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 commencer mon stage de Master 1 chez EDF dans le service Big Data, pour moi le Big Data était une notion très flou 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 nécessaire 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 adapté 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 constitué une architecture Big Data. Puis nous verrons plus en détails les architectures qui ont été crées 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 nécessaire 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 réel et on la comparera aux autres possibilités d'architecture possible pour s'assurer que c'était le meilleur choix.

Sommaire

Introduction

Le Big Data

La démocratisation des ordinateurs et l'avènement du WEB 2.0 a entraîné une digitalisation de nos connaissances. Aujourd'hui 95la 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).

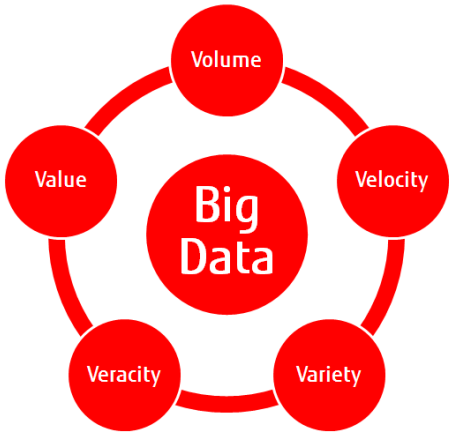

Figure 1 – Schéma des 5V du Big Data

- **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 évidement 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.

Chapitre 1

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.

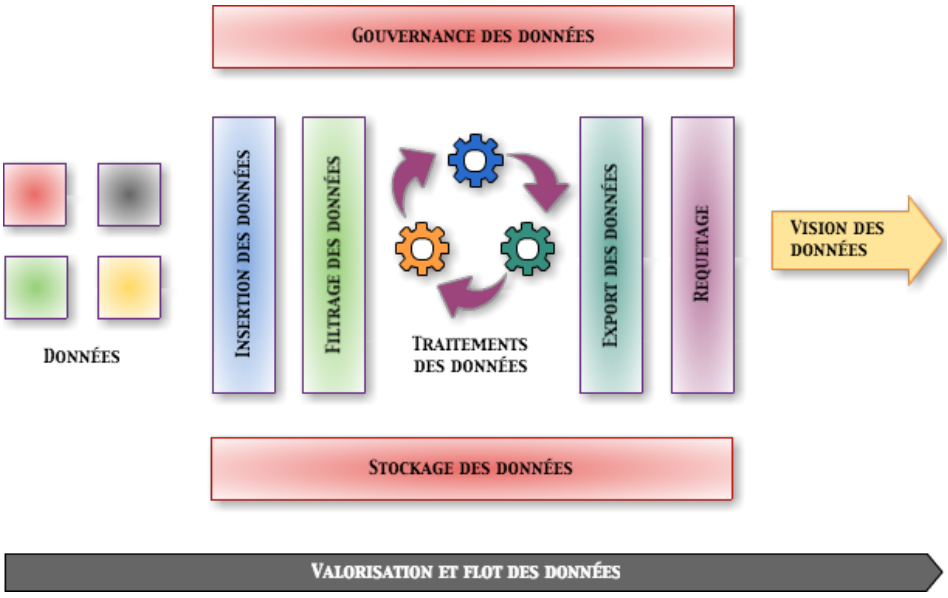

Figure 1.1 – Composants d'une architecture Big Data

Nous allons maintenant détailler le rôle de chaque composant présenté dans le digramme 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 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 lourd 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 spécialisé 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 préparer pour l'analyse. le traitement est similaire au traitement par lot, seul les outils sont différents, car ils doivent être capable de gérer le 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 ou un stockage de donnée 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 requetâge 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 requetâge 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 a 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 catégorie, et l'on constate qu'une catégorie existe sous 2 formes, le traitement des données. Les deux architectures différentes sont justement sont 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 delta [1] sous la forme code logique ou de traitement de langage naturel avec des modèles de traitement de données basés sur des évènements 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 évènement via l'exécution d'une commande ou d'une procédure delta.

La récupération d'évènements est un concept qui consiste à utiliser les évènements 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 évènements lors du chargement d'une page, de l'ajout en favoris 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 présentent.

Le traitement des données traite les flux d'évènements, 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 arrive 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 données [2].

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 source extérieures (Ex : appareils IoT), ou bien d'événement qui ont été crée 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.

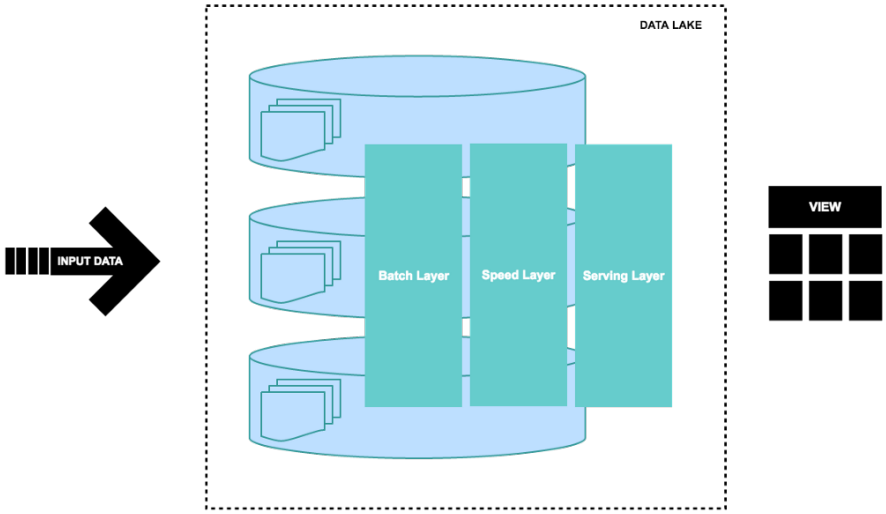

Figure 1.2 – Schéma de l'architecture Lambda

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.

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

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, quel 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 fournis au client sont les plus fraiches 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 simple 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 ou la couche de traitement pas 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 traités 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 déployé dans les cas suivants :

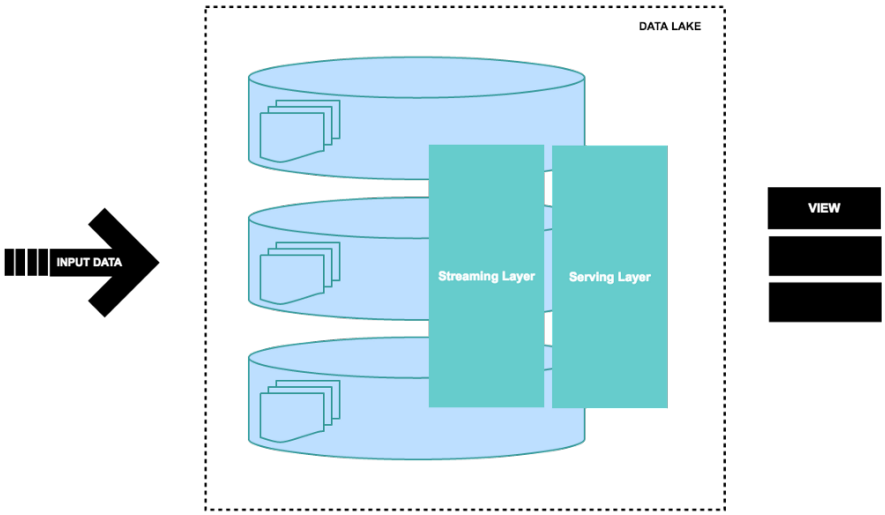

Figure 1.3 – Schéma de l'architecture Kappa

- Plusieurs événements ou requêtes sont stockés dans une file d'attente avant d'être traités traités un par un.
- L'ordre des évènements 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 utilisé 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, quel 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 requise étant 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 évènements 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.

Chapitre 2

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 topics [1] 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.

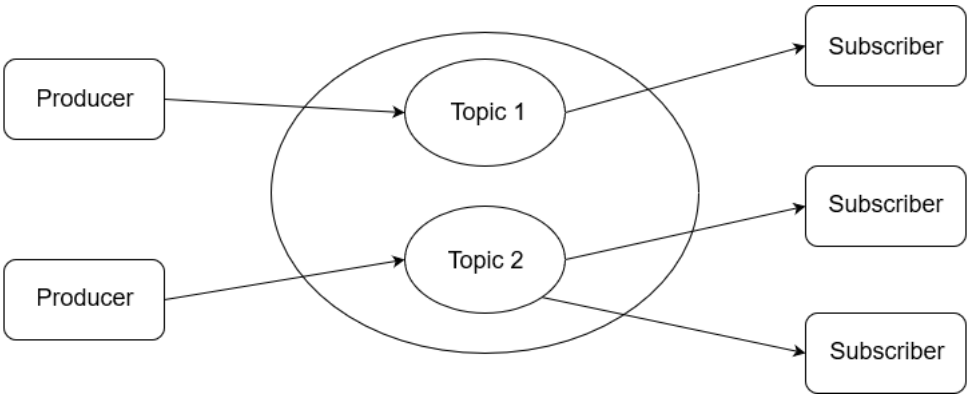

Figure 2.1 – Schéma du principe Publisher(Producer)/Subscriber

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 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 que broker que l'on souhaite afin d'augmenter le débit et la résilience du produit.

Une autre brique est utilise 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 topics, ou bien la répartition des différentes 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'ordonnancement 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 application réaliser avec des langages différents avec l'aide des API fournit.

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 un système de multicast afin de se découvrir.

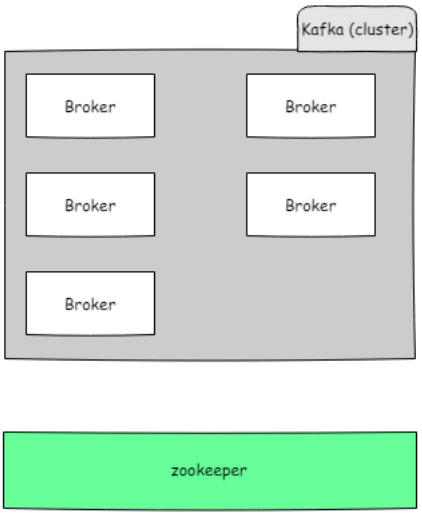

Figure 2.2 – Schéma d'un cluster Kafka et des ces composants

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 effectués, soit les données sont d'abord transformés puis stockés (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 JVM [2]. 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).

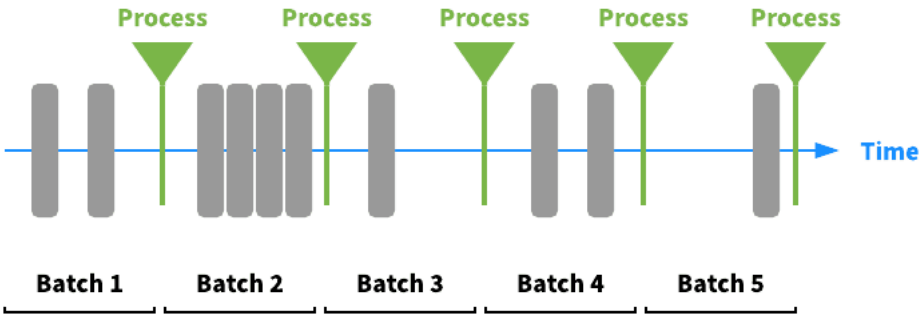

Figure 2.3 – Schéma du traitement par batch

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 librairies 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 à des performances accrue 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 tout les résultats intermédiaire, 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 couvre 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 vas

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).

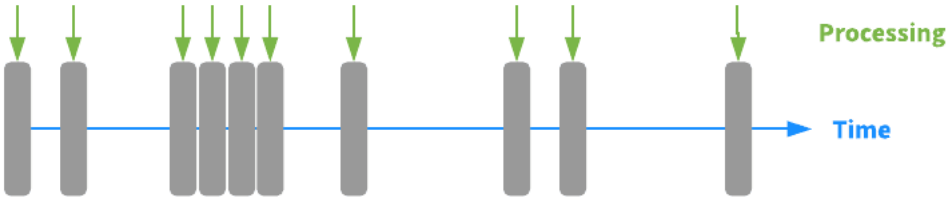

Figure 2.4 – Schéma du traitement en streaming

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 bacth 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 important qui nous permettra 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'ont 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 leurs 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 traitements (appelé lac de données) puis une autre base de données correspondant au nouveau format des données est utilisé pour améliorer les performances. Le stockage des données dans le Big Data doit s'assurer de pouvoir stocker toutes les données reçu et avoir une très haute disponibilité. Pour cela de la réplication de données et une mise à l'échelle pour la lecture et l'écriture des données. Cela implique de s'éloigner des bases de données relationnelle et de s'orienter vers des solutions NoSQL qui facilite la mise en place de ces concept. Nous n'allons pas voir en détail le fonctionnement des bases de données NoSQL, nous nous contenterons de voir si les bases de données intègrent bien ces concepts et si elles ont des particularités sur la manière de traiter les données. Si vous souhaitez voir plus en détail le fonctionnement des bases de données NoSQL je vous conseille de lire l'article de Sonia Guehis et Marta Rukoz à ce sujet [].

2.4.1 Base de données de séries temporelles

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 domaine générant le plus de donnée de séries temporelles.

Une série temporelle est tout simplement une valeur daté, 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 écrit en Java.

Afin de stocker les données, OpenTSDB se base sur une autre base de données appelé HBase, nous verrons plus en détail HBase dans la partie 2.4.6.1. Cette solution

implique donc que vous avez déjà HBase d'installer et d'avoir les connaissances nécessaire pour le configurer correctement afin de garantir des bon débit de lecture et d'écriture.

OpenTSDB ne fournit pas d'outil de requêtage 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 développé par InfluxData sous la licence MIT et écrit 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 Base de données orientée graphe

Les bases de données orientée graphe sont apparu afin de permettre de stocker facilement les relations présente entre plusieurs données. Une base de donnée de graph est basé sur les bases de données orienté objet mais avec l'utilisation de la théorie des graphes. Le stockage des données est constitué de nœuds qui représentent les données et d'arcs qui sont des pointeurs physique représentant les relations entre chaque nœuds. Ce type de bases de données s'avère très utile dans le cadre des études des relations entre de nombreuses de données, ce qui souvent le cas dans le domaine du Big Data. La figure 2.5 montre un exemple de graphe qui peut être stocké dans une base de données orientée graph.

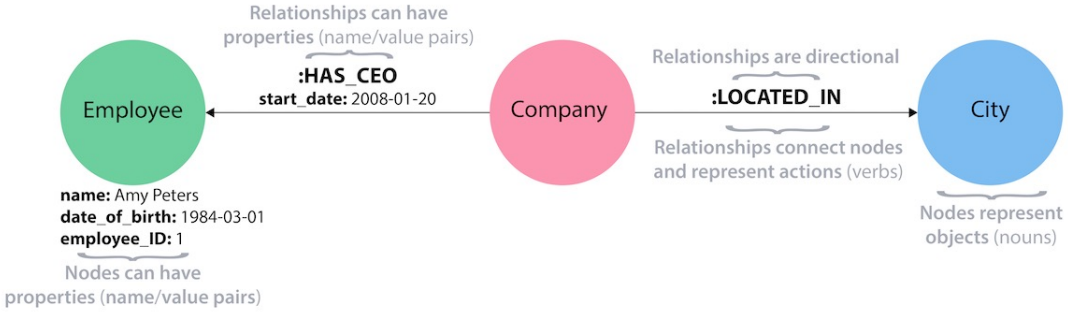

Figure 2.5 – Exemple d'un graphe dans une base de données orientée graphe
Source : https ://neo4j.com/developer/graph-database/

2.4.2.1 Neo4j

Neo4J est une base de données orientée graphe native écrite en Java et créer par l'entreprise portant le même nom. Il existe deux version de Neo4J, une version communautaire qui est open source sous la licence GPL3 et une version entreprise qui elle n'est pas open source. La solution communautaire manque de module de sécurité et ne propose de solution de mise à l'échelle.

Neo4J propose une longue liste de langage compatible, .Net, Clojure, Elixir, Go, Groovy, Haskell, Java, JavaScript, Perl, PHP, Python, Ruby et Scala.

Un atout de Neo4J est que c'est une solution native, cela permet une installation rapide et aucune dépendance nécessaire. Neo4J propose une interface web intégré pour la visualisation des graphes ainsi que pour l"écriture de requête avec leur propre langage nommé Cypher. Voici un exemple de requête utilisant le langage Cypher :

```
(p:Person {name: "Jennifer"})-[rel:LIKES]->(g:Technology {type: "Graphs"})
```

Cette requête permet de récupérer la relation qui représente le fait que Jennifer aime la technologie.

Neo4J propose une mise à l'échelle pour la lecture ce qui la rend très rapide, malheureusement l'écriture n'est pas mise à l'échelle. Les performances en écriture restent très performantes mais l'insertion d'énormément de données sur une longue période de temps peut s'avérer assez lente.

2.4.2.2 JanusGraph

JanusGraph est une solution open source sous la licence Apache 2 écrit en Java.

Contrairement à Neo4J, la liste de langage supportés par Janusgraph est beaucoup moins importante. Il ne supporte que le Java, Python et Clojure.

JanusGraph n'est pas une solution native, il s'appuie sur Hbase (voir partie 2.4.6.1) ou Cassandra (voir partie 2.4.6.2) pour le stockage des données et pour l'indexation il est fortement recommandé d'utiliser ElasticSearch (voir partie 2.4.4.1) ou Solr (voir partie 2.4.4.2) qui sont des moteurs d'indexation. Cela implique donc qu'il est nécessaire d'avoir à disposition ces solutions et de les maitriser afin de permettre leurs communication.

JanusGraph ne possède pas d'interface web, mais comme il s'appuie sur une solution de requêtage open source, cela lui permet de bénéficier d'interface web déjà développées. L'interface web la plus intéressante est Linkurious.

Le langage de requêtage utilisé par JanusGraph est Gremlin, il est aussi facile à utiliser que Cypher mais les requête ne sont pas formées de la même manière. Voici un exemple de requête avec le langage Gremlin :

```
g.V().has('name', 'hercules').out('father').out('father').values('name')
```

Cette requête permet de récupérer le nom du grand père du nœud ayant comme nom hercules.

JanusGraph propose lui aussi une mise à l'échelle pour la lecture des données et avec des performances similaire à son concurrent. Mais il ne s'arrête pas là, grâce à son utilisation de brique extérieures (HBase ou Cassandra), il propose aussi une mise à l'échelle pour l'écriture des données.

Conclusion

Nous pouvons constater que sur les deux solutions que nous avons étudier, la différence majeure réside sur le fait que Neo4J est une solution native tandis que JanusGraph s'appuie sur des technologies déjà existante.

2.4.3 Base de données clés/valeurs

Les bases de données clés / valeurs se repose sur une structure très simple, qui est probablement la structure de données la plus simple. Le principe repose sur le fait qu'on ne peut y stocker uniquement des paires de clés et de valeurs. La récupération des valeurs s'effectue via une clé connue. Ce système de stockage ressemble au HashMap que l'on peut utiliser dans différents langages. Au delà de ce principe de paires, il n'y aucune structure des données, cela permet de pouvoir stocker n'importe quelle donnée sans avoir à définir un schéma à respecter et donc de réduire l'espace nécessaire.

Ce système de stockage étant trop simple pour répondre aux besoins d'application complexes trouvent néanmoins son intérêt dans la cadre de système embarqués ou bien si il y a un besoin de traitement de données à très haute performance.

2.4.3.1 Redis

Redis est développé par Redis Lab et écrit en C, il est open source sous la licence BSD.

Redis propose un support d'un très grand nombre de langage de programmation, il en propose plus de 20 dont par exemple Java, Python, C++ et Go.

Redis opère principalement sur la mémoire RAM pour garantir la vitesse la plus élevée possible. Il propose tout de même une solution de "persistance" via des snapshots effectué régulièrement. Les snapshots sont des sauvegarde de ce qui est stocké sur la RAM à un instant T.

Redis propose également un cluster, permettant de répartir la charge sur différents nœuds. ainsi qu'une réplication des données.

2.4.3.2 RocksDB

RocksDB est une solution open source sous les licences GPLv2 et Apache 2, il est écrit en C++ et développé par Facebook.

Contrairement à Redis, RocksDB supporte beaucoup moins de langage, il ne propose le support que pour Java et C++.

A l'opposé de Redis, RocksDB opère principalement sur disque, ce qui permet de rendre les données persistante à tout moment et de pouvoir interagir avec des données plus volumineuses. Afin de palier le manque de vitesse par rapport à son concurent qui utilise la RAM, RocksDB profite du stockage flash apporté par les nouveaux SSD proposant des débits de plus en plus rapide.

Tandis que Redis est une solution accessible par différente applications sur le réseau, RocksDB lui est une solution embarqué sur un appareil, il est accessible uniquement sur la machine sur laquelle il est installé. Cela limite donc les possibilités de mise à l'échelle mais permet de réduire les interactions sur le stockage.

Conclusion

Nous pouvons constater que ces deux solutions de stockage basé sur le modèle clé/valeur sont complètement opposé. Cela va nous permettre de facilement dégager des critères pour définir les situations correspondants le mieux à chacun de ces outils.

2.4.4 Moteur d'indexation

les moteurs d'indexation sont une catégorie de bases de données qui en général ne servent de stockage permanent, mais d'outil intermédiaire pour la visualisation et l'analyse des données. Le principe de cet solution est d'indexer toutes les données afin de permettre des requêtes beaucoup plus rapide, spécialement pour des opérations de recherche. Il est en effet beaucoup plus rapide de rechercher une donnée

par avec un catalogue qu'en parcourant toutes les données stockées. Cela permet de faire des recherches plus avancées sur les données tout en pouvant facilement mettre à l'échelle l'exécution des recherches et le stockage des données. Afin de stocker des données dans ce genre de base de données, il suffit de créer un index pour un jeu de données en définissant le type de nos champs. Une fois cela fait, au moment d'injection des données dans la base de données, il suffit de spécifier l'index à utiliser. Ensuite les données seront indexées en fonction des paramètres présent dans l'index, cette opération peut prendre un peu de temps, mais une fois effectué, les recherches seront très rapide.

2.4.4.1 Elasticsearch

Elasticsearch est une solution open source sous la licence Apache 2 écrit en Java par l'entreprise Elastic.

Le moteur de recherche utilisé par Elasticsearch est Apache Lucene.

Elasticsearch est une solution native, c'est à dire qu'il ne nécessite aucune installation de dépendances et que tous ses composants ont été conçus dans le but de créer un moteur d'indexation. Elasticsearch possède un mécanisme de mise à l'échelle. De par son indexation "strict", Elasticsearch permet un traitement de données structurés en temps réel. Il propose aussi une intégration de module d'apprentissage automatique permettant la détection d'anomalies en temps réel.

Elasticsearch fait parti de la suite elastic qui propose aussi un outil d'analyse et de visualisation des données nommé Kibana (Voir partie 2.6.1). Kibana et Elasticsearch sont entièrement compatible, plus précisément Kibana s'intègre uniquement avec Elasticsearch.

2.4.4.2 Apache Solr

Apache Solr est une solution open source sous la licence Apache 2 écrit en Java par l'entreprise Apache.

Solr est très similaire à Elasticsearch en terme de performances malgré une architecture vraiment différente. En effet, Solr s'appuie sur plusieurs dépendances afin de proposer une solution complète contrairement à Elasticsearch qui lui ne nécessite aucune brique extérieure. Le seul point commun avec Elasticsearch est que Solr aussi utilise Apache Lucene. Pour le stockage des données l'utilisation d'Hadoop HDFS (Voir partie 2.4.7.1) est requise. Solr lui aussi dispose d'une solution de mise à l'échelle, mais cela nécessite d'installer Zookeeper. Solr fournit aussi un module d'apprentissage automatique, mais celui-ci est orienté pour la compréhension du langage naturel. Le mode de fonctionnement de l'indexation de Solr lui permet de traiter des données moins structuré qu'Elasticsearch, mais il supporte moins bien les traitements en temps réel.

Apache Solr dispose lui aussi de son outil de visualisation et d'analyse de données dédié, il s'agit de Banana (Voir partie 2.6.2).

Conclusion

Malgré une base commune entre ces deux solutions, leurs utilisation diffèrent de par les différences dans leur architecture.

2.4.5 Base de données orientée documents

Les bases de données orientées documents sont une forme avancé du stockage clé/valeur. Ces bases de données ont la particularité d'avoir une organisation des données sans schéma. Cela implique que :

- Les enregistrements peuvent avoir des colonnes différentes.
- Le type des valeurs associés a chaque colonne peut être différent.
- Les colonnes peuvent avoir plus d'une valeur (tableaux).
- Les enregistrements peuvent avoir une structure imbriquée.

Les bases de données orientée documents utilisent le format JSON ou XML pour le stockage des données. Le format JSON étant moins lourd est celui qui est de plus en plus privilégié pour ce genre de base de données.

2.4.5.1 CouchDB

2.4.5.2 CouchBase

2.4.5.3 MongoDB

2.4.6 Base de données à grande colonne

Une base de données à grande colonne permet de stocker des enregistrements avec la capacité de contenir un très grand nombre de colonnes dynamiques. Les noms des colonnes ainsi que les clés des enregistrement n'étant pas fixés, un enregistrement peut contenir énormément de colonnes. De part ce mode de fonctionnement, les bases de données à grande colonne sont souvent considérées comme des bases de données clé/valeur bi dimensionnels.

2.4.6.1 HBase

Disponibilité, need HDFS, meilleur lecture

La figure 2.6 représente un exemple de la manière dont sont stocker les données dans HBase.

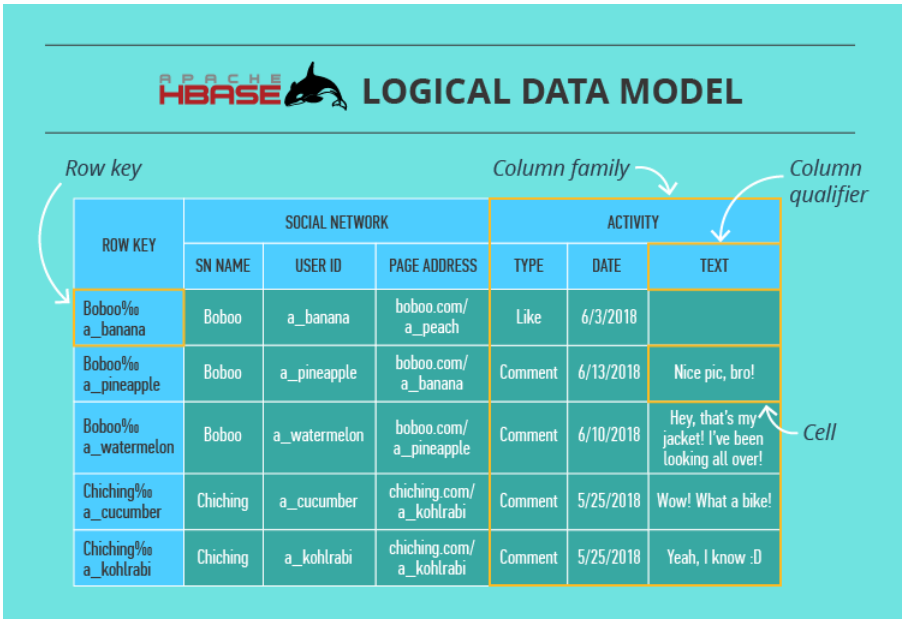

Figure 2.6 – Schéma du principe de stockage de cassandra
Source : http ://www.alliedc.com/apache-cassandra-database-4-problems-that-cassandra-developers-administrators-face/

2.4.6.2 Cassandra

Consistent, meilleur en écriture, solution native

La figure 2.7 représente un exemple de la manière dont sont stocker les données dans Cassandra.

2.4.7 Système de fichiers

2.4.7.1 Hadoop HDFS

2.5 Requêtage des données

Afin de permettre l'analyse et la visualisation des données, il est nécessaire de requêter les données stockées dans des bases de données. Pour cela il y a deux possibilités. La première solution est d'utiliser les outils de requêtage fournit par les bases de données, et la seconde est d'utiliser des librairies externes permettant d'uniformiser le requêtage entre les différentes bases de données.

Aujourd'hui, les solutions de stockage propose des solutions de requêtage qui essaient d'être le plus simple possible tout en conservant des performances suffisante. L'utilisation de librairie externe est surtout utile si vous ne souhaitez pas apprendre les requêtes pour chacune des bases de données que vous utilisez. Étant donnée que ces solutions de requêtage ne sont pas primordiale et que ce mémoire traite déjà de

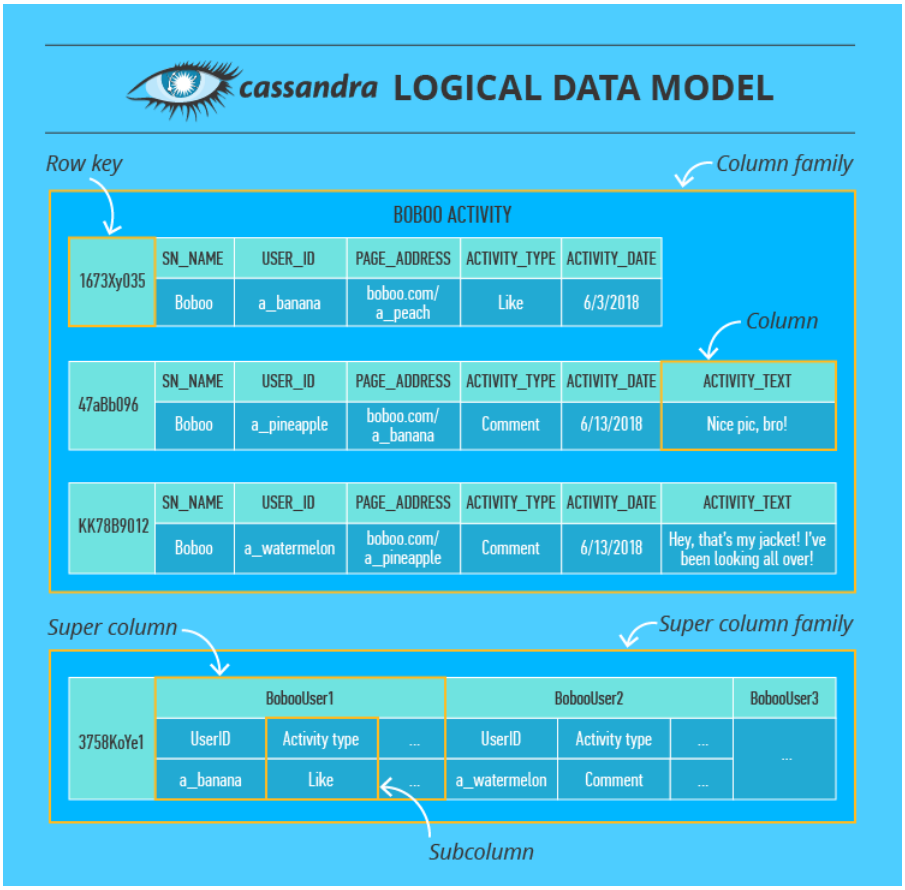

Figure 2.7 – Schéma du principe de stockage de cassandra
Source : http ://www.alliedc.com/apache-cassandra-database-4-problems-that-cassandra-developers-administrators-face/

beaucoup de technologies, nous n'allons pas nous intéresser à ces solutions. je voulais juste préciser que de telles solutions existait, et je vous laisse vous renseigner si cela vous intéresse.

2.6 Visualisation et Analyse des données

2.6.1 Kibana

Comme Banana mais pour ES.

2.6.2 Banana

Comme Kibana mais pour Solr

2.6.3 Grafana

Se plug partout. Spécialisé dans le monitoring.

2.6.4 Tableau

2.6.5 Click

2.7 Orchestration

Les solutions Big Data ont besoin d'un moyen de programmer l'exécution de certaines tâches, par exemple l'exécution de l'ingestion des données d'une source de données ou bien la création d'un rapport sur un outil d'analyse. C'est pour cela que l'on utilise un orchestrateur.

2.7.1 Apache Oozie

Apache Oozie est une solution open source sous la licence Apache 2 écrit en Java et développé par Apache.

Apache Oozie est conçu pour fonctionner avec les outils Hadoop (MapReduce et HDFS par exemple), mais il dispose aussi de fonctionnalités pour programmer l'exécution de script Shell ou Java.

Oozie permet de définir une chaîne d'actions à effectuer à un moment précis, il permet aussi d'effectuer une autre action si jamais une des tâches de la chaîne échoue. Afin de paramétrer cette chaîne d'action, Oozie propose aux utilisateurs une interface graphique pour faciliter sa configuration.

Permet de faire des chaînes d'actions. Interface graphique.

2.7.2 Cron

Une solution beaucoup plus basique et qui existe depuis longtemps, est l'utilisation de la cron tab sous Linux. Cette solution propose uniquement de lancer une tâche tout les x temps, mais avec l'utilisation des messages broker, il n'est pas forcément nécessaire de lancer une suite de tâche. En effet, si les messages brokers permettent de déclencher des évènements afin de récupérer les données, les envoyer au logiciel de traitement de données puis les stocker dans la base de données sélectionnée, la seule tâche restante est la génération d'un rapport. L'utilisation de la cron tab est suffisante pour ce genre de tâche et ne nécessite aucune installation, car elle est installée avec Linux.

2.7.2.1 Conclusion

Nous pouvons constater que la partie orchestration ne nécessite pas d'avoir une solution très avancée, même des solution basiques suffisent. Mais si l'on veut pouvoir facilement configurer l'exécution de nos tâches des solutions graphique sont tout de même à notre disposition.

Chapitre 3

Critères d'analyse

Maintenant que nous sommes familiers avec les architectures Big Data ainsi qu'avec les différentes solutions logicielles, nous allons pouvoir définir des critères afin de sélectionner l'architecture qui correspond à notre besoin. Certains critères ont déjà été dégagés 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 solutions logicielles à utiliser au sein de cette architecture. En plus des différentes informations que nous avons exposées jusqu'ici, nous aurons besoin de prendre en compte le cas d'utilisation nécessitant le déploiement d'une architecture Big Data. Il est fortement possible que plusieurs critères correspondent au cas d'utilisation que vous souhaiter mettre en œuvre, et qu'ils sonnent des solutions différentes. Dans ce cas, ce sera à vous par rapport à vos ressources de définir lequel de ces critères est le plus important pour faire votre choix. 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.

De ce que j'ai pu voir jusqu'à aujourd'hui, les entreprise ne s'intéressent pas aux architectures existantes pour le choix de leur architecture. En effet, elle ne vont pas regarder quelles sont les différences entre l'architecture Kappa et l'architecture Lambda afin de savoir vers laquelle ils vont s'orienter. Les entreprises vont s'intéresser directement aux solutions logicielles, et vont de ce fait concevoir eux même leur propre architecture. Bien évidemment, l'architecture qui sera issue des choix de l'entreprise ressemblera énormément à l'architecture Kappa ou Lambda étant donné que ces deux architectures représente les deux moyens existant actuellement pour répondre aux besoins du Big Data.

Afin de choisir leurs solutions logicielles, les entreprises vont définir leur cas d'utilisation, c'est à dire ce pourquoi ils veulent constituer une architecture Big Data. La définition du cas d'utilisation de l'entreprise passe des sources de données qu'elles vont exploiter jusqu'au visualisation et analyse qu'elles souhaitent réaliser. Les entreprises vont ensuite représenter sous forme de schéma le flux que les données vont parcourir avant d'être stockées pour l'analyse, et définir les types de traitements qu'elle devront

subir. Cela permettra de sélectionner les outils de traitements et de stockage adaptés. Une fois leur cas d'utilisation et la définition de leur flux de données bien défini, ils vont pouvoir passer à la sélection des solutions logicielles. J'ai pu constater trois manières différentes que les entreprises utilisent pour effectuer leur choix.

La solution la plus simple pour une entreprise, est de fournir les informations de son cas d'utilisation ainsi que son flux de données à une autre entreprise spécialisé qui propose des architectures complètes. Une entreprise très connue dans ce domaine est HortonWorks. Ces entreprises se spécialise dans certains cas d'utilisation et vont sélectionner les solutions logicielles adaptées à ces cas d'utilisation. Faire appel à des entreprises spécialisées est la solution la plus simple, surtout lorsque l'entreprise faisant appel à ce service ne possède pas les connaissance en interne pour le choix et l'installation d'une architecture Big Data. Par contre ce genre de solution peut être très couteuse, spécialement si c'est l'entreprise externe qui réalise l'installation et la maintenance de l'architecture.

La seconde solution utilisée par les entreprises quand elles ne souhaitent pas faire appel à une entreprise externe, est de choisir leurs solutions logicielles en installant celles utilisé par les entreprises externes. Par exemple imaginons que les entreprises spécialisées dans les architectures Big Data utilisent souvent HBase pour le stockage des données, alors l'entreprise va elle aussi utilisé HBase pour le stockage des données. Cette solution peut s'avérer payante dans certains cas, si le cas d'utilisation de l'entreprise correspond à l'utilisation de cette technologie. Par contre si ce n'est pas le cas, toute l'architecture constitué par l'entreprise ne sera pas optimale et parfois même pas du tout adaptée.

La dernière solution, est celle que j'ai effectué tout au long de ce mémoire, c'est à dire se renseigner sur chaque solutions logicielles pouvant répondre à leurs besoins, regarder leurs fonctionnement afin de sélectionner les solutions les plus pertinentes par rapport à leur cas d'utilisation. Une fois avoir réduit le nombre de solution à leur disposition, généralement les entreprises vont essayer de mettre en place les solutions logicielles qu'elles ont sélectionnées afin de faire des benchmarks pour leur permettre de faire le choix finale.

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 lors de l'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.

Pour le dernier critère, à savoir le stockage permanent des données batch avant leur traitement, si l'on suit scrupuleusement les architectures Lambda et Kappa, la solution choisie devrait être Lambda. Mais si notre cas d'utilisation correspond en tout point à une architecture Kappa et que l'on souhaite juste avoir une partir de stockage des

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

données brutes, il est plus intéressant de conserver l'architecture Kappa qui est moins complexes et juste rajouter une base de données capable de stocker ces valeurs.

3.3 Critères pour le choix des solutions logicielles

Nous allons maintenant nous attaquer aux critères permettant de départager les solutions logicielles que nous avons parcouru au long de ce mémoire. Pour cela nous allons proceder comme pour la partie précédente, c'est à dire définir des critère pour chaque catégorie de l'architecture. Bien évidemment, il sera préciser pour les parties non communes aux architectures Lambda et Kappa si cette catégorie ou tel outil est fait pour l'architecture Lambda ou Kappa pour que les choix restent cohérents.

3.3.1 Ingestion des données

Pour la partie s'occupant de l'ingestion des données, nous avons pu constater qu'un premier choix s'offrait à nous. Soit une solution complète (ETL/ELT), ou bien la réalisation d'un programme suivant l'architecture réactive. Nous allons présenter toujours sous la forme de tableau les critères permettant de définir quand utiliser ces solutions.

Comme on peut le constater, les critères de choix entre l'utilisation d'une solution complète et d'un programme personnalisé sont surtout par rapport à un problème de connaissance et de complexité de maintenance. En effet, si vous devez gérer l'extraction d'un grand nombre de sources de données, il sera plus facile de les gérer via un ETL/ELT qui mets à votre disposition une interface graphique. Interface graphique qui permet même à des personnes n'ayant pas de connaissances avancés dans la programmation de pouvoir mettre en place l'extraction des données. Tandis que le développement d'un programme personnalisé, est intéressant dans le cas ou vous avez les connaissances necessaire pour le mettre en place et que vous avez besoin de performances élevés ainsi qu'une consommation de ressources moins élevé qu'un

Critère	Solution
Manque de connaissances pour la réalisation de programmes personnalisés	ETL/ELT
Nécessité d'extraire de nombreuses sources de données	ETL/ELT
Peu de sources de données	Programme personnalisé
Nécessite d'avoir des performances élevés	Programme personnalisé

Table 3.2 – Table des critères pour le choix entre une solution complète ou d'un programme personnalisé pour l'ingestion des données.

ETL/ELT. Cela permet d'avoir d'avoir un plus petit nombre de machine et/ou moins de machine ce qui engendre un coût en matériel amoindri.

3.3.2 Message Broker

Nous avons vu trois solutions pouvant être utilisées comme agent de message, à savoir Kafka, ActiveMQ et RabbitMQ. Voici le tableau des critères permettant de les départager.

Critère	Solution
Garantir la consommation d'un message par un seul consommateur	RabbitMQ
Nécessité d'ingérer rapidement une grande quantité de messages	Kafka
Ordre des messages primordial	Kafka
Nécessité de conserver les message à plus au moins long terme	Kafka
Utilisation de protocoles spécifiques (MQTT, AMQP, ...)	RabbitMQ / ActiveMQ
Règles de routage des messages complexe	RabbitMQ / ActiveMQ

Table 3.3 – Table des critères pour le choix du logicielle d'agent de messages

Comme on peut le constater, Kafka se démarque complètement de ses deux concurrents avec des cas d'utilisation bien précis. Par contre ActiveMQ et RabbitMQ ne se démarquent pas du tout entre eux. En effet à part leur méthode de communication, ils répondent aux même besoins. Afin de les départager la solution serai de réaliser un benchmarks par rapport à votre cas d'utilisation.

3.3.3 Traitement des données

Comme on l'a vue précédemment, il existe deux types de traitement de données, le traitement par lots et le traitement en temps réel. Pour choisir lequel de ces deux traitements correspond à votre cas d'utilisation, il suffit de voir quel architecture correspondait à vos critères dans la partie précédente. Si c'était l'architecture Kappa, vous aurez uniquement besoin d'une solution de traitement en temps réel, dans le cas contraire vous aurez besoin d'intégrer les deux types de traitements de données.

3.3.3.1 Traitement par lots

3.3.3.2 Traitement en temps réel

3.3.4 Stockage des données

Comme nous avons pu le constater, il existe de nombreuses base données dans le comaine du Big Data, chacune ayant une manière différente de stocker les données. Le choix du type de base de données à utiliser va se faire par rapport aux formats des données qu'on l'on souhaite stocker et/ou l'utilisation que l'on souhaite faire de ces données.

3.3.4.1 Base de données de séries temporelles

3.3.4.2 Base de données orientée graphe

3.3.4.3 Base de données clés/valeurs

3.3.4.4 Moteur d'indexation

3.3.4.5 Base de données documents

3.3.4.6 Base de données à grande colonne

3.3.5 Orchestration

3.3.6 Visualisation et Analyse des données

3.4 Pour aller plus loin

Dans ce mémoire nous nous sommes intéresser à une très grande partie du Big Data, nous n'avons donc pas pu étudier de manière approfondit chacune des parties constituant une architecture Big Data. De ce fait, il y a moyen d'aller plus loin dans l'étude des solutions à notre disposition.

La première manière de permettre un choix plus précis de solution, serait de prendre en compte plus de solution logicielles pour chaque catégorie de l'architecture. En effet, tout au long de ce mémoire, pour chaque catégorie je n'ai traiter que deux à trois solutions, en essayant de sélectionner celles qui étaient les plus matures et les plus intéressante. Mais en prenant en compte plus de solutions logicielles, il serait possible de rencontrer des solution logicielles vraiment différentes et couvrant de manière native encore plus de cas d'utilisation. Malheureusement, ce mémoire n'étant pas centrer sur une une seule partie du Big Data, il n'était pas possible de traiter autant de possibilités.

La deuxième manière d'aller plus loin dans cet objectif de choisir l'architecture Big Data, serait de réaliser des benchmarks pour chaque solution logicielle présenté. Le fait de s'intéresser au fonctionnement de chacune des solutions étudiés n'est pas forcément suffisant pour garantir que c'est le meilleur choix par rapport à un cas d'utilisation. Afin d'avoir des tests complets, il faudrait utiliser plusieurs sources de données différentes pour les benchmarks, cela permettrait de se rendre compte facilement des différences de performances des outils en fonction des types des sources de données. Plus spécifiquement pour la partie traitement des données, il aura fallut faire des benchmarks pour plusieurs types de traitements de données afin de couvrir encore plus de cas d'utilisation et d'assurer la solution la plus optimale possible.

Chapitre 4

Comment évaluer l'architecture résultant de l'application des critères

Maintenant que nous avons réussi à définir des critères de choix pour notre architecture Big Data, il reste encore une étape pour s'assurer que les critères que nous avons défini sont corrects. Pour cela il va falloir tester notre démarche en s'appuyant sur des cas d'utilisations réels.

4.1 Application des critères définis sur des cas d'utilisations réels

Dans un premier temps, il faut appliquer notre démarche à des cas d'utilisations réels. Pour cela, nous allons détailler les types de sources de données qui correspondent au cas d'utilisation que nous avons choisis, le type de traitement nécessaire sur ces données ainsi que la valeur que l'on veut dégager à l'aide de nos analyses sur ces données. Une fois cela fait, nous allons nous appuyer sur les critères réaliser lors de la partie précédente et lister les solutions logicielles qui ont été retenu. Une fois cette étape terminée, nous allons traiter chaque cas d'utilisation un par un. Pour chacun de ces cas d'utilisation, nous allons mettre en place les solutions logicielles issues de nos critères, ce qui nous permettra de tester notre solution dans des conditions réelles.

4.2 Évaluation des résultats

Une fois la mise en place de notre solution, afin de pouvoir mesurer son efficacité, il faudra réaliser des benchmarks et laisser la solution en place pendant quelques heures. Pendant que les données arriveront dans notre architecture, nous pourrons nous concentrer sur la réalisation des visualisation et des analyses à effectuer sur les données afin de vérifier si la solution sélectionnée nous permet bien de réaliser sans encombre la mise en valeur que le cas d'utilisation nécessitait. En plus des résultats des benchmarks et de la réussite à faire les analyses nécessaire, il faudra aussi évaluer si tout le chemin de la donnée a bien été parcouru sans soucis. Par exemple est-ce qu'il n'a pas fallut faire du développement spécifique qui n'était pas prévu pour permettre la communication entre plusieurs briques, ou bien pour effectuer le traitement des données ou encore pour le stockage des données.

Conclusion

Annexe A

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 :

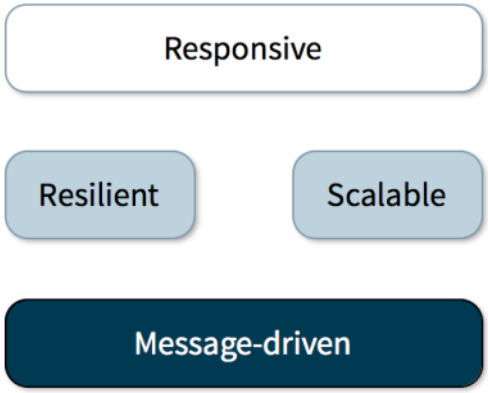

Figure A.1 – Schéma de l'architecture réactive

- 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 peut 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 na 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://streaml.io/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

Table des figures

Liste des tableaux