FAURÉ (Christian), « Les techniques de gestion du temps dans les architectures de flux », *Études digitales*, n° 5, 2018 – 1, *Religiosité technologique*, p. 201-212

DOI : 10.15122/isbn.978-2-406-09290-2.p.0201

Fauré (Christian), « Les techniques de gestion du temps dans les architectures de flux »

Résumé – L'article consacré aux "techniques de gestion du temps dans les architectures de flux" confronte le temps de l'événement à celui de son traitement et interroge l'effet de ce subtil mais constant décalage.

Mots-clés – Techniques, gestion du temps, architectures de flux, événement, traitement du temps

Fauré (Christian), « Time-management techniques in flow architectures »

Abstract – This article on "time-management techniques in flow architectures" juxtaposes the time of the event with that of its processing and questions the effect of this subtle but constant gap.

Keywords – Techniques, time management, flow architectures, event, time processing

LES TECHNIQUES DE GESTION DU TEMPS DANS LES ARCHITECTURES DE FLUX

Issues notamment du secteur de la finance et de celui des plateformes de réseaux sociaux, de nouvelles architectures digitales capables de faire traitement sur des flux de données ininterrompus commencent à se répandre. C'est l'opportunité de faire le point sur les techniques de gestion qu'utilisent ces « moteurs de flux ».

DÉFINITION D'UNE ARCHITECTURE DE FLUX

Comme les termes de « *flow* » et de « *streaming* » restent assez flous, Tyler Akidau, ingénieur chez Google, a proposé une définition assez précise et opératoire d'un système de streaming[1] :

> *A streaming engine is a type of data processing engine that is designed with infinite data sets in mind.*
>
> « Un moteur de flux est un moteur de traitement qui a été conçu pour traiter des jeux de données infinis. »

Une fois de plus, on voit que c'est le mode de collecte qui détermine l'architecture de traitement ; quand la collecte des données ne s'arrête jamais, les jeux de données deviennent infinis. Par ailleurs, ces moteurs de traitement de flux sont souvent caractérisés comme étant des systèmes à faible latence et à résultats approximatifs ou probabilistes. La « faible latence » c'est ce que l'on souhaite : répondre au plus vite aux sollicitations. Le fait d'avoir des « résultats approximatifs ou probabilistes »,

1 Tyler Akidau, *The world beyond batch : Streaming 101* : https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101.

c'est ce qu'on accepte de concéder pour y parvenir, puisqu'on travaille sur des ensembles de données ouverts et infinis, et non fermés et finis : la base sur laquelle on fait des traitements est « mouvante ».

BATCH VERSUS FLOW

Un système bancaire qui collecte les données puis les traite en *batch* (c'est-à-dire par lots) a forcément une architecture informatique conçue pour ce type de traitement. Mais si vous prenez cette architecture et que vous l'utilisez pour opérer disons un réseau social, chacun comprendra qu'il y a un problème de conception, de *design* : quelle valeur aurait un réseau social qui collecterait les informations dans la journée pour ne les mettre à jour, après traitement, que le lendemain ? Quel intérêt pour un utilisateur du réseau social de recevoir des notifications sur des interactions qui se sont passées la veille ? Cet exemple n'est pas innocent, car les architectures de flux sont largement utilisées[2] dans les systèmes de réseaux sociaux et de messageries instantanées : Twitter, Facebook, Meetic, Linkedin, etc.

La différence porte avant tout sur la manière dont se fait la collecte des données. Inévitablement, des données collectées par lots (*batch*) seront traitées par lots et celles collectées en continu (*flow*) seront traitées en continu[3] : « Dis-moi comment tu collectes tes données et je te dirai quelle est ton architecture digitale ».

La tendance, en matière d'acquisition des données, est qu'elle devienne continue et permanente. Aussi va-t-on immanquablement assister à une migration progressive des architectures digitales vers des architectures de flux. L'architecture du système d'information d'une entreprise digitale va de moins en moins avoir besoin de collecter ses données en mode *batch*. Au contraire, de nombreuses autres entreprises ne voient pas l'intérêt

2 Le secteur financier avait déjà largement utilisé ce genre de technologies.

3 Il faudrait nuancer ce propos car on peut collecter en continu et traiter en batch derrière. On peut aussi avoir besoin d'accumuler un peu de données (une fenêtre temporelle typiquement, avec le *pattern* « *micro-batch* ») avant de faire le traitement, en modifiant le curseur qui fait le compromis entre la latence, l'exactitude et le coût. Nous reviendrons en détails sur ces points dans le chapitre suivant.

de passer à des architectures de flux, selon un constat fondé sur trois raisons principales :

1. Elles estiment ne pas en avoir besoin : elles sont capables d'absorber et de traiter leur volumétrie de *data* avec des architectures classiques : « nul besoin d'une Ferrari si une Dacia fait le job ! ».
2. Construire et maintenir des architectures de flux est complexe ; ces dernières demandent des compétences pointues et, à l'heure actuelle, les *frameworks* disponibles ne sont pas tous « secs ». Il y a donc une part de risque non négligeable.
3. Les entreprises pensent que la collecte des données par lot fait partie de l'*ordre des choses* et qu'on ne peut pas y déroger. C'est probablement là leur principale réticence. En réalité, si les données sont encore collectées par *batchs*, c'est qu'il y a soit des activités manuelles dans le processus de collecte, soit un défaut de numérisation du processus, soit un reliquat historique d'une procédure non digitale (par exemple, réglementaire). Nous voyons par-là que la digitalisation des entreprises commence toujours par la définition de ses modes de collecte des données. Ceux qui n'ont pas fait évoluer leurs modalités de collecte ne voient pas l'intérêt des architectures de flux.

Il y a également d'autres arguments que l'on peut entendre et qui plaident en faveur du maintien d'une collecte et d'un traitement par *batch* ; par exemple, lorsqu'il s'agit de calculer un maximum ou une moyenne. En effet comment faire ce type de traitement si l'on n'a pas des jeux de données *lotis*, avec un début et une fin ? En réalité ces arguments sont fallacieux, une simple consultation des premiers papiers scientifiques sur les architectures de flux montre que les techniques de *windowing* (technique de *fenêtrage glissant* dans des collections de flux de données, que nous aborderons ci-dessous) ont résolu le problème posé par ce genre d'opérations bloquantes[4]. Chaque entreprise doit donc aujourd'hui se poser, à nouveaux frais, la question de la conception de ses techniques de collecte et de traitement de données.

4 Il faut quand même que les opérations de calcul soient associatives et commutatives dès lors que les données arrivent par petits bouts et en ordre dispersé. C'est le cas de la moyenne, au-delà, cela devient compliqué en mode flux. Le *batch* n'a pas cette limitation et offre aussi des possibilités de rejeu et d'idempotence à faible coût qui sont possibles en flux, mais plus complexes.

LES ARCHITECTURES *LAMBDA* POUR COMBINER *BATCH* ET *FLOW* ?

L'architecture *Lambda*, conceptualisée par Nathan Marz dans son article *How to beat the CAP Theorem*[5], vise à construire des systèmes de flux au-dessus de systèmes de traitement par *batchs*, en essayant de combiner le monde du *batch* avec celui du flux. Même si l'idée est séduisante, sa mise en œuvre s'est heurtée à la complexité de maintenir deux logiques de traitement disposant d'un code spécifique et de systèmes de stockage différents. Raison pour laquelle les architectures *Lambda* sont aujourd'hui plutôt considérées comme une phase de transition ou comme une architecture théorique que comme une solution pérenne.

Aujourd'hui, la tendance est d'avoir des systèmes de flux qui peuvent traiter aussi bien des flux de données que des *batchs* de données. Pour ces systèmes (on pense à *Apache Flink*[6] *ou Apache Spark*[7]), nul besoin de deux systèmes complémentaires (*batch* et flux) puisque le système de flux traite tout aussi bien les deux cas[8].

Mais on voit pointer un changement de paradigme puisque la spécificité d'un système (*batch* ou flux) devient une partie d'un autre système. Dit autrement : le *batch* devient un simple cas particulier du flux. Dans les faits, les *frameworks* de *streaming* évoluent à une vitesse telle que la médiation d'une architecture *Lambda* est en passe de devenir caduque. Désormais, une architecture de flux traite aussi bien des *batchs* que des flux[9], et ceci à condition que deux *challenges* soient relevés : celui de l'exactitude et celui de la manipulation du temps.

5 http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html
6 https://flink.apache.org/
7 http://spark.apache.org/
8 On a aussi parlé d'architecture « *Kappa* » : http://milinda.pathirage.org/kappa-architecture.com/
9 Une technique permettant de transformer un *batch* en flux consiste à éclater un fichier en messages unitaires qu'on injecte un à un vers une architecture de flux (pour les *batchs* faits d'enregistrements, soit 99 % de ceux pris en charge par les SI traditionnels). Mais cela ne résout pas le besoin de latence ni la nécessité d'avoir un horodatage exact à chaque message.

L'EXACTITUDE DANS LES ARCHITECTURES DE FLUX

L'exactitude est la moindre des choses si l'on veut qu'un moteur de flux puisse faire aussi bien qu'un moteur de *batch*. Mais de quelle exactitude parle-t-on ici ? Il s'agit d'idempotence. On dit qu'une opération est idempotente si elle a le même effet et produit le même résultat quel que soit le nombre de fois où qu'elle est exécutée. Par exemple, dans le style d'architecture REST, la méthode DEL a toujours le même résultat, que vous la jouiez une fois ou *n* fois[10]. Appliqué à un moteur de traitement de données, cela veut dire que les mêmes données en *input* produiront toujours le même résultat en *output*.

Garantir cette exactitude, cette idempotence, dans le traitement des flux de données suppose que le système distribué puisse être tolérant aux pannes, car si des calculs se font sur différents nœuds du réseau, il faut que des états intermédiaires puissent être gardés en mémoire[11]. Qui plus est, garder ces états en *log* est une solution au problème d'exactitude ; un des intérêts des *logs* est d'être le levier qui va permettre de collecter des données en continu et justifier l'utilisation d'un moteur de traitement de flux (d'où l'importance du *framework* Kafka dans les architectures de flux).

Au final, le traitement de flux consiste à lire une *log* (une collection de données) pour la transformer en une autre *log*, qui peut elle-même être lue pour produire une autre *log* et ainsi de suite. Il y a des traitements qui s'enchaînent, puisqu'un traitement de flux est l'application d'une fonction à une collection. On peut dire par analogie que la fonction globale du système est la composition des fonctions de chaque étape.

Après l'exactitude, le second *challenge* pour les systèmes de flux réside dans leurs techniques de manipulation du temps.

10 *Cf.* le RFC de HTTP 1.1 : « *Methods can also have the property of "idempotence" in that (aside from error or expiration issues) the side-effects of N > 0 identical requests is the same as for a single request. The methods GET, HEAD, PUT and DELETE share this property.* » https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

11 *Cf.* l'article de Kreps, *Why local state is a fundamental primitive in stream processing*, https://www.oreilly.com/ideas/why-local-state-is-a-fundamental-primitive-in-stream-processing

TEMPS DE L'ÉVÉNEMENT ET TEMPS DU TRAITEMENT

Lorsque nous regardons le ciel étoilé, la nuit, nous regardons en fait un état qui n'existe plus au moment où nous le regardons ; c'est le passé que nous voyons, du fait du temps qu'il aura fallu aux photons pour parcourir, à la vitesse de la lumière, la distance qui nous sépare des étoiles. De la même manière, votre regard qui parcourt ces lignes en voit les lettres environ une nanoseconde avant d'être déchiffrées. Que ce soit à l'échelle macroscopique ou microscopique, il y a toujours un décalage entre le temps de ce qui arrive et le temps où cela nous arrive.

Aussi, la première distinction que nous devons faire est celle qui sépare le temps de l'événement (*Event Time*), quand il arrive effectivement, et le temps du traitement (*Processing Time*), quand l'événement est constaté et traité dans le système. Dans un monde idéal, les données seraient traitées en temps réel et il n'y aurait aucune différence entre ces deux temporalités. Mais dans les systèmes distribués cela n'arrive jamais, du fait de la nature même du réseau (pertes probables de nœud du réseau, problèmes de contention) : dès lors, comment pallier les incohérences temporelles ? Comment retrouver le temps ? C'est dans les techniques de manipulation et de gestion du temps que les choses vont se jouer.

Dans les architectures de flux, c'est la nature infinie des collections de données qui pose problème : comment appliquer des calculs à une collection de données en perpétuelle extension ? Eh bien qu'à cela ne tienne : créons des collections finies virtuelles, des « vues fenêtrées » qui vont introduire des collections finies dans le flux infini. La réponse globale apportée par les moteurs de flux (*streaming engines*) dans la gestion temporelle d'un flux infini de données passe toujours par des techniques de fenêtrage (*windowing*).

Si le fenêtrage est la solution générale au problème de collections de données infinies, faire face à des situations diverses et hétérogènes exige de pouvoir décliner le fenêtrage dans des techniques particulières, adaptées à des contextes particuliers. En effet, au problème des collections de données infinies, s'ajoutent deux facteurs de complexité : les données n'arrivent pas nécessairement au moteur de traitement selon l'ordre du temps des événements (*event time*). Le plus souvent, elles arrivent même

de manière désordonnée ; l'écart entre le temps de l'événement (*event time*) et le temps du traitement (*processing time)* n'est pas constant. Sinon ce serait trop simple : il suffirait de connaître cet écart constant pour faire coïncider parfaitement les deux temporalités.

LES TECHNIQUES DE FENÊTRAGE DU FLUX DES ÉVÉNEMENTS

Les techniques de fenêtrage[12] peuvent s'appliquer soit au temps des événements, soit au temps du traitement. Le cas d'usage permettant de travailler au niveau du temps des traitements est le plus simple : le fenêtrage se fait selon l'ordre d'arrivée des données sans se soucier de savoir si elles arrivent dans le bon ordre. Cette approche permet également de savoir avec certitude que notre fenêtre s'est terminée proprement, puisqu'aucune donnée n'arrivera après coup ou en retard.

En revanche, si l'on est attentif à l'exactitude de nos calculs, on va chercher à prendre en compte le temps de l'événement comme référence, comme lorsqu'on souhaite connaître le nombre de clics sur un site web durant un créneau horaire donné : c'est l'horodatage du clic dans le temps de l'événement qui doit faire foi et pas le temps de son traitement.

Dans la majorité des cas d'usages, c'est le temps de l'événement qui est déterminant, et on ne peut plus alors utiliser des techniques de fenêtrage sur le temps des traitements : c'est ce fenêtrage sur le temps des événements, qui constitue le principal *challenge* pour les architectures de flux.

Les techniques de fenêtrage dans des collections infinies se font en jouant sur plusieurs paramètres : le temps de début et de fin de la fenêtre, la durée de la fenêtre et enfin sa fréquence. Si la durée de la fenêtre est égale à sa fréquence, nous aurons un fenêtrage fixe qui découpe le flux des données de manière régulière. Si la fréquence est inférieure à la durée, il y a un recouvrement des fenêtres ; on parle de fenêtres décalées

12 Ne perdons pas de vue que dans de nombreux cas, le traitement au fil de l'eau et sans fenêtrage est possible. Nous mettons en exergue les techniques de fenêtrage car c'est là où le plus de difficultés doivent être prises en compte.

ou glissantes (*Sliding Windows*). Enfin si la fréquence est supérieure à la durée, on aura des fenêtres espacées ; technique plutôt utilisée pour sonder ou faire des calculs sur des échantillons.

Dans ces derniers exemples, une fois fixées, les variables ne changent pas, c'est la raison pour laquelle on peut dire que ce sont des techniques de fenêtrages fixes. On peut également avoir une vision plus dynamique du fenêtrage en faisant varier les paramètres de début, de fin, de durée et de fréquence. C'est typiquement ce que l'on fait avec des fenêtres de sessions qui sont utiles lorsqu'on veut analyser les comportements des utilisateurs : le *login* et le *logout* déterminent le début et la fin de la fenêtre (avec des fonctions de *timeout*).

QUAND ET COMMENT DÉCLENCHER DES TRAITEMENTS INTERMÉDIAIRES ?

Si l'on a mis en place des techniques de fenêtrage côté *event time*, il va également falloir répondre à la question du traitement de ces fenêtres. L'exercice est difficile parce qu'on ne peut jamais vraiment savoir si les traitements se font sur des collections exactes (complétude d'une collection) : a-t-on vraiment récupéré toutes les données événementielles d'une fenêtre de temps donnée ? À quel moment décide-t-on de faire les transformations et les traitements ?

Il faut bien comprendre qu'ici la question ne porte pas sur la nature des traitements qui seront effectués mais sur le déclenchement du traitement. Pour y répondre, deux techniques différentes et complémentaires sont utilisées : les techniques de marquage (*Watermark*) et les techniques de déclenchement (*Triggers*).

Commençons par les techniques de marquage (*Watermark*) qui sont aussi appelées « tatouages numériques » ou « filigranes numériques ». Elles consistent à rajouter des informations portant sur la structure ou sur la logique des données collectées, permettant ainsi de mesurer la progression de la complétude des données dans une fenêtre de temps des événements. Cette mesure de la complétion peut être déterministe et précise quand on a une parfaite connaissance de la logique des événements

(par exemple quand on sait qu'il y aura 10 000 événements sur une plage d'une minute), mais elle peut aussi être heuristique s'il faut s'en tenir à des estimations de progression des événements dans la fenêtre.

Quand les techniques de marquage sont heuristiques, cela veut dire que le tatouage numérique interne ne suffit plus et qu'il faut faire appel à des signaux externes pour provoquer un déclenchement (*Triggers*). Ces signaux peuvent être la progression de complétude du temps de traitement, le comptage du nombre d'événements, la prise en compte d'éléments de ponctuation qui sont inclus dans les données tels que *EndOfFile*, etc. Et pour complexifier la situation, on peut également combiner différents *triggers* entre eux pour faire des *triggers* combinatoires.

LE TRIPTYQUE : COMPLÉTUDE, LATENCE, COÛT

Si un traitement est joué avant la fin de la plage temporelle de la collection de données, alors il devra être rejoué jusqu'à ce que toutes les données de la plage horaire arrivent dans le système pour *enfin* produire le résultat final. Tant que la fenêtre n'est pas finie, le traitement n'est capable de produire qu'un résultat intermédiaire et probable, car la fonction d'un ensemble de données doit toujours s'appuyer sur une complétude des données pour être correcte.

Comme des messages peuvent arriver en retard, il peut être nécessaire de laisser une marge d'erreur laissant un répit à ces derniers. Si la fonction attend trop longtemps que la plage se termine (à cause d'une donnée retardataire), cela ajoute de la latence au système qui attend le résultat pour enchaîner d'autres traitements.

Frances Perry[13], de l'équipe Google *Dataflow*, parle du triptyque complétude, latence, coût. Si l'on veut connaître exactement un résultat (par exemple les ventes de la journée) il faut attendre la complétude de la fenêtre de données. Bien souvent on souhaite avoir des résultats intermédiaires et seulement probables avant la fin de la fenêtre de calcul pour ne pas avoir à attendre et pour ne pas introduire de la latence dans le système. Mais évidemment, tous les traitements intermédiaires ont un

13 Frances Perry, dans une conférence de 2015 : https://www.youtube.com/watch?v=3UfZN59Nsk8

contrecoup : ils complexifient le système et consomment des ressources, ce qui augmente le coût global de fonctionnement du système. Il faut donc trouver pour chaque cas d'usage un *optimum* entre complétude, latence et coût.

Dernier moment, avec les techniques dites d'accumulation. En effet, dans le cas de traitements intermédiaires, il faut des techniques de gestion de cette accumulation, jusqu'au traitement final. On en distingue aujourd'hui trois principales : le *discarding* qui consiste à supprimer l'état du traitement précédent – on ne fait donc pas de lien entre les différents états. Ensuite, la technique d'accumulation (*Accumulating*) dans laquelle les états antérieurs sont repris et enrichis par les nouveaux états. Enfin, la technique *Accumulating and retracting* qui, comme son nom l'indique, permet à la fois d'accumuler des états et en même temps de supprimer des états plus anciens (une sorte de fenêtrage glissant dans le traitement).

L'équipe Google *Dataflow* a fait un formidable travail de pédagogie[14] lorsqu'elle a fait don du SDK[15] à la fondation Apache – ce qui a donné naissance à *Apache Beam*[16], une implémentation du modèle de *Dataflow* de Google qui permet de définir et d'effectuer des traitements sur des données aussi bien en *batchs* qu'en flux. L'équipe de Google a synthétisé son travail autour de quatre questions auxquelles un moteur de traitement des flux doit répondre :

Que peut-on calculer ? Cette question fait référence au type de traitement que l'on est capable d'effectuer (question que nous n'avons pas développée, mais qui renvoie aux types de fonctions de transformation que l'on peut appliquer aux collections de données fenêtrées).

À quel moment se déploie le calcul dans le temps des événements ? Cette question renvoie aux techniques de fenêtrage du temps des événements.

Quand fait-on les traitements ? Cette question renvoie aux techniques de *Watermarks* et de *Trigger*.

Comment fait-on pour gérer l'accumulation des traitements intermédiaires ? Cette question renvoie aux techniques d'accumulation.

14 Tyler Akidau, *The world Beyond Batch*, https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102

15 Kit de développement logiciel

16 https://beam.apache.org/

LE TOURNANT PROBABILISTE

Il est fort probable que le « tournant digital » des systèmes d'information de gestion corresponde à une introduction radicale des statistiques et des probabilités, c'est-à-dire à des traitements non déterministes, à l'image de ce que fit Gibbs dans la physique du début du XX^e^ siècle quand il reconnut l'existence d'un élément fondamental dans la structure de l'univers : le hasard. « Les statistiques sont la science des distributions[17] » rappelait Norbert Wiener, celui qui inventa la cybernétique, cette science du *feedback*. On comprend mieux pourquoi les systèmes distribués sont inévitablement des terrains privilégiés pour des approches statistiques et probabilistes. Cette théorie générale des messages qu'est la cybernétique, précisait Wiener, est une théorie probabiliste. Si le *feedback*, ou rétroaction, est le moyen de contrôler l'entropie, dans les systèmes distribués de flux ce contrôle s'effectue au moyen d'échanges de messages entre les différents composants du système. Nous retrouvons des jeux d'écriture et la relation épistolaire des composants logiciels introduits dont nous avons eu le loisir de traiter par ailleurs.

L'inscription « Nul n'entre ici s'il n'est géomètre » était gravée au fronton de l'Académie de Platon à Athènes, indiquant que l'enseignement de la philosophie n'était accessible que pour celui qui était déjà versé dans les sciences géométriques. Par mimétisme, nous serions tentés de dire, à propos des architectures de flux, que « Nul n'entre ici s'il n'est statisticien ». Face à la complexité croissante des échanges et des messages entre les différents composants, le recours au formalisme mathématique s'impose comme une nécessité, que l'on constate historiquement à différents niveaux. Dans les années soixante-dix et jusqu'aux années quatre-vingt-dix, il s'agissait de faire des calculs déterministes avec des modèles impératifs et de la programmation par procédures. Dans les années quatre-vingt-dix, il s'agissait de faire de la modélisation avec des modèles orientés objets[18] et de la programmation par méthodes. Depuis

17 Norbert Wiener, *Cybernétique et société*, Paris, Éditions du Seuil, 2014, p. 42.

18 Edsger Dijkstra, prix Turing 1972, dit à ce propos que « la programmation par objets est une idée exceptionnellement mauvaise qui ne pouvait naître qu'en Californie. »

les années deux mille dix, nous sommes dans des paradigmes fonctionnels où la programmation se fait à base de fonctions (*Haskell, Scala, F#, Clojure, Swift*), mais nous commençons déjà à entrevoir l'émergence du paradigme statistique avec de la programmation probabiliste couplée à des systèmes de stockage de type *Time Series DB*. Le paradigme mathématique et notamment probabiliste prend donc de l'ampleur dans les systèmes digitaux : il est le vecteur incontournable pour en maîtriser la complexité croissante.

Christian FAURÉ