

MASTER MIAGE 2ÈME ANNÉE
UNIVERSITÉ PARIS NANTERRE

MÉMOIRE DE FIN D'ÉTUDES

**Méthodes d'analyse des processus
métier pour le choix d'une
architecture Big Data adaptée**



Auteur :
LUDWIG SIMON

Tuteur :
MCF. EMMANUEL HYON

Février 2019 — Juin 2019

Remerciements

Je tiens tout d'abord à remercier mon tuteur, monsieur Emmanuel Hyon qui m'a suivi toute l'année pendant la rédaction de mémoire. Les conseils et l'aide qu'il a pu m'apporter sur la rédaction de ce dernier m'ont vraiment été très utiles.

J'aimerais aussi remercier mes collègues d'EDF avec qui j'ai pu travailler dans de bonnes conditions tout au long de l'année. Je les remercie aussi pour le temps qu'ils m'ont accordé afin de répondre à mes interrogations sur le domaine du Big Data pour m'aider dans la rédaction de ce mémoire.

Résumé

Résumé

Motivations

Avant d'avoir commencé mon stage de Master 1 chez EDF dans le service Big Data, pour moi le Big Data était une notion très floue et je ne savais pas vraiment tout ce qui composait ce domaine. Durant ce stage j'ai pu découvrir ce domaine et me rendre compte du nombre conséquent d'outils différents qui sont nécessaires pour la conception d'une architecture Big Data complète. C'est à partir de ce moment là que je me suis demandé, comment est-ce que le choix de l'architecture, et des outils Big Data s'effectue ? Il y a beaucoup plus de possibilités que les domaines que j'ai pu voir jusqu'à maintenant. C'est pour cela que j'ai décidé cette année, d'essayer de comprendre comment fonctionne une architecture Big Data afin de trouver une solution pour choisir plus facilement une architecture Big Data adaptée à nos besoins.

Objectifs

Dans ce mémoire, nous allons dans un premier temps rappeler brièvement ce qu'est le Big Data et quand il est vraiment utile de l'utiliser. Dans un second temps, nous allons présenter de manière générale comment est constituée une architecture Big Data. Puis nous verrons plus en détails les architectures qui ont été créées pour répondre aux besoins du Big Data. Ensuite, nous détaillerons pour chaque partie de ces différentes architectures, les solutions logicielles existantes permettant d'accomplir la tâche demandée. Et pour finir, nous allons à partir des études des architectures et des solutions logicielles, essayer de définir un moyen permettant de sélectionner correctement l'architecture et les outils nécessaires pour la création d'une solution Big Data correspondant à nos besoins. Afin de tester que notre méthode de choix est cohérente, on l'appliquera sur une application réelle et on la comparera aux autres possibilités d'architecture possible pour s'assurer que c'était le meilleur choix.

Sommaire

1	Les architectures Big Data	9
1.1	Architecture Lambda	11
1.2	Architecture Kappa	14
2	Analyse des solutions logicielles existantes	17
2.1	Message Broker	17
2.2	Ingestion/Extraction de données	19
2.3	Traitement des données	21
2.4	Stockage des données	24
2.5	Orchestration	26
2.6	Requetâge	26
2.7	Visualisation et Analyse des données	26
3	Critères d'analyse	27
3.1	Sélection d'architecture en entreprise	27
3.2	Critères pour le choix de l'architecture	27
3.3	Critères pour le choix des solutions logicielles	28
3.4	Aller plus loin	28
4	Implémentation : Exemple avec un processus métier	29
4.1	Définition du processus métier	29
4.2	Application des méthodes sur le processus métier	29
4.3	Évaluation du résultat	29
Annexe A	Les différents concepts du Big Data	33
A.1	Architecture Réactive	33
A.2	Architecture Répartie	35
A.3	Machine Learning	35
	Bibliographie	37

Introduction

Le Big Data

La démocratisation des ordinateurs et l'avènement du WEB 2.0 a entraîné une digitalisation de nos connaissances. Aujourd'hui la production de données est en constante évolution. Notre société produit quotidiennement 2.5 exaoctets de données (l'équivalent de 90 années de vidéos, ou 2.5 million de téraoctets) et en 2020 il est estimé que 35 zettaoctets seront produits (1 zettaoctet = 1000 exaoctet). Les réseaux sociaux et les objets connectés participent grandement à cette augmentation considérable du volume de données.

On caractérise le domaine du Big Data à l'aide de cinq facteurs appelé les 5V du Big Data [1] (Figure 1).

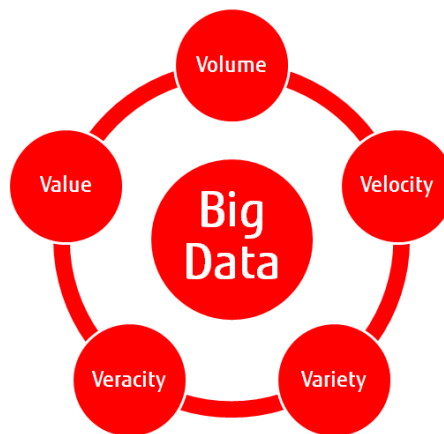


Figure 1 – Schéma des 5V du Big Data

- **Le volume** décrit la quantité de données générée. Il s'agit donc de la possibilité de gérer une masse de données créées quotidiennement. Par exemple, Facebook stocke aujourd'hui plus de 250 milliards d'images.
- **La vélocité** représente la vitesse à laquelle les données arrivent. Un des meilleurs exemples pour représenter la vélocité est l'ajout de vidéos sur la plateforme Youtube. En effet, chaque minute, ce sont pas moins de 400 heures de vidéos qui sont uploadées sur la plateforme.
- **La variété** correspond aux diverses natures que peuvent avoir les données. Par exemple, Twitter stocke du texte, des images, des fichiers vidéo, des métadonnées etc.
- **La véracité** met en avant la dimension qualitative nécessaire au bon fonctionnement des outils Big data. Lorsque les données ne sont pas qualitatives, il n'est pas possible de les traiter et de s'en servir correctement.

- **La valeur** est la plus importante des 5V. Les données auquel on s'intéresse doivent avoir une valeur réel pour que les analyse que l'on veut effectuer ne soit pas fossés. Il est donc important de trier les données avant leur exploitation.

Quand peut-on utiliser le Big Data ?

Le Big Data étant aujourd'hui "à la mode", c'est à dire que tout le monde en entend parler et donc veut aussi l'utiliser. Le problème étant, que le Big Data n'est pas toujours la solution idéale pour les applications que l'on souhaite réaliser. En effet, le Big Data a été créer dans les cas où une seule machine n'est plus suffisante pour les opérations que vous avez besoin d'effectuer. Par exemple, si vos traitements de données s'effectuent sans aucun problème sur votre machine et que votre base de données n'a aucun soucis de performance en étant installée sur une seule machine, il n'y a pas de réel intérêt pour vous de passer à une architecture Big Data. Dans le cas où une seule machine n'est plus suffisante pour vos besoins, avant de passer à une architecture Big Data vous pouvez essayer d'améliorer votre machine en changeant ses composants. Vous pouvez aussi essayer de rendre certaines parties de votre code asynchrones, afin d'améliorer les performances de votre code et de répartir son exécution au mieux sur votre machine. C'est ce qu'on appelle la mise à l'échelle verticale, c'est la première étape avant de passer à la mise à l'échelle horizontale (A.2) qui est l'un des atouts principal du Big Data. Une fois toutes ces étapes appliquées, si votre architecture n'arrive toujours pas à suivre le rythme voulu, cela veut dire que votre application est propice à l'utilisation d'une architecture Big Data. Comme vous venez de le voir, ce n'est malheureusement pas possible de définir une limite chiffrée pour définir quand l'utilisation du Big Data est requise. Cette limite peut varier en fonction du format des données reçues, de la complexité des traitements effectués sur les données, etc. Bien évidemment dans certains cas l'utilisation du Big Data est évidente (Exemple : des plateformes comme YouTube et Facebook), mais dans le cas ou vous n'avez encore jamais fait de Big Data et que vous n'avez pas un nombre de données immense, il est préférable de ne pas se diriger tout de suite vers le Big Data.

Les architectures Big Data

Une architecture Big Data est un regroupement d'outils permettant de gérer des données de leurs ingestion à leurs mise en valeur via des analyses. Il faut savoir qu'il existe plusieurs architectures dans le domaine du Big Data, et qu'elles répondent à des besoins différents. Nous allons nous intéresser aux deux architectures les plus importantes, étant donné que les autres sont des dérivées des deux architectures principales. Avant de voir en détail ces deux architectures, nous allons voir de manière plus générale les différents composants qui peuvent se retrouver dans des architectures Big Data [2][3]. Les différents composants pouvant se retrouver dans une architecture Big Data sont illustrés sur la figure 1.1.

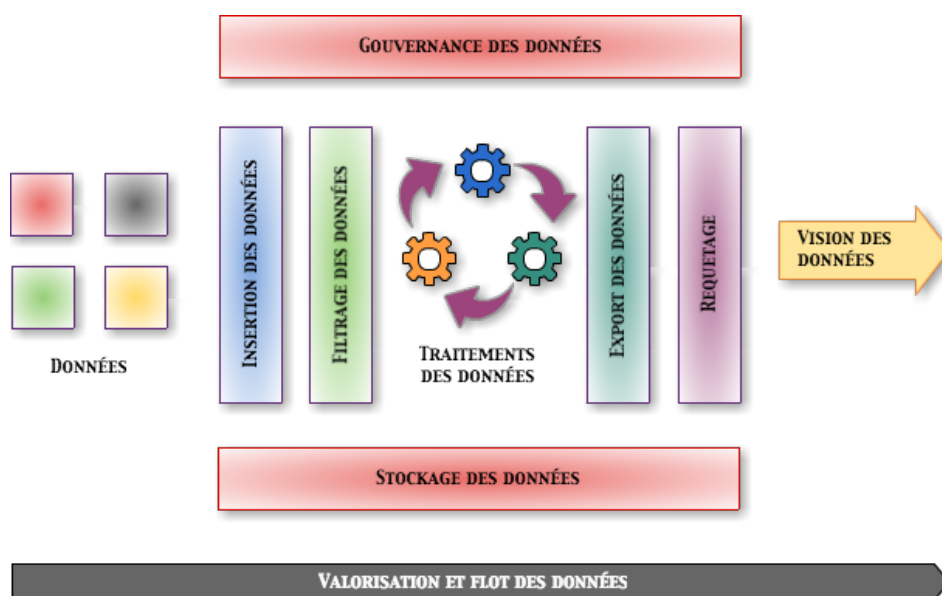


Figure 1.1 – Composants d'une architecture Big Data

Nous allons maintenant détailler le rôle de chaque composant présenté dans le diagramme ci dessus.

- **Source de données :** N'importe quelle solution Big Data a besoin d'une source de donnée en entrée. Voici quelques exemple de données qui peuvent être utilisée dans une architecture Big Data
 - Des données issues de bases de données relationnelles.
 - Des fichiers statiques produits par des applications, des fichiers de logs par exemple.
 - Des sources de données en temps réel, par exemple des données de capteurs récupérés via des appareils IoTs.

- **Ingestion des données** : La solution doit être capable d’aller chercher directement les données dans les différentes sources. Il est possible que les données soient directement envoyées dans la chaîne de traitement des données, mais c’est rarement le cas. Afin d’aller récupérer les données à la source, il y a plusieurs solutions. On peut écrire un programme permettant d’extraire les données. Cette solution est la plus efficace si le programme répond bien aux contraintes du Big Data, c’est à dire, il faut qu’il soit le plus réactif possible et le plus facilement adaptable aux flux de données entrant (Architecture réactive [A.1](#)). La seconde solution la plus simple, est l’utilisation d’un ETL (Extract Transform Load). C’est un outil graphique permettant de configurer l’extraction des données et leurs insertions dans la chaîne de traitements ou dans la solution de stockage. Néanmoins, cette solution est plus gourmande que l’écriture d’un programme car elle doit être capable de gérer énormément de sources et d’opérations différentes.
- **Traitement par lot** : Les jeux de données étant trop lourds lors d’un traitement par lots, il est nécessaire de pouvoir exécuter une tâche de traitement de longue durée afin de filtrer, agréger et préparer les données en vue de leur analyse. En général, ce genre de traitement implique une lecture de fichier source et une écriture dans des nouveaux fichiers. Le traitement des données peut s’effectuer par des programmes java, scala ou Python ou encore via des outils spécialisés comme MapReduce ou Spark.
- **Ingestion des données en temps réel** : Si la solution doit interagir avec des sources de données en temps réel, l’architecture doit impérativement implémenter un moyen de récupérer ces données et de les stocker temporairement dans une file d’attente afin de pouvoir les temporiser. Cela permet d’éviter la perte de données entrante, et permet d’envoyer les données à la solution de traitement quand elle est disponible et de ne pas la surcharger. Généralement on utilise des messages brokers pour ce genre de tâches.
- **Traitement de flux** : Une fois les données récupérées, elles doivent être filtrées, agrégées puis préparées pour l’analyse. Le traitement est similaire au traitement par lot, seuls les outils sont différents, car ils doivent être capables de gérer les traitements en temps réel.
- **Stockage des données** : Une solution de stockage de données est indispensable dans le cas de traitement des données par lot (??), et peut s’avérer utile lors de traitements des données en temps réel (??) si l’on souhaite conserver les données reçues en plus de les traiter. Le deuxième cas où un stockage de données peut être utile pour le traitement en temps réel, est si l’on a besoin d’agréger des données statiques avec les données en temps réel. Ces solutions de stockage doivent être capables de gérer divers formats de données, et surtout ils doivent être distribués ([A.2](#)).
- **Gouvernance des données** : La gouvernance des données correspond à l’ensemble des procédures mises en place afin d’encadrer la collecte des données ainsi que leur utilisation. La gouvernance des données comprend quatre dimensions. La disponibilité des données, l’utilisabilité des données, l’intégrité des données et la sécurité des données.
- **Requêtage** : Afin de pouvoir fournir les données stockées aux outils d’analyse

et de visualisation, l'utilisation d'un outil de requête peut être requis. Dans certains cas votre base de données et votre outil de visualisation communiqueront directement entre eux, mais dans d'autres cas vous aurez besoin d'utiliser un outil de requête afin de fournir les données au bon format et de pouvoir faire une sélection des données à envoyer à l'outil.

- **Analyse et visualisation des données** : La dernière étape dans une architecture Big Data est la visualisation/analyse des données. La plupart des solutions Big Data ont pour but de faire de la valorisation de données et de fournir au minimum une visualisation des données et au mieux d'effectuer des analyses dessus. Cela peut se faire par l'écriture de rapport ou bien par application d'algorithme afin de détecter et de montrer différentes corrélations entre des données par exemple. Le but principal est d'avoir une visualisation intelligente et facilement compréhensible de données étant à la base illisible par l'homme.
- **Orchestration** : La majorité des solutions Big Data consistent à effectuer des traitements de données répétés, ayant pour but de transformer les données sources puis de les stocker ou bien les envoyer directement à un outil d'analyse ou de visualisation des données. Il est donc important d'avoir un outil permettant de paramétrer les différentes actions que l'on souhaite effectuer sur nos données.

Comme on peut le voir une architecture Big Data possède beaucoup de catégories, et l'on constate qu'une catégorie existe sous 2 formes, le traitement des données. Les deux architectures différentes sont justement tournées sur cette catégorie, et proposent chacune une vision différente du traitement des données. Ces deux architectures sont l'architecture Lambda et l'architecture Kappa [4][3][5].

1.1 Architecture Lambda

L'architecture Lambda est une technique de traitement de données capable de traiter efficacement une grande quantité de donnée. L'efficacité de cette architecture provient d'un débit accru, une latence réduite et d'erreurs négligeables. Cela mène jusqu'à des applications pratiquement en temps réel. Dans le domaine du machine learning (A.3), cela permettrait aux développeurs de définir des règles delta¹ sous la forme code logique ou de traitement de langage naturel avec des modèles de traitement de données basés sur des événements afin d'obtenir de la robustesse, de l'automatisation et d'améliorer la qualité des données. Pour faire simple, toute modification de l'état des données est un événement pour le système, et il est possible de répondre à cet événement via l'exécution d'une commande ou d'une procédure delta.

La récupération d'événements est un concept qui consiste à utiliser les événements afin d'effectuer des prévisions ou bien stocker les changements effectués sur le système en temps réel. par exemple, une personne qui interagit sur un site de réseau social va provoquer des événements lors du chargement d'une page, de l'ajout en faveur d'un post, ou bien lors d'une demande d'ajout en ami. Ces événements peuvent

1. La règle Delta est une règle d'apprentissage de la descente sur gradient permettant de mettre à jour les poids des entrées des neurones artificiels dans un réseau de neurones à une seule couche

être stockés en base de données ou bien traités afin d'enrichir des données déjà présentes.

Le traitement des données traite les flux d'événements, ces événements sont stockés dans un système de données en direct. L'architecture Lambda permet le traitement des données en introduisant trois couches distinctes :

- **Batch Layer**, couche de traitement par lots.
- **Speed Layer/Stream Layer**, couche de traitement de flux
- **Serving Layer**, couche de service.

Batch Layer

Les données arrivent constamment comme un flux vers le système de données. Tout nouveau flux de données entrant dans la couche de traitements par lots est calculé et traité sur un lac de données².

Speed Layer

La couche vitesse se sert des résultats fournis par la couche de traitement par lots. Les flux de données entrant peuvent provenir de sources extérieures (Ex : appareils IoT), ou bien d'événement qui ont été créés lors du traitement des données sur la couche de traitement par lots. Le second cas est spécialement vrai si l'on veut effectuer du machine learning (A.3), afin de faire de la prédiction sur les prochaines données qui vont arriver. Comme son nom l'indique, la couche vitesse a une faible latence car elle ne traite que des données en temps réel contrairement à la couche de traitements par lots.

Serving Layer

Les sorties de la couche de traitement par lots sont sous la forme de vues par lots, et celles de la couche vitesse sont sous la forme de vues en quasi temps réel. Ces vues sont transmises à la couche de service, qui va utiliser ces données afin de stocker ces vues et donc les rendre accessibles au client qui va les exploiter à l'aide d'outil d'analyse et de visualisation.

la figure 1.2, représente un diagramme basique de ce à quoi ressemble le modèle de l'architecture lambda.

2. Un lac de données (en anglais Data Lake) est une méthode de stockage des données utilisée par le big data. Ces données sont gardées dans leurs formats originaux ou sont très peu transformées.

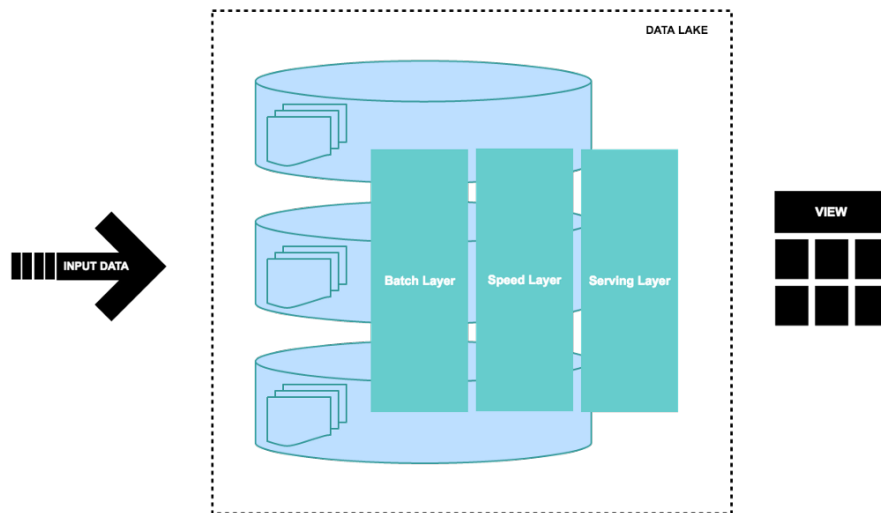


Figure 1.2 – Schéma de l'architecture Lambda

Traduisons ce fonctionnement en une equation fonctionnelle qui définit toute requête dans le domaine du Big Data. Les symboles utilisés dans cette équation sont connus sous le nom de "Lambda" et c'est de la que cette architecture s'est vu ce nom attribué.

$$\text{requête} = \lambda(\text{Données complètes}) = \lambda(\text{Données temps réel}) \times \lambda(\text{Données stockées})$$

Cette équation signifie que toutes les requêtes relatives aux données peuvent être traitées dans l'architecture lambda en combinant les résultats du stockage historique issue tu traitement par lots et des données en temps réel.

Applications de l'architecture Lambda

L'architecture Lambda peut être déployé dans les cas suivants :

- Le résultat des requêtes des utilisateurs, ne doit être calculé que lorsque que les requêtes sont émises (Pas de système de cache du résultat des requêtes).
- Les réponses doivent être rapide et le système doit être capable de gérer diverses mises à jour sous la forme de nouveau flux de données.
- Aucun des enregistrements stockés ne doit être effacé, mais l'ajout et la modification d'enregistrement doit être possible.

L'architecture Lambda peut être considéré comme une architecture de traitement de données en temps quasi réel. Comme nous l'avons mentionné précédemment, elle est tolérante aux pannes et évolutive. Elle possède une couche de traitements par lots et une couche vitesse de traitements en temps réel. Et elle garantit un stockage permanent des données. Cette architecture est utilisé par des entreprises comme Twitter, Netflix et Yahoo pour répondre aux normes de qualité de service.

Avantages et inconvénients de l'architecture Lambda

Nous allons maintenant voir à partir de tout ce qu'on a vu de cette architecture, quels sont ses avantages et inconvénients.

Avantages

- La couche de traitement par lots gère les données historiques avec un stockage distribué à tolérance de pannes, ce qui réduit les risques d'erreurs, même en cas de panne du système.
- Les données fournies au client sont les plus fraîches possible.
- Architecture à tolérance de pannes et évolutive pour le traitement des données.

Inconvénients

- La logique est implémentée deux fois (couche vitesse et couche de traitement par lots).
- Le fait de retraiter chaque traitement par lots n'est pas utile dans tous les cas.
- Il existe des solutions plus simples lorsque le besoin n'est pas complexe.

1.2 Architecture Kappa

En 2014, Jay Kreps a entamé une discussion au cours de laquelle il a souligné certaines divergences dans l'architecture Lambda. L'architecture Kappa est née en réaction à la complexité de l'architecture Lambda, notamment avec la division du traitement par lots et du traitement en temps réel.

L'architecture Kappa ne peut en aucun cas être considérée comme un substitut de l'architecture Lambda. Au contraire, elle doit être considérée comme une alternative à celle-ci, spécifiquement dans les cas où la couche de traitement par lots n'est pas au premier plan. La figure 1.3 présente le fonctionnement de l'architecture Kappa.

Traduisons le fonctionnement du traitement dans cette architecture en une équation fonctionnelle qui définit toute requête dans le domaine du Big Data.

$$\text{requête} = \kappa(\text{Nouvelles données}) = \kappa(\text{Flux de données en temps réel})$$

Cette équation montre que toutes les requêtes peuvent être traitées par l'application de la fonction kappa au flux de données en temps réel sur la couche de service.

Applications de l'architecture Kappa

L'architecture Kappa peut être déployée dans les cas suivants :

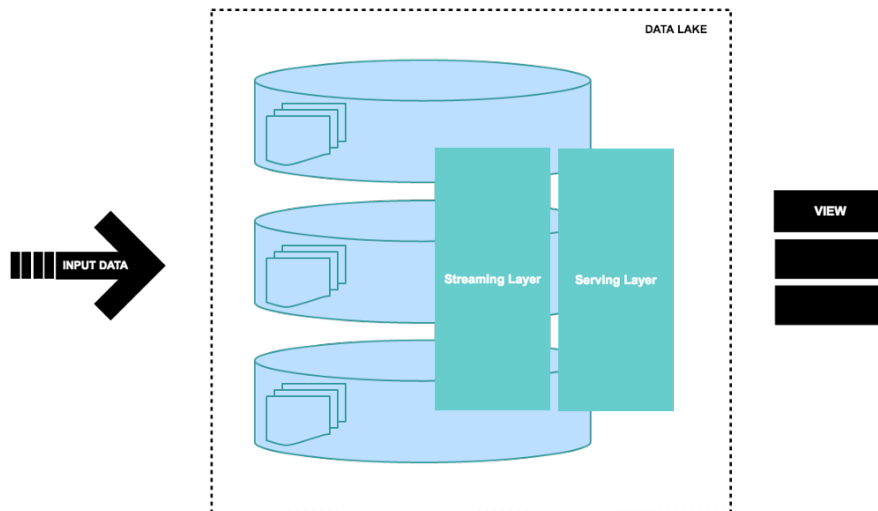


Figure 1.3 – Schéma de l'architecture Kappa

- Plusieurs événements ou requêtes sont stockés dans une file d'attente avant d'être traités un par un.
- L'ordre des événements n'est pas prédéfini, et la couche de traitement de données en temps réel doit être en mesure d'interagir avec le système de stockage n'importe quand.
- Afin de traiter des téraoctets de données, il est nécessaire que chaque nœud soit hautement disponible, résilient et supporte la réplication.

L'architecture Kappa est utilisée par des entreprises comme LinkedIn.

Avantages et inconvénients de l'architecture Kappa

Nous allons maintenant voir à partir de tout ce qu'on a vu de cette architecture, quels sont ses avantages et inconvénients.

Avantages

- Le retraitement des données est requis uniquement lorsque le code est modifié.
- Solution moins complexe que l'architecture Lambda.
- Moins de ressources requises étant donné que le traitement se fait en temps réel.

Inconvénients

- Pas de séparation entre les besoins (Temps réel et traitements par lots).
- L'absence de couche de traitement par lots peut entraîner des erreurs lors du traitement des données ou lors de la mise à jour de la base de données.

Pour conclure sur ces deux architectures, l'architecture Lambda est beaucoup plus complète que l'architecture Kappa mais cela au prix d'une complexité accrue. De nombreux cas d'utilisation en temps réel conviendront parfaitement à une architecture Lambda. On ne peut pas en dire autant de l'architecture Kappa. Dans des cas où le traitement des données par lots et en temps réel sont similaire, ou bien, si le but est principalement de fournir des données métiers aux clients, l'utilisation de l'architecture Kappa est une bonne solution. Par contre dans les cas où les traitements de données en temps réel et par lots sont complètement différents ou bien si vous avez besoin d'utiliser des modèles d'apprentissage automatique afin de faire de la prédiction sur les événements à venir, l'architecture Lambda est le meilleur choix. Maintenant que nous avons vu les différentes architecture que nous avons à notre disposition, nous allons nous intéresser aux solutions logicielles permettant de réaliser les différentes fonctions de ces architectures.

Analyse des solutions logicielles existantes

Maintenant que nous avons vu les différentes architecture Big Data existantes et que nous les avons décomposés, on va s'intéresser plus en détail à chaque composant de ces architecture. Pour chaque composant, nous allons présenter diverses solutions existantes permettant de remplir le rôle de ce dernier. Pour chacune de ses solutions, nous allons voir leurs avantages et inconvénients et leurs manière de fonctionner dans le but de pouvoir en dégager des critères de sélections.

2.1 Message Broker

Un Message broker [6], Agent de message en français, est un moyen de communication utilisant des messages entre deux applications (Ex : Communication entre un serveur et un client). Un message broker permet une communication asynchrone entre applications. L'utilisation de cette solution permet de pouvoir facilement filtrer les messages que l'on reçoit et de stocker temporairement les messages reçus afin d'éviter les pertes de données. Ce dernier cas, s'avère très utile dans le cas où l'application chargée de la réception des données n'est pas en fonctionnement pendant un certains temps. Il existe deux types de communications avec un message broker :

Publisher / Subscriber

Dans ce mécanisme, l'entité envoyant les données est nommé "Publisher" et l'entité les récupérant est nommé "Subscriber". Le publisher va envoyer des données dans des topics¹ afin que les Subscribers de ce topic puissent les récupérer. Un publisher peut envoyer des données dans un ou plusieurs topics, et les subscribers peuvent être abonné à un ou plusieurs topics (Voir figure 2.1).

Point-to-point communication

La communication point à point est la forme la plus simple de Producteur/Consommateur. Le producteur envoie ses données dans une queue et le consommateur va lire les

1. Un topic est une catégorie dans laquelle les messages produit sont stockés.

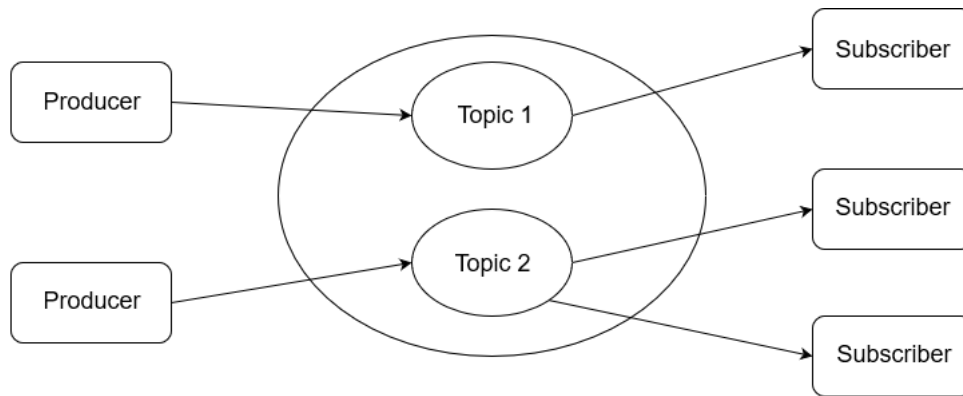


Figure 2.1 – Schéma du principe Publisher(Producer)/Subscriber

messages dans la queue. Tout comme le modèle précédent, il peut y avoir plusieurs producteur et consommateurs sur la même queue, mais si plusieurs consommateurs sont présents, ils ne recevront des portions différentes des messages afin de favoriser le traitement concurrentiel.

2.1.1 Kafka

Apache Kafka [7][8] est un système de messages distribué, développé par LinkedIn.

Kafka utilise le système de communication du Publisher(Producer) / Subscriber en utilisant un élément unique, appelé broker. Afin de faciliter la mise à l'échelle et le partitionnement des données, il est possible d'instancier autant de brokers que l'on souhaite afin d'augmenter le débit et la résilience du produit.

Une autre brique est utilisée par Kafka afin de gérer l'état des différents brokers, il s'agit de Apache ZooKeeper. Il permet de stocker facilement toutes les méta-données de chaque broker, comme par exemple le nombre de données injectées dans chaque topic, ou bien la répartition des différents topics sur chaque broker (Voir figure 2.2).

Les messages envoyés à Kafka sont stockés sur disque dans le format appelé **LOG**. Ce format n'a rien à voir avec les logs applicatifs, il s'agit d'un tableau de messages ordonnés. L'ordonnement est réalisé à partir de la date d'arrivée du message. Chaque message se voit donner un index aussi appelé offset.

2.1.2 ActiveMQ

Comme Kafka, ActiveMQ utilise le système de communication Publisher(Producer) / Subscriber. L'intérêt principal d'ActiveMQ est de connecter différentes applications réalisées avec des langages différents avec l'aide des API fournies.

De la même manière que Kafka, il permet de se mettre à l'échelle très facilement mais il n'utilise pas de brique intermédiaire afin de gérer les états des brokers (ZooKeeper). Les différentes instances d'ActiveMQ utilisent en système de multicast afin de se découvrir.

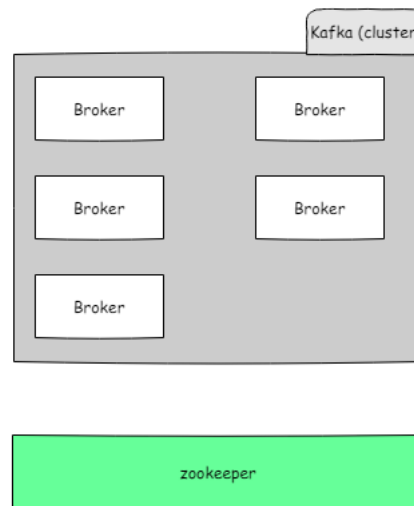


Figure 2.2 – Schéma d'un cluster Kafka et des ces composants

2.1.3 RabbitMQ

Contrairement à Kafka et ActiveMQ, RabbitMQ utilise le système de communication Point-to-Point. Cela permet de garantir qu'un seul consommateur vas être en mesure de lire le message présent dans la queue.

RabbitMQ n'utilise pas de brique extérieure afin de gérer la mise à l'échelle, mais il suit le même mode de fonctionnement que ces deux concurrents, c'est à dire un système de partition.

Conclusion

Nous pouvons constater que ces trois solutions se ressemblent, elle se se départagent uniquement sur quelques points. L'utilisation de différents protocoles, le mécanisme de communication ou bien la manière de se mettre à l'échelle.

2.2 Ingestion/Extraction de données

La première catégorie, qui est aussi la première étape d'une architecture Big Data, c'est le récupération de données. Plus précisément comment nous allons récupérer des données, soit via des requêtes sur des sources externes, soit des sources externes nous envoie directement des données.

Il existe deux approches afin d'effectuer cette tâche, soit l'utilisation d'un logiciel appelé ETL ou ELT ou bien, l'écriture de programme. Un logiciel ETL (Extract Transform & Load) ou ELT (Extract Load & Transform), sont des solutions permettant d'extraire

des données depuis des sources, appliquer une transformation sur ces données et enfin les charger dans une solution de stockage. En plus d'être solution complète, la gestion du flux des données est entièrement paramétrable à l'aide d'une interface graphique afin de faciliter l'utilisation de ces outils. La distinction entre ETL et ELT est l'ordre dans lequel les opérations sont effectuées, soit les données sont d'abord transformées puis stockées (ETL) soit l'inverse (ELT). Les deux premières solutions que nous allons aborder pour répondre à la problématique de l'ingestion des données sont des ETL/ELT.

2.2.1 Apache Nifi

2.2.2 Talend

2.2.3 Solution maison

Il est aussi tout à fait possible d'envisager d'écrire un programme simple permettant d'extraire des données des données depuis une source pour ensuite les injecter dans une base et lancer le traitement des données. Pour nous aider dans cette tâche, il existe dans une majorité des langages des connecteurs permettent d'interagir par exemple avec des bases de données. Toutefois, il est important de respecter certains critères avant de se lancer dans le développement d'un programme maison. Comme on l'a vu tout au long de notre recherche, l'un des points clés du Big Data est sa capacité de mise à l'échelle. Il faut donc choisir le langage et/ou un framework pouvant lui aussi être mis à l'échelle facilement, c'est à dire qu'il doit adopté une architecture réactive (A.1). Deux frameworks très connu utilisant cette architecture sont Akka et Vert.x. Akka utilise le principe de concurrence Acteur tandis que Vert.x utilise le principe d'évènements. Akka supporte officiellement Java et Scala et on peut facilement l'utiliser avec d'autres langages de la JVM². De son côté, Vert.x supporte de manière officielle Java, Scala, Kotlin, Javascript, Ruby et Ceylon. Il existe des modules pour Vert.x permettant de supporter d'autres langages comme le python par exemple mais il ne sont pas maintenu par Vert.x.

Conclusion

Pour conclure sur cette partie, nous avons donc le choix d'utiliser des solutions complètes configurable facilement, ou bien écrire nous même un programme réalisant l'ingestion des données. L'utilisation d'une solution complète peut paraître la plus attractive mais elle est plus consommatrice en ressources qu'un simple en programme.

2. Machine virtuelle java

2.3 Traitement des données

Après avoir récupérer des données, nous devons passer à l'étape du traitements des données. Celui ci à plusieurs rôles, en effet il peut servir à formater les données, leurs apportés de la cohérence en les combinant à des données déjà présentent. Et pour finir les rediriger vers le stockage souhaité. Le traitement des données peut se faire de deux manière différentes. La première solution est le traitement par Batch, et la seconde est le traitement en temps réel [9][10]. Chacune possède ses avantages et inconvénients, nous allons voir ça plus en détails.

2.3.1 Batch

Le traitement par Batch (Traitement par lot), consiste à traiter un important volume de données à un instant T. Le traitement par batch est surtout utilisé dans les cas ou nous avons des données stockés de manière journalière, et que nous avons besoin de tout traités en fin de journée. Il n'est pas rare de voir des tâches de traitements par Batch s'exécuter dans la nuit, étant donnée que l'on traite une masse de donnée importante, on sollicite la machine pendant une longue période. Réaliser ce traitement durant des périodes creuses, permet de largement diminuer l'impact sur l'utilisation de la plateforme (2.3).

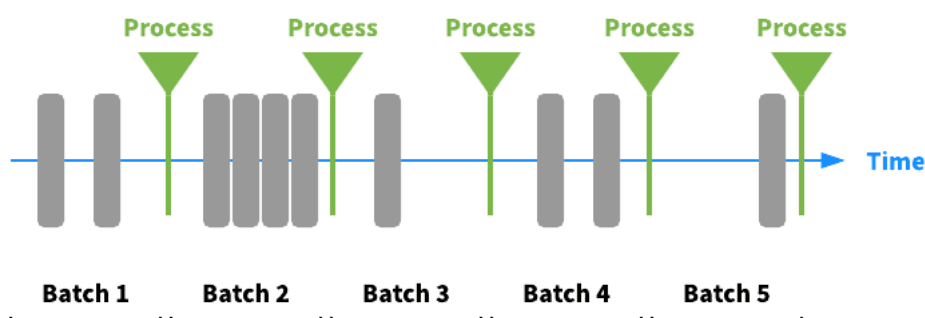


Figure 2.3 – Schéma du traitement par batch

Nous allons nous pencher sur les solutions existantes, implémentant un traitement par batch.

2.3.1.1 Apache Spark

Spark est un moteur de traitement de données distribué, il répond à de nombreux cas d'utilisation. En plus du moteur de traitement de données, Spark possède des bibliothèques SQL, d'apprentissage automatique, de calcul des graphes ainsi que le traitement de flux qui sera abordé dans la partie 2.3.2.1 sur Spark Streaming. Spark supporte différents langages de programmation, Java, Scala, Python et R.

Spark a des performances accrues car entre chaque étape des calculs, au lieu de stocker les résultats sur disque il les garde en RAM.

2.3.1.2 Hadoop MapReduce

MapReduce est aussi un moteur de traitement de données distribué, mais il répond à de moins nombreux cas d'utilisation que son concurrent Spark. Il ne possède qu'une seule librairie permettant de faire de l'apprentissage automatique. Il manque donc de librairies SQL, de calcul des graphes ainsi que de traitement de flux contrairement à son concurrent.

Au niveau des performances, MapReduce se montre moins efficace que Spark. Là où Spark va stocker en RAM tous les résultats intermédiaires, MapReduce va stocker ces résultats sur disque. Cela a un impact non négligeable sur les performances, mais il permet aussi de diminuer les coûts de la plateforme et de garantir une meilleure tolérance à la panne.

Conclusion

Ces deux solutions sont intéressantes, mais couvrent des cas d'utilisation vraiment différents. Il est donc très important de bien cibler son besoin afin de faire le bon choix pour son architecture.

2.3.2 Streaming

Un traitement de données est considéré comme étant en temps réel si il s'effectue en une seconde ou moins après la réception de la donnée. Il peut être de deux types, soit des micro batchs soit en streaming.

Micro-Batch

Le traitement par micro batch est basé sur le même principe que le traitement par batch, à l'exception qu'il s'exécute beaucoup plus régulièrement (Toutes les secondes ou moins) et que le nombre de données à traiter est donc significativement plus faible. Le micro batch est surtout utilisé dans les cas où notre système ne peut pas directement réagir lorsqu'une donnée arrive, on va donc récupérer les données très régulièrement afin de garantir un traitement en temps réel ou du moins dans le délai le plus bref possible.

Streaming

Le traitement en streaming (Traitement de flux), s'appuie sur l'architecture réactive (A.1). En effet, contrairement aux traitements par batch et micro batch, ici on ne va

pas récupérer des données de temps en temps. Dès qu'une donnée arrive on va la récupérer et la traiter immédiatement. De par son fonctionnement, le traitement en streaming ne nécessite pas de stockage en amont contrairement aux autres type de traitement (2.4).

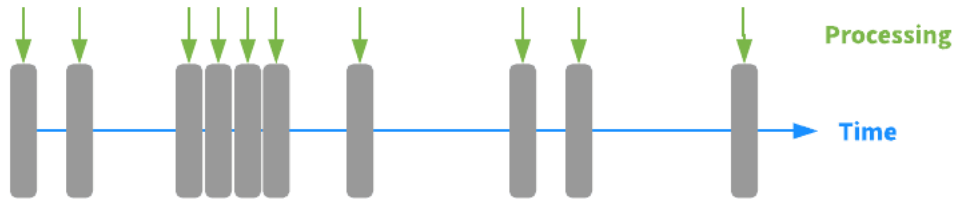


Figure 2.4 – Schéma du traitement en streaming

Nous allons maintenant nous intéresser aux différentes solutions proposant ce type de traitement.

2.3.2.1 Apache Spark Streaming

Spark Streaming est comme on l'a vu précédemment une version de Spark étudié pour le traitement en temps réel. Il bénéficie donc du large choix de librairie de Spark.

Spark Streaming à une latence assez faible mais pas suffisamment pour s'orienter vers du traitement en streaming il est plus orienté pour un traitement en micro-batch.

Spark Streaming s'avère très utile dans le cadre d'une architecture Lambda, comme on l'a vue dans cette architecture les traitements en batch et en temps réel sont effectués par des outils différents. Spark Streaming permet d'avoir la même base de code pour le traitement en batch et en temps réel, cela permet d'éviter de la duplication de code et de pouvoir n'utiliser entre autre qu'un seul outil pour le traitement des données..

2.3.2.2 Apache Storm

Apache Storm propose une sélection moins importante de langage de programmation que Spark Streaming. Il ne propose que le Java, Scala et Clojure.

Contrairement à Spark Streaming, Apache Storm possède une latence beaucoup plus faible ce qui lui permet de gérer le traitement en streaming sans aucun problème.

Tout comme Spark Streaming, Apache Storm stock les résultats intermédiaires en RAM pour garantir des performances accrues.

Apache Storm ne propose pas de librairie de calcul des graphes et n'a pas de librairie d'apprentissage automatique intégré, il faudra en installer une compatible contrairement à Spark Streaming.

Apache Storm à l'opposé de son concurrent ne permet pas de garder la même base de code entre le traitement en batch et en streaming, son utilisation se place par conséquent plus dans le cadre d'une architecture Kappa.

Conclusion

Malgré un mode de fonctionnement similaire, ces deux solutions peuvent se démarquer sur des points importants qui nous permettront de définir des critères précis par rapport à leur utilisation.

2.4 Stockage des données

Une partie très importante du Big Data est le stockage des nombreuses données que l'on reçoit. Il existe énormément de manières différentes de stocker des données selon la manière dont nous voulons les utiliser par la suite et surtout selon leur format. De plus l'utilisation de plusieurs bases de données est très courante, généralement une base de données est utilisée pour le stockage des données brutes avant leur traitement puis une autre base de données correspondant au nouveau format des données est utilisée pour améliorer les performances.

2.4.1 Time Series

La première catégorie de base de données que nous allons traiter, est la base de données de séries temporelles. Ce type de stockage est de plus en plus important avec l'explosion de l'IoT qui est un des domaines générant le plus de données de séries temporelles.

Une série temporelle est tout simplement une valeur datée, par exemple la température d'un processeur à un instant T .

2.4.1.1 OpenTSDB

OpenTSDB est une base de données Open Source sous la licence GPL3 écrite en Java.

Afin de stocker les données, OpenTSDB se base sur une autre base de données appelée HBase, nous verrons plus en détail HBase dans la partie 2.4.6.1. Cette solution implique donc que vous avez déjà HBase installé et d'avoir les connaissances nécessaires pour le configurer correctement afin de garantir un bon débit de lecture et d'écriture.

OpenTSDB ne fournit pas d'outil de requête simplifié, il faut donc être familier avec HBase pour récupérer des données stockées dans OpenTSDB.

2.4.1.2 InfluxDB

InfluxDB est une solution Open Source développée par InfluxData sous la licence MIT et écrite en Go.

Contrairement à OpenTSDB, InfluxDB n'utilise de solution externe pour le stockage des données. Ils possède sa propre solution optimisé pour les données de série temporelle. Cela à un double avantage. Premièrement, pas besoin d'installer une autre base de données et d'avoir les connaissances nécessaire pour la configurer pour des données de série temporelles. Cela permet aussi d'avoir des performances accrues, l'architecture de stockage étant crée dans le seul but de stocker des données de série temporelle, il est normal que les performances soient meilleures que pour une solution qui peut gérer plusieurs formats de données.

Par rapport à son concurrent, InfluxDB propose une solution de requetâge de données simplifiant en utilisant le langage SQL.

Conclusion

Nous pouvons constater que InfluxDB à l'air d'être la meilleure solution pour le stockage de série temporelle, mais le fait que OpenTSDB utilise HBase peut être un critère de choix pour certaines personnes pour le choix de leur architecture.

2.4.2 Graph

2.4.2.1 Neo4j

2.4.2.2 JanusGraph

2.4.3 Clés/Valeurs

2.4.3.1 Redis

2.4.3.2 RocksDB

2.4.4 Moteur d'indexation

2.4.4.1 ElasticSearch

2.4.4.2 Solr

2.4.5 Documents

2.4.5.1 CouchDB

2.4.5.2 CouchBase

2.4.5.3 MongoDB

2.4.6 Wide Column

2.4.6.1 HBase

2.4.6.2 Cassandra

2.4.7 Système de fichiers

2.4.7.1 Hadoop HDFS

2.5 Orchestration

2.6 Requetâge

2.7 Visualisation et Analyse des données

2.7.1 Kibana

2.7.2 Banana

2.7.3 Grafana

2.7.4 Tableau

Critères d'analyse

Maintenant que nous sommes familier avec les architectures Big Data ainsi qu'avec les différentes solutions logicielle, nous allons pouvoir définir des critères afin de sélectionner l'architecture qui correspond à notre besoin. Certains critères ont déjà été dégagé au fil de ce mémoire. La première étape va être de définir des critères par rapport au choix de l'architecture et ensuite des critères pour choisir les solution logicielles à utiliser au sein de cette architecture. En plus des différentes informations que nous avons exposé jusqu'ici, nous aurons besoin de prendre en compte le cas d'utilisation nécessitant le déploiement d'une architecture Big Data. Dans un premier temps, nous allons nous intéresser ce la manière dont les entreprises effectuent leurs choix d'architecture.

3.1 Sélection d'architecture en entreprise

Cette partie sera entièrement basé sur mon expérience professionnel, je ne peux en aucun cas garantir que la sélection d'une architecture s'effectue de la même manière dans toutes les entreprises.

3.2 Critères pour le choix de l'architecture

Avant de voir les différent critères par rapport au choix de l'architecture, je tiens à rappeler ce qui a été expliquer en introduction de ce mémoire, c'est qu'avant de passer sur une architecture Big Data il faut s'assurer d'en avoir l'utilité.

Nous allons exposer les critères ainsi que leur solution adapté sous la forme d'un tableau afin de permettre une visualisation simple pour effectuer notre choix.

Critère	Architecture
Prédiction d'évènement entrant à l'aide de modèle d'apprentissage automatique	Lambda
Traitement des données en temps réel et par lots radicalement différents	Lambda
Traitement des données par lots complexe	Lambda
Très faible latence entre récupération et affichage des données	Kappa
Traitement des données par lots et en temps réel similaires	Kappa
Stockage permanent des données batch avant le traitement	Lambda/Kappa

Table 3.1 – Table des critères pour le choix de l'architecture

3.3 Critères pour le choix des solutions logicielles

3.3.1 Ingestion des données

3.3.2 Message Broker

3.3.3 Traitement des données

3.3.4 Stockage des données

3.3.5 Moteur d'indexation

3.3.6 Orchestration

3.3.7 Visualisation et Analyse des données

3.4 Aller plus loin

Implémentation : Exemple avec un processus métier

4.1 Définition du processus métier

4.2 Application des méthodes sur le processus métier

4.3 Évaluation du résultat

Conclusion

Les différents concepts du Big Data

A.1 Architecture Réactive

Le nombre de données augmentant très rapidement et les clients demandant un service le plus rapide possible et disponible à tout moment, il est de ce fait important d'avoir une infrastructure facilement adaptable à ce flux. C'est le but de l'architecture réactive [11]. Elle repose sur quatre principes :

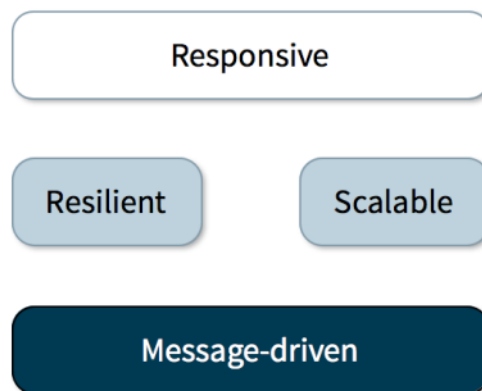


Figure A.1 – Schéma de l'architecture réactive

- Responsive : C'est l'objectif à atteindre.
- Scalable et Resilient : L'objectif ne peut pas être atteint sans remplir ces deux conditions.
- Message Driven : C'est la base qui permettra d'accueillir tous les composants afin d'obtenir une architecture réactive

Responsive : Un système responsive doit être en mesure de réagir rapidement à toutes les requêtes peu importe les circonstances, dans le but de toujours fournir une expérience utilisateur positive. La clé afin de proposer ce service, est d'avoir un système élastique et résilient, les architecture "Message Driven" fournissent les bases pour un système réactif. L'architecture Message Driven est aussi important pour avoir un système responsive, parce que le monde fonctionne de manière asynchrone tout comme elle. Voici un exemple : Vous voulez vous faire du café, mais vous vous rendez compte que vous n'avez plus de crème et de sucre.

Première approche :

- Mettre du café dans votre machine.
- Aller faire vos courses pendant que la machine fait couler le café.

- Acheter de la crème et du sucre.
- Rentrer chez vous.
- Boire votre tasse de café.
- Profiter de la vie !

Seconde approche :

- Aller faire vos courses.
- Acheter de la crème et du sucre.
- Rentrer chez vous.
- Mettre du café dans votre machine.
- Regarder impatiemment la cafetière se remplir.
- Expérimenter le manque de caféine.
- Et enfin, boire votre tasse de café.

Grâce à la première approche, on peut clairement voir la différence de gestion de l'espace et du temps. C'est grâce à cela que cette architecture est un atout primordiale pour assurer un système responsive.

Resilient :

La plupart des applications sont conçus par rapport à un fonctionnement idéal. C'est à dire qu'elle ne prévoient pas de gérer correctement les erreurs qui peuvent intervenir plus souvent que ce qu'on pourrait croire. Cela à pour effet de ne pas garantir une disponibilité continu et peut provoquer la perte de crédibilité d'un service. La résilience est là pour pallier à ce problème, elle à pour but de récupérer les erreurs émises et de les traiter correctement afin de ne pas provoquer d'interruption de service et de ne pas impacter l'expérience utilisateur.

Scalable :

La résilience et l'élasticité vont de pair dans la construction d'une architecture réactive. L'élasticité permet d'adapter facilement le système à la demande et d'assurer sa réactivité peu importe la charge qu'il subit. L'élasticité est un point très important, lorsque votre plateforme reçoit énormément de connexion cela veut dire que votre service connaît un succès important, c'est donc le pire moment pour avoir un service non disponible. Cela entrainerait une perte de crédibilité et de clients.

Il existe deux moyens différents de rendre élastique une application :

- La mise à l'échelle verticale
- La mise à l'échelle horizontale (Architecture Répartie)

La mise à l'échelle verticale consiste à maximiser l'utilisation des ressources de notre machine. Cela peut se faire par l'utilisation de programmes fonctionnant en asynchrone sur plusieurs threads. Tandis que la mise à l'échelle horizontale consiste à augmenter le nombre de machine afin d'avoir plus de ressources à notre disposition. Pour le moment on ne vas pas trop s'attarder sur la division horizontale car une partie lui est dédiée.

La mise à l'échelle verticale est certes la moins couteuse, mais ce n'est pas la plus simple à mettre en place. En effet le multithreading permet de tirer la maximum des performances de la machine, mais cela ajoute une certaine complexité au programme. Par exemple le fait de travailler avec des variables mutables entre différents threads n'est pas une tâche aisée, cela provoque donc des limitations à la mise à l'échelle

verticale. De plus, une fois avoir exploiter au maximum une machine en ayant mis en place la mise à l'échelle verticale, si on a encore besoin de ressources supplémentaire la mise à l'échelle horizontale devient indispensable.

Message-Driven :

Le Message-Driven Architecture

A.2 Architecture Répartie

A.3 Machine Learning

Bibliographie

- [1] Julie MARTIN. *Les 5 V du Big Data*. url : <https://medium.com/@jm.julie.martin/les-5-v-du-big-data-1d0462896468>.
- [2] Zoiner Tejada et olprod. *Architectures Big Data*. url : <https://docs.microsoft.com/fr-fr/azure/architecture/data-guide/big-data>.
- [3] Christophe Parageaud. *Big Data, panorama des solutions*. url : <https://blog.ippon.fr/2016/03/31/big-data-panorama-des-solutions-2016/>.
- [4] Iman Samizadeh. *A brief introduction to two data processing architectures—Lambda and Kappa for Big Data*. url : <https://towardsdatascience.com/a-brief-introduction-to-two-data-processing-architectures-lambda-and-kappa-for-big-data-4f35c28005bb>.
- [5] Michael Verrilli. *From Lambda to Kappa : A Guide on Real-time Big Data Architectures*. url : <https://www.talend.com/blog/2017/08/28/lambda-kappa-real-time-big-data-architectures/>.
- [6] <https://www.tibco.com>. *What is a Message Broker?* url : <https://www.tibco.com/reference-center/what-is-a-message-broker>.
- [7] Anthony. *Introduction à Apache Kafka*. url : <https://medium.com/@AnthonyDasse/introduction-%C3%A0-apache-kafka-d126f2bb852b>.
- [8] tutorialspoint. *Apache Kafka - Introduction*. url : https://www.tutorialspoint.com/apache_kafka/apache_kafka_introduction.htm.
- [9] Laura Shiff. *Real Time vs Batch Processing vs Stream Processing : What's The Difference?* url : <https://www.bmc.com/blogs/batch-processing-stream-processing-real-time/>.
- [10] streamlio. *Understanding Batch, Microbatch, and Streaming*. url : <https://streamlio/resources/tutorials/concepts/understanding-batch-microbatch-streaming>.
- [11] Kevin Webber. *What is Reactive Programming?* url : <https://blog.redelastic.com/what-is-reactive-programming-bc9fa7f4a7fc>.

Table des matières

1	Les architectures Big Data	9
1.1	Architecture Lambda	11
1.2	Architecture Kappa	14
2	Analyse des solutions logicielles existantes	17
2.1	Message Broker	17
2.1.1	Kafka	18
2.1.2	ActiveMQ	18
2.1.3	RabbitMQ	19
2.2	Ingestion/Extraction de données	19
2.2.1	Apache Nifi	20
2.2.2	Talend	20
2.2.3	Solution maison	20
2.3	Traitement des données	21
2.3.1	Batch	21
2.3.1.1	Apache Spark	21
2.3.1.2	Hadoop MapReduce	22
2.3.2	Streaming	22
2.3.2.1	Apache Spark Streaming	23
2.3.2.2	Apache Storm	23
2.4	Stockage des données	24
2.4.1	Time Series	24
2.4.1.1	OpenTSDB	24
2.4.1.2	InfluxDB	24
2.4.2	Graph	26
2.4.2.1	Neo4j	26
2.4.2.2	JanusGraph	26
2.4.3	Clés/Valeurs	26
2.4.3.1	Redis	26
2.4.3.2	RocksDB	26

2.4.4	Moteur d'indexation	26
2.4.4.1	ElasticSearch	26
2.4.4.2	Solr	26
2.4.5	Documents	26
2.4.5.1	CouchDB	26
2.4.5.2	CouchBase	26
2.4.5.3	MongoDB	26
2.4.6	Wide Column	26
2.4.6.1	HBase	26
2.4.6.2	Cassandra	26
2.4.7	Système de fichiers	26
2.4.7.1	Hadoop HDFS	26
2.5	Orchestration	26
2.6	Requetâge	26
2.7	Visualisation et Analyse des données	26
2.7.1	Kibana	26
2.7.2	Banana	26
2.7.3	Grafana	26
2.7.4	Tableau	26
2.7.5	Click	26
3	Critères d'analyse	27
3.1	Sélection d'architecture en entreprise	27
3.2	Critères pour le choix de l'architecture	27
3.3	Critères pour le choix des solutions logicielles	28
3.3.1	Ingestion des données	28
3.3.2	Message Broker	28
3.3.3	Traitement des données	28
3.3.4	Stockage des données	28
3.3.5	Moteur d'indexation	28
3.3.6	Orchestration	28
3.3.7	Visualisation et Analyse des données	28
3.4	Aller plus loin	28

4	Implémentation : Exemple avec un processus métier	29
4.1	Définition du processus métier	29
4.2	Application des méthodes sur le processus métier	29
4.3	Évaluation du résultat	29
Annexe A	Les différents concepts du Big Data	33
A.1	Architecture Réactive	33
A.2	Architecture Répartie	35
A.3	Machine Learning	35
	Bibliographie	37

Table des figures

1	Schéma des 5V du Big Data	7
1.1	Composants d'une architecture Big Data	9
1.2	Schéma de l'architecture Lambda	13
1.3	Schéma de l'architecture Kappa	15
2.1	Schéma du principe Publisher(Producer)/Subscriber	18
2.2	Schéma d'un cluster Kafka et des ces composants	19
2.3	Schéma du traitement par batch	21
2.4	Schéma du traitement en streaming	23
A.1	Schéma de l'architecture réactive	33

Liste des tableaux

3.1	Table des critères pour le choix de l'architecture	28
-----	--	----