism and concurrency issues to developers. The programming model is designed to facilitate the definition of low latency streaming applications. Each continuous query is defined as sequence of user-defined operators like in RStorm, but with a main difference on operator patterns. When bolts consume stream elements described by a set of attributes, operators in Millwheel consume stream elements linked to a *key* and users must implement a *key extractor*. Indeed, the business logic of an operator is applied exclusively on that key. For example, an operator computing the average price of a stock, the key extractor returns the price among the list of values describing the stock and apply mean function on the price.

Query representation: Millwheel represents continuous queries as workflows of user-defined operators.

Window support: Millwheel supports time-based sliding and tumbling windows through its API. A specific operator pattern allows to define a stateful operator over a computation window like RStorm.

Congestion management: Millwheel adapts continuous queries at operator and data levels. It supports operator parallelization and load balancing through a threshold-based algorithm which detects lack of available resources in terms of CPU and memory. When an overload is detected by local monitors, an operator can be parallelized on another processing unit to share the load. Load balancing is performed by key grouping among tasks of an operator. For example, considering an operator which consumes keys in a range from 0 to 9 and has 2 tasks, the input stream in partitioned in two partitions, one receiving keys between 0 and 4 and another receiving keys between 5 and 9. It assumes that the distribution of keys over stream elements is uniform over time.

Target streams: Millwheel supports efficiently even erratic streams in terms of volume as it relies on a threshold-based algorithm to detect and correct potential congestion of operators.

Nevertheless, its load balancing strategy based on key partitioning assumes that value distribution over keys is almost uniform at any time and processing latency associated to keys are pretty similar.

3.2.4 TelegraphCQ

TelegraphCQ [Chandrasekaran et al., 2003] is a flexible solution enabling fault-tolerant stream processing over PostgreSQL. Originally, it was designed for centralized multicore architecture but it can be extended to distributed infrastructure through an extension of the FluX module [Chandrasekaran et al., 2003].

Query definition: As TelegraphCQ is built over a PostgreSQL DBMS, it supports declarative expressions written in SQL and includes window statement. The definition is enriched with imperative expressions like *for* loops to apply treatments only on a finite number of window iterations.

Query representation: TelegraphCQ represents continuous queries as workflows having a star topology and associated to a routing policy. Indeed, the heart of each continuous query is a module named *Eddy* [Madden et al., 2002, Chandrasekaran et al., 2003] which receives the initial stream to process and route stream elements to operators according to its routing policy. Workflows in TelegraphCQ are composed of common operators (Select, Project, Join) of the relational algebra.

Window support: TelegraphCQ supports multiple window semantics: time-based and count-based sliding and tumbling windows but also *landmark* windows [Chandrasekaran et al., 2003] which have a growing size over time. Indeed, landmark windows are only defined by a slide and consider all elements since the start of the execution. Each iteration of a landmark window is superset of previous iterations.

Congestion management: TelegraphCQ integrates adaptation mechanisms at workflow, operator and data levels. To do so, TelegraphCQ relies on a threshold-based algorithm to detect potential congestion of operators. It is able to perform both operator reordering and load shedding through the Eddy module. Indeed, as the routing policy associated to the Eddy module defines the execution sequence of operators, a modification of this policy corresponds to operator reordering. The Eddy module can also discard some stream elements to reduce the workload in input of an operator. Operator parallelization and load balancing are supported in TelegraphCQ by a FluX module which is responsible to manage distribution of stream elements routed by the Eddy. FluX can add tasks of an operator and decides how inputs are distributed among them.

Target streams: TelegraphCQ has been designed to process efficiently even and uneven erratic streams. Indeed, the design of TelegraphCQ is centered on dynamic Eddy modules to route streams and shed load anytime without reconfiguration cost but with permanent overheads [Deshpande, 2004]. So, it is assumed that these overheads are compensated by frequent adaptations due to fluctuations in input rate and value distribution. Even if TelegraphCQ can also manage efficiently periodic streams, there is no model or learning-based modules to optimize adaptation to such streams. Finally, steady streams are processed by TelegraphCQ with permanent overheads that are never compensated as soon as an appropriate configuration has been found.

3.2.5 Borealis

Borealis [Abadi et al., 2005, Ahmad et al., 2005] is a distributed DSMS which inherits the stream processing engine from Aurora and its management policy of distributed infrastructures from Medusa [Balazinska et al., 2004]. The main improvement brought by Borealis is the support of *revisions* at runtime. In a real-world context, some stream sources may send stream elements out-of-order or send incorrect values and correct them later. For example, in a context of financial market analysis, some values may be out-of-date for some stock prices. A correction is sent after to update the value. Borealis integrates mechanisms which support revisions of results being computed to soften errors in a real-world context.

Query definition: Borealis inherits the graphical definition interface of Aurora and its *boxes-and-arrows* query model introduced in [Abadi et al., 2003]. Users select operators from a predefined set, named SQuAl [Cherniack et al., 2003b], and define a graph which is considered as optimal. Indeed, even if Borealis integrates predefined operators, there is no query optimization based on relational algebra. The set of operators supported by Borealis includes common operators like Select-Projection-Join and some aggregate operators as sort and union.

Query representation: Each continuous query is a graph where vertices are *boxes* which implements an operator. Boxes are linked by *arrows* which specify the routing policy of stream elements through the graph.

Window support: As Aurora, Borealis supports time-based and count-based sliding and tumbling windows through an *Aggregate* operator which takes as parameters a function, an order O for grouping, a window size and a window slide. For example, computing the average price on the last stream elements according to the chronological order hour each 30 minutes corresponds to an Aggregate operator taking as parameter the average function on the attribute price, the order *On Time*, a size 1 hour and a slide 30 minutes.

Congestion management: Borealis supports adaptation of continuous queries at each level: workflow, operator, implementation and data. It relies on a feedback control-based algorithm as presented in section 2.2. When a deviation in throughput is detected, Borealis can trigger operator reordering, operator scheduling or load shedding at runtime. Operator reordering is based on a cost model between two operators. According to SQuAl algebra, if operators are commutative and selectivity factors measured online satisfy conditions presented in section 1, Borealis inverts operators. Operator scheduling relies on bottleneck detection. When stream elements are accumulated on input queues, Borealis looks on the processing unit if it is due to a lack of CPU or bandwidth. In both cases, Borealis moves the operator on a processing unit which is more available according to the lacking resource. If the scheduler cannot find such processing unit, it enables load shedding through two phases. First, according to an algebraic analysis, Borealis may add Projection operators to reduce data volumes. If it is not feasible, load shedding is triggered on operators which affects the less result quality.

Target streams: Borealis has been designed to manage real-world even and uneven erratic streams and even support out-of-order and incorrect streams. While facing a significant variation in input rate or value distribution, Borealis tries to adapt dynamically the execution of continuous queries at query and workflow scopes and if it cannot find a satisfying configuration, it sheds

load to respect QoS constraints about performance. As TelegraphCQ, the absence of model and learning-based mechanisms denotes that Borealis has not been optimized for per-pattern streams. Finally, Borealis supports algorithm selection through a mechanism similar to *Eddy* module. The SQuAl compiler provides multiple implementations of predefined operators on each processing unit and depending of input properties, the implementation may be changed at runtime.

3.2.6 ESC

ESC [Satzger et al., 2011] is cloud-based DSMS designed for real-time demands such as online data mining. The particularity of ESC is the capacity to support elasticity at infrastructure and workflow levels. Indeed, ESC can attach and release VMs at runtime to adapt the number of configured machines to processing requirements.

Query definition: ESC offers a programming model based on key/value pairs as MapReduce model [Dean and Ghemawat, 2004]. Users define operators according to programming patterns which take as input a stream element or a set of stream elements. Operators are defined in a high definition language named Erlang [Wikström, 1994] supporting natively parallel and concurrent programming.

Query representation: Continuous queries are represented as workflows of user-defined operators. In addition to query definition, users can define rewriting rules for some operators. For example, a user-defined operator $\mathcal{O}_i$ can be associated to a rewriting rule split$\rightarrow \mathcal{O}_i \rightarrow$merge to trigger operator parallelization while needed.

Window support: Basically, ESC supports count-based tumbling windows. The order is based on arrival time making stateful operators not deterministic as window content may vary randomly. The management of other types of windows is delegated to user implementation.

Congestion management: ESC supports exclusively adaptation of continuous queries at operator level. Congestion is managed by a reinforcement learning-based algorithm in ESC. An autonomic manager implementing a Monitor-Analyze-Plan-Execute (MAPE) loop which enriches a knowledge base through interactions with ESC platform.

As mentioned above, ESC manages elasticity at physical and logical levels. At physical level, ESC integrates mechanisms to perform horizontal elasticity. After a physical scale-out, ESC can perform operator scheduling to avoid bottleneck due to a lack of CPU on active VMs. ESC relies on rewriting rules to perform operator parallelization as there are not algebraic properties on operators if users do not declare it explicitly.

Target streams: ESC centralizes the management of continuous queries and infrastructure in a single decision system. Combined to a reinforcement learning-based algorithm, ESC can add and delete processing units to execution support periodically. It makes ESC efficient to process even bounded streams after a learning phase. During that phase, ESC builds a knowledge base which covers the range of rates in input without shedding the load when data volumes become important. Intuitively, the convergence time to a complete and accurate knowledge base over any uneven or erratic streams is potentially infinite as any variation may happen anytime.

3.2.7 System S

System S [Amini et al., 2006, Jain et al., 2006] is a distributed DSMS developed at IBM supporting structured and unstructured data stream processing. It is fault-tolerant and integrates security mechanisms in a distributed context.

Query definition: System S relies on the language SPADE [Gedik et al., 2008] for continuous query definition. This language offers a set of stream-oriented operators which are specific to a domain (*e.g.*, signal processing or data mining) in addition to relational operators like Select-Project-Join. Users define a continuous query as a sequence of SPADE operators consuming and generating specified streams. SPADE relies on a logical and physical algebra turning logical operators into sets of physical operators. Each operator in SPADE may be executed in parallel without specifications from users.

Query representation: System S represents continuous queries as workflows where vertices are *Processing Elements* (PE) linked by inner-streams. A PE is composed of a working receiving input streams and three types of threads taking charge of routing and processing stream elements but also adapting the throughput of the PE to variation of workload.

Window support: System S supports time-based sliding and tumbling windows [Wu et al., 2007] through the use the operator Aggregate similar to the Aggregate operator included Borealis. Windows can also be declared through the use of the operator *Punctor* [Biem et al., 2010] which apply a stateless operator on stream elements and inserts punctuations in output streams according to a predefined interval.

Congestion management: System S integrates mechanisms to adapt continuous queries at operator and data levels. This DSMS is able to perform operator parallelization for each PE individually. Indeed, for each PE, the dispatch thread reevaluates periodically the number of worker threads to enable so it performs operator parallelization according to a reinforcement-based algorithm at PE scope. When an overload is detected, the dispatch thread increases the number of worker threads until there is no improvement of the throughput. When the workload decreases significantly, the dispatch thread puts some worker threads into sleep. Worker threads are not deleted because it is cheaper to wake up a thread than creating it. Through a continuous monitoring of window contents, System S is able to perform load shedding when data volumes are critical in input of PEs running stateful operators.

Target streams: PEs manage parallelization individually according to a reinforcement learning algorithm. This design fits to even bounded streams as each PE builds its own knowledge base and is able to adapt quickly its parallelism degree according to input rate after a learning phase. Moreover, when some working threads are not necessary to handle the current workload, they are only put into sleep to speed up adaptations to future increases. Such mechanism is costly while processing steady streams as fluctuations are rare. Processing uneven streams involve several modifications of the knowledge base which lower its interest.

3.2.8 Infosphere

IBM Infosphere [Biem et al., 2010] is a distributed DSMS able to process static data and streams with the same processing engine. It is extensible as it allows to add domain specific operators

implemented in a high level language like C++ in addition to predefined operators offered by the declarative language SPADE [Gedik et al., 2008].

Query definition: As System S, Infosphere relies on the declarative language SPADE for query definition. The declaration of continuous queries sits between a declarative expression like a CQL expression and an implementation in a high level language using stream-oriented APIs. User-defined operators follow *templates* which allow or not data parallelism.

Query representation: SPADE expressions are turned into workflows of PE as explained above. However, Infosphere classifies PEs into 3 main categories:

- *Source* PEs are connected to stream generating services. Infosphere integrates several connectors to receive streams from GPS devices or databases.
- *Sink* PEs convert a stream into a relation stored in a file or database system. Generally, relations produced by a Sink are meant to be used by another system.
- All other PEs consume and generate streams or set of streams.

Window support: Infosphere supports time-based and tumbling windows exactly like System S as they both rely on the same query language.

Congestion management: Infosphere adapts continuous queries at operator level exclusively. Indeed, Infosphere supports operator parallelization and scheduling on user demand and automatically. Indeed, through SPADE primitives, parallelism degrees and assignment on processing units can be specified for each PE. In addition, a reinforcement learning-based algorithm allows Infosphere to analyze online properties of PEs as input and output volumes to collocate operators on same processing units in order to avoid network bottlenecks.

Target streams: As Infosphere turns continuous queries into graphs of PEs like System S, it fits also to even bounded streams but with the difference that its initial configuration is defined by users.

3.2.9 Flink

Apache Flink [Carbone et al., 2015] is an open-source DSMS processing indifferently static data and data streams as continuous sequences of stream elements running through fault-tolerant *dataflows*.

Query definition: Flink provides several API for data manipulation and transformation. These APIs include relational operators like Select-Project-Join or machine learning operators like k-mean clustering. Like Sonora, continuous queries are defined as sequences of user-defined operators. As some operators are user-defined, their semantics are hidden to Flink's optimizer.

Query representation: From the definition of a continuous query, Flink generates an optimized graph of stateless and stateful operators. In opposition to other workflow systems, Flink may buffer stream elements between two operators before emission. Indeed, as illustrated in [Carbone et al., 2015], two stateless operators processing stream elements in pipeline exchange stream element as soon as they are produced. These exchanges are performed concurrently and the throughput does not increase when the rate exceeds a certain threshold due to this concurrency. Buffering stream elements and sending them into microbatches increase the throughput after this threshold.

Window support: Flink supports time-based and count-based sliding, tumbling and landmark windows. Windows are declared as options within stateful operators. Flink takes advantages of overlaps between consecutive iterations of a sliding window to compute incrementally results. This incremental processing model reduces significantly computation latency of stateful operators over sliding windows.

Congestion management: Flink only supports adaptation of continuous queries at operator level. Flink performs resource-aware scheduling according to user specifications about CPU and memory requirements for each operator like RStorm. As explained above, Flink relies on dynamic microbatching of stream elements between operators to improve throughput instead of shedding the load or parallelize operators at runtime.

Target streams: Flink does not integrate operator parallelization and load shedding at runtime. Nevertheless, Flink is able to maintain a near optimal throughput through dynamic microbatching. It makes it reliable for even erratic streams under the assumption that it exists an optimal size of microbatches which compensates an overflow in input. Otherwise, Flink is appropriate for steady streams if the bandwidth supports microbatches.

	RStorm	Sonora	Millwheel	TelegraphCQ	Borealis	ESC	System S	Infosphere	Flink
Workflow terminology	topology	application	dataflow	application	application	workflow	dataflow	dataflow	dataflow
Vertices terminology	spout and bolt	operator	operator	eddy and steM	box	processing element	processing element	processing element	operator
Target streams	steady and even	steady and even	erratic and even	erratic	erratic	bounded and even	bounded and even	bounded and even	erratic and even
Query language	API	API	API	SQL-derived	SQuAl or graphical	Erlang	SPADE	SPADE	API
Supported windows	time-based and count-based sliding windows	time-based and count-based sliding windows	time-based and count-based sliding windows	time-based, count-based and landmark sliding windows	time-based and count-based sliding windows	count-based tumbling windows	time-based sliding windows	time-based sliding windows	time-based and count-based sliding windows
Adaptation levels	operator	operator and data	operator and data	workflow, operator, data	workflow, operator, implementation and data	operator	operator and data	operator	operator
Adaptation strategy	reactive (user demand)	not mentioned	reactive (threshold-based)	reactive (threshold-based)	reactive (control-based)	reactive (reinforcement learning-based)	reactive (reinforcement learning-based)	reactive (reinforcement learning-based)	reactive (user demand)

Table 2: Summary of workflow-based DSMSs

3.3 MapReduce-based solutions

All solutions based on the MapReduce framework (see chapter 2 section 2.1.4) support the key/value data model with possible extensions. By default, we consider in the section that all approaches represent continuous queries as pipelines of MapReduce jobs except if mentioned differently.

3.3.1 C-MR

Continuous-MapReduce(C-MR) [Backman et al., 2012] is a DSMS enabling the pipeline execution of MapReduce jobs over unbounded data streams. C-MR extends the MapReduce framework to support window definition and ensures order preservation between parallel nodes processing partitions of a global input stream.

Query definition: C-MR extends MapReduce framework with window definition. Continuous queries are defined as standard MapReduce jobs (see chapter 2 section 2.1.4). Map and Reduce operators are enriched with windowing parameters (size and slide).

Query representation: A continuous query is represented as a DAG of MapReduce jobs running in pipeline. C-MR benefits from optimization of MapReduce jobs like sorting and merging between Map and Reduce phases. Note that Map and Reduce phases are executed asynchronously in C-MR to process a microbatch of stream elements as soon as it arrives.

Window support: C-MR supports time-based sliding and tumbling windows. Windows are declared directly as parameters of Map and Reduce operators. They are implemented through punctuations (see section chapter 2 2.1.1) to maintain chronological order within streams. C-MR takes advantage of sliding windows to compute incrementally results like Flink. Intermediate results are computed by *Combine* operators which keep updated results over sliding windows.

Congestion management: C-MR supports the adaptation of continuous queries at operator level. C-MR performs operator scheduling according to *affinities* between operators. As a continuous is represented by a DAG of MapReduce jobs processing stream elements in pipeline, C-MR collocate operators exchanging highest amount of data to limit network bottlenecks.

Target streams: C-MR aims even and uneven erratic streams as presented in [Backman et al., 2012]. To absorb fluctuations in input rate, C-MR relies on incremental computations over sliding windows as discussed above. According to this optimization, C-MR is able to process streams with short latency independently of input rate. The dynamic scheduling of operators allow C-MR to compensate critical overflows in input rate.

3.3.2 iMR

In-situ MapReduce (iMR) [Logothetis et al., 2011] is designed to processing timestamped data without preliminary storing phase. It allows to process data with frequent updates (e.g., server logs) where they are produced.

Query definition: iMR extends the MapReduce framework with window definition like C-MR. It allows to discretize timestamped data and apply MapReduce jobs to process stream elements in pipeline. The main difference with C-MR is that iMR requires that data are stored on disk to replay them as a sequence of finite substreams. Nevertheless, an iMR application processes stream elements continuously and generates new results as soon as updates are stored.

Query representation: As C-MR, iMR represents a continuous as MapReduce jobs processing stream elements in pipeline. Nevertheless, iMR applications does not integrate Combine operators.

Window support: Time-based sliding and tumbling windows are supported by iMR. Like C-MR, iMR takes advantage of overlaps between consecutive iterations of a sliding window to process incrementally results. However, iMR does not require a Combine phase but discretize streams into *panes* [Logothetis et al., 2011]. Let consider a sliding window of size R and slide S, a pane is a subwindow of size R/S. Map operators compute results on panes instead of complete windows and results are updated incrementally during the Reduce phase.

Congestion management: iMR supports the adaptation of continuous queries at data level exclusively. iMR enables load shedding when the workload lead to a violation of QoS constraint about end-to-end latency. Indeed, the number of stream elements included in a pane requires a processing time greater than user specification, the pane is discarded in case of a violation of QoS constraints. A fidelity metric, denoted C^2, reflects the quality of results according to losses during computation.

Target streams: According to its load shedding policy, iMR processes even and uneven erratic streams as it is designed to handle any massive increase in input rate. But contrary to Sonora, iMR does not sample inputs to obtain a steady stream. Indeed, when iMR detects a critical workload, it discards entire logic substreams (i.e., elements included in a subwindow) until the global workload becomes acceptable.

3.3.3 Spark Streaming

Apache Spark Streaming [Zaharia et al., 2012b] is a stream processing engine built over Apache Spark[14]. It aims at bringing the efficiency and simplicity of MapReduce paradigm to stream processing. Apache Spark outperforms other MapReduce implementations (Hadoop, Hive) because of its better memory management to perform most computation on main memory and avoid disk accesses.

Query definition: A continuous query is defined as a MapReduce job with window clauses. Spark Streaming benefits from Spark APIs to query relational data or use domain specific operators (*e.g.*, machine learning methods). Each operator Map and Reduce is declared with input and output streams.

[14] https://spark.apache.org/

Query representation: Continuous queries correspond to MapReduce jobs processing stream elements in pipeline like C-MR and iMR applications. Data are grouped and sent in *Resilient Distributed Datasets* (RDDs) [Zaharia et al., 2012a]. RDDs rely on the same principle presented for Flink. As it assumed that Spark Streaming processes huge volume of data, global throughput is significantly improved if data are buffered before being sent between operators.

Window support: Spark Streaming supports time-based and count-based sliding and tumbling windows. Windows are declared through Spark Streaming API potentially for each operator. Size of RDDs is adapted to window size and slide in order to perform incremental computations.

Congestion management: Spark Streaming only supports adaptation of continuous queries at operator level. It optimizes the execution of Spark applications to fit most requirements induced by a stream context. Operator scheduling is performed according to a threshold-based algorithm which detects overload in entry of operators and move them in order to avoid computation bottlenecks on some processing units. It does not take network overheads into account.

Target streams: Spark Streaming takes advantage of its threshold-based mechanism for congestion management to process even and uneven erratic streams. A significant fluctuation in input rate or value distribution may be compensated by reallocation of operators on processing units. The absence of reconfiguration traces like knowledge base shows that Spark Streaming is not specifically optimized for bounded stream processing.

3.3.4 Samza

Apache Samza [Noghabi et al., 2017] is a distributed DSMS supporting stateful processing and recovering fast from failures. Samza relies on partitioned local states updated by a low-overhead mechanism. It allows states to scale to hundred of terabytes without increasing failover latency.

Query definition: As other MapReduce approaches, Samza relies on the key/value data model. To manipulate such data, it provides an API integrating three types of operators:

- One-to-one operators correspond to operators taking in input a stream and producing a single stream in output. One-to-one operators offered by Samza are map, filter, window and partition operators.
- Many-to-one operators take as input many streams in input and produce a single stream in output. Samza suggests join and merge operators for this type. The operator merge returns the union of all input streams.
- One-to-many operators correspond to partitioning operators and are user-defined.

With these operators, users can define a continuous query as a sequence of operators subscribing and publishing some streams.

Query representation: Continuous queries are represented as Samza jobs. A job is a workflow where vertices are one of the operator mentioned above.

Window support: Samza supports time-based tumbling windows through the operator window mentioned above. It takes as parameter a size and a function which could be user-defined.

Congestion management: Apache Samza integrates mechanisms to adapt continuous queries at operator level and can potentially adapt queries at data level. In addition to fast failover, Apache Samza supports operator parallelization and scheduling at runtime triggered by a threshold-based monitoring module. Load balancing is performed by a user-defined partitioner.

Target streams: Like Spark Streaming, Samza adapts execution of operators according to a threshold-based algorithm so even and uneven erratic streams can be processed efficiently. Indeed, Samza monitors continuously operators and adapts parallelism degrees and assignments on processing units accordingly.

3.3.5 S4

Yahoo Simple Scalable Streaming System (S4) [Neumeyer et al., 2010] is a general purpose stream processing system built for large clusters and supporting massively parallel applications. S4 offers a decentralized architecture where all nodes share same functions and responsibilities.

Query definition: S4 offers a simple programming interface based on the key/value data model. A continuous query is defined as a sequence of user-defined operators following a pattern. A method *processEvent* specifies which keys are consumed by the operator and a method *output* describes keys produced the operator. These methods have stateful variations which require window specification.

Query representation: S4 represents continuous queries as workflows of *Processing Elements* (PE) different from PE presented previously. In S4, a PE is the combination of a *functionality* which corresponds to the operator logic, the type of consumed events, the list of keyed attributes the operator consumes and optionally considered values of keyed attributes. A PE may also be associated to a Time-To-Live (TTL). If no stream elements enters a PE during its TTL, the PE is discarded.

Window support: Time-based sliding and tumbling are supported in S4 through parameters included in stateful operators. As iMR and C-MR, S4 takes advantage of sliding window definition to compute incrementally results.

Congestion management: S4 integrates automatic failover but does not integrate mechanisms to adapt continuous queries. Indeed, no mechanism can tackle the apparition of network and computation bottlenecks. Moreover, routes are defined statically. It is also assumed that key partitions defined by users do not involve major imbalance. Only the TTL may remove a PE receiving no stream elements.

Target streams: S4 is the less flexible DSMS presented in this classification so it is assumed that input streams are steady and even. Even if the TTL allows the deactivation of idle PEs, it is supposed that the value distribution does not change often.

3.3.6 Dataflow

Google Cloud Dataflow[15], or just Dataflow [Akidau et al., 2015], is MapReduce-based DSMS designed natively for stream processing. Dataflow supports unordered data streams and provides multiple windowing schema. It results of the extension of FlumeJava API [Chambers et al., 2010] developed over Google Millwheel. FlumeJava is an API for the development of pipelines of MapReduce jobs. It facilitates the creation, the execution and the modification of such pipelines through a simple programming interface providing high-level operators mapped automatically on optimized MapReduce jobs.

Query definition: Dataflow provides two primitive operators:

- *ParDo* is the generic parallel function. It receives in input a stream element or a collection of stream elements and apply a function *DoFn* on input elements. DoFn is defined either by users in a high level imperative language like Java either with a high level operator provide by FlumeJava API. For each stream element, it returns zero, one or many outputs.
- *GroupByKey* is a key grouping operator applying a function on collections of stream elements sharing a same key.

Users define continuous queries through Dataflow API as sequences of *ParDo*/*GroupByKey* pairs.

Query representation: Continuous queries are represented as pipelines of MapReduce jobs composed of *ParDo* and *GroupByKey* operators. According to query definition, Dataflow can merge *ParDo* functions on a single operator if they share same keys.

Window support: To apply windowing on stream elements, Dataflow enriches the key/value data model with two additional metadata: *event time* and *window*. *Event time* corresponds to the arrival date of a stream element and window refers to the window it belongs for *GroupByKey* operators.

Congestion management: Dataflow only supports the adaptation of continuous queries at implementation level. Dataflow performs implementation selection at runtime through FlumeJava. Indeed, while using predefined operators, FlumeJava may have multiple implementations and select the most appropriate one according to execution features like input size.

Target streams: The absence of load shedding and adaptation of operators limit the ability of Dataflow to process even and uneven erratic streams. The algorithm selection is limited to compensate a substantial variation in input rate and value distribution for all operators.

[15] https://cloud.google.com/dataflow

	C-MR	iMR	Spark Streaming	Samza	S4	Dataflow
Microbatch terminology	pane	subwindow	RDD	none	pane	subwindow
Vertices terminology	map and reduce	map and reduce	map and reduce	map, filter, window, partition, merge and join	processing elements	ParDo and GroupByKey
Target stream	erratic	erratic	erratic	erratic	steady and even	steady and even
Query language	API	API	SparkSQL or API	API	API	API
Supported windows	time-based sliding windows	time-based and count-based sliding windows	count-based tumbling windows	time-based and count-based sliding windows	time-based tumbling windows	time-based sliding windows
Adaptation levels	operator	data	operator	operator and data	none	implementation
Adaptation strategy	reactive (threshold-based)	reactive (control-based)	reactive (threshold-based)	reactive (threshold-based)	none	reactive (control-based)

Table 3: Summary of MapReduce-based DSMSs

4 Discussion

After a formalization of basic concepts about data streams, we introduced the problem of congestion. In a real-time context, resources may not be adapted for computations during the complete lifetime of a query. It requires to *elastically* adapt resources (i.e., infrastructure) and continuous queries to processing requirements over time. To adapt the global capacity of the cluster, we presented solutions for substituting or adding processing units at runtime. Substituting processing units allows to benefit from more resources on a single processing unit but has many inconveniences. It requires to overprovision the execution support or to allow important migration overheads which is not acceptable for low-latency applications processing streams continuously. Adding processing units at runtime is supported in [Satzger et al., 2011] but is limited by the number of available machines and fits exclusively to cloud environments.

Elasticity at logical layer can be applied independently of the execution support. Main patterns presented in section 1 allow removing congestion due to computation and network bottleneck. Moreover, they can improve significantly performance of DSMSs (*e.g.*, operator parallelization) when used at runtime. The triggering of such elastic techniques at runtime require a continuous monitoring of submitted queries coupled to a detection mechanism. This mechanism relies either on external intervention (users or tier application) or on an automatic method. Relying on external intervention limits the elasticity of DSMSs to user reactivity and expertise. Automatic solutions based on queuing theory are limited by the static model used to predict the behavior of DSMSs. Indeed, clusters executing continuous queries evolve over time (*e.g.*, node failure [Yang et al., 2012]) so a static model is not appropriate in a stream processing context. Threshold-based approaches [Zaharia et al., 2012b, Chandrasekaran et al., 2003, Noghabi et al., 2017] are exclusively reactive as they only trigger reconfiguration according to the current state of the system. Control-based [Abadi et al., 2005] and reinforcement learning-based approaches [Amini et al., 2006, Biem et al., 2010] can anticipate congestion of operators through a continuous adaptation of workflows to workload. Nevertheless, finding an optimal configuration may require an important number of interactions with the DSMS and induce reconfiguration overheads.

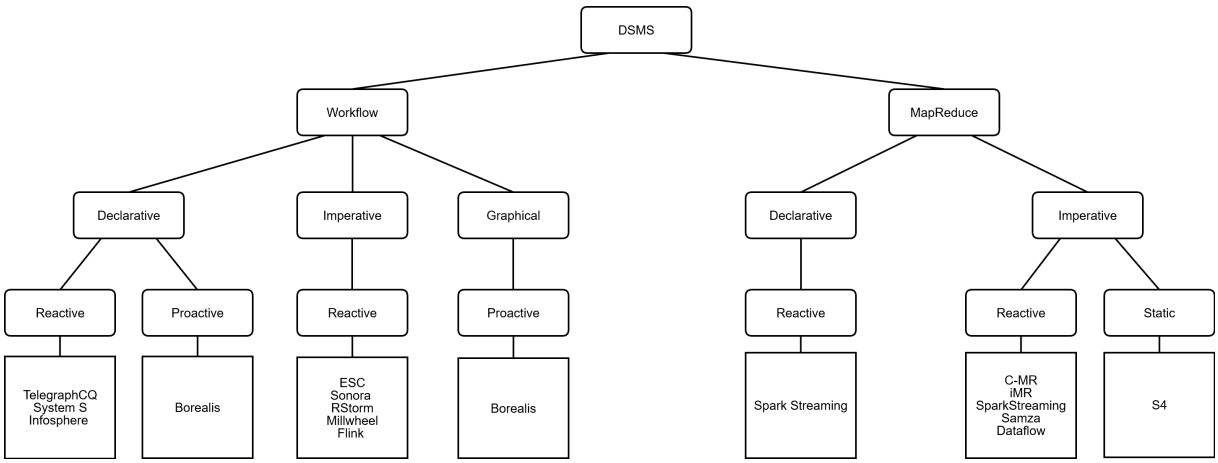


Figure 8: Classification of distributed DSMSs

To our knowledge, it appears that no solution integrates elastic mechanisms able to anticipate critical fluctuations in input rate and value distribution.

4

A generic framework for elastic stream processing: a global picture

Contents

We presented a wide variety of DSMSs and suggest a classification to highlight differences in terms of target streams and congestion management between categories of DSMSs. It shows that different optimization techniques allow DSMSs to process streams elastically. For example, let consider a continuous query Q_1 consuming an uneven erratic stream S_1. Adapting Q_1 consists in avoiding processing and network bottlenecks due to overload in input of operators while critical increases in input rate happen. Triggering operator parallelization and scheduling is an appropriate solution in that case. Moreover, as the distribution of values has an impact on processing latency too, adapting computations to fluctuations in distribution of values consists in balancing the load to avoid processing bottlenecks due to imbalance between equivalent tasks. Nevertheless, there may be no satisfying solution based on load balancing if the parallelism degree is undersized according to the input rate. It shows that the efficiency of a configuration may not only depend on a single pattern but a combination of patterns.

According to the classification of DSMSs suggested in the previous chapter, it appears, to our knowledge, that no DSMS can handle efficiently any type of streams. It highlight the difficulty to extract features of DSMSs which bring a benefit without limiting global elasticity or degrading performance.

In this section, we focus on different adaptation levels enabling elastic stream processing through the identification of their roles, which triggers are used to detect reconfiguration needs and relations between these levels. It facilitates the identification of key aspects for elastic stream processing and their impacts on execution properties (i.e., processing latency and result quality).

Finally, we present common strategies used in literature and discuss their efficiency while facing different stream types.

1 The ORACL loop

1.1 Steps for query optimization

In the previous chapter, we expose some patterns commonly used to optimize the treatment of continuous queries presented in chapter 3 Table 1 (see page 26), a pattern may modify a continuous query at different scopes: the entire workflow, a single operator or data lifecycle. Nevertheless, modifications performed at each level may have an impact on other ones. In this chapter, we aim at specifying the different steps for query optimization. We present and detail their roles and the relation order between them. Indeed, optimizing a continuous query requires to follow a specific process composed of two main steps: the logical and the physical steps. The logical step aims at defining an optimal workflow according to input streams and composed of operators potentially parallelized. The physical step aims at defining optimal assignment and customization of tasks on processing units. By customization, we mean the choice of the implementation and the selection of input data. For both logical and physical steps, we identify two substeps: the inter and the intra-operator optimization. It defines an optimization process composed of 4 steps performed in the following order:

- The logical inter-operator step aims at defining a near-optimal workflow corresponding to the submitted continuous query and according to stream fluctuations in terms of distribution of values.
- The logical intra-operator aims at defining appropriate parallelism degrees (*i.e.*, number of tasks) for each operator belonging to the workflow according to stream fluctuations in input rate.
- The physical inter-operator step looks for an optimal assignment plan of all tasks composing a workflow on available processing units. Criteria defining an optimal assignment plan depends on user objectives in terms of resource usage.
- The physical inter-operator step defines optimal implementations for tasks according to input stream properties (*e.g.*, distribution of values) and manages data lifecycle.

1.2 Adaptation levels

Through the identification of these optimization steps, we suggest an abstract framework which formalizes the function of each step and existing dependencies between them according to query and stream characteristics.

As illustrated on Figure 1, these levels define an adaptation loop, denoted *ORACL* loop, which is composed of steps *Order* (logical level and inter-operator step), *Replicate* (logical level and intra-operator step), *Assign* (physical level and inter-operator step) and *Custom Locally* (physical level and intra-operator step). Adaptation steps are revised according to a top-down model. For example, a revision at *Replicate* involves revisions of *Assign* and *Custom Locally* steps but not of *Order* step.

- The *Order* step performs an algebraic optimization of continuous queries. Let consider a continuous query Q applying a global function f on inputs to generate outputs. An optimal workflow W of Q, from logical point of view, respects the following properties:

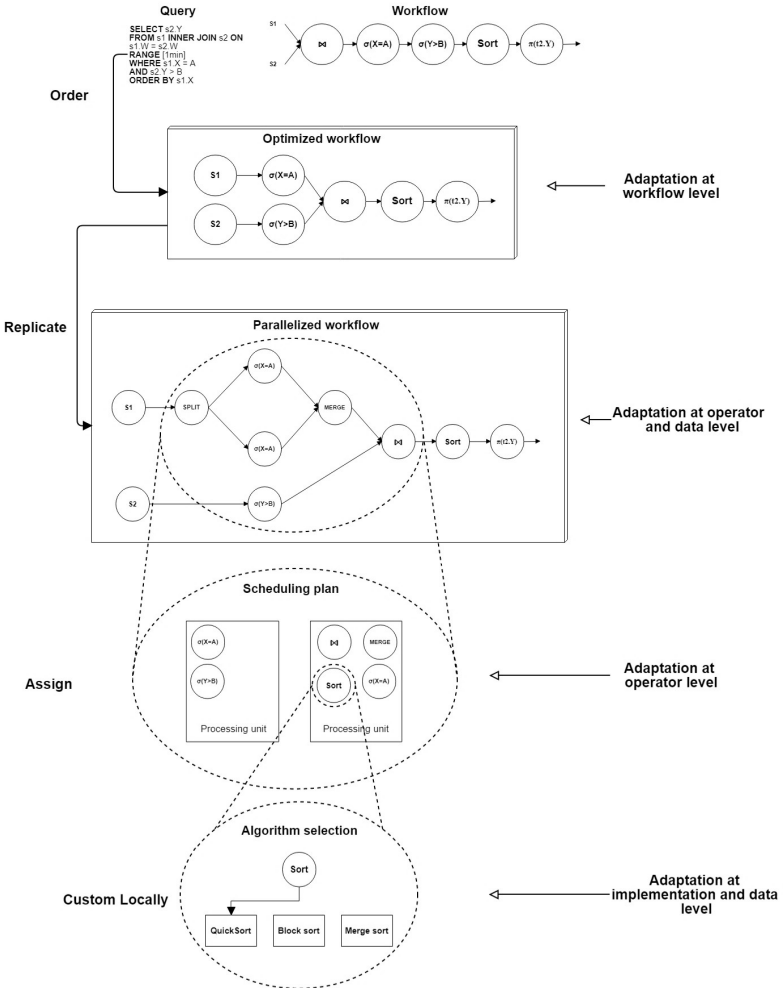


Figure 1: ORACL loop

- W applies the function f on inputs so it is logically equivalent to Q.
- The sequence of operators composing W is defined such as the workflow minimizes a cost function.

- The *Replicate* step adapts parallelism degrees of operators in order to maintain processing rates greater or equal to input rate. Let consider an operator $\mathcal{O}_i$, with an input rate r_i and an average processing latency lat_i. As lat_i can be defined in second per stream element, the inverse value $\frac{1}{lat_i}$ corresponds to the processing rate pr_i of $\mathcal{O}_i$. Maintaining pr_i greater than r_i is the major challenge of operator parallelization. Actually, if k tasks associated to an operator process stream elements in parallel, they virtually multiply pr_i by k. Nevertheless, partitioning and transmission overheads limit the benefit brought by parallelization. In addition, depending on the number of available processing units, increasing the parallelism degree of an operator over a certain threshold may create concurrency between tasks.

- The *Assign* step revises assignments of tasks on processing units, denoted scheduling plan, in order to adjust resource usage to processing requirements. As mentioned above, the definition of an optimal scheduling plan is guided by a targeted usage of resources. As most continuous queries are composed of heterogeneous operators in terms of processing requirements (*e.g.*, stateless filters and stateful joins), usage of resources on each machine depends on the subset of operators assigned on it. An optimal scheduling plan should ensure that each operator benefits of enough resources such as its processing rate is not limited by resources.

- Finally, the *Custom Locally* step aims at taking advantage of local implementations of operators in order to select an algorithm adapted to execution context. Indeed, the computational complexity of equivalent implementations of an operator may differ in term of time (*e.g.*, nested loop join or hash join). Selecting the most appropriate implementation may increase significantly the processing rate of a given operator.

1.3 Optimization strategies

As presented above, each step of the ORACL loop aims at producing a output which has specific properties. However, if the behavior of each step can be defined in a generic way, it exists several strategies to produce targeted output. The choice of the strategy depends on user objectives in terms of performance and resource usage but also on technical constraints (*e.g.*, query language used to define continuous queries). We provide a brief overview of common strategies used to implement each step and detail their impacts on outputs.

1.3.1 Improvement through Order step

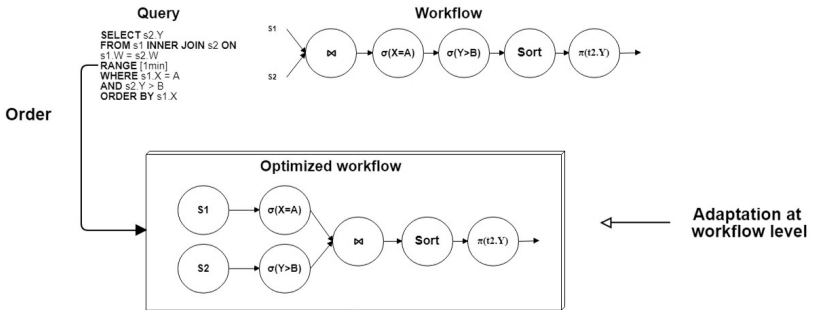


Figure 2: Order step

The Order step considers continuous queries defined by users. As presented in chapter 2 section 2.1, continuous queries can be declarative expressions (e.g., CQL expressions), se-

quences of operators declared in a high level language [Peng et al., 2015, Zaharia et al., 2012b, Yang et al., 2012, Akidau et al., 2013] or graphs of connected operators defined through a graphical interface [Abadi et al., 2003, Abadi et al., 2005]. This step is only relevant for DSMSs relying on a declarative language. Moreover, a cost function is necessary to evaluate each possible workflow associated to a continuous query.

The generation of an optimized workflow is totally dependent of the query language and possibilities of algebraic optimization. While using a declarative language, some approaches [Arasu et al., 2004, Chandrasekaran et al., 2003] use algebraic properties like traditional DBMS [Garcia-Molina, 2008]. Nevertheless, an optimal workflow cannot be defined for an entire stream as its properties cannot be anticipate at the initialization of the continuous query. Only some choices will not be contradicted at runtime (*e.g.*, projection of attributes before selective operators if there are commutative), other permutations of operators are performed at runtime according to operator properties (e.g., selectivity factor). Approaches using graphical interfaces for query definition [Abadi et al., 2003, Abadi et al., 2005] may use formal semantics of operators in order to permute safely operators. Approaches relying on imperative languages [Peng et al., 2015, Akidau et al., 2013, Noghabi et al., 2017, Akidau et al., 2015] cannot benefit of such optimization as there is no explicit semantic nor algebraic properties (*e.g.*, commutative operator) linked to operators.

1.3.2 Improvement through Replicate step

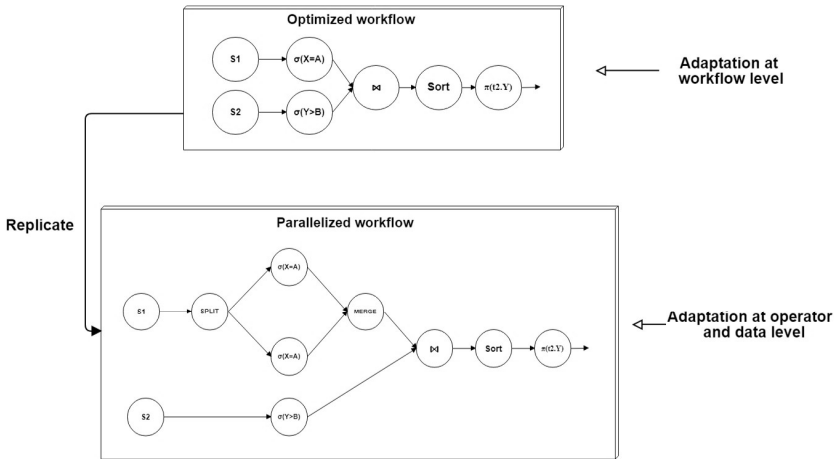


Figure 3: Replicate step

Once an optimized workflow has been defined, parallelism degrees are set. Each operator of the optimized workflow is associated to a set of equivalent tasks. Stream elements are routed to tasks according to a load balancing strategy. It is worth noting that operator parallelization and load balancing are intrinsically linked to exploit fully data parallelism.

The parallelization of operators can be performed on user demand or automatically. Parallelization of operators on user demand [Xu and Peng, 2016] relies on user expertise. Automatic

parallelization of operators is triggered automatically at runtime to compensate a significant gap between input and processing rate. Some approaches [Schneider et al., 2009, Gedik et al., 2014] rely on trial-and-error algorithms to explore parallelism degrees for each operator independently of all others and associate a rank of input rates managed efficiently.

A major risk of operator parallelization is that the frequency of reconfigurations becomes too important. Indeed, creating a thread on the fly and balance partitions between new tasks involve important overheads. To limit this negative effect, authors [Gedik et al., 2008, Biem et al., 2010] suggest to create threads once and instead of deleting them when they are not necessary, threads are only put into sleep.When additional working threads are necessary, the DSMS just wake up some sleeping threads. Another major inconvenient common to most solutions is the parallelization of operator independently of the effect on downstream operators. Indeed, modifying the parallelism degree of an operator may have an impact on its processing rate and throughput. Missing this relation between operators fixed during *Order* step may cause instability due to contradictory reconfiguration.

Once parallelism degrees have been set for each operator, it is necessary to define a strategy for load balancing between tasks. As presented in chapter 3 section 1, maintaining a balance between tasks of an operator is crucial to benefit completely from data parallelism. For example, let consider an operator $\mathcal{O}_i$ divided into tasks $\mathcal{T}_1$ to $\mathcal{T}_n$. During a period of time Δ, each task $\mathcal{T}_j$ receives in input a partition $\mathcal{P}_j$ of global inputs $\mathcal{P}$ which may differ from other partitions in term of volume and distribution of values. According to this and as all tasks apply the function corresponding to $\mathcal{O}_i$, overall processing times may be different for each task. So, the global processing time of $\mathcal{P}$ by $\mathcal{O}_i$ is at least the greatest of all processing times of its tasks. In this context, it is crucial to minimize imbalance in term of processing time between tasks.

To do so, we distinguish three categories of load balancing strategies:

- Round-Robin strategies [Abadi et al., 2003, Abadi et al., 2005, Chandrasekaran et al., 2003, Peng et al., 2015] distribute fair numbers of stream elements between tasks. This strategy is efficient to balance load under two assumptions. First, the input stream is even so partitions of same size involve theoretically same processing times. Secondly, all tasks has same available resources (CPU, RAM) to process stream elements.

- Key-based strategies [Neumeyer et al., 2010, Akidau et al., 2013, Akidau et al., 2015] associate a key set to each task and route stream elements accordingly. It assumes that input streams are even and key distribution is uniform over time. To remove the uniformity constraint, authors in [Rivetti et al., 2015] suggest an approach which builds balanced key groups associated to each task according to key distribution at runtime. This approach serves as guarantee that key grouping maintains balanced partitions in term of volume over time.

- Load-aware strategies aim at balancing processing times between tasks. In [Rivetti et al., 2016], a load balancing strategy based on *sketches* associates average processing times to encountered keys in order to route dynamically stream elements on tasks able to process them with shortest delays. This solution has the advantage to be applicable on both even and uneven streams with a reduced footprint on memory.

1.3.3 Improvement through Assign step

The definition of the scheduling plan is guided by an objective function describing an optimized usage of resources. We distinguish three main objective functions for operator scheduling:

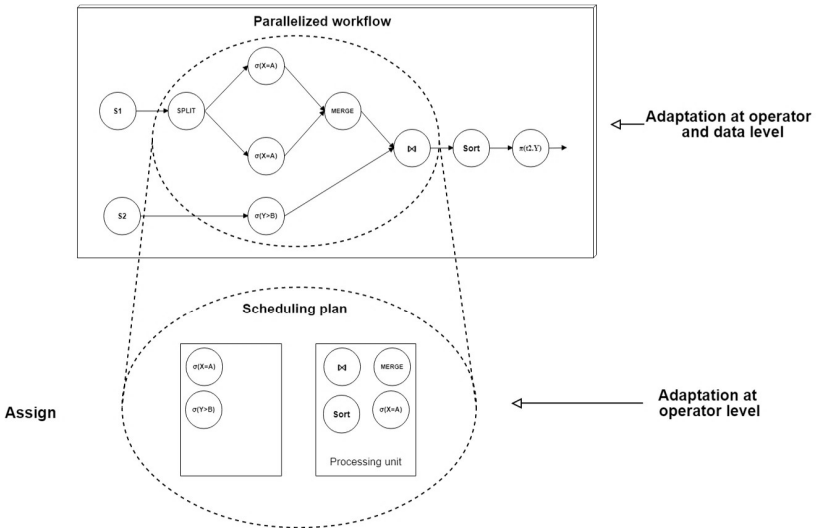


Figure 4: Assign step

- The first objective function $f_{equality}$ can be denoted as *equality-aware*. They aim at balancing processing requirements evenly between all processing units. More formally, let consider n processing units $\mathcal{M}_1$, $\mathcal{M}_2$,..., $\mathcal{M}_n$ described by resource usages $\mathcal{U}_1, \mathcal{U}_2,..., \mathcal{U}_n$. A optimal scheduling plan assigns operators such as:

$$f_{equality} = argmin \left(\sqrt{\frac{1}{n} \left(\sum_{i=1}^{n} \mathcal{U}_i^2 \right) - \bar{\mathcal{U}}^2} \right) \tag{1}$$

  where $\bar{\mathcal{U}}$ is the mean value of all resource usages. It corresponds to minimize standard deviation of resource usage among processing units. Spreading computations uniformly over processing units has the advantage to spread the impact of an overload over all processing units. Commonly, this strategy is implemented through a Round-Robin distribution [Xu et al., 2014, Zaharia et al., 2012b] of operators on slots or more generally processing units. Nevertheless, it assumes that all operators require same resources which may be wrong in real-world applications.

- Traffic-aware objective function $f_{traffic}$ aims at minimizing the global network traffic. As presented in [Xu et al., 2014], transmitting important volumes of data over network has a significant impact on global processing latency. Indeed, considering a parallelized workflow, edges define transmissions of stream elements between operators, denoted inner-streams. These inner streams can be supported through shared memory or network involving serialization/deserialization and transmission overheads. Assigning operators on processing units such heaviest inner streams, in term of volume, are supported by shared memory improves significantly overall latency and throughput. Formally, considering inner streams $\mathcal{S}_1$, $\mathcal{S}_2$,..., $\mathcal{S}_n$ belonging to a parallelized workflow, an optimal scheduling plan is defined as follow:

$$f_{traffic} = argmin\left(\sum_{i=1}^{n} |\mathcal{S}_i|\right) \quad (2)$$

where $|\mathcal{S}_i|$ is the volume of stream elements transmitted through the inner stream $\mathcal{S}_i$. A side-effect of this strategy is to concentrate consecutive operators on same processing units. In comparison to equality-aware strategy, it limits significantly the risk of network bottlenecks as heaviest inner-streams are supported by shared memory but it increases the risk of processing bottlenecks as operators manipulating greatest volumes of data are assigned on same resource.

- Resource-aware objective function $f_{resource}$ aims at minimizing combined usage of all resources (i.e., CPU, memory and bandwidth). This approach extends the traffic-aware strategy through the consideration of CPU and memory constraints. In [Peng et al., 2015], authors suggest to consider static CPU and memory requirements for each operator and to assign operators such as network is minimized and each operator meets its resource requirements. It corresponds to a multi-dimensional multi-choice knapsack problem which is solvable through a heuristic algorithm with a polynomial complexity. The main difference with $f_{traffic}$ is the consideration of explicit resource requirements which bound the minimal network traffic.

1.3.4 Improvement through Custom Locally step

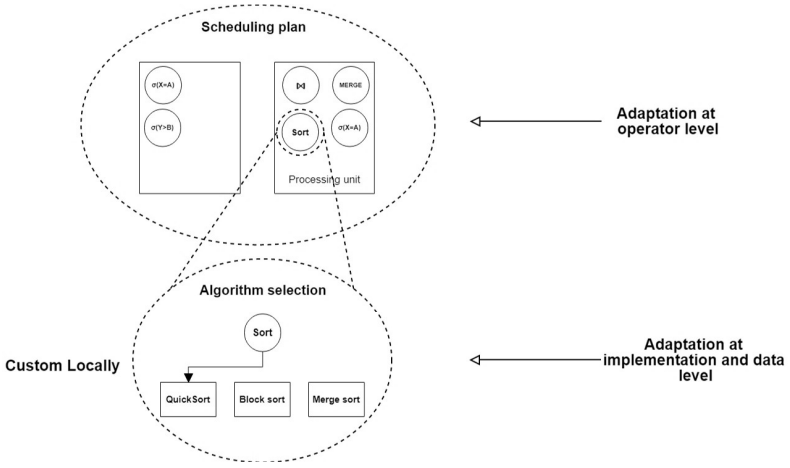


Figure 5: Custom Locally step

As presented in section chapter 3 1, this step applies algorithm selection to adapt operator implementation to inputs. Of course, this step is relevant only for DSMSs offering many implementations per operator. It mainly corresponds to DSMSs based on a declarative language. The choice of the algorithm may have a significant impact on processing latency. The example of join algorithms is the most representative as equivalent implementations have different

time complexity. Borealis [Abadi et al., 2005] takes advantage of its algebra to provide different implementations of its predefined operators. Each processing unit is provisioned with all implementations and operators select at runtime which one fits the best to execution context.

In [Welsh et al., 2001], authors give a different semantic to algorithm selection as it corresponds to a controlled load shedding. Indeed, in case of overload, operators managed by SEDA can apply a degraded service. It corresponds to a implementation of the service integrating a sampling mechanism to reduce the workload.

An other load shedding strategy has been introduced in [Babcock et al., 2004] and developed in [Tatbul et al., 2007]. It relies on the introduction of shedders in workflows composed of aggregation operators (*e.g.*, sum or count). The aimed shedding ratio of the workflow, i.e. the ratio between the input rate before and after shedding, is used to compute where shedders with their respective shedding ratios should be placed such as the relative error due to shedding does not exceed a maximal threshold.

2 Orchestration of optimization

2.1 Adaptation triggers

To enable dynamic adaptation of continuous queries, a DSMS needs a monitoring module which observes continuously relevant metrics and optionally a set of parameters which discriminate normal execution of continuous queries. We describe here metrics observed at each level and optional parameters used to trigger each step of the ORACL loop.

2.1.1 Order step

As mentioned above, the Order step aims at optimizing continuous queries according to algebraic properties of operators. It corresponds to dynamic operator reordering (see 3 section 1). Reordering is based on evolution of selectivity factors. For reminder, considering two commutative operators $\mathcal{O}_1$ and $\mathcal{O}_2$, if $\mathcal{O}_2$ is more selective than $\mathcal{O}_1$ and is executed downstream, permuting $\mathcal{O}_1$ and $\mathcal{O}_2$ reduces the volume of data exchanged between these operators without modifying the global semantic of the query. In addition to selectivity factors and volumes of data, DSMSs need algebraic properties of all operators composing the workflow.

2.1.2 Replicate step

Detecting a need of operator parallelization relies on a continuous monitoring of input and processing rates. As explained above, modifying the number of tasks associated to an operator has an impact on its global processing rate. Adjusting parallelism degrees allows operators to process stream elements as soon as they arrive and prevent processing bottlenecks due to an accumulation of pending stream elements. In some approaches [Schneider et al., 2009, Gedik et al., 2014], throughput is observed instead of processing rate. It relies on the assumption that selectivity factor does not vary significantly so for a given input rate, a maximal throughput could be identified and remain valid for the entire stream.

2.1.3 Assign step

Two event types can trigger reassignment of operators at runtime:

- *Overload and underload of processing units.* While a processing unit is overloaded according to some resources (CPU and RAM), it is necessary to move some operators assigned to this processing unit on others which have enough available resources. An overload means that usage of some resources have exceeded predefined thresholds. This detection could be done according to a resource monitoring at different scopes. Some DSMS like Apache Storm or Apache Flink allow users to instantiate multiple processing units on a single machines which have declared resources. Depending on the resource sharing policy between processing units (i.e., if resources are exclusive to a processing unit or not), monitoring of resources is relevant only at machine or processing unit scope.

- *Network traffic.* The appearance of network bottlenecks requires to revise the scheduling plan [Xu et al., 2014] in order to prevent an important degradation of the overall latency. Such bottlenecks are detected through an observation of inner-stream rates coupled with the current scheduling plan. For each inner-stream supported by network interface, if the rate is limited by the bandwidth, assignments of associated operators should be revised. More generally, authors show in [Xu et al., 2014] the interest to keep dynamically heaviest inner-streams on same machines.

2.1.4 Custom Locally step

Like the Order step, the Custom Locally step is only relevant for DSMSs based on a declarative language. Indeed, changing the local implementation of an operator depends on both inputs and implementation properties. For example, considering a common operator like join, two frequent implementations are nested-loop join and hash-join. Depending on the ratio between cardinalities of driving and probed relations, nested-loop implementation may be faster or slower than hash-join implementation. So, while changing the algorithm, specific metrics may be required according to implementation properties.

If implementations associated to an operator only suggest more or less degraded versions of the same algorithm (i.e., versions integrating a sampling or load shedding policy), selecting an other implementation at runtime relies on the ratio between input and processing rate. It is closed from the triggering condition for operator parallelization but it is commonly used when operator parallelization is not applicable (*e.g.*, maximal usage of resources or absence of parallelization support at runtime).

2.2 Challenges and dependencies between adaptation levels

As presented on Figure 1, there is an execution order between steps . Considering an adaptation level, it results that choices performed at previous step have an impact on optimization possibilities. We present here dependencies be adaptation levels through a generic description of each intermediate output.

In output of the *Order* step, an optimal workflow is defined. It fixes data transmission between operators which has an impact on data volumes transferred during processing. Depending on these transmissions, more or less important volumes must be exchanged to execute the query defined by a user. Generating a non-optimal workflow in output of this step may cause overloads on inner-streams which must be managed during next steps despite they could be avoided through algebraic optimization.

This optimized workflow is used by the *Replicate* step to generate an optimal and parallelized workflow. It is composed all operators to assign on processing units. It fixes the parallelism degree for each operator which bounds theoretical processing rates. In addition, for parallel

areas, the load balancing has been defined and should compensate fluctuations in distribution of values while processing uneven streams. So, choices made during this phase define which fluctuations in input rate and distribution of values can be handled without reconfiguration. According to the current input rate, underestimating parallelism degrees for some operators may lead to processing bottlenecks which degrade the global latency. The single solution to manage an overload at fixed parallelism degree is load shedding. So, it requires to degrade result quality to maintain acceptable latency according to user constraints. At the opposite, overestimating parallelism degrees add more operators to schedule without benefit in terms of performance. Moreover, it degrades global latency because of partitioning and routing overheads.

According to a set of available processing units, a scheduling plan assigning each operator of the parallelized workflow on processing units is generated. It defines the set of active processing units and the network traffic between them. In case of effective overload (*e.g.*, CPU overload) or deviation from the optimal scheduling plan, the *Assign* step is revised to find to avoid node failure. In the case of heterogeneous processing units in terms of available resources (*e.g.*, processing units shared by multiple continuous queries) or available implementations of operators, assignments restricts opportunities of local customization.

Finally, when all operators are assigned, the *Custom Locally* step associated them to implementations. Stream elements can be routed to operators and computed according to selected implementations.

To sum up, the management of fluctuations in input rate and value distribution can be done at different steps. Nevertheless, according to the order on adaptation levels, optimizing the execution of a continuous query early offers favors performance and result quality.

3 Discussion

The ORACL loop gathers key aspects for elastic stream processing and order them into steps. It highlights possibilities of elastic optimization while processing streams and points impacts of each step on global execution. For example, the Replicate step sets parallelism degrees of operators and load balancing strategy between tasks. It has a defines average processing rates of operators which have a direct impact on maximal input rate operators are able to absorb. Moreover, it shows that the global performance of a DSMS is the result of a sequence of optimization decisions. To obtain a full elastic stream processing, this sequence should be revised dynamically to maintain a near-optimal configuration according to fluctuations in input rate and distribution of values.

Nevertheless, the ORACL loop is not fully applicable to any DSMS to enable a full elastic stream processing. For example, DSMS based on imperative definition languages cannot perform the Order step as operators are user-defined and do not have explicit algebraic properties (*e.g.*, commutativity). In consequence, such DSMS rely on user expertise for the definition of optimal workflows. Moreover, algorithm selection cannot be performed during Custom step due to the absence of different implementations. It limits significantly optimization at query and operator scopes.

Concerning Assign step, several works [Aniello et al., 2013, Xu et al., 2014, Peng et al., 2015] suggested solutions using heuristic algorithm to obtain near-optimal scheduling plans according to an objective function as discussed above. The definition of a scheduling plan is mainly offline [Aniello et al., 2013, Peng et al., 2015] and takes as inputs a set of operators, a set of available processing units and optional constraints specified by users. Then, this initial scheduling plan may be revised dynamically through migrations of operators at runtime to remove machine

overloads when necessary [Xu et al., 2014].

It appears that Replicate step lacks strategies able to reconfigure efficiently parallelism degrees of operators at runtime. In [Xu and Peng, 2016], a solution named Stela is able to identify critical operators in a workflow and according to available resources, adjust parallelism degrees in order to maximize global throughput. Nevertheless, this solution relies completely on user expertise and reactivity to add or delete resources when necessary. In several cases, end users cannot have such control on execution (*e.g.*, execution of continuous queries on the cloud through a service provider). In [Schneider et al., 2009, Gedik et al., 2014], authors suggest solutions enabling automatic parallelization of operators at runtime over System S [Gedik et al., 2008]. Their approaches relies on processing elements (see 3 section 3) to modify parallelism degree of each operator independently of all others. The parallelization strategy relies on reinforcement learning based only on input volumes to build a specific knowledge base for each processing element without consideration for modification performed upstream. So, the global convergence time may be infinite in presence of erratic streams. Moreover, as parallelism degrees are exclusively associated to ranges of input rates, uneven streams may extend considerably the duration of the learning phase. Finally, all solutions for automatic parallelization estimate appropriate parallelism degrees according to a stable input stream, *i.e.*, steady streams like GPS signals or temperature sensors in a smart building context. It means that in the case of an input stream increasing quickly, reconfiguration of parallelism degrees will be frequent and involve important overheads. Identifying evolution trend of input stream may avoid these overheads in most cases. Moreover, adapting the parallelism degree of an operator without taking upstream operators into account degrades the consistency and the stability of the DSMS. For example, while observing a significant increase of input rate upstream a costly operator (*e.g.*, join) it is interesting to adapt its parallelism degree in a proactive way so the impact of the overload is soften by the quick adaptation.

Concerning load balancing, load-aware strategy presented in [Rivetti et al., 2016] builds balanced stream partitions between tasks based on effective processing latency. So, it is independent of stream type according to distribution of values and application (i.e., even or uneven).

5

Preventive auto-parallelization approach for elastic stream processing

Contents

Through the presentation of the ORACL loop and bibliography, we highlighted the need of automatic, reactive and accurate strategies for operator parallelization. Once on user demand approaches require user presence and expertise, the efficiency automatic approaches based on reinforcement learning are limited by stream properties in terms of fluctuations in input rate and distribution of values. To tackle this issue, we suggest an auto-parallelization strategy, named AUTOSCALE, which prevents operator congestion and limit degradation of result quality. This approach relies on an automatic and dynamic adaptation of resource consumption for each continuous query. This solution takes advantage of i) a metric estimating the activity level of operators in the near future ii) the AUTOSCALE approach which evaluates the need to modify parallelism degrees at operator and workflow scope.

1 Execution context

1.1 Assumptions

According to the execution context presented in chapter 2 section 2.2, we assume that there are enough available resources to process all queries (**H1**).

We consider that the DSMS manages state migration when some tasks of operators are added or deleted (**H2**). Otherwise, it requires to manage *states* [Hirzel et al., 2014] as they allow the system to distinguish stream elements waiting in pending queues from ones being processed and waiting for the completion a current computation window. This management is out of the scope of this chapter.

Concerning the execution of each query, we consider that all operators can be processed in parallel by multiple tasks and scheduled potentially on different machines. Nevertheless, the global incoming load is divided evenly between tasks applying a same operator (**H3**).

These tasks are assigned on machines according to a scheduling strategy. We assume that the scheduling strategy revises periodically assignments of operators. Moreover, this strategy assigns at most one task for each operator on a given processing unit (**H4**) so there is no concurrency on resource usage (CPU,RAM) between two tasks of a same operator.

Finally, the auto-parallelization strategy is integrated to a system processing streams elastically according to the ORACL loop (see chapter 4 section 1). So, modifications of operator ordering and scheduling are performed respectively before and after any modification of parallelism degree (**H5**).

1.2 Challenges

Each task applies a function defined by a user on each stream element in input. Depending on the time complexity of its function and available CPU, an operator can process, in average, a certain number of items per time unit. This number is called the *capacity* of the task. This capacity limits the input rate an operator can handle. According to his, a congestion may happen when the input rate is greater than the capacity. In order to limit the impact of a critical input rate, a solution consists in modifying the parallelism degree of the operator in order to distribute the incoming load between more tasks.

In order to prevent congestion, a DSMS should be able to detect when the rate in entry of an operator reaches or exceeds its capacity. Indeed, a detection based on resource consumption (CPU or memory) only allow to remedy an effective congestion. It is not satisfying because the quality of treatments is deteriorating before the system reconfigures itself. Even if no stream element is lost, the overall latency suffers from the congestion of one or many operators.

Yet, it is not easy to decide judiciously when increasing (*scale-out*) or decreasing (*scale-in*) the parallelism degree of an operator. Actually, given a task, if its input rate exceeds its capacity, the associated operator tends to congestion. But the congestion is effective only if the input rate remains equal or higher than the capacity for a significant time. Otherwise, a scale-out is triggered too early lead to the creation and assignment of one or many unnecessary tasks. Thus, it degrades the stability of the system and generates important reconfiguration overheads. It is crucial that a relevant parallelization strategy takes the stability of the system into account. Moreover, it is important that this strategy reduces the parallelism degree of underused operators. It fits global capacity to processing needs and, depending on the scheduling strategy, it allows to free unnecessary processing units which become available for other queries.

To sum up, the automatic and dynamic adaptation of capacities of operators requires that a DSMS is able to detect potential congestion before it becomes effective in most cases. Moreover, a

relevant strategy should not overreact to sudden peaks of input rate. It degrades the stability and the performance of the DSMS because of reconfiguration overheads. Finally, the system should fit capacities of operators to their processing needs in order to consume only necessary resources. The objective of an auto-parallelization strategy is then to detect when a reconfiguration is needed and to adjust accurately the parallelism degree. Finding a satisfying compromise between these issues is a major challenge for elastic stream processing. Indeed, with the growing popularity of *pay-as-you-consume* solutions (Amazon EC2, Microsoft Azure...) and the emergence of *Green IT*, it is crucial for the current generation of DSMS to take elasticity of treatments into account.

1.3 Overview of the AUTOSCALE approach

From these observations, let consider streams evolving progressively on a time interval Δ. Extracting knowledge from the recent history of stream fluctuations through time-series analysis allows to estimate appropriate parallelism degrees of operators with an accuracy depending on the regression model (see chapter 3 section 2.2). Indeed, time-series analysis presents the advantage to forecast future states of a system without an *a priori* knowledge so parallelism degrees of operators can be adapted in constant time independently of stream rates encountered previously.

In the remainder of this chapter, we suggest an auto-parallelization strategy, named AUTOSCALE based on time-series analysis on recent history of operators as presented in section2. These histories are built by a monitoring module which observes metrics describing the execution of each operator. A derived metric discriminating potential processing bottlenecks is computed from monitored ones. The computation of this metric and the interpretation of its possible values are detailed in section 3. Once a suggestion of reconfiguration is available for each operator, an algorithm checks the consistency of each suggested reconfiguration at workflow scope. Checking the consistency at workflow scope consists in identifying the set of necessary reconfigurations according to processing requirements. It aims at improving the stability of the system. Steps of this algorithm are presented in section 4 and we motivate its choice through highlighting examples. Then, we detail, in section 5 how we compute parallelism degrees according to globally consistent estimations of workloads. Finally we discuss limits and advantages of our approach in section 6 and expose an empirical study justifying the selected strategy for consistency checking.

2 Monitoring management

To identify potential congestion of operators through time-series analysis, we need to build recent histories of operators. These histories are composed of metrics describing execution states of operators. In this section, we suggest an attribute/value model to represent operators. Values are gathered over sliding *monitoring windows* presented below.

2.1 Operator model

We consider as operators, in the monitoring model, logical operators belonging to the optimized workflow (see chapter 4 section 1). Thereafter, we describe operators at any timestamp t with three metrics: input load, pending queue size and average processing latency.

- The input load corresponds to the number of received stream elements.
- Pending queue size matches the number of stream elements pending in operator queue as illustrated on Figure 1.

- The average processing latency is the average processing time per stream element without distinction for each value appearing in the stream.

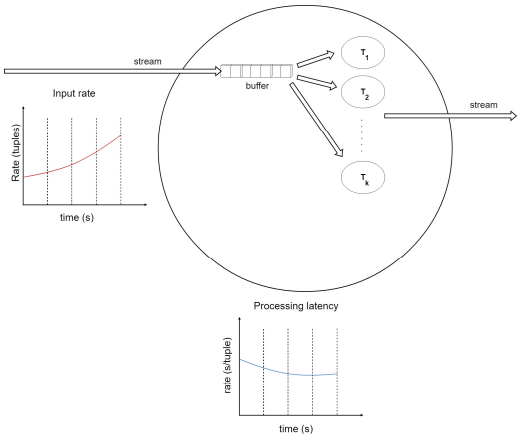


Figure 1: Operator model

For each metric, we consider the recent history. As logical operators may be applied by many tasks, each value belonging to a history is the aggregation of measured values for all tasks. For example, let consider an operator $\mathcal{O}_i$ applied by tasks T_i^1 and T_i^2, the number of stream elements in input of $\mathcal{O}_i$ at a timestamp t_0 is the sum of input stream elements of T_i^1 and T_i^2. We also consider the sum to aggregate pending queue size. Nevertheless for the average processing latency, we select the maximal value between average processing latency of T_i^1 and T_i^2. So, we consider that the operator $\mathcal{O}_i$ is limited by its slowest task, *i.e.* the task with the greatest processing latency.

2.2 Formalization

Let $\mathcal{T} = (\mathcal{O}, \mathcal{V})$ be the topology of a continuous query represented as a direct acyclic graph where $\mathcal{O}$ is the set of operators and $\mathcal{V}$ the set of streams.

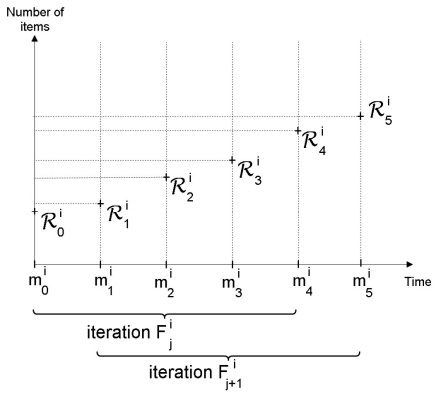


Figure 2: Monitoring window

Let $\mathcal{F}$ be a set of monitoring sliding windows $\mathcal{F}_i = \{(F_j^i)\}_{j \in \mathbb{N}^+}$. Each window $\mathcal{F}_i$ is associated

with the operator $\mathcal{O}_i$, as illustrated on Figure 2, and is composed of iterations F_j^i. Each F_j^i is defined by a duration Δ and groups measurements collected during this interval. These measurements are done according to a predefined set of timestamps $\mathcal{M}_i = \{m_1^i, m_2^i, ..., m_n^i\}_{n \in \mathbb{N}^+}$. For each operator $\mathcal{O}_i$, we collect measurements taking into account items received and processed in the interval $[m_{k-1}^i, m_k^i[$ with k=1,...,n. It is worth noting that a master process (*e.g. Nimbus for Storm or JobTracker for Hadoop*) serves as guarantee that measurements are collected synchronously on each processing unit.

In order to compute relevant metrics from these monitored values, some constraints must be considered:

- The interval $[m_{k-1}^i, m_k^i[$ should be greater than the time required to pre-process and store measurements in a standard database management system, which is around a second. It reduces redundant measurements and massive monitoring overheads.
- The duration Δ defining the size of the monitoring should be greater than all processing window sizes of stateful operators belonging to $\mathcal{T}$. It ensures that all metrics presented in the remainder of this section can be computed for both stateless and stateful operators. Indeed, if a stateful operator computes results during a duration greater than Δ, it is impossible to analyze multiple measurements within a single window.
- A *grace* period must be considered after each reconfiguration triggered by AUTOSCALE. Variations in input rates due to system stabilization are not considered during this period of time. In the remainder of this chapter, we consider a grace period of Δ after each reconfiguration.

Let $\mathcal{R}^i$ be the set, potentially infinite, of stream elements received by operator $\mathcal{O}_i$. We consider $\mathcal{R}^{i,j}$ as the subset of stream elements received by $\mathcal{O}_i$ during the iteration F_j^i, and $\mathcal{R}_k^i$ the subset of elements received between $[m_{k-1}^i, m_k^i[$. In the example presented on Figure 3, $\mathcal{R}^{i,j}$ is the sum of measurements $\mathcal{R}_0^i$ to $\mathcal{R}_4^i$, and $\mathcal{R}^{i,j+1}$ the sum of measurements $\mathcal{R}_1^i$ to $\mathcal{R}_5^i$.

In addition to the number of stream elements received, we collect the processing latency per stream element of the operator observed during F_j^i, denoted $Lat_{F_j^i}$. This does not include the time an item may spend in pending queues.

3 Detection of reconfiguration needs

Now, that we have detailed how each operator belonging to a workflow is monitored, we focus on how reconfiguration needs and opportunities are detected from observed metrics. An auto-parallelization strategy prevents congestion without degradation in result quality only if it can anticipate appearance of processing bottlenecks. According to this requirement, it is necessary to estimate input load and capacity in near future for each operator. On the contrary, if the input load decreases significantly, the capacity should be decreased accordingly to avoid waste of resources. In the remainder, we detail how AUTOSCALE detects reconfiguration needs through estimations of input load and capacity in near future.

3.1 Estimation of input load in near future

Let consider an operator $\mathcal{O}_i$ observed at the start of an iteration F_j^i of the monitoring window $\mathcal{F}_i$. The global workload of $\mathcal{O}_i$ during F_j^i, denoted $Input_j^i$ corresponds to the number of stream

elements received during F_j^i added to the total number of stream elements pending in input queues of tasks applying $\mathcal{O}_i$ at the end of the previous iteration F_{j-1}^i of the monitoring window. This workload is expressed in formula (1).

$$Input_j^i = |\mathcal{R}^{i,j}| + pending_{F_{j-1}^i} \tag{1}$$

As illustrated on figure 3, the effective number of received elements $|\mathcal{R}^{i,j}|$ cannot be computed exactly before the end of F_j^i. Nevertheless, to anticipate an eventual congestion of $\mathcal{O}_i$ during F_j^i, it is possible to approximate $|\mathcal{R}^{i,j}|$ at the end of F_{j-1}^i.

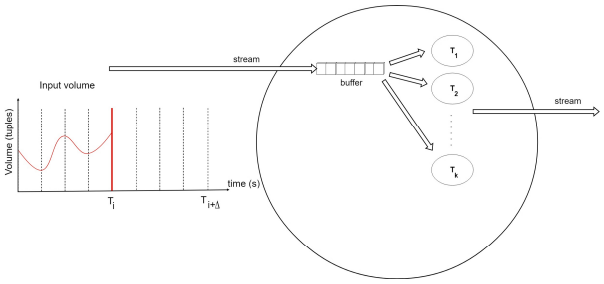


Figure 3: Recent history F_{j-1}^i at the start of iteration F_j^i

The estimation of $|\mathcal{R}^{i,j}|$ can be performed through a linear regression model based on measurements collected during F_{j-1}^i. Let f_{j-1}^i be the affine function computed by linear regression as illustrated on figure 4. For each timestamp m_k^i corresponding to a future measurement in F_j^i, f_{j-1}^i is applied to estimate the number of received stream elements $\mathcal{R}_k^i$. So, the total number of stream elements received during F_j^i is estimated as in formula (2) by $|Estim\mathcal{R}^{i,j}|$:

$$|Estim\mathcal{R}^{i,j}| = \sum_{m_k^i \in \mathcal{M}_i} \lceil f_{j-1}^i(m_k^i) \rceil \tag{2}$$

It is worth noting that some regression models have been tested without improving significantly the accuracy of estimations.

With the knowledge of $|Estim\mathcal{R}^{i,j}|$, it is possible to estimate the expected workload during F_j^i at the end of F_{j-1}^i according to formula (3).

$$EstimInput_{F_j^i} = |Estim\mathcal{R}^{i,j}| + pending_{F_{j-1}^i} \tag{3}$$

According to formulas (1) and (2), the workload of each operator can be estimated with a maximal anticipation Δ, where Δ is the size of each iteration of the monitoring window. The estimation of the workload is updated at each end of an iteration which corresponds to the acquisition of a new measurement. It allows to keep estimations close from effective workloads. To optimize the computation of the estimated workload, sums used for regression are updated incrementally instead of recalculating aggregates common to overlapping iterations of the monitoring window.

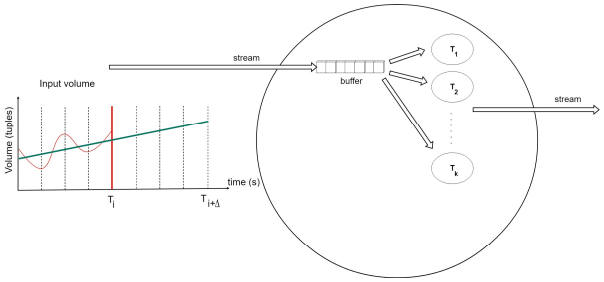


Figure 4: Estimated number of received stream elements over F_j^i

3.2 Estimation of processing capacity in near future

Now that an estimation of future workload has been computed for each operator, it is necessary to estimate the capacity in near future for each of them. Thus, it will be possible to determine if the capacity of an operator fits the expected workload during next iteration F_j^i. To do so, we aim at estimating the number of stream elements each operator should be able to process during F_j^i. The processing capacity of an operator $\mathcal{O}_i$ depends on the average number of stream elements $\mathcal{O}_i$ has been able to process during the previous iteration F_{j-1}^i. This number can be computed from the average latency per stream element of $\mathcal{O}_i$ during F_{j-1}^i, denoted $Lat_{F_{j-1}^i}$. Thus the capacity $Capacity_{F_{j-1}^i}$ is computed according to the formula (4).

$$Capacity_{F_{j-1}^i} = \frac{1}{Lat_{F_{j-1}^i}} \times \Delta \times deg_{j-1}(\mathcal{O}_i) \tag{4}$$

As the processing latency may fluctuate over time, we take the covariance between $Capacity_{F_{j-1}^i}$ and previous iterations into account to estimate the expected capacity during F_j^i. The estimated capacity of $\mathcal{O}_i$ is computed as follow:

$$EstimCapacity_{F_j^i} = Capacity_{F_{j-1}^i} + \epsilon_i \tag{5}$$

where ϵ_i is the covariance between the capacity during F_{j-1}^i and the capacities observed during the previous iteration of the monitoring window.

3.3 Identification of potential congestion at operator scope

We have presented methods to estimate input workload and processing capacity in near future at operator scope. To identify if a modification of parallelism degree will be beneficial to an operator, it is necessary to evaluate if the operator will be able to process its estimated workload. Intuitively, for a given operator $\mathcal{O}_i$, if the estimated workload exceeds the capacity, $\mathcal{O}_i$ will accumulate stream elements on its pending queue and may become a processing bottleneck. At the contrary, if the capacity exceeds significantly the estimated workload, it means that the parallelism degree of $\mathcal{O}_i$ is oversized for future processing requirements and could be reduced to save resources.

3.3.1 Local estimation of activity level

To represent the balance between estimated input load and capacity, we suggest the notion of activity level.

Definition 16. (Activity level) *Let consider an operator $\mathcal{O}_i$ applied by a set of tasks observed at the start of the iteration F_j^i of a monitoring window. The activity level of $\mathcal{O}_i$ is the ratio between estimations of its input load and its processing capacity on F_j^i.*

Applied at operator, or *local*, scope, the *Local Activity Level*, denoted LAL, is defined in formula (6).

$$LAL_{F_j^i} = \frac{EstimInput_{F_j^i}}{EstimCapacity_{F_j^i}} \tag{6}$$

The LAL is said local because it relies exclusively on operator history without consideration for upstream operators. It is computed at the end of each iteration of the monitoring window independently for all operators belonging to a workflow. From the value of the LAL, a modification of parallelism degree can be suggested to fit current capacity to estimated workload in near future.

3.3.2 Identification of activity states

Let θ_{min} and θ_{max} be two thresholds delimiting respectively a low and a high activity level, with θ_{min}, $\theta_{max} \in]0;1]$. For a given operator $\mathcal{O}_i$, modifications of parallelism degrees are suggested according to the following policy:

- If $LAL_{F_j^i} \leq \theta_{min}$, the local activity of the operator is 'low' because operator capacity is at least $\frac{1}{\theta_{min}}$ greater than the estimated workload $EstimInput_{F_j^i}$.
- If $\theta_{min} < LAL_{F_j^i} \leq \theta_{max}$, the local activity of the operator is 'medium' because the operator is able to process all items during F_j^i but $EstimInput_{F_j^i}$ is greater than $\theta_{min} EstimCapacity_{F_j^i}$.
- If $\theta_{max} < LAL_{F_j^i} \leq 1$, the local activity of the operator is 'high' because the operator has just the capacity to process stream elements waiting to be processed during F_j^i.
- If $LAL_{F_j^i} > 1$, the local activity of the operator is then 'critical' because the operator is not able to process $EstimInput_{F_j^i}$ with its estimated capacity $EstimCapacity_{F_j^i}$ during F_j^i.

4 Consistency at workflow scope

We have determined processing requirements of each operator in near future according to their own histories. From activity states recommended locally for each operator, we could trigger scale-in and scale-out without considering the upstream and downstream operators. Nevertheless, when the activity level is critical, a modification of the parallelism degree of an operator may affect the throughput of the operator. By consequence the input rate of downstream operators may be affect accordingly. If parallelism degrees of multiple operators are changed simultaneously, it may lead to inconsistent reconfigurations while considering a workflow in its entirety.

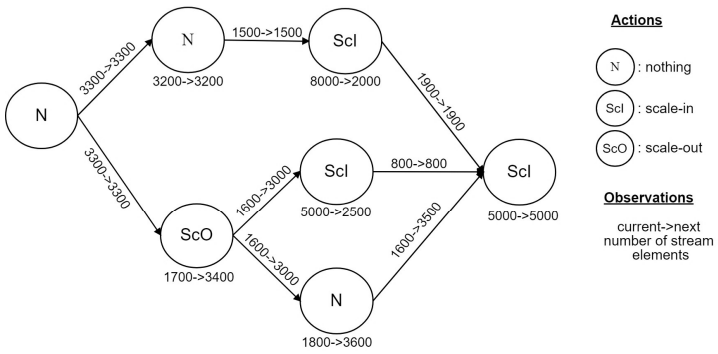


Figure 5: An example of inconsistent reconfiguration at workflow scope

For example, let consider a workflow W and let suppose that the system applies suggestions of reconfiguration computed locally.

As illustrated on figure 5, performing a scale-out increases the throughput of an operator so the input load of downstream operators will increase accordingly after reconfiguration. Considering that modifications of parallelism degrees are performed simultaneously for all operators in order to minimize reconfiguration overheads, three inconsistent cases may occur after reconfiguration:

- If a scale-out is performed upstream and a scale-in is performed locally (see figure 6), it may cause a congestion because the input rate increases while the capacity decreases. Of course, it depends on amplitudes of scale-out and scale-in.

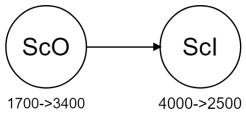


Figure 6: Potential inconsistent case 1

- In the same logic, if a scale-out is performed upstream and nothing is changed locally (see figure 7), the operator may have a high or critical activity level in near future and it is necessary to reevaluate if the current parallelism degree will handle the increasing load.

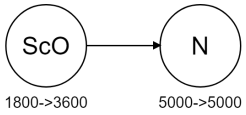


Figure 7: Potential inconsistent case 2

- Finally, when scale-out is performed upstream and locally (see figure 8), and it is necessary to reevaluate if the expected capacity after reconfiguration will be appropriate to handle the input load. Indeed, a scale-out suggested locally may be based on a undersized estimation of local input load. By consequence, it may involve an additional scale-out later which degrades the stability of the system.

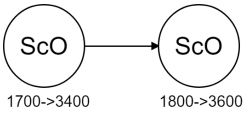


Figure 8: Potential inconsistent case 3

It appears necessary to consider dependencies between operators while computing the set of reconfigurations to perform. Nevertheless, it requires to evaluate the impact of a reconfiguration on downstream operators.

4.1 Construction of the instantaneous graph of local activities

To tackle the consistency issue at workflow scope, we suggest an *instantaneous graph of local activities* (IGLA) which allows to analyze the activity of an operator and take activities of upstream operators into account. The IGLA sums up metrics necessary to evaluate the impact of a reconfiguration on downstream operators.

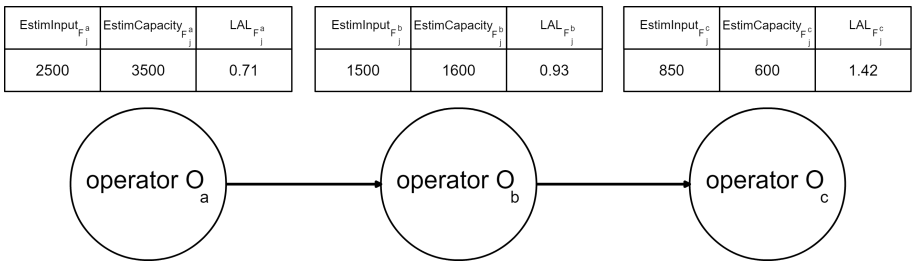


Figure 9: An example of IGLA

As illustrated on figure 9, an IGLA is an attributed graph where each vertex corresponds to an operator of the workflow to reconfigure. Each vertex is associated to a vector of attribute/value pairs where each attribute corresponds to a metric computed locally as shown on figure 9. Indeed, from local estimations, we can infer which modifications should be applied and detect inconsistencies as explained above.

4.2 Evaluation of reconfiguration impact

Let consider a monitored operator $\mathcal{O}_i$ during an iteration F_j^i. We have measured the total number of stream elements it has processed, denoted $processed_{F_j^i}$, and the total number of stream elements it has emitted during F_j^i, denoted $output_{F_j^i}$. Considering that $\mathcal{O}_i$ can be selective (a filter, a join...), we compute its selectivity factor $SF_{F_j^i}$ during the iteration F_j^i according

to formula (7). It is worth noting that $processed_{F_j^i}$ is a measurement and not a theoretical estimation like $Capacity_{F_j^i}$.

$$SF_{F_j^i} = \frac{output_{F_j^i}}{processed_{F_j^i}} \tag{7}$$

According to the estimation of incoming load $EstimInput_{F_j^i}$ and the current capacity $Capacity_{F_j^i}$ of the operator $\mathcal{O}_i$, we define the estimated number of processed elements during the next iteration, denoted $EstimProcessed_{F_j^i}$, according to formula (8).

$$EstimProcessed_{F_j^i} = min(EstimInput_{F_j^i}, Capacity_{F_j^i} \times \Delta) \tag{8}$$

Indeed, an operator can at most process the number of items corresponding to its capacity per time unit multiplied by the duration of an iteration. With this estimation and the selectivity factor $SF_{F_j^i}$, we estimate then number of items emitted by $\mathcal{O}_i$ on F_{j+1}^i, denoted $EstimOuput_{F_{j+1}^i}$ thanks to the following formula:

$$EstimOutput_{F_{j+1}^i} = EstimProcessed_{F_j^i} \times SF_{F_j^i} \tag{9}$$

According to this value, it is possible to have a complementary estimation of the incoming load of next operators. Actually, let consider a child operator $\mathcal{O}_c$ receiving its inputs from a parent operator $\mathcal{O}_p$. The value $EstimOutput_{F_{j+1}^p}$ is intrinsically different from $EstimInput_{F_j^c}$ because it is not based on items already received by $\mathcal{O}_c$ as illustrated on Figure 10. Indeed, $EstimOutput_{F_{j+1}^p}$ is computed from items received and processed by the previous operator, in this example, $\mathcal{O}_p$. Intuitively, it gives a greater anticipation of critical variations of the global input rate.

4.3 Consistency checking at workflow scope

So, still considering an operator $\mathcal{O}_c$ receiving its inputs from an operator $\mathcal{O}_p$, we have at disposal two distinct estimations of the incoming load of $\mathcal{O}_c$: its local estimation $EstimInput_{F_j^c}$ and the global estimation $EstimOutput_{F_{j+1}^p}$. The choice of the estimation to consider depends on which aspect the DSMS should favor.

Indeed, if the DSMS serves as guarantee that the capacity of each operator remains great enough to absorb its incoming load, the maximal value between $EstimInput_{F_j^c}$ and $EstimOutput_{F_{j+1}^p}$ is considered to adjust the parallelism degree of $\mathcal{O}_c$. According to available estimations and **H1**, it ensures that each operator is able to process all incoming elements. Nevertheless, it prevents to perform a scale-in until local and global estimations confirms that it does not lead to a potential congestion.

In opposition, if the DSMS aims at using only necessary resources, the minimal value between $EstimInput_{F_j^c}$ and $EstimOutput_{F_{j+1}^p}$ is used. According to that combination strategy, the DSMS decreases the capacity of operators as soon as it is locally or globally advisable. Yet, this strategy presents as drawback to degrade the stability of the system. Indeed, decreasing capacities of operators to save resources as soon as possible also means increasing them each time the incoming load increases significantly.

For both combination strategies, we consider the globally consistent estimation as the result of a function *combine* which takes both estimations as input and returns the globally consistent estimation according to the DSMS' objective.

Considering a combination strategy, we consider that the consistent estimation of input volume is the result of a function *combine*. This function takes as input the estimation $EstimInput_{F_j^c}$ computed locally and the estimation $EstimOutput_{F_{j+1}^p}$ computed upstream.

4.3.1 Global estimation of activity level

To evaluate accurately which operators should be revised, it is necessary to estimate and propagate the effect of each reconfiguration along the workflow. Thus, AUTOSCALE is able to detect if a reconfiguration recommended locally worth being triggered according to reconfigurations performed upstream. Formally, we will override the value of the $LAL_{F_j^i}$ for an operator $\mathcal{O}_i$ by an activity level taking into account estimations of inputs performed upstream. We compute this *Global Activity Level*, or GAL, according to formula (10).

$$GAL_{F_j^i} = \frac{combine(EstimInput_{F_j^i}, \sum EstimOutput_{F_{j+1}^{P_i}})}{EstimCapacity_{F_j^i}} \tag{10}$$

where $\sum EstimOutput_{F_{j+1}^{P_i}}$ is the sum of the estimated outputs of all parent operators of $\mathcal{O}_i$.

Once the LAL has been replaced by the GAL, AUTOSCALE can reconsider the activity state (low, medium, high and critical) of each operator and decide which reconfiguration should be triggered.

4.3.2 Algorithm

Algorithm 1 Global consistency checking

Require: sources, local-based IGLA
Ensure: globally consistent IGLA
  $sources \leftarrow \emptyset$;
  $checked \leftarrow \emptyset$;
  for all *source* in *sources* **do**
    $operators \leftarrow operators \cup children(source)$;
  end for
  for all $\mathcal{O}_i$ in *operators* **do**
    $current \leftarrow currentDegree(\mathcal{O}_i)$;
    if $activity(source) ==$ 'critical' $\wedge$ $unchecked(\mathcal{O}_i)$ **then**
      $EstimParentOutput \leftarrow \sum EstimOutput_{F_{j+1}^{P_i}}$;
      $EstimInput_{F_j^i} \leftarrow combine(EstimInput_{F_j^i}, EstimParentOutput)$;
      $next \leftarrow degree_j(\mathcal{O}_i)$;
      if $current > next$ **then**
        $setScaleIn(IGLA, \mathcal{O}_i, next)$;
      end if
      if $current == next$ **then**
        $setNothing(IGLA, \mathcal{O}_i, current)$;
      end if
      if $current < next$ **then**
        $setScaleOut(IGLA, \mathcal{O}_i, next)$;
      end if
      $checked \leftarrow checked \cup \{\mathcal{O}_i\}$;
    end if
  end for
  if $operators \neq \emptyset$ **then**
    $checkConsistency(operators, IGLA)$;
  end if

As presented in Algorithm 1, the consistency checking algorithm explore the IGLA from sources. Estimations of input loads computed on source operators cannot be combined with estimations computed on upstream operators as there are entries of the workflow. Then, the function *children*() look for all children operators of each source. If the function *activity*() returns that a child operator has a critical parent at local scale and the method *unchecked*() returns that it has not been checked already, we compute the globally consistent estimation of its incoming load and replace its local estimation. It propagates the effect of critical estimation to all operators processing stream elements emitted by a source. Then, we compare the current parallelism degree given by function *currentDeg*() and the adequate parallelism.

We can map each operator to a modification of its parallelism degree as presented in Table 1. This decision takes into account the global activity of a given operator and the evolution trend of its incoming load. As a reminder, the affine function f_j^i is computed with linear regression to estimate the load of an operator. This function allows to evaluate the evolution trend of the load according to its derivative value. If this value is strictly positive, the load is considered increasing. Otherwise the load is estimated as decreasing or constant.

Table 1: Decision Matrix for IGLA computation

Evolution trend of inputs / Operator activity	Decreasing or constant	Increasing
$GAL_{F_j^i} < \theta_{min}$	*scale-in*	*nothing*
$\theta_{min} \leq GAL_{F_j^i} < \theta_{max}$	*nothing*	*nothing*
$\theta_{max} \leq GAL_{F_j^i} < 1$	*nothing*	*scale-out*
$1 \leq GAL_{F_j^i}$	*scale-out*	*scale-out*

To sum up, AUTOSCALE estimates the activity at local and global scope for each operator as illustrated on Figure 10. At local scope, AUTOSCALE computes an estimation of the incoming load thanks to monitoring data on received items and pending queues. This incoming load is divided by the estimated capacity of the operator to give a value of its local activity level. To propagate local estimations to next operators, the estimated output is computed. It relies on an estimation of processed items and the selectivity factor of the operator. For children operators, this estimation is combined to their local estimation of the incoming load to help the DSMS to reach its objective as introduced above.

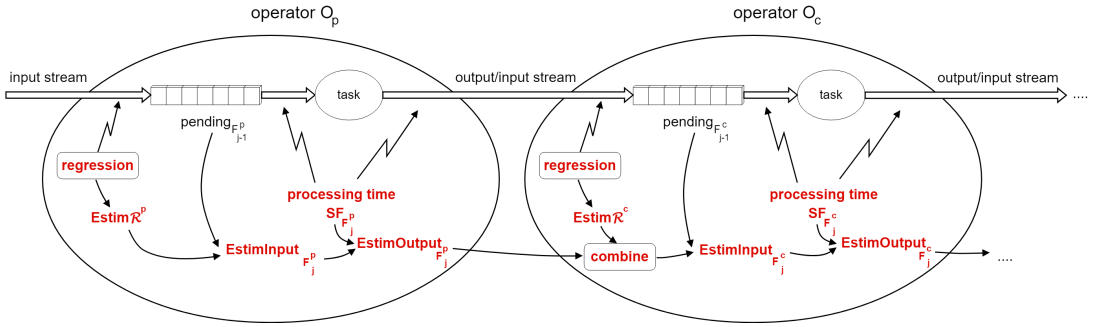


Figure 10: Estimations at local and global scope

5 Quantification of reconfiguration

For each operator requiring scale-in or scale-out, we have to evaluate an appropriate parallelism degree. As we consider that users do not have an *a priori* knowledge of stream fluctuations, the system cannot be trained on relevant input rates to build workload→parallelism degree mappings upstream. So, we have to approximate the parallelism degree under the assumption that the total capacity of an operator is strictly proportional to the number of tasks. It assumes that overheads induced by input split and merge are negligible.

Let $deg_{j-1}(\mathcal{O}_i)$ be the parallelism degree of $\mathcal{O}_i$ during the iteration F_j^i. Let $maxP_{\mathcal{O}_i}$ be the maximal parallelism degree of $\mathcal{O}_i$, we consider that its appropriate parallelism degree is defined according to formula (11). It is worth noting that we consider a maximal parallelism $maxP_{\mathcal{O}_i}$ for $\mathcal{O}_i$ because most DSMSs limit the number of tasks associated to a single operator. Moreover,

when the parallelism degree of an operator a threshold, overheads involved by data routing and potential network transmissions balance the benefit brought by the parallel execution.

$$deg_j(O_i) = \begin{cases} min(maxP_{\mathcal{O}_i}, deg_{j-1}(\mathcal{O}_i) + 1), \text{if activity is 'high'} \\ min(maxP_{\mathcal{O}_i}, \lceil deg_{j-1}(\mathcal{O}_i) \times GAL_{F_j^i} \rceil), \text{otherwise} \end{cases} \quad (11)$$

We distinguish the specific case where an operator has a high activity and an increasing input rate. Indeed, the value of $GAL_{F_j^i}$ is smaller than 1, but a scale-out is recommended (see Table 1). In this case, we simply increment the parallelism degree of the operator by 1. In any other case, AUTOSCALE considers as the appropriate parallelism degree, the smallest parallelism degree greater than the current parallelism weighted by the value of $GAL_{F_j^i}$.

6 Discussion

We have presented an auto-parallelization strategy estimating common metrics of the literature for each operator and in near future. It allows to anticipate potential congestion of operators instead of removing effective ones. In addition, we suggest an algorithm for consistency checking at workflow scope in order to avoid inconsistent reconfiguration of operators.

Nevertheless, the time complexity of this algorithm is in $\mathcal{O}(V + E)$ where V is the set of operators and E the set of streams between operators, or *inner streams*. If the workflow is composed of many operators highly connected, the computation of the global activity for each operator may add some overheads.

So, to improve the anticipation of congestion, two alternative policies to congestion may be used: a policy considering exclusively local estimations for each operator, denoted *LocalOnly* and a policy checking the consistency considering only the type of reconfiguration performed upstream, denoted *Straight* policy.

scale-in and scale-out may be decided from estimated metrics at local scope (*i.e.*, exclusively from operator history).

While the local activity has been computed, scale-in opportunities and scale-out needs can be identified as presented in Table 2.

Evolution trend in input rate \ Operator activity	Low activity	Medium activity	High activity	Critical activity
Decreasing or stable	*scale-in*	*nothing*	*nothing*	*scale-out*
Increasing	*nothing*	*nothing*	*scale-out*	*scale-out*

Table 2: Local decision matrix for reconfiguration evaluation

We can apply the straight policy for consistency checking which does not take the impact into account but only *symbolic* inconsistencies. By nominal, we mean scale-in, scale-out and nothing. After deciding locally which reconfiguration suits to an operator, the decision matrix presented in Table 3 can be used to replace inconsistent decisions. We explore the IGLA according to a breadth-first search, or BFS, from sources to final operators. Considering the global decision matrix, we identify the consistent subset of actions to perform for each workflow. For instance, if a scale-in is recommended at local scope but a scale-out has been validated upstream then the

current parallelism degree of the operator is remains unchanged. It is interesting to notice that if a scale-in intervenes upstream current operator and nothing has been recommended locally, this policy prefers maintaining the current parallelism degree than decreasing it because it presents more risks to decrease parallelism degree before local and global recommendations confirm such reconfiguration. In comparison to the consistency checking algorithm of AUTOSCALE, it saves computations for global consistency as the selectivity factor and the combine function.

Prevailing action upstream operator \ Local suggestion for operator	*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*

Table 3: Decision matrix for global consistency

Nevertheless, an issue occurs when the current operator has multiple parents with different suggestions of reconfiguration. To solve this issue, we suggest a relation order on actions. So, a scale-out prevails on a scale-in which prevails on a nothing action. To determine if a local action is globally consistent, we introduce the global decision matrix (see Table 3).

According to the relation order on possible actions, the *Straight* policy can determine the predominant action for all nodes upstream. Thus, if at most a scale-in is validated upstream the current node, the local action is validated otherwise it is replaced by a nothing or a scale-out in order to avoid foreseeable congestion in near future.

This policy increases the consistency of reconfiguration at workflow scope but it assumes that each upstream scale-out causes a significant increase of input rate which is not systematically the case. So, we suggest to enrich the IGLA with additional metrics in order to evaluate accurately the impact of each reconfiguration on downstream operators.

6.1 Empirical study of consistency checking

In order to evaluate the interest of the consistency checking algorithm integrated in AUTOSCALE, we test it in front of AUTOSCALE without consistency checking (LocalOnly) and AUTOSCALE with nominal consistency checking. The stream illustrated on Figure 11 has major increases in input rate followed by decreases to highlight adaptations performed by AUTOSCALE.

This stream is played in entry of the workflow illustrated on Figure 12. This linear workflow is composed of a source emitting the stream illustrated on Figure 11 to an operator (FastNonFilter) transmitting each stream element within a millisecond without filtering inputs. Outputs of this operator are sent to an operator (SlowNonFilterMid) applying heavy treatments without filtering. Each stream element is processed around 100ms. Outputs of the SlowNonFilter operator are consumed by an quick filter (FastFilter) which processes each input within a milliseconds but also filter its inputs. The operator FastFilter has a fixed selectivity factor of 0.1. Finally, the operator SlowNonFilterEnd has the same properties than the operator SlowNonFilterMid.

This workflow is interesting to evaluate consistency checking because the heterogeneity of processing latency and selectivity factors represent real-world complex applications. So, in front

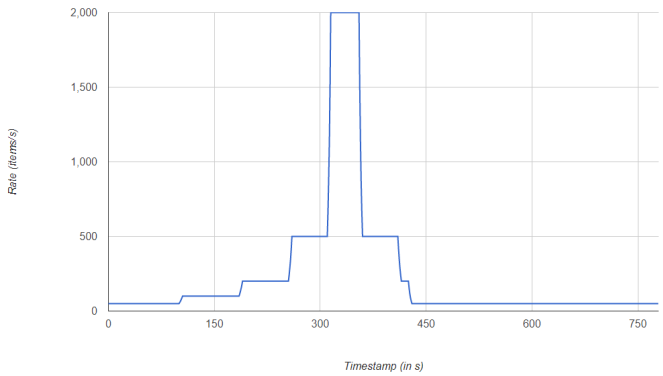


Figure 11: Stream fluctuations in input rate

of fluctuations in input rate, scale-in and scale-out of operators will not need same modification of their parallelism degrees.

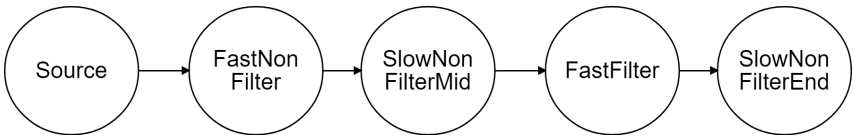


Figure 12: Heterogeneous workflow

For each strategy for consistency checking, we observe modification of parallelism degree performed for each operator, the average end-to-end latency of the workflow, the throughput and the number of stream elements processed over a predefined threshold. In this case, we set this threshold to 30 seconds.

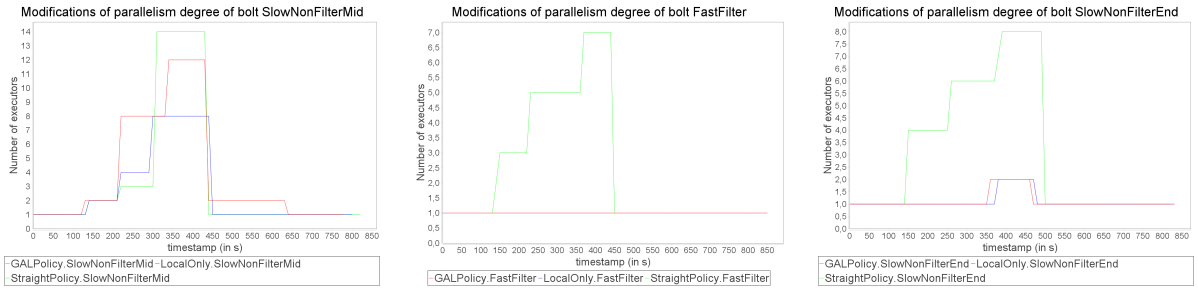


Figure 13: Modification of parallelism degree

From left to right, we can observe on Figure 13 modifications of parallelism degrees respectively for operators SlowNonFilterMid, FastFilter and SlowNonFilterEnd. When AUTOSCALE takes only local estimations into account (LocalOnly), the parallelism degree of the operator SlowNonFilterMid is increased and decreased according to fluctuations in input rate. Indeed, upstream operator FastNonFilter does not modify significantly variations in input rate. The parallelism degree of the operator FastFilter remains unchanged according to local metrics. It can be explained by the fact that the input rate is limited by the throughput of the operator

SlowNonFilterMid and the average processing latency of the FastFilter is greater or equal to the processing latency of upstream operators. Finally, the parallelism degree of the final operator SlowNonFilterEnd is increased by 1 when the stream is at its maximal rate. It does not have same modifications of parallelism degree than SlowNonFilterMid because the original input rate has been limited by SlowNonFilterMid processing rate and reduced by FastFilter which has a selectivity factor of 0.1.

When a nominal consistency checking strategy is applied (Straight policy), the operator SlowNonFilterMid is initially reconfigured according to local estimations but this reconfiguration implies a scale-out of the operator FastFilter according to the decision matrix presented in Table 3. The parallelism degree of FastFilter is adapted according to the linear projection on SlowNonFilterMid workload. It involves an additional reconfiguration which is not necessary as the maximal throughput SlowNonFilterMid can deliver with 14 tasks is smaller than the processing rate of FastFilter with a single task. So, the operator FastFilter is reconfigured each time SlowNonFilter requires a modification of parallelism degree. In the same logic, the final operator SlowNonFilterEnd is reconfigured each time FastFilter is reconfigured.

When **autoscale** uses a consistency checking strategy computing the GAL for each operator (GALPolicy), we observe that the parallelism degree of SlowNonFilterMid is increased initially according to local metrics. But, at the second reconfiguration, we note that the parallelism degree is increased by 7 tasks instead of 3 because the estimation of inputs based on upstream operators is greater than the local estimation. The same phenomenon can be noticed on the third scale-out. Then, instead of taking local estimations into account to get down to 1 task, the parallelism degree of SlowNonFilterMid is decreased to 2 before being decreased to 1 later. As recommended by local estimations the parallelism degree of SlowNonFilterEnd is increased by 1 like it was without consistency checking.

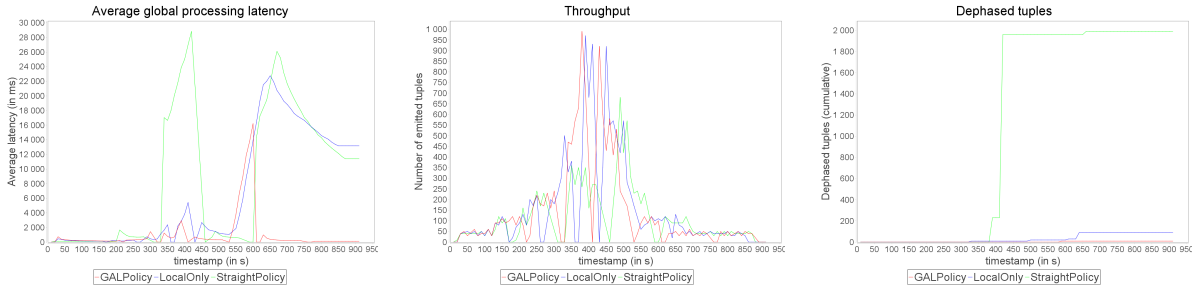


Figure 14: Impact of consistency checking strategy on performance and quality

The impact of each sequence of scale-out/in can be evaluated in terms of performance and result quality at workflow scope. The GALPolicy strategy improves both aspects compared to StraightPolicy and LocalOnly strategies. Concerning performance, GALPolicy maintains a smaller end-to-end latency because it does not trigger unnecessary scale-out like StraightPolicy and it anticipates predictable fluctuations of workload at workflow scope contrary to LocalOnly strategy. The throughput is significantly greater when the input rate reaches its maximal value compared to StraightPolicy because of less reconfiguration and a better accuracy while estimating future workload. It can also be observed from out-of-time elements happening almost only with StraightPolicy because of consecutive reconfigurations of multiple operators which delays significantly processing latency. In this testing simulation, the major improvement of GALPolicy compared to LocalOnly strategy is the combine method which avoid to underestimate processing requirements. So, the processing latency can decrease significantly with GALPolicy because

there are enough tasks to empty pending queues and process incoming stream elements without accumulating them for a long period of time.

6.2 Estimation of capacity

We justified empirically the interest of the consistency checking with GAL metric. Through the computation of the GAL, AUTOSCALE reconsiders the local estimation of input load. Nevertheless, the estimation of the capacity performed locally is only considered locally. Indeed, a local estimation of the capacity that suppose that two assumptions are true. First, the average processing latency of the operator will not change in near future. Such variations may be due to a sensitivity to values in input. To ensure that AUTOSCALE adapts accurately operators in front of even and uneven streams, the processing latency of each operator should remain as stable as possible. In the case of operators executed by multiple tasks, it can be done with a load-aware strategy for load balancing. Second, it is assumed that on an estimation period Δ, the operator will have as much available CPU time as it requires. Depending on the scheduling strategy and available processing units, some operators may be assigned on a same processing unit and share CPU time. To improve the accuracy of estimations, it is crucial to take concurrency into account for the computation of operator capacity.

6

Preventive auto-parallelization enhancements

Contents

1 Motivation

1.1 Impact of thread concurrency on operator capacity

We presented the preventive auto-parallelization strategy AUTOSCALE which anticipates modification of parallelism degrees according to local and global estimations. The aim of this approach is to prevent congestion and limit increases of average processing latency due to accumulation of stream elements. So, it is assumed that each scale-out and scale-in respectively increase and decrease resources reserved for a given operator. For example, when AUTOSCALE estimates that an operator $\mathcal{O}_i$ will have two times more stream elements to process in near future, its parallelism degree is at least multiplied by two to double available resources. Nevertheless, considering a scheduling strategy assigning tasks such as there could be concurrency for CPU and memory usage [Xu et al., 2014, Peng et al., 2015], this assumption may be wrong. Considering the operator $\mathcal{O}_i$, if its parallelism degree is multiplied by two but some tasks are assigned on processing units

having less available resources than necessary, the processing capacity is not effectively doubled. Some solutions[Peng et al., 2015] offer the possibility to associate resource constraint for each operator. Still considering the operator $\mathcal{O}_i$, users can specify that a task of $\mathcal{O}_i$ is assignable on a processing unit only if there are at least 20% of idle CPU time and 256Mb of free memory space. Then, the scheduling strategy defines the scheduling plan such as each task has at least required resources declared by users. It assumes two conditions:

- Users have a complete knowledge of relative time and space complexity of each operator. This knowledge is particularly difficult to build if there are some user-defined operators without declared algebraic properties.
- Resources are fragmented such as if a task uses less resource than required, the difference between required and used resources should be available at any time. In practice, idle resources may be used by other processes. The scheduling of active threads varies in according to CPU architecture and operating system.

So, such systems are designed to let users give indicative requirements for each operator with an accuracy depending totally of user expertise. While estimating processing capacities, the difference between required and used resources by an operator affects significantly the accuracy of the estimation. When there are concurrency on resource usage, taking effective resource usage into account is crucial to improve the accuracy of modification of parallelism degrees. In the remainder of this chapter, we consider DSMSs considering reservation of resources for each operator as described above.

1.2 Managing load between tasks in presence of uneven streams

To estimate the processing latency of an operator $\mathcal{O}_i$ applied by many tasks, we consider the average latency of all tasks applying $\mathcal{O}_i$ without evaluating the potential imbalance between tasks. Indeed, let consider a task T_1^i which processed values AAAB and another task T_2^i which processed values BBAB. Let consider that the key A takes 1 time unit to be processed and the key B takes 10 time units, the average latency of $\mathcal{O}_i$ is 5.5 time units per stream element but the standard derivation is more than 2. This imbalance is negative for performance as it implies different processing requirements for task of a same operator. To tackle this issue, some load balancing strategies [Rivetti et al., 2015] have been developed. They aim at compensating such skew in value distribution through adaptive routing policy based on incoming values.

In the remainder of this chapter, we aim at defining an auto-parallelization strategy which takes concurrency on resource usage into account while estimating processing requirements of operators. This solution should not rely on static knowledge provided by users as it assumes user expertise and processing requirements may vary over time. In addition, this strategy should be paired with a load balancing strategy to serve as guarantee that value distribution does not lower benefits brought by modification of parallelism degree.

2 Resource-aware auto-parallelization of operators

In this section, we present the auto-parallelization strategy AUTOSCALE+ which aims at tackling problems mentioned above. AUTOSCALE+ relies on the monitoring module of AUTOSCALE to get recent history of operators. In addition to these information, AUTOSCALE+ collects information about CPU usage for each task of operator. It allows to perform resource-aware parallelization

of operators. In the remainder of this section, we detail the computation of input workload in near future at local and global scope within a single pass. Then, we present the evaluation of processing capacity taking potential concurrency on resource usage into account. Finally, we suggest a configurable adaptation policy offering the possibility to users to control the stability of the system.

2.1 Enhancement of workload estimation

As presented in chapter 5 section 3, AUTOSCALE+ aims at computing an estimation of the volume of stream elements to process in near future. AUTOSCALE+ reduces the computation time of workload estimation and increases the accuracy through the following improvements:

- Instead of estimating the activity level through a local and a global estimation, AUTOSCALE+ explores each workflow according to a breadth-first search. For each operator $\mathcal{O}_i$, the value of $|EstimR^{i,j}|$ and $EstimOutput_{F_j^i}$ is computed according to formulas (2) and (9) presented in chapter 5 section 3. For sources, *i.e.* operators without upstream operators, the value of $EstimInput_{F_j^i}$ is computed according to formula (3) presented in chapter 5 section 3. For other operators, AUTOSCALE+, the value of $EstimInput_{F_j^i}$ is computed according to formula (1).

$$EstimInput_{F_j^i} = combine(EstimR^{i,j}, \sum_{p \in par(\mathcal{O}_i)} EstimOutput_{F_j^i}) + pending_{F_{j-1}^i} \quad (1)$$

  where par($\mathcal{O}_i$) returns all parent operators of $\mathcal{O}_i$

- To improve the accuracy of the regression model, AUTOSCALE+ applies linear, logarithmic and exponential regression models and selects the model fitting the best to the previous iteration of the window. It implies light computation overheads but allow to detect characteristic stream fluctuations with improved accuracy.

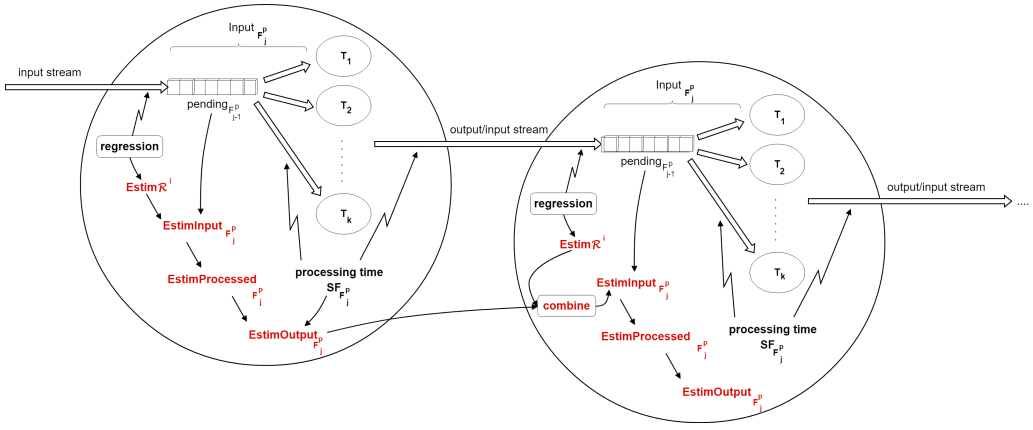


Figure 1: Metrics

2.2 Estimation of available resources

Now that we have estimated incoming and outcoming volumes for each operator, we need to approximate the maximal number of stream elements each operator can process during an iteration of duration Δ. We denote this number of stream elements the capacity of the operator. According to both estimations, then we are able to detect potential congestion. In an ideal case, the capacity is defined as follow:

$$IdealCapacity_{F_j^i} = \frac{\Delta}{Lat_{F_j^i}} \tag{2}$$

where $Lat_{F_j^i}$ is the processing latency. This processing latency does not take into account time spent in pending queue. Nevertheless, using this approximation means that each thread associated to the operator is able to maintain its execution during duration Δ without being interrupted by other threads. In a distributed and multi-threaded environment, this assumption favors over-estimations of operator capacities.

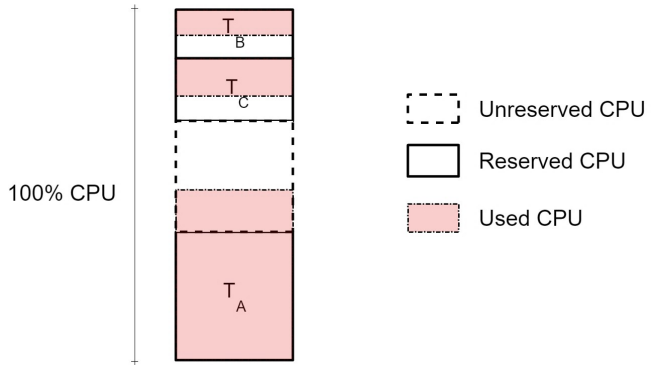


Figure 2: Usable CPU for threads on one core

As illustrated on Figure 2, the thread T_A requires more CPU time than its reservation. At the opposite, threads T_B and T_C use less CPU time than they reserved. A percentage of the CPU time is neither reserved nor used, so it could be used by any thread. Considering current CPU reservation and usage, we aim at estimating usable CPU time for each thread. Each thread is associated to a CPU *constraint* used by the scheduler. The interest of such kind of constraint is to avoid assignments leading to resource starvation.

Thus, on a given CPU, let consider three operators $\mathcal{O}_A$, $\mathcal{O}_B$ and $\mathcal{O}_C$ executed respectively by threads T_A, T_B and T_C as shown on figure 2. Each thread is associated to a *resource constraint* [Peng et al., 2015] and a current usage which could be greater, equal and smaller than the constraint. As CPU usage may vary suddenly overt time, we also consider a weighting factor $\alpha \in [0;1[$. Thus, we underestimate lightly available CPU time to avoid fast overload. Let T_X be a thread associated to a reservation constraint $ResaCPU_X$. This thread is in concurrence with n other threads for the usage of a CPU C. We estimate that the usable CPU time by T_X is defined according to formula (3).

$$UtilCPU(T_X, C) = \alpha \times max(UsedCPU(T_X, C), ResaCPU_X) + \frac{1}{n}(100 - \sum_{\forall T_Y \neq T_X} UsedCPU(T_Y, j)) \quad (3)$$

Considering all threads T_i^1, T_i^2,..., T_i^m applying an operator $\mathcal{O}_i$, we define the global CPU time $UtilCPU_i$ for T_i^1, T_i^2,..., T_i^m consuming resources of CPUs $CPU(T_i^1)$, $CPU(T_i^1)$,..., $CPU(T_i^m)$ as follow:

$$UtilCPU_i = \min_{x=1...x=m}(UtilCPU(T_i^x, CPU(T_i^x)) \quad (4)$$

We assume that an increase in input rate affects all threads executing the same function equally.

With this estimation, we can approximate usable CPU time with a greater precision according to current thread assignments. Thus, we can improve the definition of the capacity as follow:

$$Capacity_{F_j^i} = \frac{\Delta}{Lat_{F_j^i}} \times UtilCPU_i \quad (5)$$

2.3 Balance between processing requirements and resources

Now, we have an estimation of incoming volumes and accurate capacities for all operators, we can detect imbalance between processing requirements and resource usage. There are three possibilities: detecting a need of scale-out, a possibility of scale-in or doing nothing. While AUTOSCALE+ detects a need of scale-out, the system should reconfigure itself to avoid at least a degradation of performance due to the accumulation of stream elements on pending queues. In the case of a scale-in, the system should benefit from a decrease of parallelism degree in terms of performance and active resources. Nevertheless, performing a scale-in brings overheads that future benefits does not systematically compensate. For example, if performing a scale-in reduces overall resource usage by 2% but increases massively the average processing latency until migrations of pending queues are completed, it does not worth for users wanting results with short latency or paying for all available resources without distinction between active and inactive ones. In such case, it appears relevant to let users define which benefit should bring a scale-in, in terms of resources saved, to compensate reconfiguration overheads.

2.3.1 Working interval

Now that we have an estimation of the incoming workload and the capacity taking resource usage into account, we can approximate the ideal parallelism degree, denoted $idealK$ according to formula (6).

$$idealK = \frac{EstimInput_{F_j^i}}{Capacity_{F_j^i}} \quad (6)$$

The problem is then to define if it worth modifying the current parallelism degree. Indeed, if the parallelism degree $idealK$ is lower than the current parallelism degree k but does not bring a minimal benefit in terms of performance, triggering a scale-in involves reconfiguration overheads that are not compensated by a significantly better performance.

To represent this notion of benefit, we suggest a controllable *working interval* associated to each parallelism degree k. This interval has as upper bound the current parallelism degree k and as lower bound a parallelism degree $\min_k$ which is function of the current parallelism degree k. The value of $\min_k$ is defined according to formula (7).

$$min_k = \beta \times k \tag{7}$$

where $\beta \in]0;1]$ is a controllable parameter. If β is close from 0, it means that AUTOSCALE+ performs scale-in only when input volumes are very small compared to operator capacities. If β is close from 1, AUTOSCALE+ performs scale-in as soon as a smaller parallelism degree k' guaranties theoretically that input volumes can be processed.

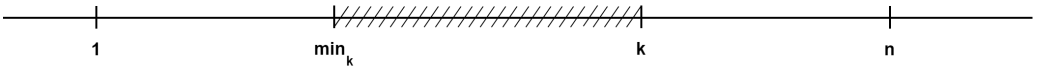


Figure 3: Working interval

2.3.2 Modification of parallelism degree

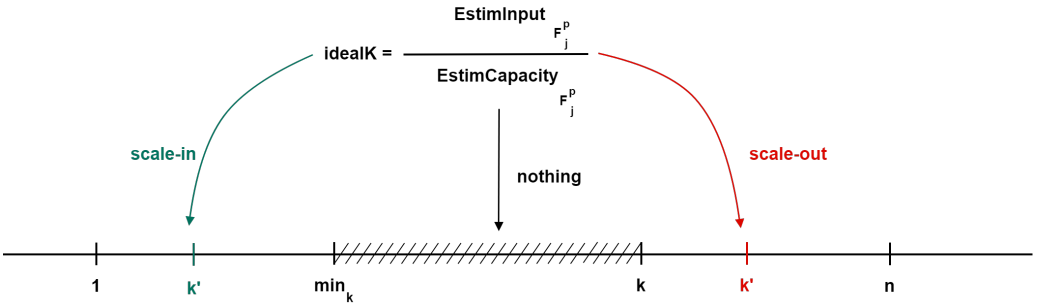


Figure 4: Modification of parallelism degree

A scale-out should be performed for an operator $\mathcal{O}_i$ when $idealK$ exceeds the working interval as represented on figure 4. It means that k will be under-evaluated during next iteration of the monitoring window and may lead to congestion.

On the contrary, if $idealK$ is smaller than min_k, it means that k will be over-evaluated for the next iteration F^i_{j+1}.

Within this working interval, a possibility of scale-in is not considered as it does not save enough resources to compensate reconfiguration overheads.

More formally AUTOSCALE+ recommends to keep the current parallelism for an operator $\mathcal{O}_i$ as long as the following condition is valid:

$$min_k \leq idealK \leq k \tag{8}$$

It is worth noting that the greater is k, the greater is the associated working interval. We opt for such property because the more there are tasks to merge into less tasks, the more it takes time to merge pending queues distributed over the cluster and re-route stream elements.

2.3.3 Computation of the appropriate parallelism degree

Thus, AUTOSCALE+ computes an appropriate parallelism degree k' according to formula (9).

$$argmin_{k'}(\frac{EstimInput_{F_j^i}}{ResCapacity_{F_j^i}} \leq k) \tag{9}$$

where $ResCapacity_{F_j^i}$ is the capacity of $\mathcal{O}_i$ considering the CPU constraint $ResCPU_i$. The capacity $ResCapacity_{F_j^i}$ is defined according to formula (10).

$$ResCapacity_{F_j^i} = \frac{\Delta}{Lat_{F_j^i}} \times ResCPU_i \times \alpha \tag{10}$$

where $\alpha \in]0;1]$ is a parameter allowing AUTOSCALE+ to consider a relative margin between effective CPU usage and CPU reservation. It means that AUTOSCALE+ takes into account the fact that some threads may need more than their reservation at runtime. As β, the parameter α can be defined through several methods like empirical study, reinforcement learning or user expertise.

It is worth noting that the new parallelism degree is computed considering only the CPU requirement declared by users and not the last value of $UtilCPU_i$. Indeed, a modification of parallelism degree will lead to a modification of thread assignments so $UtilCPU_i$ may change after reconfiguration of the system.

3 Load management

3.1 Auto-parallelization of operator with load imbalance

Load imbalance may occur in a stream processing context. Let consider an operator $\mathcal{O}_i$ applied by k tasks T_1^i to T_k^i. In addition, the average processing latency per stream element Lat_i of $\mathcal{O}_i$ is a function of the value v_j in input. Each value v_j belongs to an ordered set of values V=$\{v_1, v_2, ..., v_m\}$ and is associated to a specific processing latency $Lat_i(v_j)$ such as $Lat_i(v_j) < Lat_i(v_{j+1})$. Let consider a stream sequence S of size q×k stream elements such as the distribution of values over S is uniform.

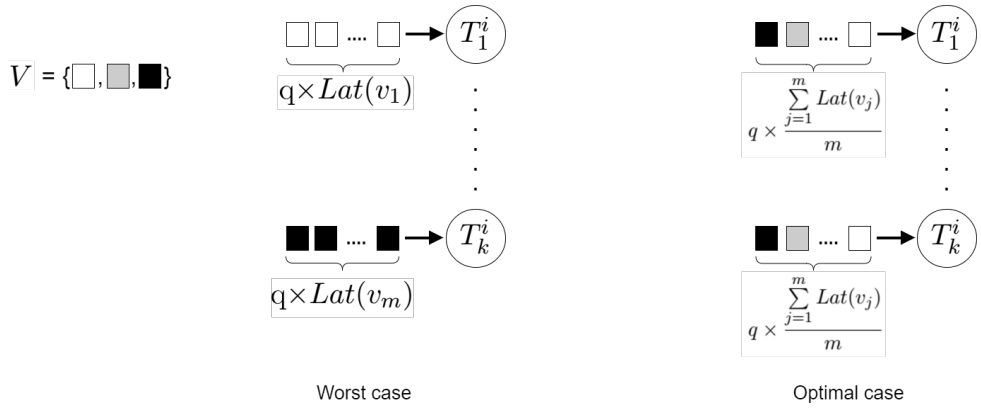


Figure 5: Worst and optimal cases of load balancing

Now, let consider a load balancing strategy distributing stream elements in a Round-Robin fashion. As illustrated on figure 5, after the distribution of q×k stream elements, processing requirements in terms of computation of times are q×$Lat(v_1)$ for T_1^i and q×$Lat(v_m)$ for T_k^i in the worst case. Considering that the capacity of $\mathcal{O}_i$ is computed taking the average processing latency into account, tasks associated to $\mathcal{O}_i$ and having greater processing requirements will be under-provisioned in CPU time due to load imbalance. It causes more pending stream elements and increases future estimation of the workload. At the opposite, an optimal load balancing strategy would have distributed the same stream sequence of q×k such as each task has an average processing latency per stream element equals to the average latency $\overline{Lat_i(V)}$ with $\overline{Lat_i(V)}$ defined as follow:

$$\overline{Lat_i(V)} = q \times \frac{\sum_{j=1}^{m} Lat(v_j)}{m} \tag{11}$$

By consequence, the imbalance due to load balancing may lead the auto-parallelization strategy to perform unnecessary scale-out which does not improve the performance but at the contrary bring additional overheads with reconfiguration. It appears then necessary to combine AUTOSCALE+ with a load balancing strategy.

3.2 Compatibility issues

Combining an auto-parallelization strategy with a load balancing strategy may raise some compatibility issues. Indeed, an auto-parallelization approach like AUTOSCALE+ relies on the fact that a congestion is due to an overload in input of all tasks associated to an operator, *i.e.*, adding more tasks reduces effectively the workload of each task. In the specific case of load imbalance, a task may be overloaded while others process stream elements normally. This overload comes from an uneven distribution of stream elements according to their values. As some operators are sensitive to stream element values, the processing latency can significantly vary depending on which value is read in input.

To tackle this issue, the auto-parallelization should rely on a load balancing strategy guaranteeing that the workload is evenly distributed between tasks of an operator with regards to the distribution of values. So, load balancing strategies relying of key grouping to build partitions [Neumeyer et al., 2010] are not efficient in this context. Indeed, they route statically stream elements to tasks according to their values without consideration for the volume of each partition. On the opposite, load balancing strategies balancing statically the number of stream elements in each partition may create imbalance if the operator is sensitive to values in input and the distribution of values changes over time.

To sum up, it appears necessary to combine AUTOSCALE+ with a load balancing strategy which takes into account the processing latency for each value appearing in the stream. Thus, the workload of each partition may be balanced at runtime according to the distribution of values.

4 Discussion

We presented the preventive auto-parallelization strategy AUTOSCALE+ which takes effective resource usage into account to modify parallelism degree of operators. For a given operator, AUTOSCALE+ assumes that the incoming load is evenly distributed over its tasks. This property

is guaranteed by the resource-aware load balancing strategy OSG. In this section, we aim at evaluating benefits brought by the combined approach through a comparison with static parallelism management.

4.1 Online Shuffle Grouping for resource-aware load balancing

As explained above, AUTOSCALE+ should modify parallelism degree of operators more accurately if it is combined with an optimal load balancing strategy. Indeed, the optimal load balancing strategy ensures that each task has the same workload. So, the estimation of the required capacity computed at operator scope fits to the input load of each task.

Computing an optimal routing policy of stream elements requires an *a priori* knowledge of $Lat(v_j)$ for all $v_j \in V$. Nevertheless, in a stream processing context, the set V is generally unknown at the beginning of treatments as the time complexity of $\mathcal{O}_i$. Moreover, the scheduling of tuples to tasks must be performed online *i.e.*, the load balancing algorithm does not know the sequence of stream elements to schedule.

To solve this issue, we choose to associate AUTOSCALE+ to the resource-aware load balancing strategy OSG [Rivetti et al., 2016], for *Online Shuffle Grouping*. OSG performs online load balancing through the combination of two *Count Min Sketches* [Cormode and Muthukrishnan, 2005] and a *Greedy Online Scheduler*. OSG associates two matrices of controllable sizes to each operator $\mathcal{O}_i$. For each matrix, each row is mapped to an exclusive 2-universal hash function [Carter and Wegman, 1979]. Each time a stream element t is read in input, the Count Min Sketch algorithm updates the occurrence of t in the first matrix and the completion time in the second matrix. Thus, according to both sketches, OSG is able to return the average completion time of any value seen previously. This estimation is approximated with a bounded error as analyzed in [Rivetti et al., 2016]. A Greedy Online Scheduler uses these estimations to route an incoming stream element to the task able to process it with the shortest delay. This delay is computed as the time needed to execute all pending stream elements. More details about OSG and theoretical analysis can be find in [Rivetti et al., 2016].

4.2 Empirical study of the combination AUTOSCALE+ and OSG

We implemented AUTOSCALE+ and OSG over the stream engine Apache Storm 1.0.2[16]. We perform tests on a the workflow illustrated on figure 6.

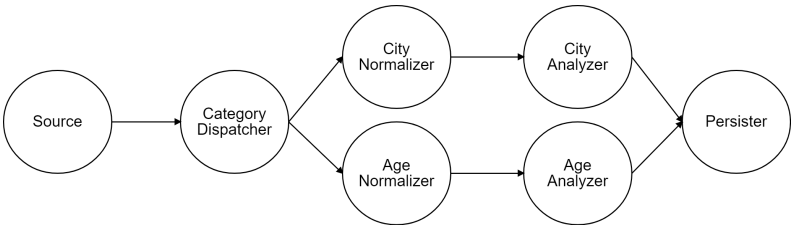


Figure 6: Complex sensitive topology

This workflow is composed of several operators with various selectivity factors and average processing latency. The spout (OpinionSource) emits stream elements concerning opinions submitted by users about a topic. Each opinion is described by information on the user, like its age

[16] https://storm.apache.org/

and code representing its location, the topic and user opinion. Stream elements are sent to a bolt (CategoryDispatcher) filtering unnecessary attributes and depending on the branch downstream. In addition, it filters stream elements concerning a predefined list of irrelevant topics. A branch starts with a bolt (CityNormalizer) retrieving information on user location from the code. This bolt has the exact same properties than the sensitive bolt of the simple sensitive topology. Indeed, depending on the code, retrieving information on the city takes more or less time. It allows us to compare the impact of workflow structure and complexity on bolt behavior and dynamic adaptation of its parallelism degree. Then, a bolt (CityAnalyzer) extracts relevant subgroups according to opinion and location. The other branch starting from the bolt CategoryDispatcher performs similar treatments in order to define subgroups on user opinion and age. Finally, the Persister takes in input descriptions of subgroups and persists them in a storage file system.

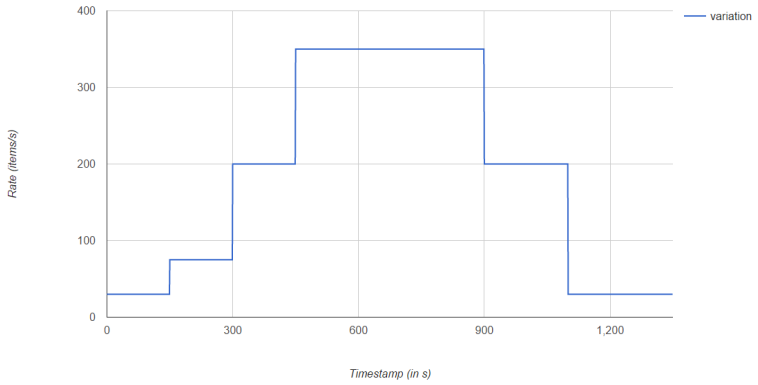


Figure 7: Fluctuations in input rate of synthetic streams

We generated two synthetic streams following fluctuations in input rate illustrated on figure 7. The first stream has an uniform distribution of values. The second stream follows a Zipf distribution law of values such as keys requiring greatest processing time have smallest occurrences. The skew of the Zipf law is set to 1.5 in the remainder of this section.

In a first time, we focus on benefits brought by the association of AUTOSCALE+ and OSG introduced in [Rivetti et al., 2016]. We observed the behavior of the workflow presented on figure 6 while receiving the synthetic streams in function of following configurations:

- The default configuration defining statically parallelism degree of operators for the complete lifetime of the workflow and balancing the load of each operator stream elements according to a Round-Robin (shuffle) policy.
- The same configuration but using OSG as load balancing strategy.
- AUTOSCALE+ as the auto-parallelization strategy and the shuffle policy for load balancing.
- AUTOSCALE+ as the auto-parallelization strategy and OSG for load balancing.

For each configuration we observe the average processing latency and the throughput of the workflow to evaluate the performance. Concerning result quality, we define a processing latency threshold discriminating final results computed within a satisfying end-to-end latency.

Size of monitoring window (in s)	90
Monitoring frequency (in s)	90
α (AUTOSCALE+)	0.3
β (AUTOSCALE+)	0.7
θ (OSG)	0.05
ϵ(OSG)	0.05

Table 1: Parameters for AUTOSCALE+ and OSG

Finally, we analyze the effect on resource usage through the cumulative CPU usage of the critical operator. Parameters for AUTOSCALE+ and OSG configurations are summarized in table 1:

Parameters α and β have been defined through an empirical study. Indeed, we have evaluated the impact of these parameters on scale-in and scale-out and choose these values as the most appropriate ones.

4.2.1 Evaluation with uniform distribution

While processing the synthetic stream with the different configurations, we notice major differences between static management of parallelism degrees and AUTOSCALE+. Indeed, configurations using AUTOSCALE+ maintain treatments while facing critical fluctuations in input rate as illustrated on figure 8. So, result quality is improved with 94.8% of all stream elements processed under the time threshold against 26.1% with the static management of parallelism degree.

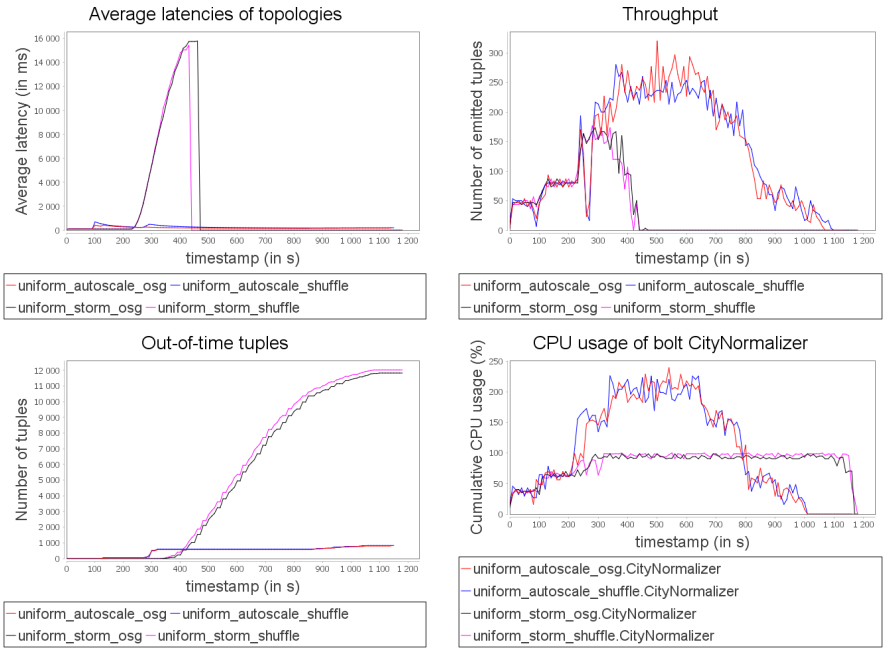


Figure 8: Comparison between static parallelization of operators and AUTOSCALE+

With the static solution, we can observe that the CPU usage of the critical operator (CityNormalizer) increases until it uses all available CPU of the machine. When the machine is overloaded, the throughput degrades until complete congestion. Then, stream elements are considered as

processed out-of-time. Contrary to static configurations, AUTOSCALE+ adapts the parallelism degree of the critical operator such as it could use more than 200% cumulative CPU time on multiple hosts and avoid congestion.

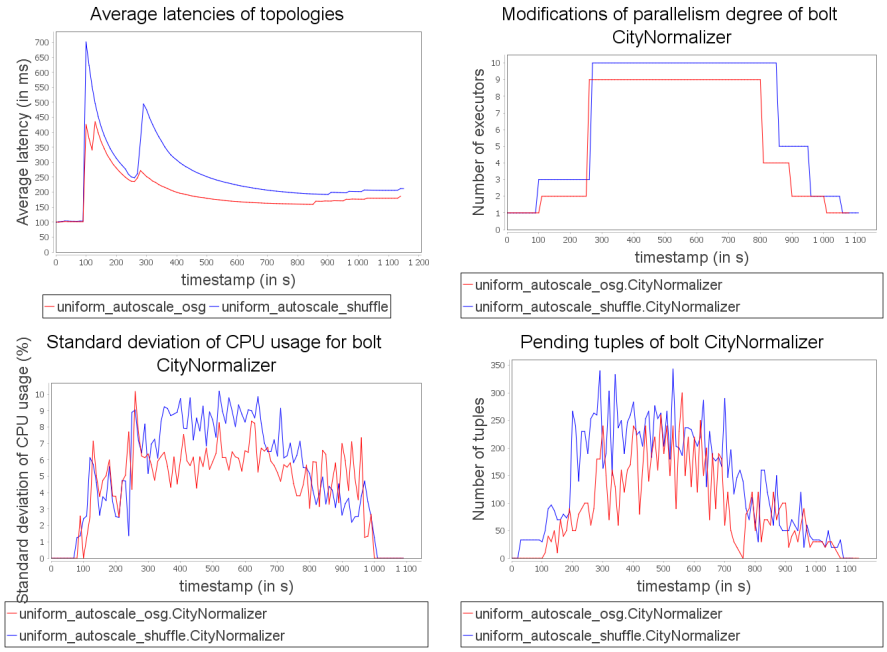


Figure 9: Comparison between Round-Robin and OSG

Now, let observe in more details differences between AUTOSCALE+ with Round-Robin and OSG strategies for load balancing. Concerning performance, it is worth noting that OSG keeps latency 39% lower than Round-Robin strategy after reconfiguration. Load balancing between tasks of the critical operator is improved faster with OSG. Actually, when OSG detects new tasks, it routes new stream elements in priority to thee new tasks. At the opposite, the Round-Robin strategy keeps an imbalance between old and new tasks for a longer duration.

To highlight this imbalance, we focus on configurations using AUTOSCALE+ with the Round-Robin strategy and with OSG. It is worth noting that the standard deviation of CPU usage of all tasks of the critical operator is in average 4.4% with OSG and 5.6% even with a uniform distribution of values. Moreover, the fill ratio of pending queues is in average 26% lower with OSG than with the Round-Robin strategy. The lower fill ratio of pending queue has an impact on input workload and estimations performed by AUTOSCALE+. So, with OSG, AUTOSCALE+ estimates more accurately parallelism degree to handle maximal load and delivers better performance.

4.2.2 Evaluation with biased distribution

We repeat the same experiment but this time, we play in input the biased version of the synthetic stream described in introduction of experimental study. For remainder, fluctuations in input rate remain unchanged but the distribution of values follows a Zipf law. Each value in the stream is associated to a specific processing time going from 10 to 90 milliseconds.

Between static configurations and configurations using AUTOSCALE+, we observe similar behaviors mentioned above. So, adaptations performed by AUTOSCALE+ allows the system to process 94.6% of all stream elements against 26.8% in average for static configurations.

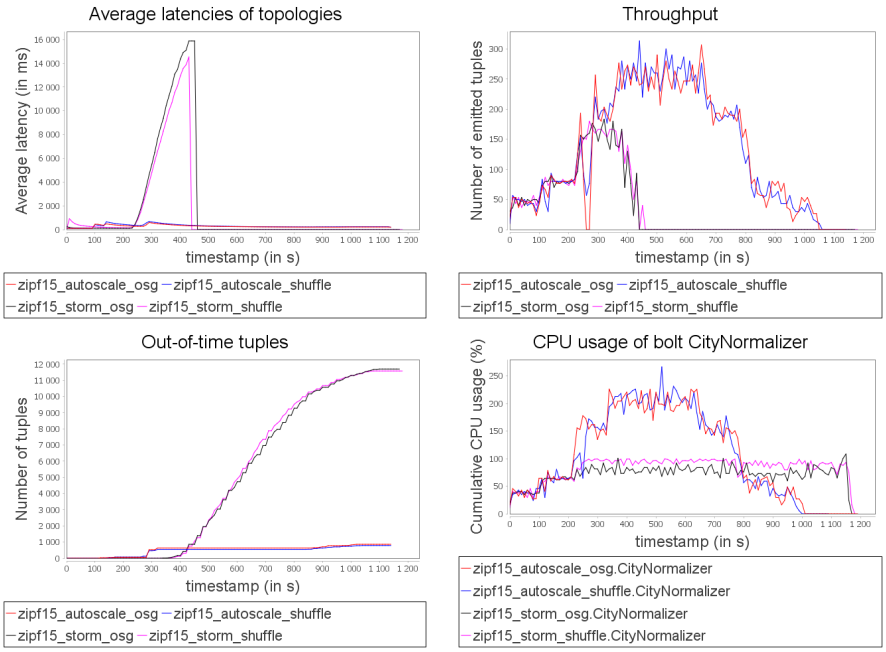


Figure 10: Comparison between static parallelization of operators and AUTOSCALE+

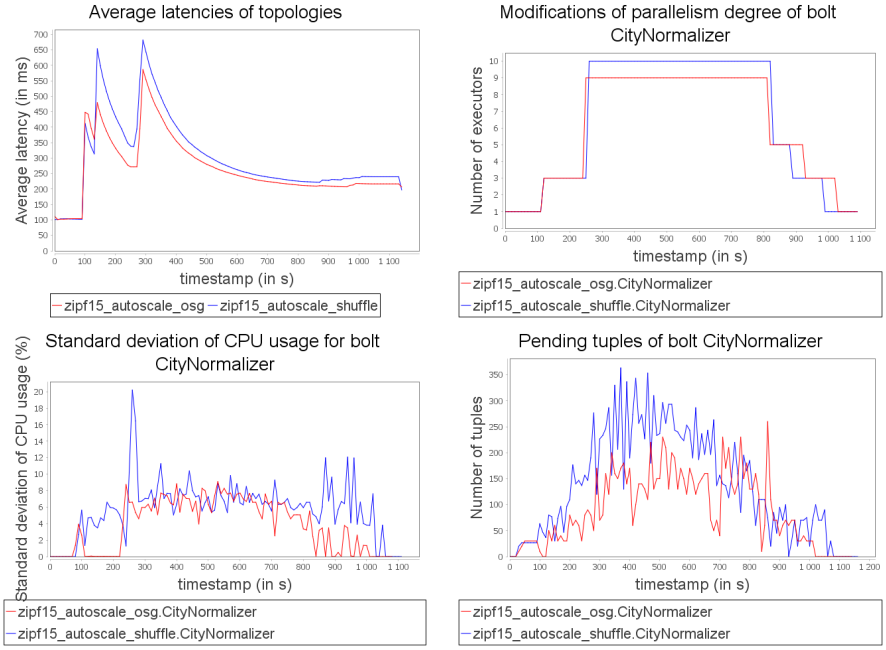


Figure 11: Comparison between Round-Robin and OSG

For configurations using AUTOSCALE+, we observe same behaviors too. Nevertheless, the bias in data distribution has an impact on load balancing. Indeed, compensating load imbalance with such stream is harder than with a uniform distribution of values. So, the standard deviation of CPU usage of all tasks is increased to 6.8% with the Round Robin strategy and reduced to

3.8% with OSG. Nevertheless, the fill ratio of pending queues is in average 11% less important with OSG than the Round-Robin strategy.

In conclusion, the auto-parallelization strategy AUTOSCALE+ allows to adapt parallelism degrees of operator before the host of a critical operator is overloaded. It maintains treatments while facing critical fluctuations in input rate. The association with the load strategy OSG reduces the imbalance between tasks of an operator. It has two effects: decrease the fill ratio of pending queues which improves the end-to-end latency of the workflow. Then, it reduces workload estimations performed by AUTOSCALE+ which results in better performance in front of important input rate.

4.3 Limits of AUTOSCALE+

We presented the approach AUTOSCALE+ which is able to anticipate the congestion of operators. The estimation of workload in near future is accurate under the assumption that the evolution trend of stream rate does not vary significantly in near future. Indeed, if AUTOSCALE+ detects a significant increase input rate for an operator, it may trigger a scale-out. If the input decreases deeply after the analysis performed by AUTOSCALE+, a scale-in can be triggered and involve reconfiguration overheads.

7

Experiments

Contents

1 Overview of solutions

In this section, we summarize solutions for parallelization, load balancing and scheduling of operators that we experiment in the remainder of this chapter (see Figure 1). Each solution solution has selected for its popularity over existing solutions and is described briefly with its specific features.

For the parallelization of operators, we consider five approaches:

- The static approach offers only the opportunity to set parallelism degree of operators on user demand. To prevent the congestion of operators with this approach, users must monitor continuously states of operators and trigger scale-in or scale-out manually. Moreover, users must have an expertise in order to determine which parallelism degrees satisfy current processing requirements.

- The approach AUTOSCALE presented in chapter 5 prevents the congestion of operators according to an estimation of processing requirements in near future.

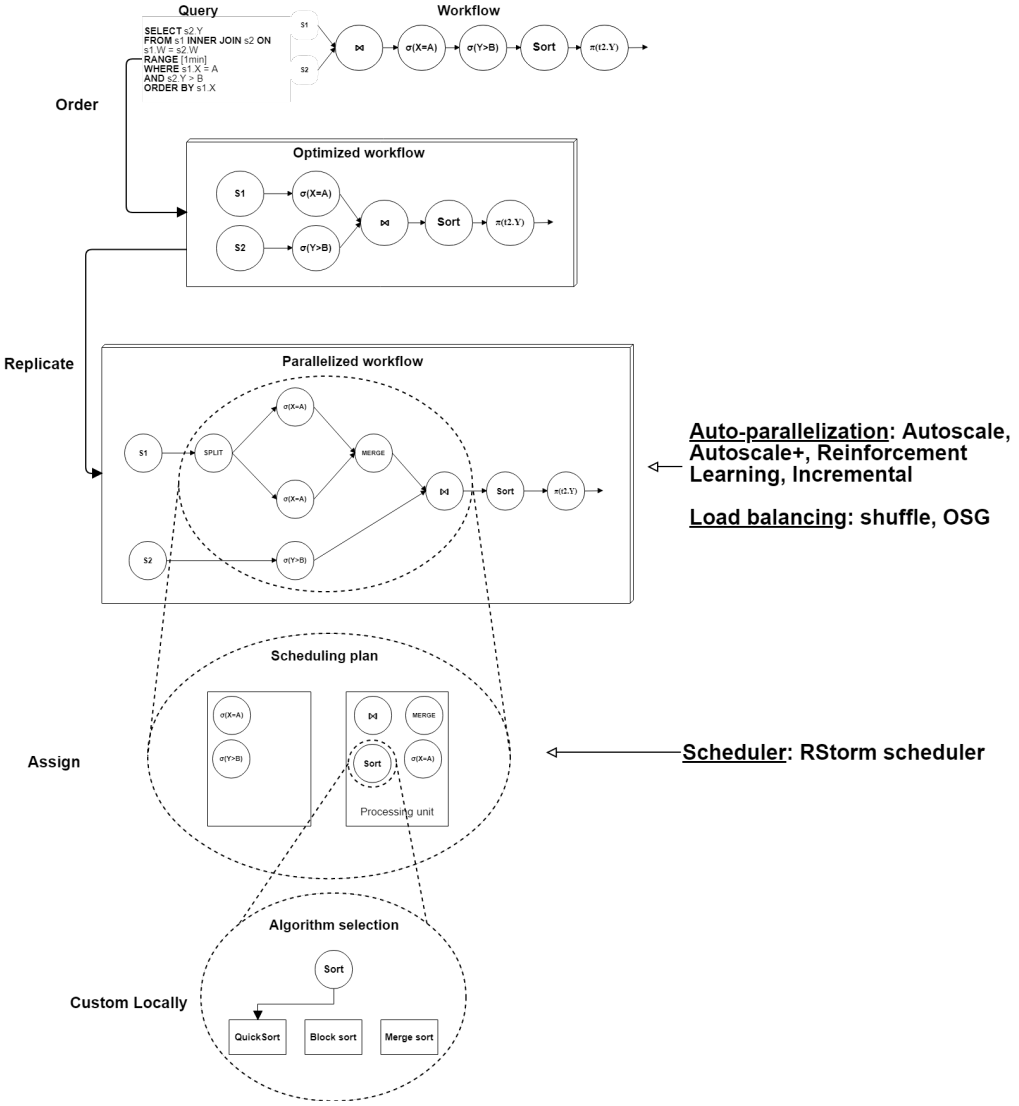


Figure 1: Experimented solutions

- The extension of AUTOSCALE, named AUTOSCALE+ (**asplus**) and introduced in chapter 6, takes resource usage into account to adapt the parallelism degree of operators according to stream variations in input rate.

- An incremental (**incr**) strategy observes the input rate and the throughput of each operator. Considering two thresholds *min* and *max*, the INC strategy increases the parallelism degree by one if the ratio input rate on throughput exceeds *max* and a scale-in if this ratio is lower than *min*.

- A reinforcement learning-based (**rlearn**) strategy mapping input rates to appropriate parallelism degrees at runtime. The RL strategy is initialized with a knowledge base covering stream fluctuations in input rate. It assumes that users can train the system with representative fluctuations which is not always the case in practice.

In order to balance the load within operators, we consider two solutions:

- The *shuffle grouping* (**shuffle**) which corresponds to a Round-Robin distribution of stream elements to tasks (see chapter 4). This solution aims at balancing the number of stream elements processed by each task of an operator without consideration for the distribution of values.
- The OSG (**osg**) approach presented in chapter 6 which estimates the processing time necessary to compute each value of stream element. Thus OSG can balance the load between tasks of an operator even if the distribution of values varies significantly over time.

Finally, we consider a single scheduling strategy named *resource-aware scheduling* and presented in [Peng et al., 2015]. This strategy takes in input constraints on resource availability for each operator (CPU and memory) and returns a near-optimal scheduling plan which minimize the consumption of resources.

All those approaches have been implemented in the DSMS Apache Storm due to the lack of open-source and extensible solutions integrating at least one the solutions mentioned above. Moreover, we have chosen this solution over other efficient SPEs such as Apache Spark Streaming [Zaharia et al., 2012b] due to data management. Yet, Spark Streaming systematically groups stream elements into batches, called *Resilient Distributed Datasets* (RDD). Nevertheless, if RDD sizes are large compared to incoming volumes, detection of congestion and over-consumption of resources is delayed. Thus, RDD size must be managed dynamically in addition to parallelism degrees. Compared to Apache Flink, Storm also provides technical support to implement an auto-parallelization strategy without affecting the core of the system. In addition, at the beginning of developments, Apache Flink was not released yet and lack documentation about its design and performance.

2 Design and implementation of AUTOSCALE and AUTOSCALE+

2.1 Overview of Apache Storm

Apache Storm is an open-source SPE, allowing users to define continuous queries as graphs of operators, called *topologies*. Users define each operator in a high-level programming language such as Java, Python or Clojure.

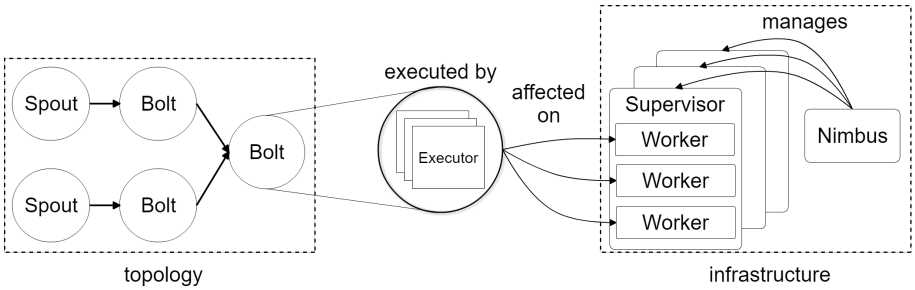


Figure 2: Storm architecture

To summarize, operators, named *components* in Storm terminology, belong to one of two categories: *spouts* or *bolts*. A spout is a connector to a raw stream source and represents an

entry of a topology. It distributes stream elements to components to which it is connected and can process filtering operations if required. Bolts consume stream elements from any component and compute a result for each element received (*stateless* bolt) or for a set of stream elements (*windowed* bolt).

Each component is executed in parallel by *executors*. An executor is an instance of an operator. Each executor is assigned to a processing unit by the scheduler (see figure 2). The number of executors for a given spout/bolt is revised at runtime only at user request [Xu and Peng, 2016].

Concerning the execution support, Storm relies on two types of processing nodes: *Nimbus* and *supervisor*. The Nimbus acts as a JobTracker for Hadoop . As illustrated on figure 2, each supervisor manages a pool of *workers*, *i.e.* processing units, and monitors executors assigned on them.

2.2 Implementation of AUTOSCALE

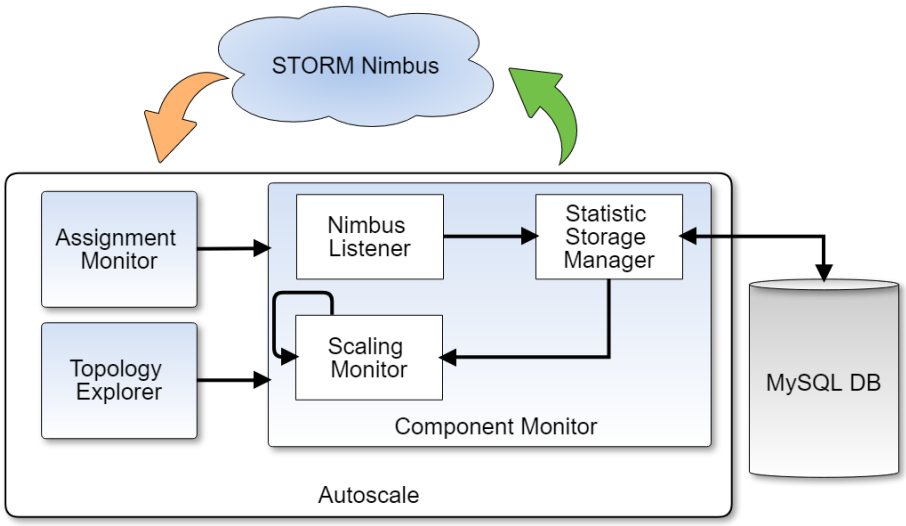


Figure 3: AUTOSCALE architecture

We have implemented AUTOSCALE and AUTOSCALE+ over Storm 1.0.2. It implements the *IScheduler* interface of the Storm API. AUTOSCALE and AUTOSCALE+ modify the parallelism degree of operators and then let the resource-aware scheduler introduced in [Peng et al., 2015] define a near-optimal scheduling plan. To perform auto-parallelization according to monitored metrics, AUTOSCALE and AUTOSCALE+ rely on three modules:

- *Component Monitor*: this module is in charge of monitoring continuous queries at operator scope and triggering reconfigurations when necessary. To collect all measurements at runtime, a client listens to internal metrics gathered by the Nimbus node. This client is integrated within a sub-module named the *Nimbus listener*. These raw data are pre-processed and stored in a MySQL database through a *Statistic Storage Manager* sub-module. The *Statistic Storage Manager* is also in charge of providing methods to group measurements by iterations of the monitoring window. Measurements are analyzed by the *Scaling Monitor* sub-module which includes the auto-parallelization strategy presented in chapter 5 or 6 depending on the version.

- *Assignment Monitor*: this module is in charge of collecting information on the execution support and assignments of operators on processing units. Through this module, AUTOSCALE can extract the scheduling plan and compute the parallelism degree of each operator, which resources are available on each processing unit and the amount of CPU and memory resources each task requires.
- *Topology Explorer*: this module is in charge to facilitate the exploration of each submitted queries. Indeed, each time a new continuous query is submitted, this module builds static knowledge on its structure for faster exploration. For example, it provides methods to identify entry and output operators of a topology but also methods to identify the children and parents operators of a given operator.

3 Evaluation of AUTOSCALE

In this section, we present the an experimental evaluation of AUTOSCALE. Due to the lack of open-source implementations of auto-parallelization strategies compatible with Storm, we focus our efforts on the comparison of AUTOSCALE with the static approach (see section 1) natively integrated in Storm. We evaluate the benefit brought by Storm depending on the expertise level of the user on a micro-benchmark.

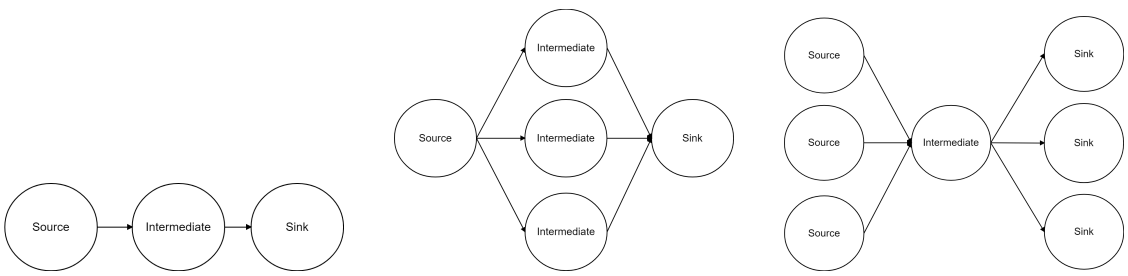
3.1 Experimental protocol

3.1.1 Execution support

Our test cluster is composed of 7 VMs. Each VM has at its disposal a dual-core CPU Intel(R) Xeon(R) E5-2620 running at 2.00GHz, 4Gb of RAM and 40Gb of hard disk space. A machine runs the Nimbus daemon and is dedicated to cluster coordination. Each supervisor manages 4 workers. On the Nimbus host, a MySQL database is also deployed in order to store historical data as illustrated above.

3.1.2 Workflows and streams

To validate our approach, we choose to study its impact on three elementary topologies: a linear, a diamond and a star topology. Each elementary topology is composed of two types of bolts: *intermediate* bolts with low latency and *sink* bolts with high latency.



(a) Linear topology (b) Diamond topology (c) Star topology

In this section, we choose to present only some results relative to the linear, diamond, star and a complex topology. More detailed results are also available on our website[17]. Moreover,

[17] https://perso.liris.cnrs.fr/roland.kotto-kombi/autoscale/v2/

implementation, datasets and topologies can be downloaded for reproductibility.

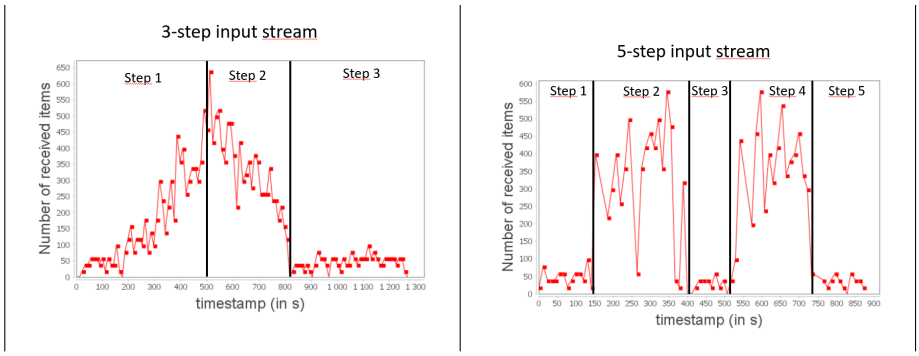


Figure 4: 3-step and 5-step streams

We built a 3-step synthetic stream with the following characteristics: 1) distribution with a small standard derivation 2) significant increase and decrease in load. Indeed, as illustrated on figure 4, input load is constant at a low rate. Load increases progressively before stabilizing at a high rate. Finally, rates decrease markedly until it reaches the initial low rate. We also add small and irregular fluctuations in order to simulate fluctuations in a real stream. To comply with good Storm practices, we implemented the replay of out-of-time stream elements.

Then, we apply a 5-step stream with sudden input rate peaks (see figure 4) to test the reactivity of our approach. This second stream moves from a low input to a very high one without a progressive transition as presented above. The input rate decreases suddenly before increasing again.

3.1.3 Criteria of evaluation

We summarize the main experimental parameters in table 3.1.3.

Table 1: Control parameters

window size	60s
monitoring frequency	10s
θ_{min} (AUTOSCALE)	0.3
θ_{max} (AUTOSCALE)	0.8
processing timeout (AUTOSCALE)	30s
combine strategy (AUTOSCALE)	max

We collect all measurements each 10 seconds and group them in windows of 60 seconds. AUTOSCALE considers that the activity level of an operator is low if it is lower than 0.3 and high or critical if it exceeds 0.8 (see chapter 5 section 3). A stream element is considered as out of time, or obsolete, if it has spent 30 seconds or more within the topology. This timeout takes time spent in pending queues and network latency into account in addition to processing times within operators. Finally, the consistency checking of reconfiguration at workflow scope (see chapter 5) considers the max as the combine function.

For each configuration, we measured the global latency of the topology (performance) and the number of dephased stream elements (result quality). Concerning system reactivity and the usage of resources, we observed parallelism degrees of each bolt.

3.2 Results on the microbenchmark

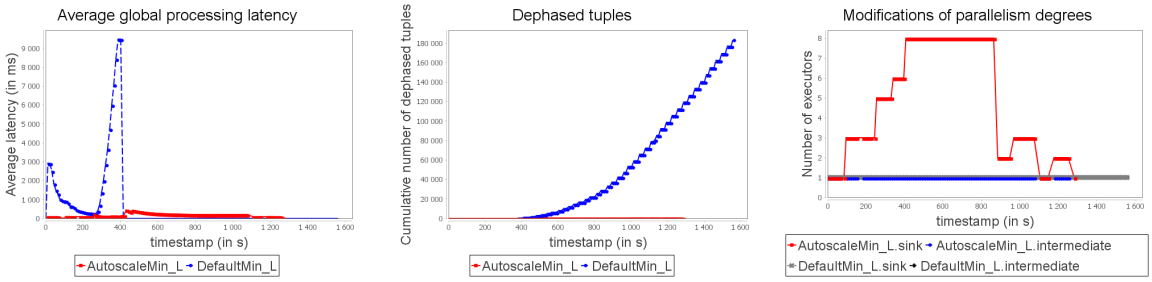
3.2.1 Application on the linear topology

We compare AUTOSCALE to the native scheduler of Apache Storm according to two configurations. We summarize experimental configurations for the linear topology in table 2:

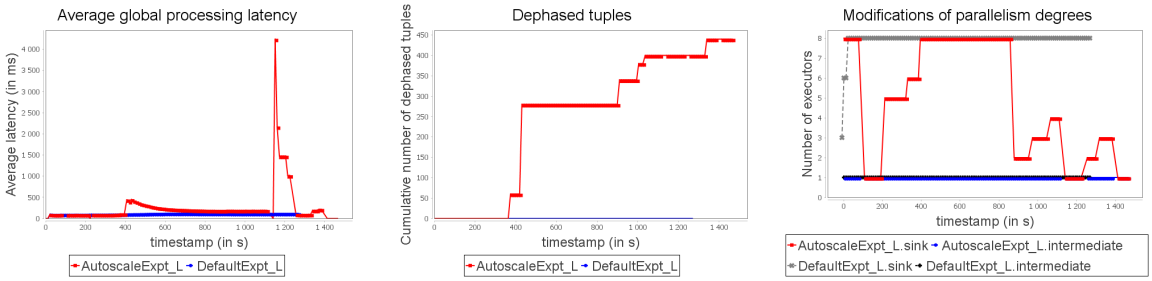
Table 2: Configuration of operators for Linear topology

	intermediate	sink
average latency	2ms	80ms
min degree	1	1
expert degree	1	8
max degree	8	8
CPU reservation	20.0	80.0
memory reservation	256Mb	512Mb

With the configuration *ConfMin*, the initial number of executors per bolt corresponds to minimal degrees (see table 2). Intuitively, the configuration *ConfMin* is adapted to small incoming loads but cannot handle large ones. With the configuration *ConfExpt*, initial numbers of executors correspond to expert degrees (see table 2). Expert degrees have been chosen with full knowledge of stream variation and latency of operators. This configuration can in fact handle the maximal load without wasting resources.



(a) Comparison between Storm (Default) and AUTOSCALE for the Linear topology in front of the 3-step stream with *ConfMin*.



(b) Comparison between Storm (Default) and AUTOSCALE for the Linear topology in front of the 3-step stream with *ConfExpt*.

Figure 5: Experimental results for the Linear topology

With ConfMin, we observe that the incoming load cannot be handled, thus leading to the complete congestion of the topology. Indeed, the topology is not able to process stream elements completely. As soon as congestion occurs, new stream elements emitted by the spout are dephased

and replayed indefinitely until a user intervenes (see figure 5a). On the contrary, our auto-parallelization strategy increases dynamically and automatically the parallelism degree of critical operators in order to adjust their capacities to future incoming loads. When the stream rate decreases, the parallelism degree decreases accordingly. It also prevents overusing resources that are no longer necessary.

With ConfExpt (see Figure 5b), we start with a configuration able to handle large loads. Nevertheless, this configuration overuses resources when the stream rate is low. It corresponds to the start and the end of the synthetic stream. Our auto-parallelization strategy reduces the parallelism degree when operators do not need large capacities. In this case, just as with ConfMin, the parallelism degree is adapted dynamically. Thus, AUTOSCALE achieves equivalent performance with approximately 37.5% less CPU and memory resources. The significant increase in topology latency with AUTOSCALE is due to a scale-in from three to one supervisor, which re-routes multiple stream elements and implies this significant overhead.

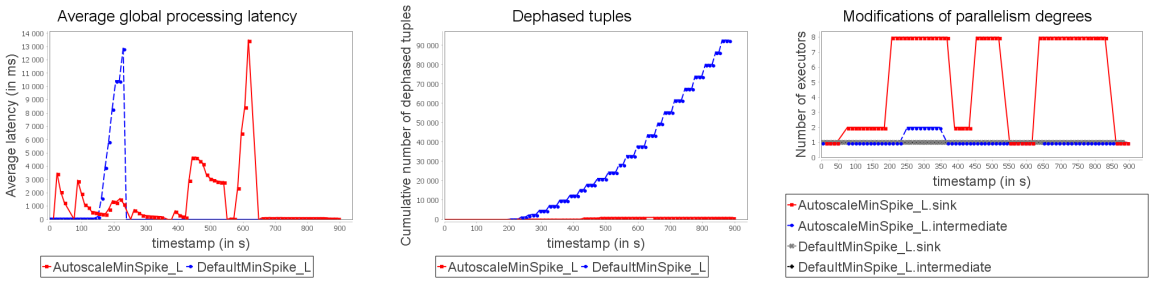


Figure 6: Comparison between Storm (Default) and AUTOSCALE for the Linear topology in front of the 5-step stream with *ConfMin*.

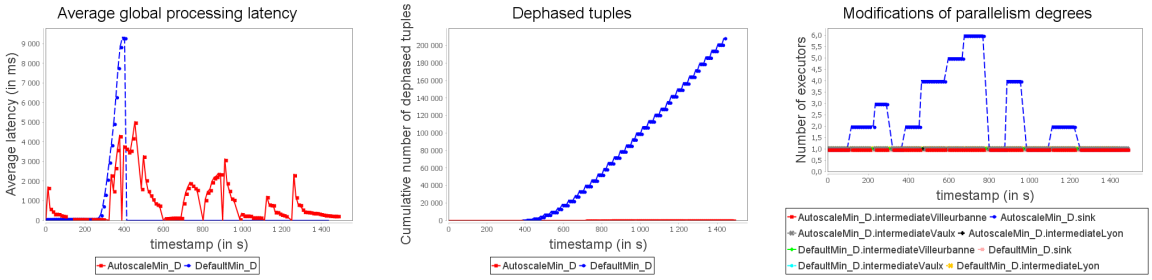
Unsurprisingly, we can see on Figure 6, that with ConfMin, Storm is unable to handle the sudden increase in input rate and that topology is completely congested. Even the decrease in input rate is not enough to restore normal operator activity. Indeed, due to replay of out-of-time stream elements more and more emissions are carried out by the spout, with the result that pending queues remain full. On the contrary, the AUTOSCALE approach reacts in multiple stages to adapt the capacity of each operator to fluctuations in input rate. Even the intermediate bolt, which has a very low latency, performs a scale-out as a precaution thanks to the global context. As a result of this adaptation, operators can consume their respective pending queues fast enough to benefit from the decrease in input rate. AUTOSCALE adapts dynamically the parallelism degree. Finally, a reconfiguration is performed as soon as a new peak appears.

3.2.2 Application on the diamond topology

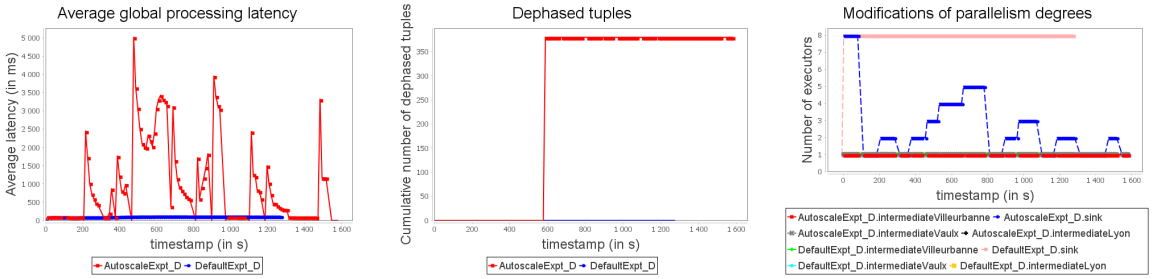
We reused same configurations for the diamond topology (see table 2).

For the configuration ConfMin, we observe that the default scheduler of Storm is not able to avoid the congestion as illustrated on figure 7a. As exposed for the linear topology, operators accumulate stream elements on their pending without being able to respect the maximal timeout. So, stream elements are replayed indefinitely which leads to complete congestion. AUTOSCALE detects a congestion risk before it becomes effective and performs scale-out on sink operator in order to adapt dynamically its parallelism degree.

With ConfExpt, AUTOSCALE decreases the parallelism of the sink operator after a short time because the input rate does not require an important parallelism degree. Then, AUTOSCALE



(a) Comparison between Storm (Default) and AUTOSCALE for the Diamond topology in front of the 3-step stream with *ConfMin*.



(b) Comparison between Storm (Default) and AUTOSCALE for the Diamond topology in front of the 3-step stream with *ConfExpt*.

Figure 7: Experimental results for the Diamond topology

adapts dynamically the parallelism of the sink operator as shown on figure 7b. Thus, it allows to uses around 35% less resources than the default scheduler.

When sudden increases of input rate are applied on the diamond topology, the default scheduler is not able to absorb the input stream and operators are completely congested. So, AUTOSCALE is able to performs scale-out to absorb the input stream almost as fast as it arrives. So, when the input rate decreases deeply between two peaks, AUTOSCALE performs scale-in to fit dynamically and automatically capacity of operators to their processing needs.

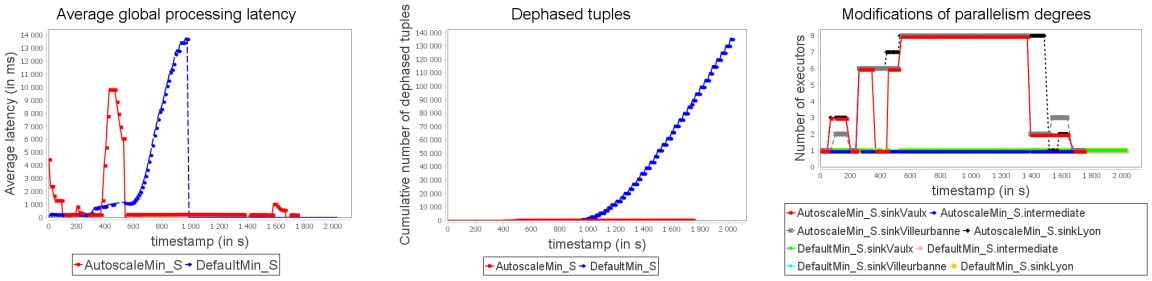
3.2.3 Application on the star topology

For the star topology, we used parameters presented in table 3 to define configurations ConfMin and ConfExpt. It is important to note that the average processing latency of sink operators is significantly than it is for linear and diamond topologies. It relies on the fact that a star topology contains multiple sink operators so they receive their input load is divided by a factor of 3 in comparison to linear and star topologies.

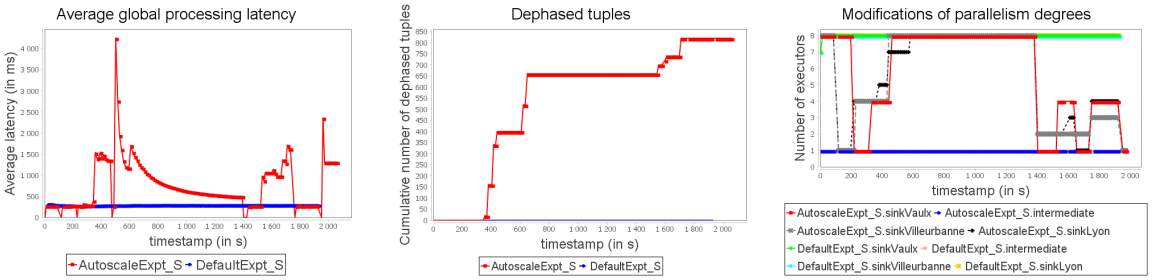
Table 3: Configuration of operators for Star topology

	intermediate	sink
average latency	2ms	240ms
min degree	1	1
expert degree	1	8
max degree	8	8
CPU reservation	20.0	30.0
memory reservation	256Mb	512Mb

Concerning the star topology, a similar behavior is observed. It is worth noting that because of the structure of the star topology, AUTOSCALE is able to detect the risk of congestion (see figure 8a) before it actually happens. This can be done according to the awareness of the global context offered by AUTOSCALE. Finally, AUTOSCALE reduces resource usage around 24.6% as illustrated on figure 8b. It is less than linear and diamond because there are more slow operators. Indeed, slow operators tend naturally to be more often in high or critical activity than fast ones.



(a) Comparison between Storm (Default) and AUTOSCALE for the Star topology in front of the 3-step stream with *ConfMin*.



(b) Comparison between Storm (Default) and AUTOSCALE for the Star topology in front of the 3-step stream with *ConfExpt*.

Figure 8: Experimental results for the Star topology

3.3 Results on advertising topology

We test our approach on an advertising topology mainly inspired from a topology used in [Peng et al., 2015] and available on Github[18] to validate our approach in a real context. We essentially modify the source to be able to reproduce the same stream with different configurations and add two operators (ip projection and ip processor) to obtain a complex topology. Moreover, we apply the 3-step input stream illustrated on figure 4.

This topology takes as input, logs representing an event linked to an advertisement on a web page. Each log is first deserialized before being transmitted to an event filter. Two projection operators receive stream elements from this filter, one looking for user IP addresses and the other for information on the ad. A join with a static dataset is performed to link the ad to a promotion campaign. Finally, IP and campaign processors increase users and campaign counts to update a remote database. The main interest of this topology is the significant selectivity of a filter operator (see figure 9), as this implies that a large increase in input rate will have a minor impact on final operators even if they have large latency in comparison with other operators.

[18] `https://github.com/yahoo/streaming-benchmarks/`

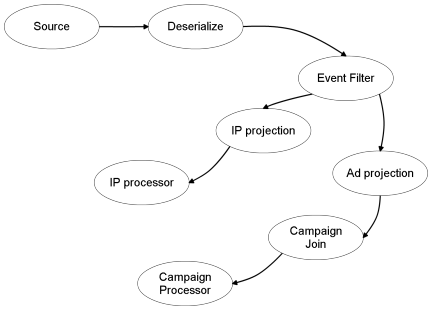
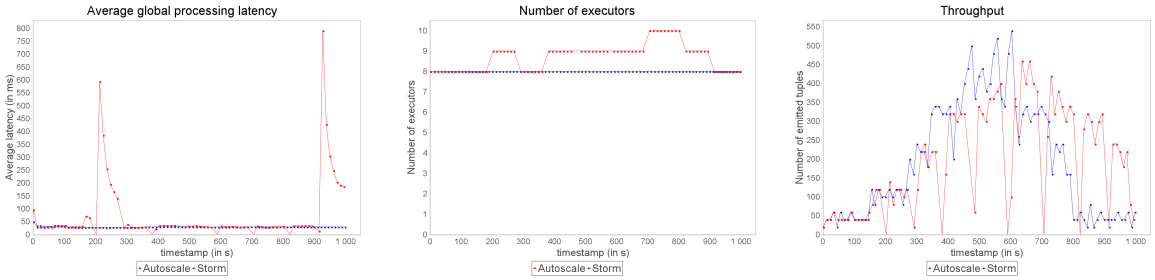
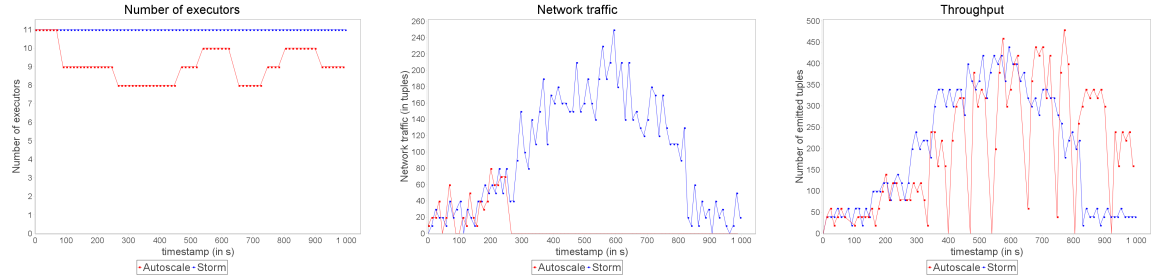


Figure 9: Advertising topology for stream benchmarking



(a) Comparison between Storm (Default) and AUTOSCALE for the Advertising topology in front of the 3-step stream with *ConfMin*.



(b) Comparison between Storm (Default) and AUTOSCALE for the Advertising topology in front of the 3-step stream with *ConfExpt*.

Figure 10: Experimental results for the Advertising topology

We observe that even if the topology is not congested, the AUTOSCALE approach performs some scale-outs in order to adapt operator capacity to their respective input rates as illustrated on figure 10a. This is due to the *combine* strategy (see chapter 5 section 4), which takes into account the maximum between local and global estimations as the globally consistent one. Therefore, when a slow operator begins to accumulate some stream elements on its pending queue, the AUTOSCALE approach performs a scale-out to avoid congestion. Nevertheless, AUTOSCALE performs a similar throughput even if there are some unnecessary reconfigurations in one case. We can estimate overheads induced by AUTOSCALE to 12% in comparison to actual needs in terms of CPU and memory requirements.

If a user bases his/her choice of parallelism degree exclusively on latencies, he/she will start the topology with some unnecessary executors (see figure 10b). The AUTOSCALE approach performs scale-in to fit capacities of operators to their respective processing needs. It is important to notice that the dynamic adaptation made by AUTOSCALE, combined with the scheduler, allows

all treatments to be collected on a single supervisor. With AUTOSCALE, Storm is able to handle biggest amount of data without generating network traffic, which is a large overhead factor, as explained in [Xu et al., 2014], and using 50% less resources.

4 Evaluation of AUTOSCALE+ with OSG

In this section, we evaluate the behavior of DABS on a different micro-benchmark composed of topologies sensitive and insensitive to stream element values. As a reminder, a topology is sensitive to stream element values if the processing latency is function of the value read in input. As we have demonstrated the effectiveness of AUTOSCALE compared to the native solution of Storm, we suggest here implementations of INC and RL strategies (see section 1) to evaluate the performance of DABS in front of common auto-parallelization strategies integrated in other DSMSs. We assume that these implementations are not as optimized as they are in other systems but they have characteristic behaviors that we want to observe ad analyze.

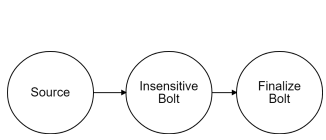
4.1 Experimental protocol

We used the same cluster described in section 3.1 but extended the number of processing units to 10. The module managing the distribution of stream elements between executors implements the *CustomStreamGrouping* interface of Storm API. We also deploy a MySQL database on Nimbus to store monitoring data. We summarize main experimental parameters in table 4.

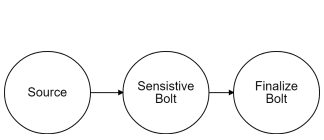
Table 4: Main parameters

window size	90s
monitoring frequency	10s
processing timeout	30s
α (AUTOSCALE+)	0.3
β (AUTOSCALE+)	0.8
combine strategy (AUTOSCALE+)	max
θ (OSG)	0.05
ϵ (OSG)	0.05

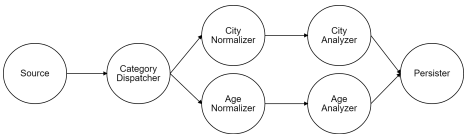
To validate our approach, we demonstrate its effectiveness on three topologies.



(a) Simple insensitive topology



(b) Simple sensitive topology



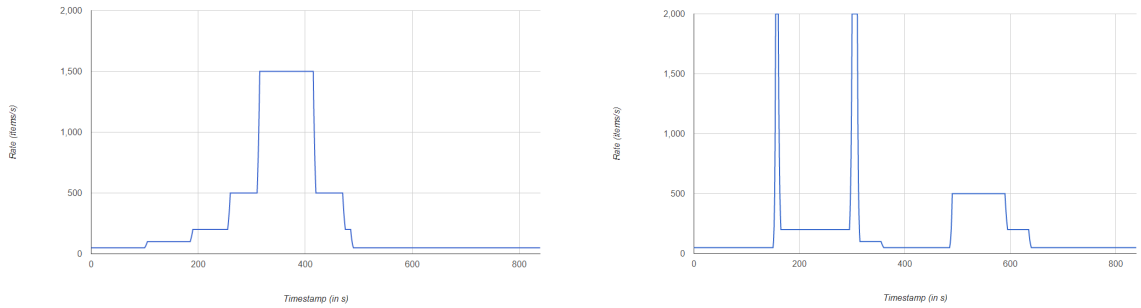
(c) Complex sensitive topology

The simple insensitive topology (see figure 11a) composed of a spout (Source) emitting stream elements without filtering them. These stream elements are processed by a bolt (InsensitiveBolt) applying a function with a time complexity independent of the value read in input. So, streams are all even when played in input of this topology. Finally, a bolt (FinalizeBolt) ends the computation of each stream element by sending a termination signal to Storm monitor.

The simple sensitive topology (see figure 11b) has the same structure as the simple insensitive topology but the function applied by the intermediate bolt (SensitiveBolt) has a time complexity

which depends directly from the value read in input.

The complex sensitive topology (see figure 11c) is composed of several operators with various selectivity factors and average processing latency. The spout (OpinionSource) emits stream elements concerning opinions submitted by users about a topic. Each opinion is described by information on the user, like its age and code representing its location, the topic and user opinion. Stream elements are sent to a bolt (CategoryDispatcher) filtering unnecessary attributes and depending on the branch downstream. In addition, it filters stream elements concerning a predefined list of irrelevant topics. A branch starts with a bolt (SensitiveBolt) retrieving information on user location from the code. This bolt has the exact same properties than the sensitive bolt of the simple sensitive topology. Indeed, depending on the code, retrieving information on the city takes more or less time. It allows us to compare the impact of workflow structure and complexity on bolt behavior and dynamic adaptation of its parallelism degree. Then, a bolt (CityAnalyzer) extracts relevant subgroups according to opinion and location. The other branch starting from the bolt CategoryDispatcher performs similar treatments in order to define subgroups on user opinion and age. Finally, the Persister takes in input descriptions of subgroups and persists them in a storage file system.



(a) Progressive stream

(b) Erratic stream

As illustrated on Figures 11a and 11b, we build two synthetic streams with following common features: 1) at least one critical increase in input rate leading the system to congestion with a minimal (one executor per operator) and static configuration 2) decrease of input rate to evaluate the elasticity of the system. For each stream, we can set the distribution law. It can be uniform over all possibles values or biased according to a zipf law with a predefined skew. These streams allow us to determine which impact has DABS while facing critical fluctuations in both input rate and value distribution.

4.2 Results on simple insensitive topology

4.2.1 Simple insensitive topology in front of the progressive stream

As presented above, the simple insensitive topology has an average processing latency independent of input values. It exclusively depends of the volume of stream elements to process. For this reason, we decide to enable only AUTOSCALE+ for parallelism management and let the default grouping solution of Storm, denoted *shuffle* grouping, route stream elements to executors.

As presented above, the reinforcement learning strategy relies on a knowledge base such as it can associate a parallelism degree to an input rate. While processing the progressive stream (see figure 11a), the reinforcement learning strategy increases the parallelism degree of the operator InsensitiveBolt (see figure 11a). We notice that the parallelism degree decreases

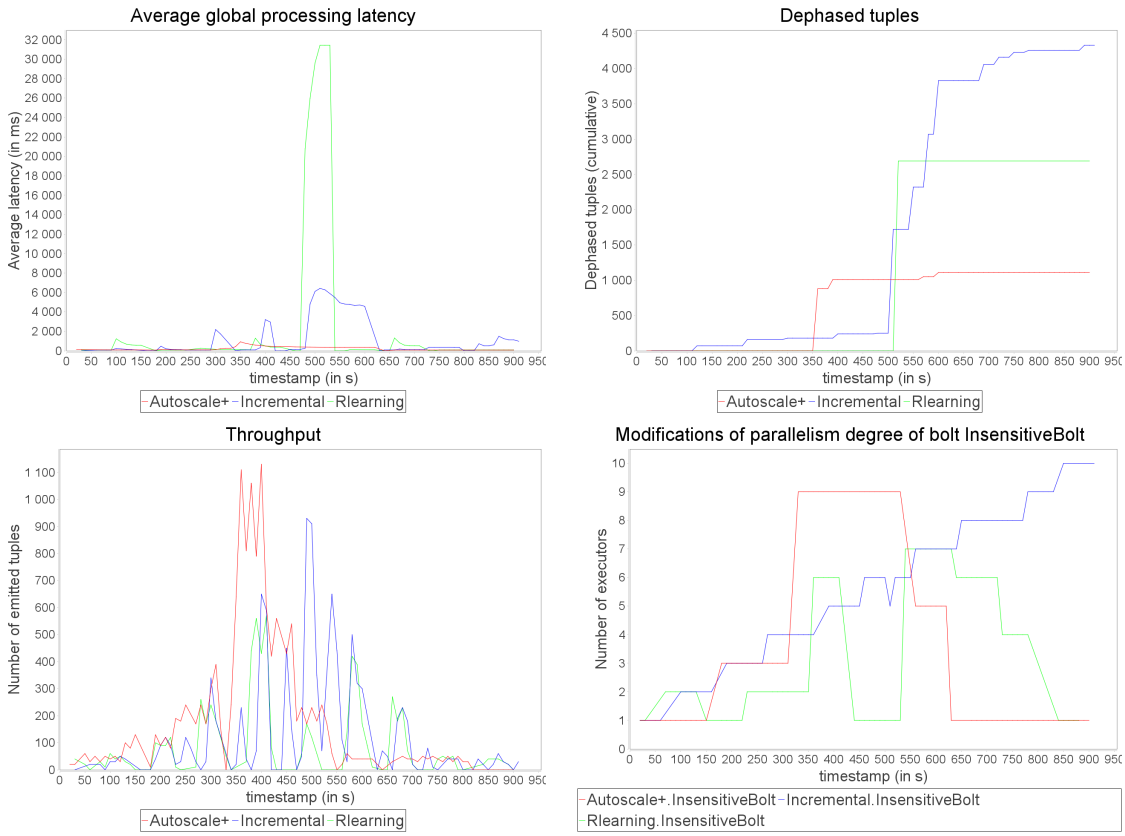


Figure 11: Simple insensitive topology with progressive stream

significantly as soon as the peak in input rate decreases. Nevertheless, most stream elements are just delayed by the processing rate and pending in input queue. So, the parallelism degree must be increased to avoid congestion. These modifications have a major impact on average processing latency and result quality. Indeed, reconfiguring the system while large volumes of data are running between operators causes increases of average processing latency exceeding the maximal threshold. It is due to reconfiguration overheads including migrations of pending queues and activation/deactivation of tasks on machines. It has also an impact on result quality because 17% stream elements cannot be processed under the maximal threshold.

In comparison, the incremental strategy increases continuously the parallelism degree of the operator as long as long the workload exceeds the processing rate. By workload, we refer to the sum of incoming and pending stream elements. Even if the parallelism degree is increased, it cannot reach a suitable value to handle maximal fluctuations in input rate. It implies large increases of average processing latency causing 29% losses of stream elements over the complete execution. In addition, in terms of resource usage, the incremental strategy requests 64% more active processing units than the reinforcement learning and 18% more than AUTOSCALE+.

While using AUTOSCALE+, Storm is able to anticipate suitable parallelism degrees over the complete execution. Even if AUTOSCALE+ tends to overestimate the required parallelism degree due to regression, reconfiguration overheads are compensate by benefits on processing latency. Actually, the average processing latency remains stable over the complete execution reducing losses to 7%. It can also be observed on throughput as AUTOSCALE+ is able to maintain a throughput close to input rate with short time shift.

4.2.2 Simple insensitive topology in front of the erratic stream

We applied the erratic stream (see figure 11b page 117) in input of the simple insensitive topology to test the reactivity of each auto-parallelization strategy while facing sudden and large peaks in input rate.

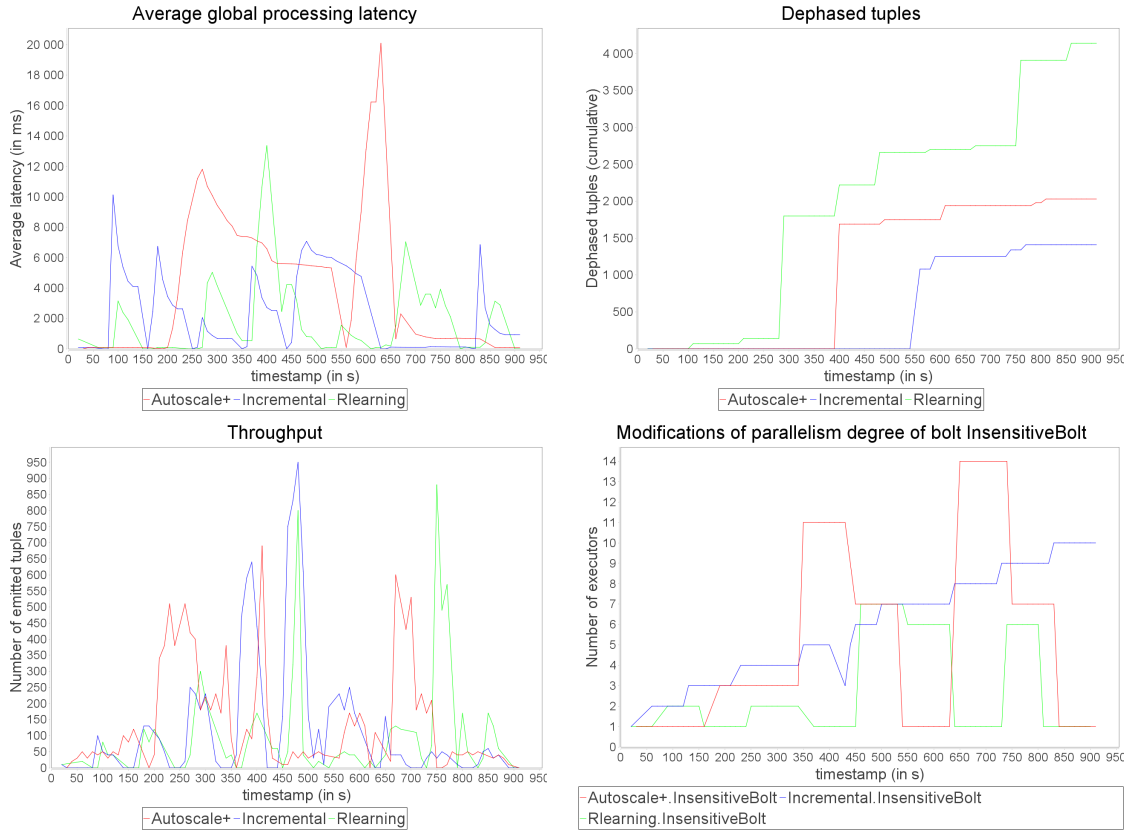


Figure 12: Simple insensitive topology with erratic stream

The reinforcement learning strategy increases and decreases the parallelism degree of the operator InsensitiveBolt according to two main peaks corresponding essentially to the increase in input rate happening at the end of the erratic stream. Indeed, brief increases in input rate does not imply important scale-out. They do not increase significantly the average input rate on recent history so the parallelism degree does not require large increase of its parallelism degree. Nevertheless, the sudden accumulation of a large number of stream elements on pending queues increases the average processing latency. The impact on result quality remains negligible with only 0.6% of stream elements lost over the complete execution.

The incremental strategy benefits from the short duration of peaks in input rate. Indeed, as the incremental strategy over-provisions the operator, resources necessary to handle brief increases in input rate are available. So, the average processing latency increases significantly only when the input rate remains high for a long duration like it is happening at the end of the erratic stream. Losses of stream elements are reduced to 19% over the complete execution but the usage of processing units remains 85% higher than the reinforcement learning strategy and 4% higher than AUTOSCALE+.

Concerning AUTOSCALE+, the appearance of sudden increases in input rate has an impact on the regression model used to anticipate processing requirements. So, the parallelism degree of the

operator is increased sooner causing a degradation of the average processing latency as the the input rate decreases immediately. AUTOSCALE+ overestimates processing requirements as the immediate decrease in input rate cannot be predicted. This overestimation allows to maintain a throughput close to the input rate and process the entire stream with 18% losses. As critical increase and decrease of input rate are sudden and brief, they cannot be anticipated and affect the processing latency before AUTOSCALE+ reconfigures the system.

4.3 Results on simple sensitive topology

4.3.1 Simple sensitive topology in front of the progressive stream

We present now same results with the simple sensitive topology. Input streams follow same fluctuations in input rate but the distribution of values is biased. It follows a zipf distribution with a skew of 1,5 as used in [Rivetti et al., 2016]. To compensate the skew in value distribution, we used the OSG grouping solution for all configurations. So, auto-parallelization strategies are not penalized by imbalance between executors of same operator.

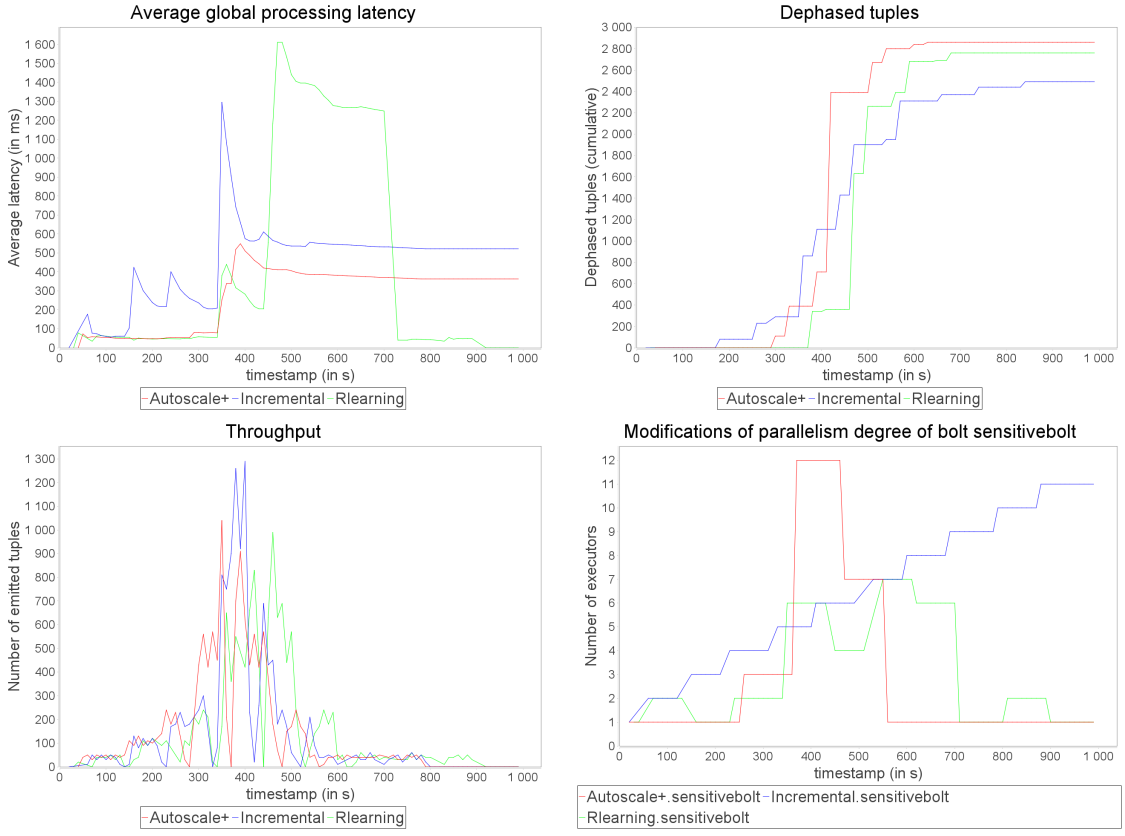


Figure 13: Simple sensitive topology with progressive stream

As observed with the simple insensitive topology, AUTOSCALE+ anticipates processing requirements and is able to maintain a smaller processing latency while the stream is at its maximal rate. Nevertheless, the reinforcement learning strategy is able to decrease significantly the processing latency when the input rate decreases. The incremental strategy is penalized by OSG because the frequent modification of parallelism degree forces OSG to reevaluate its routing policy. Indeed, as OSG tracks the load of each task associated to an operator, the frequent

modification of parallelism degree involves frequent updates of the monitoring structure.

In terms of result quality, the incremental strategy is able to keep loss of stream elements at 17% but it requires 94% more active processing units than reinforcement learning strategy and 83% more than DABS (AUTOSCALE+ with OSG). The reinforcement learning strategy reacts to increase in input rate and cannot prevent 19% of losses while AUTOSCALE+ looses 20% of stream elements during the complete execution.

Concerning throughput, all solutions deliver close performance even if AUTOSCALE+ remains the auto-parallelization strategy keeping the smaller time shift between fluctuation in input rate and throughput.

4.3.2 Simple sensitive topology in front of the erratic stream

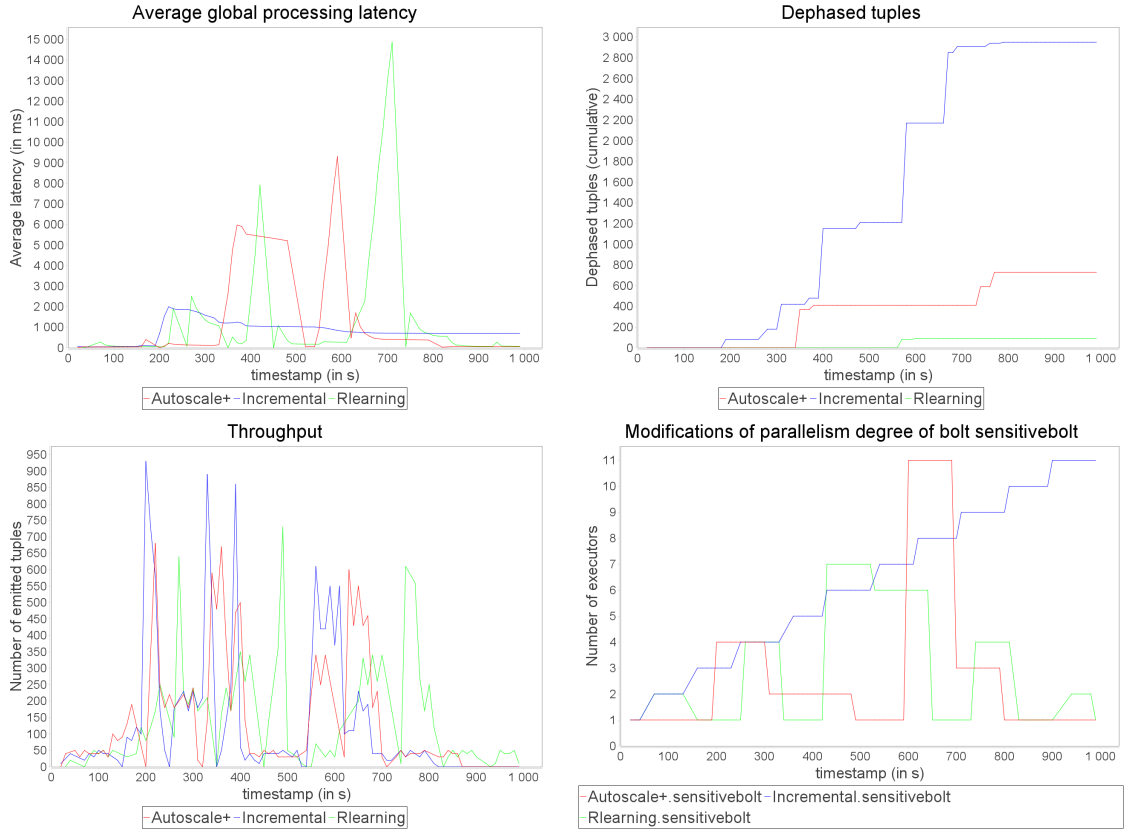


Figure 14: Simple sensitive topology with erratic stream

As discussed above, the erratic stream has two short fluctuations in input rate spaced by a significant decrease. We notice that DABS is significantly more accurate than AUTOSCALE+ alone. Indeed, the average processing latency remains low except two punctual increases during the first two peaks in input rate and before the last increase in input rate which lasts longer (see figure 11b). DABS reduces losses of stream elements to 4.8% even if sudden increases cannot be anticipate. It is due to OSG which delivers fast routing when slightly overprovisionned in executors, what DABS does through regression. It makes the major difference with the reinforcement learning strategy which provides only the suitable number of executors to avoid congestion. The incremental strategy maintains a low processing latency and delivers a throughput close to the input rate but it still uses considerably more resources to complete the treatment of the entire

stream. Moreover, losses are more important than reinforcement learning solution and DABS and go up to 20% over the complete execution. It is also interesting to notice that DABS maintains a lower time shit between fluctuations in input rate and throughput than the reinforcement learning solution. So, even if stream elements arrive at important rates, preventive reconfiguration performed by DABS does not delay their treatment.

4.4 Results on complex sensitive topology

4.4.1 Complex sensitive topology in front of the progressive stream

After testing the behavior of auto-parallelization strategies in front of simple topologies, we applied same biased streams in input of a complex topology (see figure 11c). This topology is more representative of real-world continuous queries. It includes common operators as filters on values and attributes, joins with static bases and also user-defined functions from expert domains like data mining.

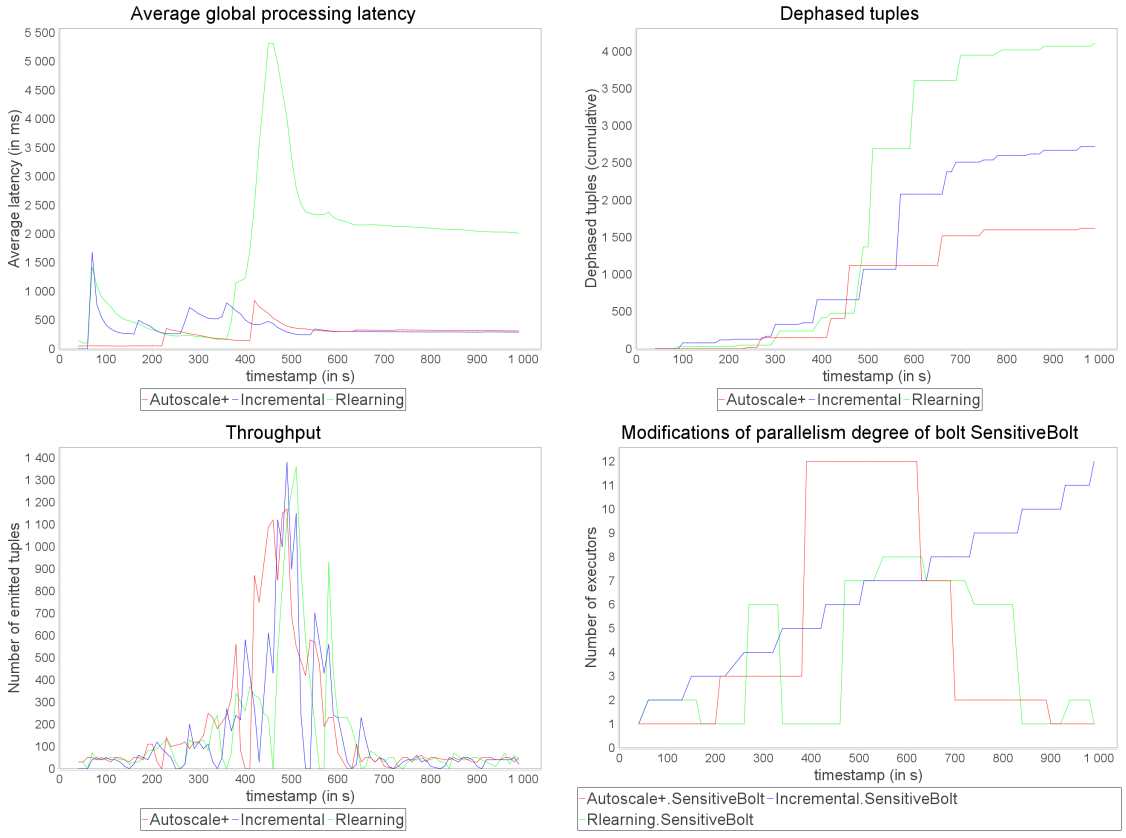


Figure 15: Complex sensitive topology with progressive stream

There are two significant differences between this experimental setup and ones presented above:

- The operator becoming critical is not directly connected to the source. Stream elements emitted by the source are filtered and transformed by other operators before reaching the operator. Its inputs may not present same fluctuations than original ones.

- Operators processing stream elements upstream critical operators can be analyzed to evaluate consistency at workflow scope. Indeed, simple topologies composed of a single operator cannot benefit from analysis at workflow scope because there is no complementary estimation to compare with local estimations.

While playing the progressive stream in input of the complex topology, DABS anticipates the first scale-out compared to AUTOSCALE+ with the simple topology. It also maintains an important parallelism degree longer than AUTOSCALE+ does for the simple topology. According to these modifications of parallelism degree, DABS maintains a lower processing latency than the reinforcement learning strategy and requests 30% less active resources than incremental strategy for equivalent performance.

In terms of result quality, DABS benefits from the analysis at workflow scope to improve anticipation of scale-out and to reduce losses to 10%. In comparison, reinforcement learning strategy reduces losses only to 27% and the incremental strategy to 18%. It is due to the accumulation of stream elements which has a greater impact of processing latency when multiple operators are affected downstream.

Except a time shift when the stream reaches its maximal input rate, all strategies maintain a throughput close to input rate. DABS is slightly more reactive than other auto-parallelization strategies and deliver a maximal throughput sooner.

4.4.2 Complex sensitive topology in front of the erratic stream

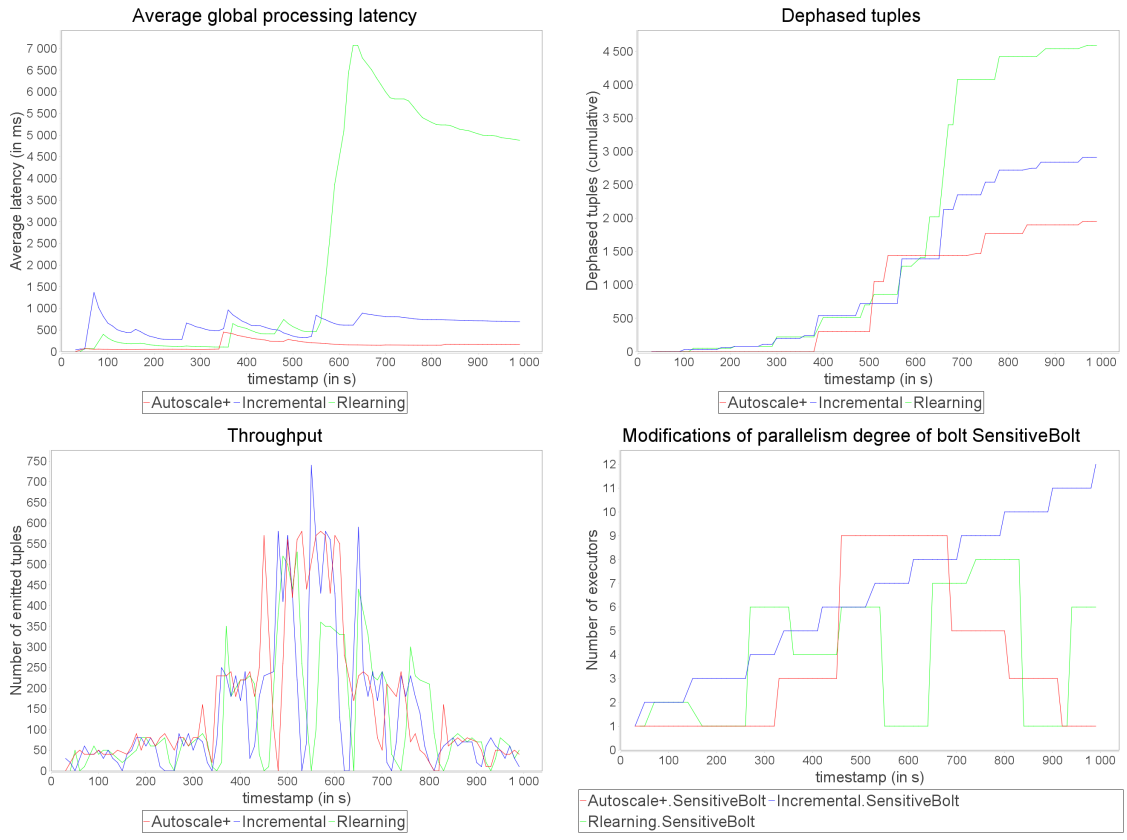


Figure 16: Complex sensitive topology with erratic stream

The complex structure of the topology makes an important difference in the treatment of the erratic stream. For all auto-parallelization strategies, we can observe that brief but large fluctuations in input rate do not affect significantly the operator. These fluctuations are absorbed by the processing rate of upstream operators and inter-operator transmissions.

In this context, DABS performs all scale-out before any scale-in. It allows the system to maintain a low latency even when it faces critical variations in input rate. The incremental strategy also keep a low processing latency but uses 36% more active processing units to achieve treatments. On its side, the reinforcement learning strategy keeps reacting to average input rate to adjust parallelism degree. It results in an inconsistent scale-in at workflow scope which is contradicted afterwards. It has a major impact on processing latency and result quality. Actually, when DABS and the incremental strategy maintains losses at 13% and 19% respectively, the reinforcement learning strategy looses 26% of stream elements at the end of the observation.

5 Discussion

To sum up, we presented the evaluation of AUTOSCALE and AUTOSCALE+ approaches for auto-parallelization of operators. They both present the advantage to avoid congestion of operators through an automatic of the parallelization. It appears that AUTOSCALE may involve some reconfiguration overheads when it overestimates the workload in near future. Nevertheless, it consistently reduces the usage of resources at runtime for equivalent performance.

Moreover, the experimental evaluation highlights the ability of AUTOSCALE+ to maintain a low processing latency compared to auto-parallelization strategies commonly used in DSMSs. In most cases, AUTOSCALE+ looses less stream elements than other auto-parallelization strategies while facing critical fluctuations in input rate. The combination with the load balancing strategy OSG improves the performance of the system while processing streams with high skew in data distribution.

8

Conclusion

Contents

1 Summary of our contributions

The focus of this research is congestion management. As presented in chapter 3, congestion may be due to different reasons. For a given operator belonging to a continuous query, an input rate significantly greater than the processing rate induce an accumulation of stream elements on the input queue. This accumulation increases the time spent by each stream element in the input queue up to saturation. In such case, exploiting data parallelism can be a solution. Indeed, duplicating an operator increases its overall processing rate.

To answer this problem, users may set maximal parallelism degree for each operator. It ensures that all available resources are exploited to process streams. Nevertheless, this solution is costly as it requires to consume resources on all available machines. In a *Green IT* context and from a economical point of view, this solution implies major waste of resources.

Thus, we suggest the auto-parallelization strategy AUTOSCALE which performs scale-in and scale-out according to fluctuations in input rate. It adapts usage of resources to processing requirements avoiding congestion of operators and overconsumption of resources at the same time. Each reconfiguration is triggered after the following steps:

- At operator scope, a monitoring module collects metrics (input rate and processing latency) describing the execution of the operator. From the recent history, AUTOSCALE estimates the input workload and the processing capacity in near future. According to these estimations, we suggest a metric of local activity used to detect reconfigurations requirements in a proactive way.

- From these estimations of processing requirements in near future, AUTOSCALE identifies which operators may benefit from scale-in or scale-out according to metrics computed locally.

- We propose an algorithm checking the consistency of reconfiguration requirements at workflow scope. Through an exploration of the workflow from sources to sinks, the algorithm

validates or invalidates suggested local reconfiguration requirement according to modifications validated upstream. Indeed, AUTOSCALE computes a global activity metric relying on the processing capacity estimated locally and a combination of input workloads estimated locally and upstream. It aims at triggering a consistent set of reconfigurations, including anticipated scale-out of operators. It improves system stability by avoiding antagonist scale-in and scale-out.

The execution model considered by AUTOSCALE does not take concurrency for resource usage and load imbalance into account. To bridge this gap, we extend AUTOSCALE to AUTOSCALE+ through the following improvements:

- The monitoring module collects the CPU time allocated to each task associated to an operator during the recent history. It allows to evaluate which CPU is effectively available for each task and take the concurrency between tasks into account.
- From these new metrics, we suggest a resource-aware variant of our global activity metric. This variant evaluates the processing capacity of a given operator according to available resources of machines running its associated tasks. So, each scale-in/out is triggered accurately with regards to resources of processing units.
- We modify the consistency checking at workflow scope to perform both local and global estimations in one-pass. On complex topologies with several operators, it reduces the computation time of the set of reconfigurations globally consistent.
- We associate AUTOSCALE+ to the load balancing strategy OSG [Rivetti et al., 2016] such as the accuracy of scale-in/out is not biased by a skew in data distribution. Thus, this combined solution is able to adapt dynamically and automatically workflows while facing critical fluctuations in input rate and data distribution at the same time.

Experimental studies of AUTOSCALE and AUTOSCALE+ show respectively that consistency checking at workflow improves significantly the stability of the system and the association with a load balancing strategy serves as guarantee that even and uneven streams are processed similarly.

Concerning AUTOSCALE, experiments on a micro-benchmark shows that AUTOSCALE avoids congestion of workflows when the input rate becomes critical. According to our experiments, AUTOSCALE is able to save up to 37,5% less CPU and memory resources and at least 12% for equivalent performance. While computing several continuous queries on the same cluster, it improves significantly the scalability of the system.

Improvements included in AUTOSCALE+ let users define more accurately their priorities in terms of performance and resource usage. With an appropriate configuration, AUTOSCALE+ is able to consume up to 18% less resources than a reactive auto-parallelization strategy based on reinforcement learning. Moreover, AUTOSCALE+ is able to limit losses to 7% while receiving streams at critical input rates.

Combining AUTOSCALE+ with the load balancing strategy OSG significantly improves the performance and usage of resources of the DSMS. Indeed, even when a biased stream in terms of distribution of values is received at rates changing suddenly, losses are limited to 10% and resource usage is reduced by 30% compared to common solutions for elastic stream processing implemented in the DSMS Apache Storm. We develop some auto-parallelization strategies as custom modules integrated into Apache Storm, including AUTOSCALE and AUTOSCALE+ and demonstrate the effectiveness of AUTOSCALE and AUTOSCALE+ approach in order to manage

congestion of operators. We develop microbenchmarks in order to evaluate benefits brought by these solutions while processing a variety of continuous queries over different streams. We give the possibility to users to reuse these workflows and streams for further developments and works.

2 Perspectives

Contributions suggested during this research can be improved and extended in several directions to reach a fully elastic stream processing with a better synergy between adaptation levels presented in chapter 4.

Exploiting user profile

In chapter 5, we present a combine strategy which takes local and global workload estimations into account to return the considered estimation. We can offer the possibility to users to prioritize performance (*i.e.*, performing scale-out as soon as it can prevent any increase of the end-to-end latency) or resource usage (*i.e.*, performing scale-out only when congestion cannot be avoided and performing scale-in as soon as possible). Indeed, depending on the critical aspect of a continuous query and the financial cost of computations, users may not have same priorities. The problem is then to integrate user profile within the auto-parallelization strategy. It raises several issues such as the identification of metrics describing user profile and the integration of these metrics in the decision algorithm.

Generalizing the resource-aware scheduler

As studied in chapter 6, AUTOSCALE+ takes advantage of the load balancing strategy OSG to perform accurate scale-in/out in presence of bias in data distribution. In the same idea, when a task suffers from a lack of resources on its host (*e.g.*, the scheduling strategy assigned several complex operators on the same machine), imbalance between processing capacities appears because of concurrency between threads. To compensate such kind of imbalance, it would be interesting to associate AUTOSCALE+ with a resource-aware scheduling strategy. Such strategies [Peng et al., 2015, Aniello et al., 2013] have already been investigated in the literature but they take as input static user constraints defining resource requirements for each operator. Even if it is assumed that users are experts knowing the time and space complexity of each operator, it remains particularly difficult to estimate resource requirements such as the ratio of allocated CPU time and memory space. Moreover, it induces that there is a minimal subset of available machines which will stay active even if the input stream is significantly lower than the expected input rate (*e.g.*, a road monitoring stream at different hours of a day). Instead of considering static user constraints, it would be interesting to consider user preferences which can be modified at runtime depending on effective resource usages. For example, let consider an operator associated to a preference of 20% on CPU usage. If this operator requires in average 50% at runtime, the scheduler will consider that each task applying this operator requires 50% instead of user preference. The main issue consists in the evaluation of processing requirements of each operator at runtime as it depends on several parameters like the type of the operator (stateless or stateful) and its sensitivity to values of stream elements. Another challenge is to evaluate when it is beneficial to move an operator from a processing unit to another as resource usage (*e.g.*, CPU usage) may vary quickly.

Introducing operator model for auto-parallelization

Auto-parallelization strategies suggested in this research focus on adapting the processing capacity to the estimated workload without considering the type of the operator (stateless or stateful). As stateful operators process stream elements micro-batch by micro-batch, evaluating the accurately workload requires to distinguish stream elements being processed and stream elements pending in input queue until the next micro-batch. More generally, as several DSMS support declarative languages for query definition, it could be interesting to exploit algebraic properties of predefined operators (selectivity factor, time complexity) to improve the regression model used by AUTOSCALE. The problem is then to define for each class of operators, specific features refining the regression model in order to anticipate congestion with a better accuracy.

Auto-parallelization on limited resources

During this research, we assumed that users submit their continuous queries to a cluster managed by a service provider. With the democratization of small single-board computers (*e.g.*, Rapsberry Pi), establishing a cluster of multiple processing units located at different places becomes a viable solution according to the economical aspect. It allows to process data closer from stream sources [Logothetis et al., 2011] and have a control on distributed execution. Nevertheless, such solution is limited by the performance of each processing unit. Depending on the input rate and the complexity of continuous queries, treatments may be limited by resources [Yang et al., 2012]. To perform elastic stream processing, it is then necessary to prioritize some operators. The prioritization of scale-out has been studied in [Xu and Peng, 2016] and has been presented as the impact of an operator on the throughput of a workflow. Depending on the application, some operators may have a great impact on results in terms of quality without generating the greatest volume of outputs. It appears interesting to refine the notion of impact of an operator in a stream processing context. Thus, AUTOSCALE+ can be extended with a prioritization of operators combined to the resource-aware activity metric in order to maximize the result quality under resource constraints. It could also be extended to the preemption of resources by some operators even if it implies reducing reserved resources of other operators. In such context, AUTOSCALE+ could also be combined with a load shedding strategy to discard judiciously stream elements of non-priority operators.

Bibliography

[Abadi et al., 2005] Abadi, D. J., Ahmad, Y., Balazinska, M., Cherniack, M., hyon Hwang, J., Lindner, W., Maskey, A. S., Rasin, E., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. (2005). The design of the borealis stream processing engine. In *In CIDR*, pages 277–289.

[Abadi et al., 2003] Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. (2003). Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139.

[Ahmad et al., 2005] Ahmad, Y., Berg, B., Cetintemel, U., Humphrey, M., Hwang, J.-H., Jhingran, A., Maskey, A., Papaemmanouil, O., Rasin, A., Tatbul, N., Xing, W., Xing, Y., and Zdonik, S. (2005). Distributed operation in the borealis stream processing engine. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 882–884, New York, NY, USA. ACM.

[Akidau et al., 2013] Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., and Whittle, S. (2013). Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044.

[Akidau et al., 2015] Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., and Whittle, S. (2015). The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8:1792–1803.

[Amini et al., 2006] Amini, L., Andrade, H., Bhagwan, R., Eskesen, F., King, R., Selo, P., Park, Y., and Venkatramani, C. (2006). Spc: A distributed, scalable platform for data mining. In *Proceedings of the 4th international workshop on Data mining standards, services and platforms*, pages 27–37. ACM.

[Andrzejak et al., 2010] Andrzejak, A., Kondo, D., and Yi, S. (2010). Decision model for cloud computing under sla constraints. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 257–266.

[Aniello et al., 2013] Aniello, L., Baldoni, R., and Querzoni, L. (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*, pages 207–218.

[Arasu et al., 2004] Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Ito, K., Motwani, R., Srivastava, U., and Widom, J. (2004). Stream: The stanford data stream management system. Springer.

[Arasu et al., 2006] Arasu, A., Babu, S., and Widom, J. (2006). The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142.

[Babcock et al., 2002] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. (2002). Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA. ACM.

[Babcock et al., 2004] Babcock, B., Datar, M., and Motwani, R. (2004). Load shedding for aggregation queries over data streams. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 350–361. IEEE.

[Babu and Widom, 2001] Babu, S. and Widom, J. (2001). Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120.

[Backman et al., 2012] Backman, N., Pattabiraman, K., Fonseca, R., and Cetintemel, U. (2012). C-mr: Continuously executing mapreduce workflows on multi-core processors. In *Proceedings of Third International Workshop on MapReduce and Its Applications Date*, MapReduce '12, pages 1–8, New York, NY, USA. ACM.

[Balazinska et al., 2004] Balazinska, M., Balakrishnan, H., and Stonebraker, M. (2004). Load management and high availability in the medusa distributed stream processing engine. In *In Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 929–930.

[Biem et al., 2010] Biem, A., Bouillet, E., Feng, H., Ranganathan, A., Riabov, A., Verscheure, O., Koutsopoulos, H., and Moran, C. (2010). Ibm infosphere streams for scalable, real-time, intelligent transportation services. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1093–1104. ACM.

[Carbone et al., 2015] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4).

[Cardellini et al., 2016] Cardellini, V., Nardelli, M., and Luzi, D. (2016). Elastic stateful stream processing in storm. In *High Performance Computing & Simulation (HPCS), 2016 International Conference on*, pages 583–590. IEEE.

[Carter and Wegman, 1979] Carter, J. L. and Wegman, M. N. (1979). Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154.

[Castro Fernandez et al., 2013] Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E., and Pietzuch, P. (2013). Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA. ACM.

[Cetintemel, 2003] Cetintemel, U. (2003). The aurora and medusa projects. *Data Engineering*, 51(3).

[Chambers et al., 2010] Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R. R., Bradshaw, R., and Weizenbaum, N. (2010). Flumejava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM.

[Chandrasekaran et al., 2003] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S. R., Reiss, F., and Shah, M. A. (2003). Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA. ACM.

[Chaudhuri, 1998] Chaudhuri, S. (1998). An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, pages 34–43, New York, NY, USA. ACM.

[Chen et al., 2000] Chen, J., DeWitt, D. J., Tian, F., and Wang, Y. (2000). Niagaracq: A scalable continuous query system for internet databases. *SIGMOD Rec.*, 29(2):379–390.

[Cherniack et al., 2003a] Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., and Zdonik, S. (2003a). Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA.

[Cherniack et al., 2003b] Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., and Zdonik, S. (2003b). Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA.

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

[Das et al., 2005] Das, R., Tesauro, G., and Walsh, W. E. (2005). Model-based and model-free approaches to autonomic resource allocation. *IBM Ressearch Report, RC*, 23802.

[De Matteis and Mencagli, 2016] De Matteis, T. and Mencagli, G. (2016). Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 13:1–13:12, New York, NY, USA. ACM.

[Dean and Ghemawat, 2004] Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA. USENIX Association.

[Deshpande, 2004] Deshpande, A. (2004). An initial study of overheads of eddies. *SIGMOD Rec.*, 33(1):44–49.

[Ding et al., 2015] Ding, J., Fu, T. Z., Ma, R. T., Winslett, M., Yang, Y., Zhang, Z., and Chao, H. (2015). Optimal operator state migration for elastic data stream processing. *arXiv preprint arXiv:1501.03619*.

[Garcia-Molina, 2008] Garcia-Molina, H. (2008). *Database systems: the complete book*. Pearson Education India.

[Gedik, 2014] Gedik, B. (2014). Partitioning functions for stateful data parallelism in stream processing. *The VLDB Journal*, 23(4):517–539.

[Gedik et al., 2008] Gedik, B., Andrade, H., Wu, K.-L., Yu, P. S., and Doo, M. (2008). Spade: The system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1123–1134, New York, NY, USA. ACM.

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

[Golab et al., 2004] Golab, L., Garg, S., and Özsu, M. T. (2004). *On Indexing Sliding Windows over Online Data Streams*, pages 712–729. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Graefe, 1993] Graefe, G. (1993). Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169.

[Gulisano et al., 2012] Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., Soriente, C., and Valduriez, P. (2012). Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365.

[Heinze et al., 2014a] Heinze, T., Jerzak, Z., Hackenbroich, G., and Fetzer, C. (2014a). Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 13–22, New York, NY, USA. ACM.

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

[Hirzel et al., 2014] Hirzel, M., Soulé, R., Schneider, S., Gedik, B., and Grimm, R. (2014). A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4):46:1–46:34.

[Hochreiner et al., 2015] Hochreiner, C., Schulte, S., Dustdar, S., and Lecue, F. (2015). Elastic stream processing for distributed environments. *IEEE Internet Computing*, 19(6):54–59.

[Hummer et al., 2013] Hummer, W., Satzger, B., and Dustdar, S. (2013). Elastic stream processing in the cloud. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 3(5):333–345.

[Ishii and Suzumura, 2011] Ishii, A. and Suzumura, T. (2011). Elastic stream computing with clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 195–202. IEEE.

[Jain et al., 2006] Jain, N., Amini, L., Andrade, H., King, R., Park, Y., Selo, P., and Venkatramani, C. (2006). Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 431–442. ACM.

[Jain et al., 2008] Jain, N., Mishra, S., Srinivasan, A., Gehrke, J., Widom, J., Balakrishnan, H., Çetintemel, U., Cherniack, M., Tibbetts, R., and Zdonik, S. (2008). Towards a streaming sql standard. *Proc. VLDB Endow.*, 1(2):1379–1390.

[Jiang and Chakravarthy, 2003] Jiang, Q. and Chakravarthy, S. (2003). Queueing analysis of relational operators for continuous data streams. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, CIKM '03, pages 271–278, New York, NY, USA. ACM.

[Kalashnikov, 2013] Kalashnikov, V. V. (2013). *Mathematical methods in queuing theory*, volume 271. Springer Science & Business Media.

[Kang et al., 2003] Kang, J., Naughton, J. F., and Viglas, S. D. (2003). Evaluating window joins over unbounded streams. In *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, pages 341–352.

[Karp, 1988] Karp, R. M. (1988). A survey of parallel algorithms for shared-memory machines. Technical report, Berkeley, CA, USA.

[Kendall, 1953] Kendall, D. G. (1953). Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain. *The Annals of Mathematical Statistics*, pages 338–354.

[Lei and Rundensteiner, 2014] Lei, C. and Rundensteiner, E. A. (2014). Robust distributed query processing for streaming data. *ACM Trans. Database Syst.*, 39(2):17:1–17:45.

[Li and McLeod, 1981] Li, W. and McLeod, A. (1981). Distribution of the residual autocorrelations in multivariate arma time series models. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 231–239.

[Lin et al., 2008] Lin, W., Yang, M., Zhang, L., and Zhou, L. (2008). Pacifica: Replication in log-based distributed storage systems.

[Logothetis et al., 2011] Logothetis, D., Trezzo, C., Webb, K. C., and Yocum, K. (2011). In-situ mapreduce for log processing. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 9–9, Berkeley, CA, USA. USENIX Association.

[Lohrmann et al., 2015] Lohrmann, B., Janacik, P., and Kao, O. (2015). Elastic stream processing with latency guarantees. In *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*, pages 399–410. IEEE.

[Lorido-Botran et al., 2014] Lorido-Botran, T., Miguel-Alonso, J., and Lozano, J. A. (2014). A review of auto-scaling techniques for elastic applications in cloud environments. *J. Grid Comput.*, 12(4):559–592.

[Madden et al., 2002] Madden, S., Shah, M., Hellerstein, J. M., and Raman, V. (2002). Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 49–60, New York, NY, USA. ACM.

[Maurer et al., 2011] Maurer, M., Brandic, I., and Sakellariou, R. (2011). *Enacting SLAs in Clouds Using Rules*, pages 455–466. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Mehta and DeWitt, 1995] Mehta, M. and DeWitt, D. J. (1995). Managing intra-operator parallelism in parallel database systems. In *Proceedings of the 21th International Conference on*

Very Large Data Bases, VLDB '95, pages 382–394, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Mishra and Eich, 1992] Mishra, P. and Eich, M. H. (1992). Join processing in relational databases. *ACM Comput. Surv.*, 24(1):63–113.

[Murugesan and Gangadharan, 2012] Murugesan, S. and Gangadharan, G. (2012). *Harnessing green IT: Principles and practices*. Wiley Publishing.

[Nasir, 2016] Nasir, M. A. U. (2016). Fault tolerance for stream processing engines. *arXiv preprint arXiv:1605.00928*.

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

[Nikolov et al., 2014] Nikolov, V., Kächele, S., Hauck, F. J., and Rautenbach, D. (2014). Cloudfarm: An elastic cloud platform with flexible and adaptive resource management. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC '14, pages 547–553, Washington, DC, USA. IEEE Computer Society.

[Noghabi et al., 2017] Noghabi, S. A., Paramasivam, K., Pan, Y., Ramesh, N., Bringhurst, J., Gupta, I., and Campbell, R. H. (2017). Samza: stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645.

[Pearce et al., 2012] Pearce, O., Gamblin, T., de Supinski, B. R., Schulz, M., and Amato, N. M. (2012). Quantifying the effectiveness of load balance algorithms. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 185–194, New York, NY, USA. ACM.

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

[Petit et al., 2010] Petit, L., Labbé, C., and Roncancio, C. L. (2010). An algebric window model for data stream management. In *Proceedings of the Ninth ACM International Workshop on Data Engineering for Wireless and Mobile Access*, MobiDE '10, pages 17–24, New York, NY, USA. ACM.

[Pruett, 2007] Pruett, M. (2007). *Yahoo! pipes*. O'Reilly.

[Qiao et al., 2003] Qiao, L., Agrawal, D., and Abbadi, A. E. (2003). Supporting sliding window queries for continuous data streams. In *15th International Conference on Scientific and Statistical Database Management, 2003.*, pages 85–94.

[Rivetti et al., 2016] Rivetti, N., Anceaume, E., Busnel, Y., Querzoni, L., and Sericola, B. (2016). Online scheduling for shuffle grouping in distributed stream processing systems research paper. In *ACM/IFIP/USENIX Middleware 2016*.

[Rivetti et al., 2015] Rivetti, N., Querzoni, L., Anceaume, E., Busnel, Y., and Sericola, B. (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, pages 80–91, New York, NY, USA. ACM.

[Rosenthal et al., 1989] Rosenthal, A., Chakravarthy, U. S., Blaustein, B., and Blakely, J. (1989). Situation monitoring for active databases. In *Proceedings of the 15th International Conference on Very Large Data Bases*, VLDB '89, pages 455–464, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

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

[Satzger et al., 2011] Satzger, B., Hummer, W., Leitner, P., and Dustdar, S. (2011). Esc: Towards an elastic stream computing platform for the cloud. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 348–355.

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

[Schreier et al., 1991] Schreier, U., Pirahesh, H., Agrawal, R., and Mohan, C. (1991). Alert: An architecture for transforming a passive dbms into an active dbms. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 469–478, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Sedaghat et al., 2013] Sedaghat, M., Hernandez-Rodriguez, F., and Elmroth, E. (2013). A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, CAC '13, pages 6:1–6:10, New York, NY, USA. ACM.

[Shukla and Simmhan, 2017] Shukla, A. and Simmhan, Y. (2017). Toward reliable and rapid elasticity for streaming dataflows on clouds. *arXiv preprint arXiv:1712.00605*.

[Stephens, 1997] Stephens, R. (1997). A survey of stream processing. *Acta Informatica*, 34(7):491–541.

[Stonebraker et al., 2005] Stonebraker, M., Çetintemel, U., and Zdonik, S. (2005). The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47.

[Tatbul et al., 2007] Tatbul, N., Çetintemel, U., and Zdonik, S. (2007). Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 159–170. VLDB Endowment.

[Tatbul et al., 2003] Tatbul, N., Çetintemel, U., Zdonik, S., Cherniack, M., and Stonebraker, M. (2003). Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 309–320. VLDB Endowment.

[Tatbul and Zdonik, 2006] Tatbul, N. and Zdonik, S. (2006). Window-aware load shedding for aggregation queries over data streams. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 799–810. VLDB Endowment.

[Tucker et al., 2003] Tucker, P., Maier, D., Sheard, T., and Fegaras, L. (2003). Exploiting punctuation semantics in continuous data streams. *Knowledge and Data Engineering, IEEE Transactions on*, 15(3):555–568.

[Welsh et al., 2001] Welsh, M., Culler, D., and Brewer, E. (2001). Seda: an architecture for well-conditioned, scalable internet services. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 230–243. ACM.

[Wikström, 1994] Wikström, C. (1994). Distributed programming in erlang. In *In PASCO'94-First International Symposium on Parallel Symbolic Computation*. Citeseer.

[Wu et al., 2007] Wu, K.-L., Hildrum, K. W., Fan, W., Yu, P. S., Aggarwal, C. C., George, D. A., Gedik, B., Bouillet, E., Gu, X., Luo, G., et al. (2007). Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on system s. In *Proceedings of the 33rd international conference on Very large data bases*, pages 1185–1196. VLDB Endowment.

[Wu and Tan, 2015] Wu, Y. and Tan, K.-L. (2015). Chronostream: Elastic stateful stream computation in the cloud. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 723–734. IEEE.

[Xu and Peng, 2016] Xu and Peng, G. (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 et al., 2014] Xu, J., Chen, Z., Tang, J., and Su, S. (2014). T-storm: Traffic-aware online scheduling in storm. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 535–544.

[Yang et al., 2012] Yang, F., Qian, Z., Chen, X., Beschastnikh, I., Zhuang, L., Zhou, L., and Shen, J. (2012). Sonora: A platform for continuous mobile-cloud computing. *Technical Report. Microsoft Research Asia, Tech. Rep.*

[Zaharia et al., 2012a] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012a). 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*, pages 2–2. USENIX Association.

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

[Zhu and Agrawal, 2012] Zhu, Q. and Agrawal, G. (2012). Resource provisioning with budget constraints for adaptive applications in cloud environments. *IEEE Transactions on Services Computing*, 5(4):497–511.

FOLIO ADMINISTRATIF

THESE DE L'UNIVERSITE DE LYON OPEREE AU SEIN DE L'INSA LYON

NOM : KOTTO KOMBI

DATE de SOUTENANCE : 29/06/2018

Prénoms : Roland Olivier

TITRE : Distributed query processing over fluctuating streams

NATURE : Doctorat

Numéro d'ordre : 2018LYSEI050

Ecole doctorale : INFOMATHS

Spécialité :Informatique

RESUME : Le traitement de flux de données est au cœur des problématiques actuelles liées au *Big Data.* Face à de grandes quantités de données (Volume) accessibles de manière éphémère (Vélocité), des solutions spécifiques tels que les systèmes de gestion de flux de données (SGFD) ont été développés. Ces SGFD prennent en entrée des flux et des requêtes continues pour générer de nouveaux résultats aussi longtemps que des données arrivent en entrée. Dans certains domaines, les flux considérés ont des débits qui varient en termes de nombre de données produites par unité de temps ou en termes de distribution de valeurs des données. Ces variations peuvent impacter fortement les besoins en ressources nécessaires au traitement des requêtes continues.

Dans le contexte de cette thèse, qui s'est réalisée dans le cadre du projet ANR Socioplug (ANR-13-INFR-0003), nous considérons une plateforme collaborative de traitement de flux de données. Chaque utilisateur peut soumettre des requêtes continues et contribue aux ressources de traitement de la plateforme. Cependant, chaque unité de traitement mise à disposition pour le traitement des requêtes dispose de ressources limitées en termes de processeur et de mémoire ce qui peut engendrer la congestion du système en fonction des variations des flux en entrée. Le problème est alors de savoir comment adapter dynamiquement les ressources utilisées par chaque requête continue par rapport aux besoins de traitement ? Cela soulève plusieurs défis : i) comment détecter un besoin de reconfiguration ? ii) quand reconfigurer le système pour éviter sa congestion ? iii) comment éviter des reconfigurations n'ajustant pas l'usage des ressources aux besoins des traitements ?

Durant ces travaux de thèse, nous nous sommes intéressés aux différentes étapes de traitement d'une requête continue sur une infrastructure distribuée. De cette analyse, nous avons pu identifier les limites de l'existant et les mécanismes permettant d'adapter dynamiquement les ressources utilisées pour l'exécution d'une requête continue. Nous avons focalisé nos efforts sur la gestion automatique de la parallélisation des opérateurs composant le plan d'exécution d'une requête. Nous proposons une approche originale basée sur l'observation des opérateurs et une estimation des besoins de traitement dans un futur proche. Ainsi, nous pouvons augmenter (*scale-out*) ou diminuer (*scale-in*) le niveau de parallélisme des opérateurs composant une requête continue de manière proactive afin d'ajuster les ressources utilisées aux besoins des traitements. Par rapport à une configuration statique définie par un expert, nous montrons qu'il est possible à la fois d'éviter la congestion du système dans certains cas ou de la retarder dans les cas les plus critiques. Nous montrons également qu'il est possible de réduire significativement la consommation de ressources tout en maintenant une performance et une qualité des résultats équivalentes.

Nous proposons également de combiner cette approche avec des mécanismes complémentaires tels que l'équilibrage de charge pour l'adaptation dynamique de requêtes continues. Ces différents travaux ont été implémentés et validés dans un SGFD largement utilisé avec différents jeux de tests reproductibles.

MOTS-CLÉS : flux de données, requête continue, traitement distribué, adaptation dynamique

ABSTRACT :
In a *Big Data* context, stream processing has become a very active research domain. In order to manage ephemeral data (Velocity) arriving at important rates (Volume), some specific solutions, denoted data stream management systems (DSMSs), have been developed. DSMSs take as inputs some queries, called *continuous queries*,defined on a set of data streams. A continuous query generates new results as long as new data arrive in input. In many application domains, data streams have input rates and distribution of values which change over time. These variations may impact significantly processing requirements for each continuous query.

This thesis takes place in the ANR project Socioplug (ANR-13-INFR-0003). In this context, we consider a collaborative platform for stream processing. Each user can submit multiple continuous queries and contributes to the execution support of the platform. However, as each processing unit supporting treatments has limited resources in terms of CPU and memory, a significant increase in input rate may cause the congestion of the system. The problem is then how to adjust dynamically resource usage to processing requirements for each continuous query ? It raises several challenges : i) how to detect a need of reconfiguration ? ii) when reconfiguring the system to avoid its congestion at runtime ? iii) how to avoid reconfigurations that do not improves the performance of the system ?

In this work, we are interested by the different processing steps involved in the treatment of a continuous query over a distributed infrastructure. From this global analysis, we extract mechanisms enabling dynamic adaptation of resource usage for each continuous query. We focus on automatic parallelization, or auto-parallelization, of operators composing the execution plan of a continuous query. We suggest an original approach based on the monitoring of operators and an estimation of processing requirements in near future. Thus, we can increase (*scale-out*), or decrease (*scale-in*) the parallelism degree of operators in a proactive way such as resource usage fits to processing requirements dynamically. Compared to a static configuration defined by an expert, we show that it is possible to avoid the congestion of the system in many cases or to delay it in most critical cases. Moreover, we show that resource usage can be reduced significantly while delivering equivalent throughput and result quality.

We suggest also to combine this approach with complementary mechanisms for dynamic adaptation of continuous queries at runtime. These differents approaches have been implemented within a widely used DSMS and have been tested over multiple and reproductible micro-benchmarks.

KEYWORDS : stream processing, continuous query, distributed computing, dynamic adaptation

Laboratoire (s) de recherche : LIRIS

Directeur de thèse : Philippe LAMARRE

Président de jury :

DEFUDE, Bruno	Professeur Télécom Sud Paris	

Composition du jury :

AMANN, Bernd	Professeur Sorbonne Université	Rapporteur
MORVAN, Franck	Professeur Université Paul Sabatier	Rapporteur
SKAF-MOLLI, Hala	Professeure Université de Nantes	Examinatrice
LAMARRE, Philippe	Professeur INSA de Lyon	Directeur de thèse
LUMINEAU, Nicolas	Maître de Conférences Université Claude Bernard Lyon 1	Co-directeur de thèse