Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

THESIS / THÈSE

MASTER EN INGÉNIEUR DE GESTION À FINALITÉ SPÉCIALISÉE EN DATA SCIENCE

Stockage et traitement des données

Nollevaux, Augustin

Award date:
2022

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 15. Feb. 2023

Cours de Big Data : Stockage et traitement des données

Augustin Nollevaux

Directeur: Prof. S. FAULKNER

Mémoire présenté
en vue de l'obtention du titre de
Master 120 en ingénieur de gestion, à finalité spécialisée
en data science

ANNEE ACADEMIQUE 2021-2022

Table des matières

1.Introduction 4
1.1. Objectifs du mémoire 4
1.2. Intérêts du Big Data 5
Quelques chiffres 5
Cas d 'étude du Big Data 7
1.3. Origine du Big Data 9
1.4. Caractéristiques du Big Data 13
1.5. Modèles de distribution 13
Single-Server 14
Sharding 14
Réplication 16
1.6. Type de stockage 19
1.7. Overview d'un projet de Data Science 19
2. Bases de données NoSQL 21
Evolution des bases de données 21
Caractéristiques des bases de données NoSQL 23
Avantages et désavantages de NoSQL 25
Consistance des données (ACID vs BASE) : 26
Le Théorème du CAP 28
Extensibilité horizontale et extensibilité verticale 32
3.Base de données orientée clé-valeur 33
3.1. Introduction 33
3.2 Fonctionnement 33
3.3 Utilité et usage 39
3.4 Avantages 39
3.5 Désavantages 40
3.6 Applications 40
3.7 Acteurs principaux 41
4.Base de données orientée colonnes 43
4.1 Introduction 43
4.2 Fonctionnement 44
4.3 Utilité et usage 48
4.4 Les bases de données orientées colonnes sont-elles NoSQL ? 48
4.5 Type de stockage 48
4.6 Avantages 49
4.7 Désavantages 49
4.8 Applications 50
4.9 Acteurs principaux 50
5.Base de données orientée documents 52
5.1 Introduction 52
5.2 Fonctionnement 53
5.3 Utilité et usage 54
5.4 Avantages 55
5.5 Désavantages 55

5.6 Applications......55
5.7 Acteurs principaux......56
6.Base de données orientée graphe......57
5.1 Introduction......57
5.2 Fonctionnement......57
5.3 Utilité et usage......59
5.4 Avantages......60
5.5 Désavantages......60
5.6 Gestion des accès......61
5.7 Applications......61
5.8 Acteurs principaux......61
7. Comparaison et choix de la base de données......63
Comparaison entre les bases de données......63
Choix du type de base de données......64
8. Comparaison et choix de la base de données......66
8.1. Couche des sources de distribution et d'ingestion......67
8.2. Couche de stockage distribué......68
HDFS......69
MapReduce......72
8.3. Couche d'infrastructure Hadoop......73
8.4. Couche de management Hadoop......74
8.5. Couche d'analyse......75
8.6. Couche de visualisation......77
8.7. Couche de sécurité......78
8.8. Couche de monitoring......78
8.9. Cloud Computing......79
Conclusion......**Erreur ! Signet non défini.**

1.Introduction

1.1. Objectifs du cours

L'objectif de ce cours est de fournir une présentation globale du Big Data ainsi que d'expliquer les aspects techniques qui se cachent derrière les différentes approches du Big Data. L'histoire de la gestion des données nous permettra d'expliquer comment et pourquoi nous sommes passé de la business intelligence analytique au Big Data. Furht et Villanustre (2016) ont défini le défi du Big Data comme : « la discipline qui consiste à fournir une architecture matérielle et des systèmes logiciels connexes capables de transformer des données très volumineuses en connaissances utiles ». Il n'y a pas de consensus dans la communauté scientifique à propos du cadre que recouvre la notion de Big Data. Certains (Debortoli, Muller et vom Brocke, 2014) l'associent simplement comme stockage et traitement de données tout en le dissociant des applications telles que le *data mining*, le *deep learning* ou le *machine learning*. D'autres considèrent le Big Data dans son intégralité et en élargissent le spectre en passant du stockage par le traitement à l'analyse et au reporting de l'analyse comme le font Furht et Villanustre (2016). D'après nos différentes recherches, nous sommes en accord avec la vision du Big Data dans son intégralité, bien que ce qui caractérise les plus grandes implications du paradigme Big Data passe par la différence qui existe entre la phase de stockage et celle du traitement des données.
Comme le propose Akerkar (2013), nous divisons le Big Data en 3 parties : 1/ la partie sur les données (stockage et traitement), 2/ la partie analytique et 3/ la partie management des données Big Data.

Ces 3 parties peuvent être traduites en 3 challenges (Akerkar, 2013) :

1. La nécessité d'une architecture capable de gérer les caractéristiques particulières du big data (les 5V), de gérer les compromis entre les différents besoins non-fonctionnels d'un système de gestion de données (consistance, disponibilité, extensibilité) et, enfin, capable de supporter un système distribué.
2. L'utilisation de nouvelles techniques d'analyse et de traitement des données de type Big Data (*machine learning*, *deep learning*, *data mining*, *text mining*,…). Ce second challenge inclut également la visualisation des résultats obtenus par l'analyse et le traitement des données.
3. Le management des données est la partie permettant d'encadrer les 2 premiers challenges : la sécurité des données, la sécurité de l'infrastructure (des sources de données jusqu'à la visualisation des données), la confidentialité des données, la protection des données, la gestion des accès, le respect des lois liées à la manipulation des données (par exemple le RGPD en Europe), les questions éthiques (par exemple, discrimination liée à un algorithme), etc.

Dans le cadre de ce cours, nous répondrons principalement et en détails aux questions posées par le premier challenge et nous laisserons le détail des deux seconds challenges à d'autres cours de l'Université de Namur (le cours de Machine Learning notamment pour le deuxième challenge et le cours de sécurité informatique, entre autres, pour une partie du troisième challenge) mais également à de futurs travaux qui pourraient venir compléter cette présentation globale du Big Data.

Dans le chapitre 1, correspondant au premier challenge, il s'agira de présenter les nouvelles technologies et techniques qui ont été développées pour y répondre. Les deux principales nouvelles technologies sont les bases de données NoSQL (Chapitre 2) et l'environnement Hadoop qui est une architecture spécifique (Chapitre 3) pour le Big Data et qui a deux grands composants : HDFS et MapReduce.

Pour le second challenge, différentes techniques d'analyse, non-exhaustives, des données seront présentées. Lors de cette présentation, nous irons moins en profondeur et en technicité car ce challenge renvoie à une autre branche de la discipline qui nécessiterait de nombreuses précisions

dont le présent coursco ne peut que faire l'esquisse. Pour le troisième challenge, une briefe présentation sera faite comme pour le second challenge.

1.2. Intérêts du Big Data

Afin de bien saisir l'importance et la place du Big Data dans le monde actuel, quelques chiffres, des uses-cases ainsi que des applications du Big Data sont présentés.

Quelques chiffres

- Le New York Stock Exchange génère plus d'un TeraByte de données chaque jour (Petrov, 2021).
- Chaque minute (Marr, 2018) : plus de 450 000 tweets sont écrits ou encore 500 000 photos sont partagées sur Snapchat.
- Chaque jour : plus de 500 TeraByte de données sont générées sur Facebook et 95 millions de photos et vidéos sont postées sur Instagram.
- La figure 1.1 représente la taille des revenus du marché du Big Data, qui est en constante augmentation et est prévue d'atteindre plus de 100 milliards en 2027 (Mlitz, 2021).
- Kambau et Hasibuan (2017) mettent en valeur la domination de la quantité de données non-structurées, comme nous le voyons sur la figure 1.2, qui représente l'évolution de la quantité de données générées par type de données : structuré (en gris) et non-structuré (en vert).
- Google répond à *40.000 recherches par seconde* et il y a plus de 5 milliards de recherches, tous navigateurs confondus, par jour (Marr, 2018).
- En 2020, le *marché global* du Big Data était d'environ *200 milliards de dollars.*
- *Entre 2010 et 2020, le volume des données numériques a été multiplié par 30*, passant de 2 zettaoctet à 64 zettaoctet. Il est estimé que, pour 2025, le volume devrait dépasser les 180 zettaoctet. Dans les faits, seul un petit pourcentage de ces données nouvellement créées est actuellement conservé puisque seulement 2 % des données produites en 2020 ont été sauvegardées en 2021 (Gaudiaut, 2021).

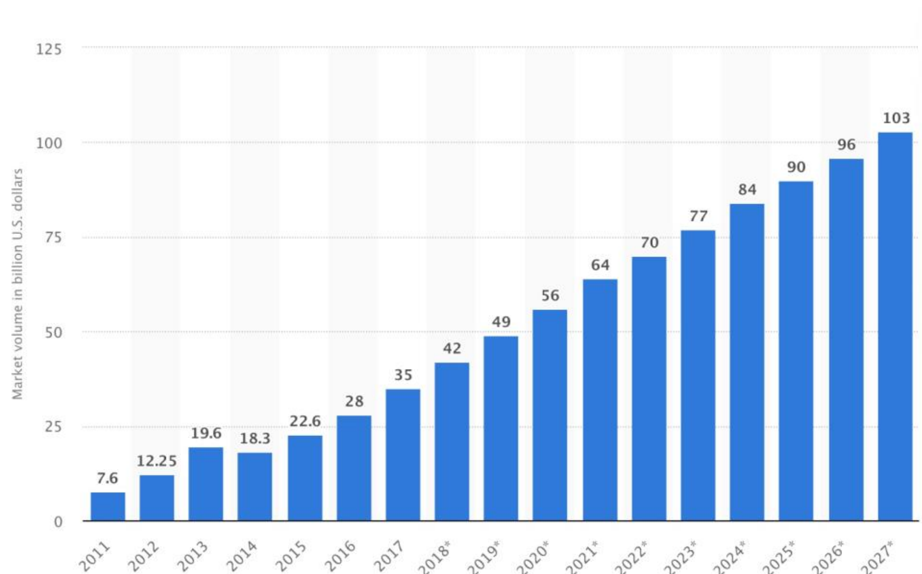


Figure 1.1 : Histogramme de taille des revenus du marché du Big Data dans le monde entier depuis 2011 à 2017 et en prévision de 2018 à 2027.

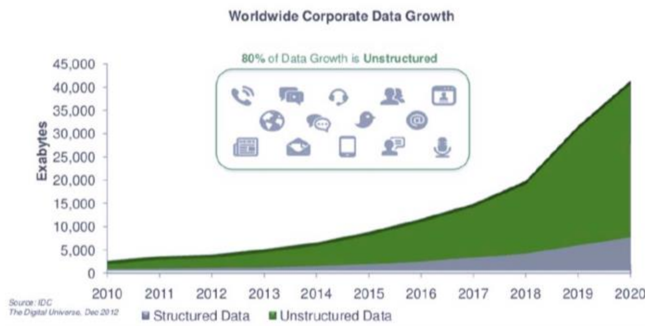


Figure 1.2: Graphique à aire de l'évolution de la quantité de données générée par type de données.

Cette explosion de données détient une double origine :

1/ La première origine est à situer dans le progrès technique des capacités (de stockage, de processeurs, de latence, etc.) qui peut être compris comme *une augmentation de la capacité de l'hardware* couplée à une *diminution des coûts* de production. Il est commun de faire référence à la Loi de Moore (Wikipedia contributors, 2021) (figure 1.3) pour expliquer que, avec le temps qui avance, les ordinateurs deviennent de plus en plus petits et de plus en plus rapides mais également de moins en moins chers.

Techniquement parlant, la loi de Moore prévoit que le nombre de transistors que l'on peut mettre sur un microprocesseur double tous les deux ans. Cependant, depuis 2015-2017, la loi de Moore ne s'applique plus car les transistors vont atteindre (ou ont presque atteint) leur taille minimale (quelques nanomètres). Les coûts de calcul des processeurs a également diminué comme sur la figure 1.4 où le nombre de watts pour effectuer 1 million de requêtes par seconde est passé de 1W en 1975 à 0.00001 W en 2015.

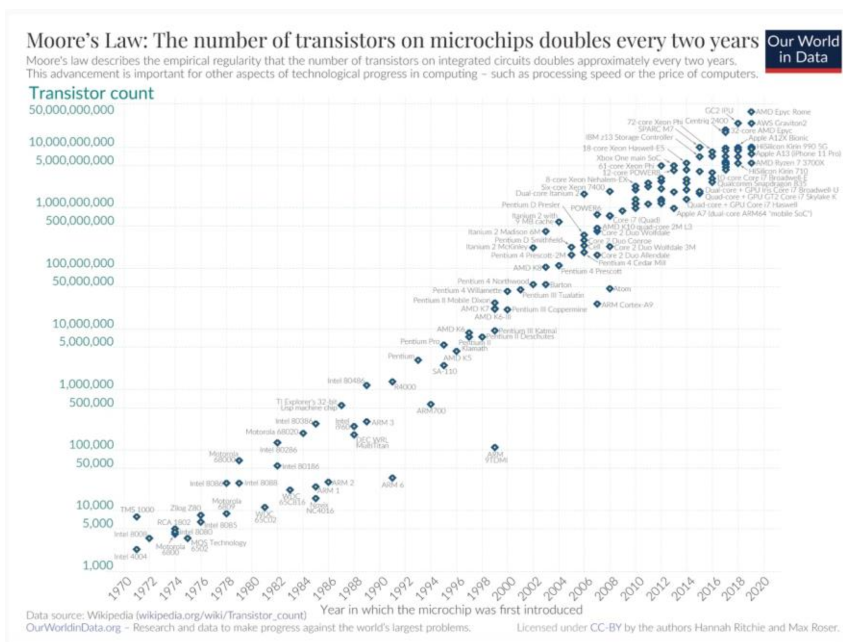


Figure 1.3 : Graphique de l'évolution du nombre de transistor par microship entre 1970 et 2020 (Roser M., 2013).

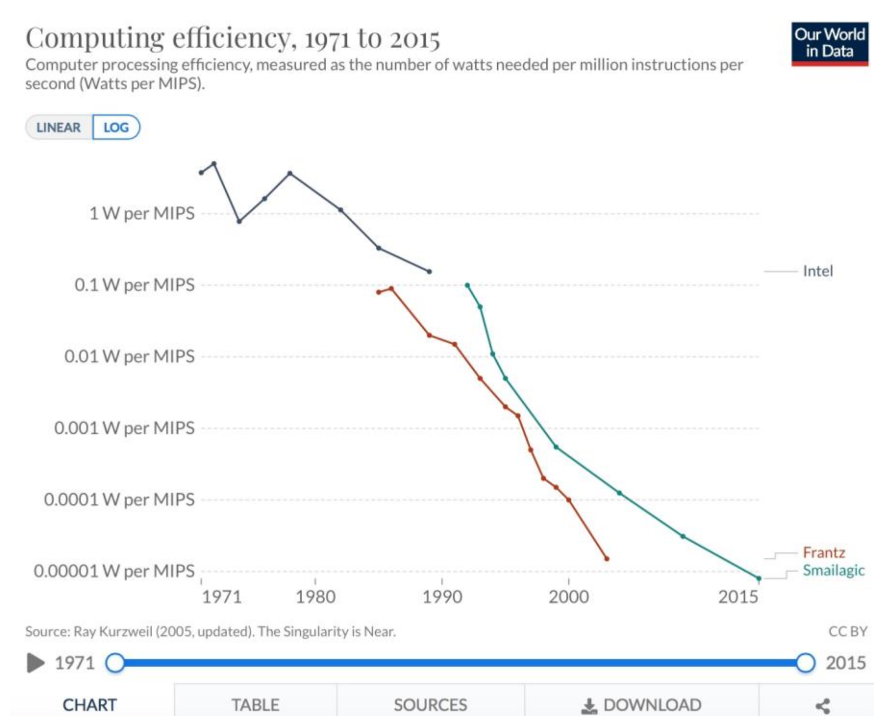


Figure 1.4 : Graphique de l'évolution de la consommation de Watts nécessaire pour un million d'instruction par seconde (Watts/MIPS) entre 1971 et 2015 ((Roser M., 2013).

2/ La seconde origine est l'apparition de technologies comme Internet et les réseaux sans fil au sein desquels des applications tels que les réseaux sociaux et le *Cloud Computing* ou encore l'IoT (Internet of Things) se sont développées.
L'évolution technologique des appareils photos et vidéos expliquent également l'explosion des données sur les réseaux sociaux ainsi que l'utilisation de l'IoT. Il y a une évolution de la qualité des photos pour une diminution du coût des appareils (Roser M., 2013).

Cas d 'étude du Big Data

Le champ d'application du Big Data est très large et possède des cas d'application pour la plupart des industries. Ci-dessous, nous pouvons retrouver deux exemples d'application en droit et en médecine :

- **Les applications de droit (Law Industry)**: utilisation du Big Data pour l'accélération des procédures judiciaires (Muskan, 2021).
Les données du secteur juridique possèdent 3 caractéristiques typiques du big data : elles sont en très grandes quantités, elles sont de différents types et différents formats et elles sont actualisées constamment lors de procédures judiciaires. L'utilisation adéquate des bonnes données au bon moment est essentielle pour rendre le processus judiciaire plus rapide. La gestion de ces données par le paradigme Big Data permet d'optimiser l'utilisation de ces données et facilite l'accès aux références et aux preuves lors de procédures judiciaires (en audience ou en préparation d'audience), rendant ainsi le processus total moins couteux.
- **Les applications en médecine (HealthCare Industry)** (Durcevic, 2020). Pour cette industrie, les applications du Big Data sont très nombreuses, et l'exemple le plus utilisé est la prédiction de cancers à partir des caractéristiques des patients. Pour cette raison, nous présentons deux

applications moins communément utilisées.

<u>- Prédiction du nombre de patients pour optimiser la gestion du personnel.</u> Trop de personnel pour une période donnée signifierait des surcoûts inutiles mais trop peu de personnel signifierait des services médiocres aux conséquences fatales. Cet exemple peut s'appliquer à tous les secteurs nécessitant d'assigner du personnel pour une période donnée.
<u>- Alerte en temps réel.</u> Par exemple, si la tension artérielle d'un patient augmente de façon alarmante, le système enverra une alerte en temps réel au médecin qui pourra alors prendre des mesures afin de réguler la tension artérielle du patient, que ce soit à distance ou sur place.

Afin de mieux saisir l'importance du Big Data, il nous a paru judicieux de montrer un exemple concret pour lequel le développement et l'application du Big Data a, d'une part, été un succès, et, d'autre part, un relatif échec.

Il d'agit de la présidentielle des États-Unis en 2012 entre Obama et Romney qui avaient chacun prévu d'utiliser le Big Data afin de les aider durant leurs élections. Pour le second, le mauvais développement des outils Big Data n'a apporté aucune valeur et s'est révélé être une dépense assez inutile alors que, pour le premier, cet usage fut une vraie réussite.

La présidentielle aux Etats-Unis de 2012 est un bon *use-case* qui montre comment deux utilisations de Big Data peuvent bien fonctionner ou mal fonctionner (Whet M., 2012).
En effet, en 2012, Barack Obama a lancé l'opération « Narwal » d'Obama qui voulait rassembler toutes les données en relation avec l'élection (collecteurs de fonds, agents de terrain, réseaux sociaux) dans une seule base de données. « […], l'équipe d'Obama a introduit ces données dans des modèles prédictifs avancés qui ont permis aux responsables de cibler les électeurs avec des messages spécialisés. Par exemple, la campagne d'Obama a pu identifier les partisans de Planned Parenthood vivant dans des codes postaux majoritairement chrétiens et leur envoyer des e-mails sur la "guerre contre les femmes" de Romney sans risquer de s'aliéner les démocrates pro-vie de Reagan qui pourraient reculer devant un tel langage. » (Thiessen, M. 2012).

De plus, l'équipe d'Obama a également eu recours au Big data pour étudier les tendances électorales : là où les instituts de sondage classiques utilisaient un échantillon de 1000 personnes, ils ont, eux, développé un profil de données de sondage sur 29 000 personnes et ont ainsi pu suivre l'évolution des tendances électorales en réaction avec les actions mises en places (débats, messages personnalisés, etc.).

Face à cela, le candidat républicain Romney avait lancé le projet « Orca » - pour l'anecdote, Orca (orque) est le prédateur de Narwal (baleine) - qui était une plateforme de Big Data mise en place afin d'en savoir plus sur les électeurs et les tendances de l'élection. Cependant, le développement de cette solution Big Data avait été mal pensé et leur plateforme Orca rencontrait beaucoup de problèmes.

- Due à un problème lié à l'architecture de la plateforme, elle s'effondrait lors de surcharge sur les serveurs : la partie mobile d'Orca était une application Web pris en charge par un seul serveur Web et un seul serveur d'applications. Nous le verrons plus tard, l'architecture du Big data repose justement sur un système distribué qui permet de répartir les charges de travail et éviter les crashs ou les « point-of-failure ».
- Problème de gestion des accès : à cause de problèmes de mots de passes et d'URL, « hundreds (or more) volunteers who couldn't use the app or the backup phone system » (Thiessen, M. 2012).
- Problème de management des données : ils n'avaient jamais testé le système avant le jour de l'élection et de nombreux problèmes sont survenus en son jour de lancement.

Au delà du débat sur l'éthique d'avoir recours à des technologies de manipulation tel que le Big Data pour l'élection du dirigeant d'un Etat, cet exemple nous montre que malgré la volonté commune d'utiliser le Big Data, les deux implémentations ont eu des résultats fort différents et cela souligne l'importance pour tout manager d'avoir une idée plus précise du fonctionnement du Big Data (de l'architecture au management des données).

1.3. Origine du Big Data

Pour comprendre l'histoire du Big Data, il faut démarrer de l'informatique décisionnelle (également appelée Business Intelligence, BI) et suivre son évolution qui a abouti à un nouveau paradigme, appelé Big Data.
Comme l'expliquent Sawant et Shah (2013) dans leur livre, initialement, la manipulation des données était uniquement transactionnelle mais au fur et à mesure que la quantité de données générée grandissait et s'accumulait, l'ensemble des données a commencé à être analysé en utilisant des datawarehouses et data marts.
L'informatique décisionnelle, définie par Chen, Chiang et Storey (2012), est : «l'ensemble des techniques, technologies, systèmes, pratiques, méthodologies et applications qui analysent les données commerciales essentielles pour aider une entreprise à mieux comprendre son activité et son marché et à prendre des décisions commerciales opportunes. Outre les technologies sous-jacentes de traitement des données et d'analyse, la BI&A comprend des pratiques et des méthodologies centrées sur l'entreprise. ». Plus simplement, nous définissons la BI par son objectif qui est la récupération et traitement des données pour en tirer de la connaissance (par exemple, *par la mise en valeur d'un produit qui se vend beaucoup*) ou de la valeur (*par exemple, par de meilleures performances de processus*). La BI traditionnelle stocke les données, que ce soit de manière structurée et normalisée, dans un serveur central unique dans un SGBD relationnel et les données sont analysées une fois les transactions terminées (analyse hors ligne).

La figure 1.5 est un exemple de schéma des étapes clés d'un processus de BI (Jeremy R., 2021) :

On peut voir qu'il y a 4 parties essentielles : les **sources de données**, l'**ETL**, le **Datawarehouse** et le **reporting** (interface utilisateurs et applications).

Le processus de BI commence toujours par les sources de données (venant du CRM, de l'ERP, bases de données internes, des différents départements, etc, et également source externe à l'entreprise). Il s'agit de données structurées, avec une taille de données raisonnable et un traitement des données qui peut se faire en « batch processing » (c'est-à-dire pas nécessairement en temps réel).

Ensuite, la partie ETL (Extract, Transform, Load) va se charger de se connecter aux différentes sources pour extraire, transformer (nettoyer, trier, organiser, etc.) et charger les données dans le Datawarehouse sous forme structurée. L'idée est d'automatiser le processus de l'ETL pour que le processus de BI puisse se faire tout seul.
Le Datawarehouse s'occupe de stocker les données, structurées et formatées par l'ETL, sous formes de tables de faits et de tables de dimensions.
La dernière étape du processus de BI est le reporting et la diffusion des connaissances et valeurs obtenues et créées par le traitement des données. Lors de cette étape, les données sont récupérées du DataWareHouse et sont modélisées pour en faire des outils d'analyses et de prises de décisions (tableau de bords, création d'alerte, schémas, etc.). C'est dans cette partie également que se passe ce que l'on appelle les processus OLAP : online-analytical-processing. La dernière partie de cette étape est le partage et la diffusion de ces outils.
Ilieva, Yankova et Klisarova (2015) présentent, sur la figure 1.6, un exemple de la diffusion de connaissances auprès des utilisateurs : il s'agit d'un tableau de bord détaillant différents aperçus des données de ventes d'une entreprise.

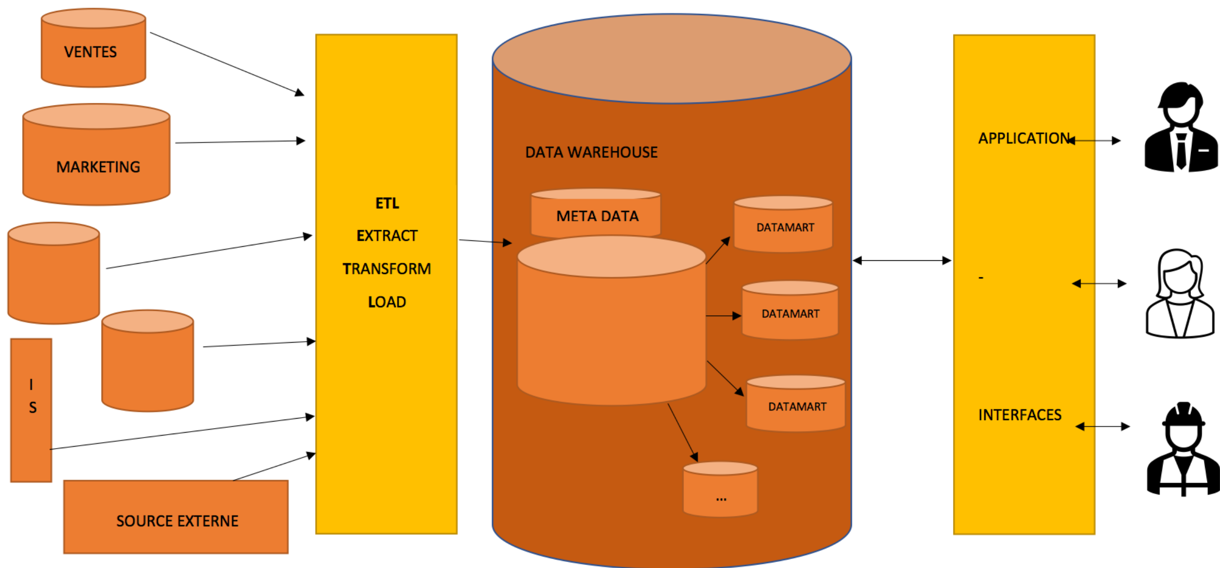


Figure 1.5 : Schéma d'architecture de business intelligence (Linden, 2021).

On peut également définir la BI à l'aide de ses *caractéristiques (Lecomte, 2021)* :

- Le processus BI démarre de besoins d'utilisateurs pour arriver à une solution IT qui répond à ces besoins (**from business to IT**). La structure de la solution informatique (la plateforme IT) va être conceptualisée par rapport à un besoin orienté business.
- Les données stockées sont sur un **serveur central** (Datawarehouse).
- Les données traitées sont **structurées ou semi-structurées** et proviennent la plupart du temps de bases de données *internes* à l'entreprise.

Figure 1.6 : Exemple de tableau de bord à propos des données de ventes d'une entreprise : chiffre d'affaires des ventes par segment, chiffres d'affaires des ventes par produit et part du chiffre d'affaire par segment (Ilieva, Yankova et Klisarova, 2015).

A partir des années 2000, les avancées techniques (augmentation des performances et diminutions des coûts) et technologies (Internet, réseaux sociaux, Cloud Computing…) ont transformé le paradigme des données. On est passé des données *structurées et centralisées* à des données *semi-structurées ou non-structurées*, de données stockées sur un *seul serveur* à des données *distribuées sur une architecture à plusieurs serveurs*, des données sous un *format uniforme* à des données *très variées* (document, fichier audio, images, BSON,…), des données avec une *taille raisonnable* (Gb) à des données *très volumineuses* (Pb ou Eb), des données s'établissant sur une *durée raisonnable*

(heures ou jours ou mois) à des données de *haute vélocité* (presque-en-temps-réel ou en temps réel),…
Avec cette évolution, de nouveaux types de besoins sont apparus :

- Pourvoir stocker des immenses quantités de données. Le très grand volume de données ne pouvait plus être stocké dans un seul serveur parce que la puissance de stockage d'un serveur à des limites physiques (l'extension verticale n'est pas illimitée). Il a fallu faire de l'extension horizontale (scale-out), c'est-à-dire stocker le volume de données sur plusieurs serveurs distribués, pour gérer ces immenses volumes.
- Pouvoir traiter tout type de données. La variété des données a augmenté et il a fallu trouver de nouvelles méthodes pour gérer différents types de données simultanément.
- Pouvoir traiter rapidement les données. Les besoins de vitesse ayant augmentés, il a fallu augmenter en performance pour savoir répondre à ces besoins de vitesse.

Cependant, le paradigme et les outils existants de la BI traditionnelle ne pouvaient pas répondre à ces besoins à cause de leurs structures mêmes (système centralisé, peu flexible, incapacité à gérer des données non-structurées,) (Ali et al., 2019).
Ainsi pour venir en aide à la BI, différents outils *(framework, langages, système de fichier, système de gestion de bases de données, méthodes,…)*, principalement open-source et permettant de répondre à ces besoins, ont vu le jour et se sont développés : **Hadoop** (un framework open-source spécialement conçu pour stocker et analyser des grandes bases de données) ; les **bases de données NoSQL** (BigTable de Google, DynamoDB de Amazon ou Redis open-source,…) ; **MapReduce** (patron d'architecture qui permet le calcul parallèle à travers une distribution de nœuds);… Cette *nouvelle manière de traiter les données* a donné naissance à ce **nouveau paradigme** qu'est le **Big Data**.

Pour expliquer ce qu'est le Big Data, on le présente souvent au travers de ses caractéristiques qui sont les 3V: pour *Volume*, *Variété* et *Vélocité.* Après de nombreuses recherches et lectures et particulièrement grâce aux travaux de Fowler et Sadalage (2013), de Furht et de Villanustre (2016) desquels nous nous sommes inspiré, je définirais les 3V de cette façon :

- **Volume** : cela fait référence à la très *grande quantité de données* qui est stockée et manipulée.
- **Variété** : cela fait référence à la *variété des types de données* qu'une application Big Data peut traiter : données *structurées*, *semi-structurées*, *non-structurées* ou une combinaison de ces 3 types. Les données peuvent être des documents, des articles, des emails, des photos, des fichiers, des données graphiques, des fichiers videos, données de senseurs, click de souris, etc, tout les types de formats qui existent à l'heure actuelle. La variété fait également références aux multiples sources dont peuvent provenir les données : le Web dont les réseaux sociaux, OpenData/bases de données publiques, IoT, données internes à une entreprise, données externes d'experts, etc. Comme nous l'avons vu sur la figure 1.2, les données de type non-structuré représentent plus de 80% de la quantité de données générée.
- **Vélocité** (Xiaoqian, 2019) : cela fait référence à la *vitesse à laquelle les données sont traitées* - lorsqu'on parle de traitement, on parle de toute manipulation possible sur des données (stockage, requêtes, analyse, visualisation, partage). Elle varie en fonction des besoins de l'application Big Data et peut être en *temps réel*; *presque-en-temps-réel* (exemple: trading haute-fréquence) ; *en streaming* ; ou encore pour un laps de temps plus long (traitement OLAP). Sawant et Shah (2013) suggèrent que cette vélocité trouve son origine dans la démocratisation d'appareils mobiles ainsi que le cloud computing. En effet, en 2013, 90% du volume de données qui existait avait été créés les deux dernières années et 70% avaient été créés par des individus et non des entreprises. La courte durée de vie des données implique qu'une gestion trop lente de celles-ci (en stockage ou en analyse) peut signifier l'obsolescence de données sans en avoir extrait la valeur qu'on leur cherche.

A l'heure actuelle, on ne parle plus uniquement des 3V du Big Data mais des 5V du Big Data.Certains étendent également le Big Data aux 9V ou plus (volume, variété, vélocité, véracité, value, variabilité, validité, vulnérabilité, volatilité, etc) Voici ci-dessous une courte définition des deux autres V :

- **Véracité**: cela fait référence à la fiabilité et à la qualité de la source de données ainsi qu'à la dimension qualitative des données. Les grands volumes de données n'implique pas forcément de

grandes quantités de connaissances extraites. Il y a un besoin de méthodes et de techniques pour évaluer la véracité des données.

- **Value**: cela fait référence à la valeur (ou bénéfice) obtenue par les applications Big Data mises en place. L'objectif de l'utilisation du Big Data est d'améliorer la valeur de l'entreprise, par des connaissances. La valeur apportée par ces applications doit être supérieure aux coûts liés au développement de celles-ci.

Ces 3 caractéristiques font que les données ne peuvent plus être traitées de la même manière qu'avant. L'idée derrière les 3V est que le Big Data n'est pas juste une question de capacité de stockage des données en plus grande quantité, il s'agit d'un nouveau paradigme de la science des données. Ce nouveau paradigme implique de, non seulement de repenser la gestion de la capacité de stockage de grande quantités de données, mais également de repenser les méthodes traditionnelles d'analyses de données qui ne sont plus applicables pour cette nouvelle ère du Big Data. Précisons que les méthodes traditionnelles ne sont pas devenues obsolètes ou incompatibles avec le nouveau paradigme, et elles sont d'ailleurs encore très largement utilisées dans le monde des données. De plus, le Big Data ne remplacera pas les systèmes BI mais ceux-ci vont plutôt évoluer ensemble. Par exemple, il est possible d'intégrer des analyses Big Data ou data mining dans des solutions de BI (par exemple, une analyse Big Data peut être une des sources de l'étape 1 du processus BI). La figure 1.7 présente les caractéristiques de la BI et du Big Data afin de bien saisir leurs similarités et leurs différences.

Business Intelligence	Big Data
Récupération et traitement de la données pour créer de la valeur	Récupération et traitement de la données pour créer de la valeur
On sait ce que l'on cherche: From Business Need to IT solution	On ne sait pas ce que l'on cherche : From IT to Business Need
Sources de données en interne (principalement)	Source de données en interne et en externe
Données structurées	Données structurées et non structurées
Serveur central	Systeme de fichier distribué
Logiciel privé	Open-source
Analyse offline	Analyse en temps réel et en offline

Figure 1.7: Tableau de comparaison entre la BI et le Big Data.

1.4. Caractéristiques du Big Data

Les caractéristiques du Big Data ne se limitent pas seulement aux 5V, Furht et Villanustre (2016) suggèrent que ce nouveau paradigme comporte un ensemble de technologies et techniques propres qui le rend particulier:

- **Système de fichier distribué et bases de données distribuées**, c'est-à-dire que les données sont réparties sur plusieurs serveurs, grâce à HDFS et aux NoSQL.
- **L'extension horizontale** du volume des bases de données. Cette technique est le fait de pouvoir augmenter la quantité de stockage disponible pour une base de donnée en augmentant le nombre de serveurs. On la contraste avec l'extension verticale qui, elle, fait la même chose mais en augmentant la puissance du serveur. Le problème avec cette dernière est la limite physique de la puissance de stockage d'un serveur.
- **MPP** (massive parallel processing) : l'utilisation d'un grand nombre de processeurs pour effectuer un ensemble de calculs coordonnés en parallèle, c'est-à-dire en simultané (Wikipedia contributors, 2018). Il y a 3 approches pour le MPP : Grid computing ; Computer Cluster et Massive Parallel Processor array.
- **Cloud computing** : correspond à l'accès à des services informatiques (serveurs, stockage, mise en réseau, logiciels) via Internet, à partir d'un fournisseur (Wikipedia contributors, 2018). Il y a 3 types de services dans le Cloud Computing : SaaS (Software as a service), le PaaS (Platform as a Service) et le IaaS (Infrastructure as a Service). Nous détaillerons plus profondément ces 3 services plus loin.
- **Les techniques d'analyse** propres au Big Data, comme l'ont développé Furht et Villanustre (2016) : text analytics (fait référence au processus d'analyse de textes non structurés pour en extraire des informations pertinentes) ; audio analytics (sont utilisées pour analyser et extraire des informations de données audio non structurées ; les applications typiques de l'analyse audio sont les centres d'appels et les entreprises du secteur de la santé) ; video analytics (consiste à analyser et à extraire des informations significatives des flux vidéo ; l'analyse vidéo peut être utilisée dans diverses applications de vidéosurveillance) ; social media analytics (comprend l'analyse de données structurées et non structurées provenant de diverses sources de médias sociaux, notamment Facebook, Linkedin, Twitter, YouTube, Instagram, Wikipedia, etc) ; predictive analytics (comprend des techniques pour prédire les résultats futurs sur des données passées et actuelles).

1.5. Modèles de distribution

Comme nous venons de le voir, une des caractéristiques principales du Big Data est l'architecture distribuée (via un système de fichier distribué et des bases de données distribuées), connaitre les différents modèles de distribution est essentiel pour comprendre l'architecture propre au Big Data. De plus, comme nous allons le voir dans le chapitre 2, une des motivations du mouvement NoSQL est de pouvoir utiliser des bases de données sur des larges clusters (sur différents ordinateurs). Nous nous sommes basés sur le travail Fowler et Sadalage (2013) afin de présenter les modèles de distribution. Ils suggèrent notamment que les modèles distribués possèdent différents avantages tels que:

- La capacité de manipuler de très grandes quantités de données ;
- La capacité d'améliorer la performance des requêtes de lecture ou d'écriture ;
- Augmenter la disponibilité de la base de données, même lors de la défaillance de certaines machines.

Faire fonctionner un système de fichiers ou une base de données sur un cluster de machines est complexe et nécessite de nouvelles méthodes pour le traitement et la manipulation des données.
Il y a 2 manières de distribuer les données: 1/ la **réplication** ou 2/ le **sharding.** On peut utiliser une seule des manières ou on peut utiliser les deux manières.
La réplication consiste à faire *plusieurs copies des données sur différentes machines* appelées nœuds dans un système distribué. Le sharding consiste à *diviser la donnée en plusieurs parties* et copier ces différentes parties sur *différents nœuds*.

La **grande différence** entre les deux manières est que pour la réplication, de mêmes données existent plusieurs fois sur des machines différentes, ce qui implique *une gestion plus complexe de la consistance* des données mais en contrepartie *une plus grande robustesse* des données tandis que pour le sharding, une donnée n'existe qu'une seule fois à un seul endroit, ce qui implique une *gestion de la consistance plus facile* mais une *moins bonne robustesse* (en cas de perte d'un nœud, les données sont perdues).

Comme l'ont fait Fowler et Safalage (2013), on va distinguer 3 méthodes de distribution des données : single-server ; sharding et réplication.

Single-Server

La façon la plus simple de stocker les données est de les stocker dans une base de données qui existe sur une seule machine qui s'occupe de toutes les opérations. Cependant, comme nous l'avons vu précédemment, l'extensibilité horizontale sur des clusters de machines/serveurs est indispensable pour la gestion des très grandes bases de données et donc le modèle « single-server » ne convient pas pour ce type de bases de données (mais peut convenir pour d'autres types de bases de données).

Sharding

Le sharding divise la base de donnée en différentes parties et répartie ces différentes parties sur différents nœuds qui sont chacun responsable des opérations de lecture et d'écriture de leur propre partie de données.
La figure 1.8 est une illustration du sharding d'une collection de données de 1TB qui est répartie sur 4 serveurs distincts.

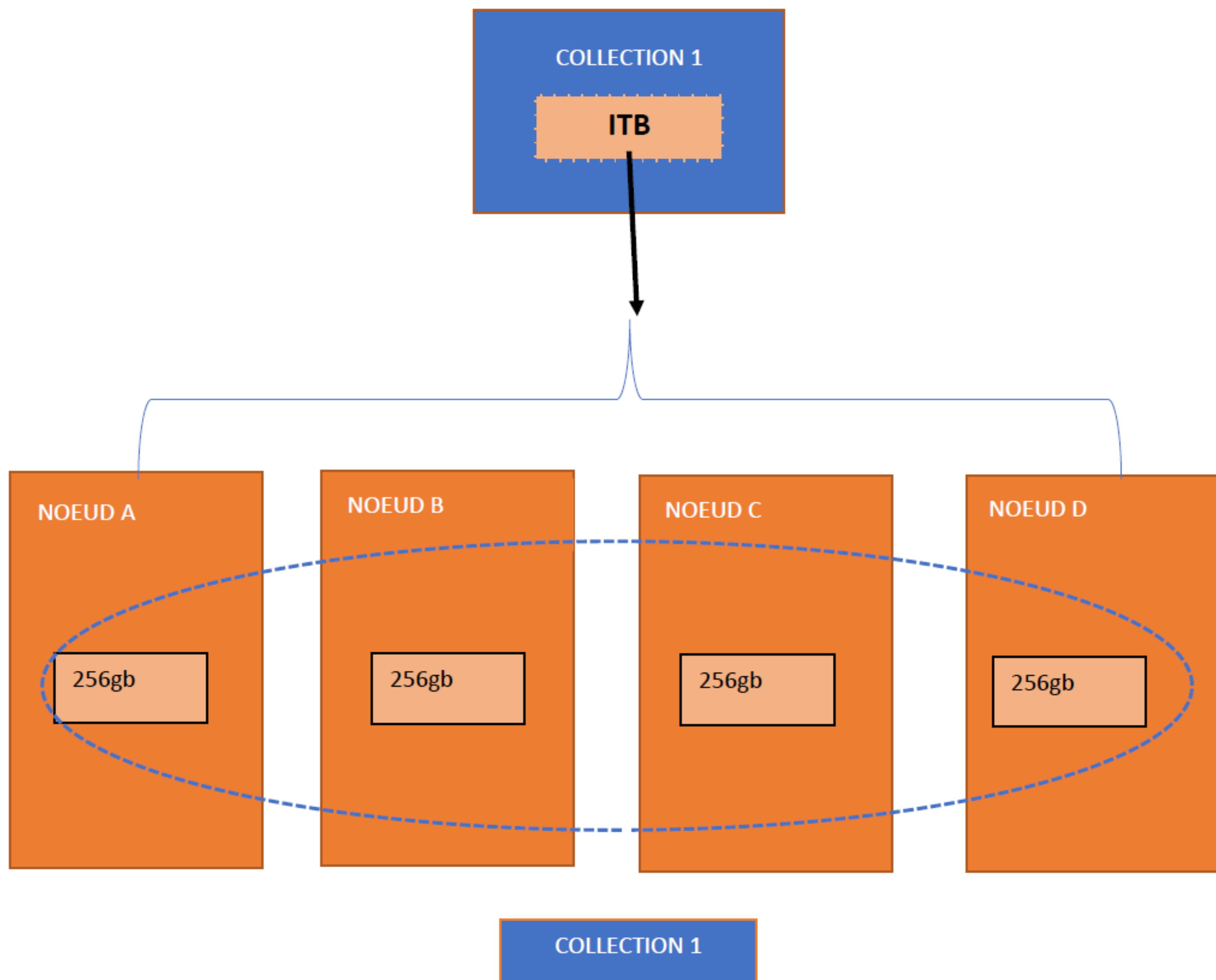


Figure 1.8: Schema du sharding d'une collection de données de 1 TB en 4 parties différentes de 256GB sur 4 saveurs distincts (McCormick, 2021).

Dans l'idéal, différents utilisateurs communiquent avec des machines différentes et la charge de travail - qui correspond aux calculs à effectuer pour répondre aux requêtes des utilisateurs - est répartie parfaitement à travers les différentes machines, à savoir 25% des calculs à faire pour chacun des serveurs A;B;C;D. Pour se rapprocher de cet idéal, il faut grouper ensemble les données liées qui reçoivent des requêtes similaires (faire des agrégats de données similaires) afin que, lorsqu'un utilisateur fait une requête, il ne doive le faire que sur un seul des serveurs — ou du moins qu'il récupère la plupart des données dont il avait besoin sur un seul nœud. Dans le pire des cas, avec une mauvaise répartition des données, la requête de l'utilisateur va devoir passer auprès de chaque serveur pour obtenir une partie du résultat du calcul.
C'est ainsi que les agrégats prennent tout leur sens dans les bases de données NoSQL (sauf graphe) car on va conceptualiser la structure et la répartition des données sur différents nœuds pour faire des agrégats des unités de distribution.
Pour l'amélioration de la performance, Fowler et Sadalage (2013) proposent deux recommandations intéressantes :

- si la plupart des requêtes se font sur des agrégats liés à une localisation physique, il est bien de **stocker les données dans un data center proche de la localisation physique.** Exemple: *pour une base de données d'un magasin de ventes en ligne, si les agrégats se font sur le pays où vit le client, il est bien de stocker les données dans un data center proche du pays où le client vit.*
- **Garder la répartition du travail équitable entre les différents nœuds**, c'est-à-dire essayer d'arranger les agrégats d'une manière à ce que le travail soit réparti sur les différents nœuds. Exemple : il ne faut pas que 80% du travail soit sur le shard A, 5% sur le shard B et 2,5% sur le shard C et D. Attention, il arrive que la répartition du sharding varie avec le temps ou les cycles (Exemple: *jour de la semaine*).

Le sharding est particulièrement intéressant pour **augmenter la performance de l'écriture** et de la **lecture** là où la réplication est performante pour la lecture mais beaucoup moins performante pour l'écriture.
Un des **désavantages du sharding est la résilience**. Tout comme la méthode single-server, la défaillance d'un nœud rend le nœud indisponible ou entraine une perte de données, ce qui implique qu'une partie de la base de donnée sera manquante. Lorsque l'on a qu'une seule machine, il n'est pas très coûteux de mettre en place des procédures pour garder cette machine toujours opérationnelle ou pour récupérer des données perdues. Dans le cas des clusters, on utilise généralement des serveurs de moins bonne qualité (étant donné que l'on en a beaucoup) et on considère que la défaillance des serveurs est plutôt la norme que l'exception. C'est pourquoi le sharding sans réplication diminue la résilience de la base de données.

Réplication

La réplication consiste à recopier les données sur différents nœuds/serveurs.

1. **Maitre-esclave** (Master-slave)

Dans un système distribué maitre-esclave, il y a 2 types de nœuds : le nœud **maitre**, qui est unique, et les nœuds esclaves, qui représentent le reste des nœuds du système. Le maitre est la *source autoritaire* et est *responsable de la modification des données* dans la base de données (ou des fichiers s'il s'agit d'un système de fichier distribué) sur les différents nœuds. Un processus de réplication synchronise tout les nœuds esclaves avec le maitre.
Le nœud maitre peut servir pour des opérations d'écriture et de lecture et les nœuds esclaves peuvent servir uniquement pour des requêtes de lecture. Concrètement sur la figure 1.9, si un client (ici l'application) fait une requête d'écriture, elle sera soumise au maitre qui va faire l'opération d'écriture en son sein et ensuite cette opération d'écriture sera répliquée au reste des nœuds esclaves. Ainsi, les nœuds esclaves ne peuvent pas faire d'opération d'écriture directement mais peuvent recevoir des *ordres* de réplication d'opération d'écriture venant du maitre. On parle d'extensibilité de lecture et non d'écriture. Ce type de réplication est particulièrement utile lorsqu'il y a beaucoup de requêtes de lecture. Si on veut améliorer les performances de lecture, il suffira de rajouter des nœuds esclaves pour répondre plus rapidement aux requêtes. Cependant, on est tout de même limité par la capacité du maitre à gérer les modifications et à actualiser les nœuds esclaves.
Le deuxième avantage est la résilience pour les requêtes de lecture. En effet, si un nœud esclave échoue, il y a d'autres nœuds esclaves qui sont disponibles pour répondre aux requêtes. Et si c'est le maitre qui échoue, il y a toujours les nœuds esclaves qui peuvent répondre aux requêtes de lecture, le temps que le master soit restauré ou qu'un nouveau maitre soit désigné. Cependant, un maitre qui échoue peut potentiellement entrainer de la perte de donnée si les modifications récentes n'ont pas été répliquées sur les nœuds esclaves.
Le master peut être désigné manuellement ou automatiquement (parmi le cluster de nœuds, un nœud est élu pour être le master). L'avantage de la désignation automatique est la réduction du downtime du système.
Le désavantage de la réplication est le danger d'inconsistance des données en lecture. Il se peut que deux utilisateurs lisant sur deux nœuds différents récupèrent des données différentes parce que la réplication n'est pas synchronisée sur un des deux nœuds.

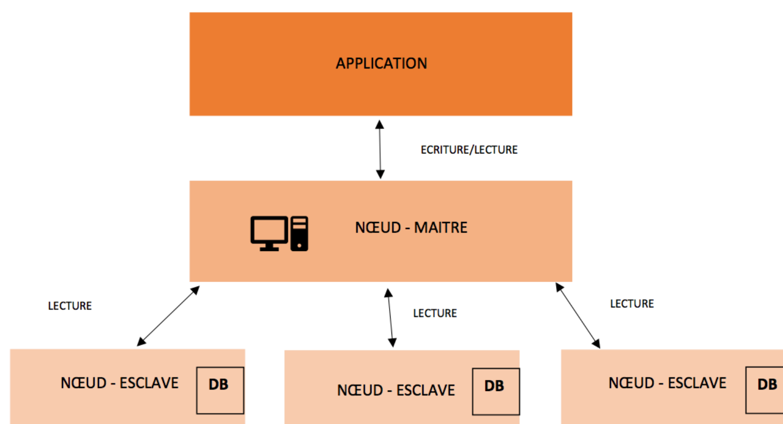


Figure 1.9: Schéma d'un système distribué maitre-esclave avec 3 nœuds-esclaves.

2. Peer-to-peer

Le point faible du fonctionnement maitre-esclave est le maitre lui-même, car il est un *single point of failure* et il est *l'unique nœud pouvant faire des opérations d'écriture*. La méthode de réplication peer-to-peer résout ce problème en ayant aucun maitre et en ayant des nœuds égaux pouvant tous faire des opérations d'écriture. Cette méthode augmente la robustesse des opérations d'écriture et de lecture mais implique également une extensibilité horizontale pour à la fois la lecture et également l'écriture : il suffit d'ajouter de nouveaux nœuds pour avoir une meilleure performance. La figure 1.10 est un exemple de système distribué peer-to-peer fully-connected, où chaque nœud est en communication avec tout les autres nœuds du cluster. Dans un tel système, le client peut faire une requête d'écriture ou de lecture sur n'importe quel nœud du cluster et ensuite le résultat de la requête va se répliquer sur les autres nœuds du cluster.
À nouveau, le désavantage est le risque d'inconsistance des données en lecture et en écriture : dans le pire des cas, deux utilisateurs vont modifier différemment les mêmes données au même moment (*write-write conflict*), ce qui va entrainer de l'inconsistance dans la base de données (ou le système de fichiers distribué) étant donné que l'on aura une même donnée avec deux informations différentes sur deux nœuds différents. Et une inconsistance en écriture reste pour toujours.

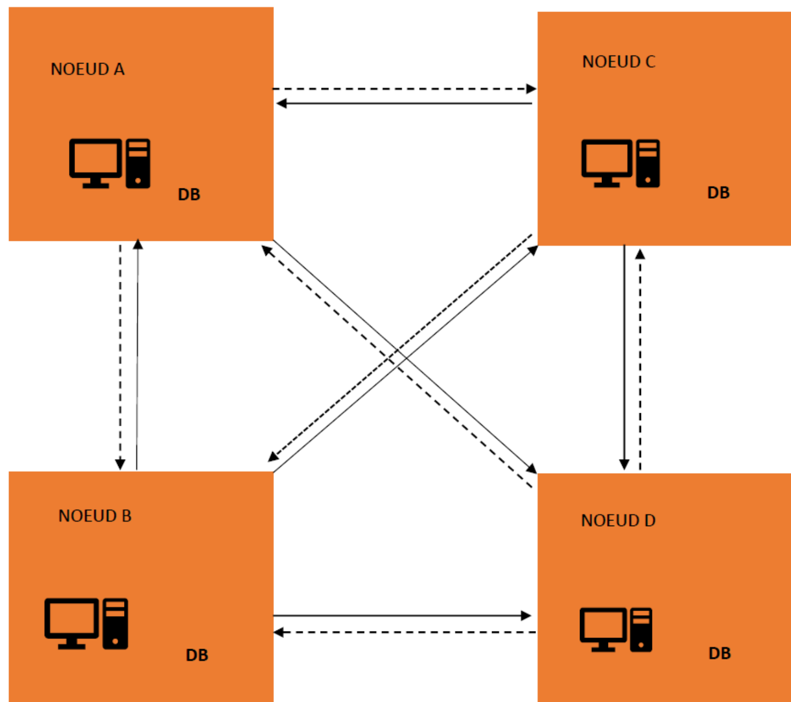


Figure 1.10: Schéma d'un système distribué peer-to-peer fully connected (Kituta, Argawal et Kant, 2019).

Il existe plusieurs approches pour la distribution peer-to-peer dont la plus connue est l'approche *protocole de bavardage* (*Gossip Protocol*), comme sur la figure 1.11. Les nœuds agissent en parallèle, chaque nœud engage une conversation avec un nombre choisi de machines et après un certain laps de temps, une information connue d'un seul nœud le devient pour tous. Une autre approche possible est l'approche fully connected comme sur la figure 1.10.

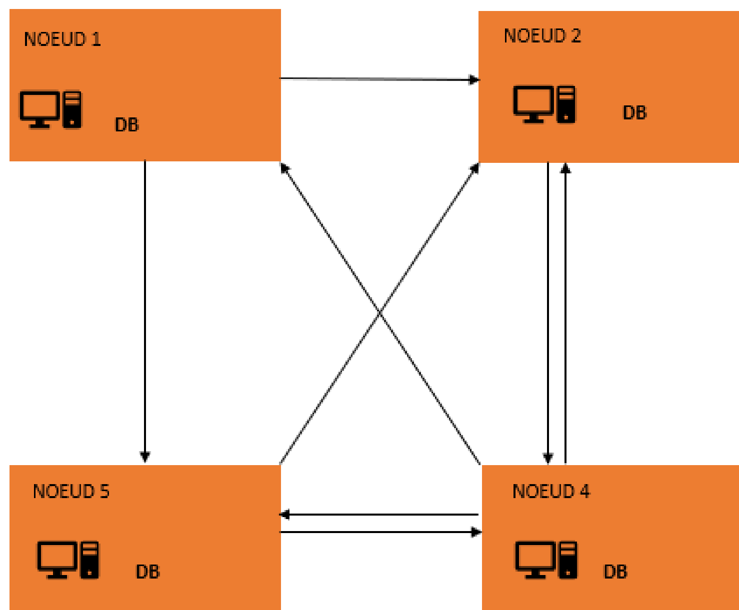


Figure 1.11: Schéma d'un système distribué peer-to-peer gossip protocol (Kituta, Argawal et Kant, 2019).

En résumé, la technique maitre-esclave réduit le risque de « write-write conflict » mais le maitre est un point faible et la technique peer-to-peer évite d'avoir le problème de *single point-of-failure* et d'opération d'écriture mais augmente le risque de « write-write conflict ».

1.6. Type de stockage

Afin de mieux comprendre comment les données sont stockées sur un ordinateur, reprenons les explications de Emarcore et Kendrick (2016) à propos des types de mémoires d'ordinateurs. Pour fonctionner, un ordinateur a besoin de deux types de mémoires :

- **La mémoire vive (in-memory ou RAM)** : pour faire fonctionner un programme et conserver les données que l'on a besoin lors de l'utilisation de l'ordinateur. Les données de la mémoire vive sont supprimées lorsque l'ordinateur est éteint ou lorsque l'ordinateur échoue.
- **La mémoire de stockage (on disk)** : pour stocker les données même lorsque l'ordinateur est éteint ou échoue mais également pour stocker les programmes installés. Il y a de nouveaux 2 types de mémoires de stockage :
  1. Les disques durs mécaniques (HHD)
  2. Les disques à mémoire flash (SSD), qui sont plus rapides que les HHD mais qui sont plus chers (bien que le marché des SSD devient de plus en plus démocratique).

La mémoire vive est plus performante que la mémoire de stockage, cependant la mémoire de stockage a des capacités de stockage bien plus grand que la mémoire vive.

Il existe aussi la mémoire virtuelle qui est un espace sur la mémoire de stockage qui a été dédiée pour agir comme une mémoire vive.

1.7. Overview d'un projet de Data Science

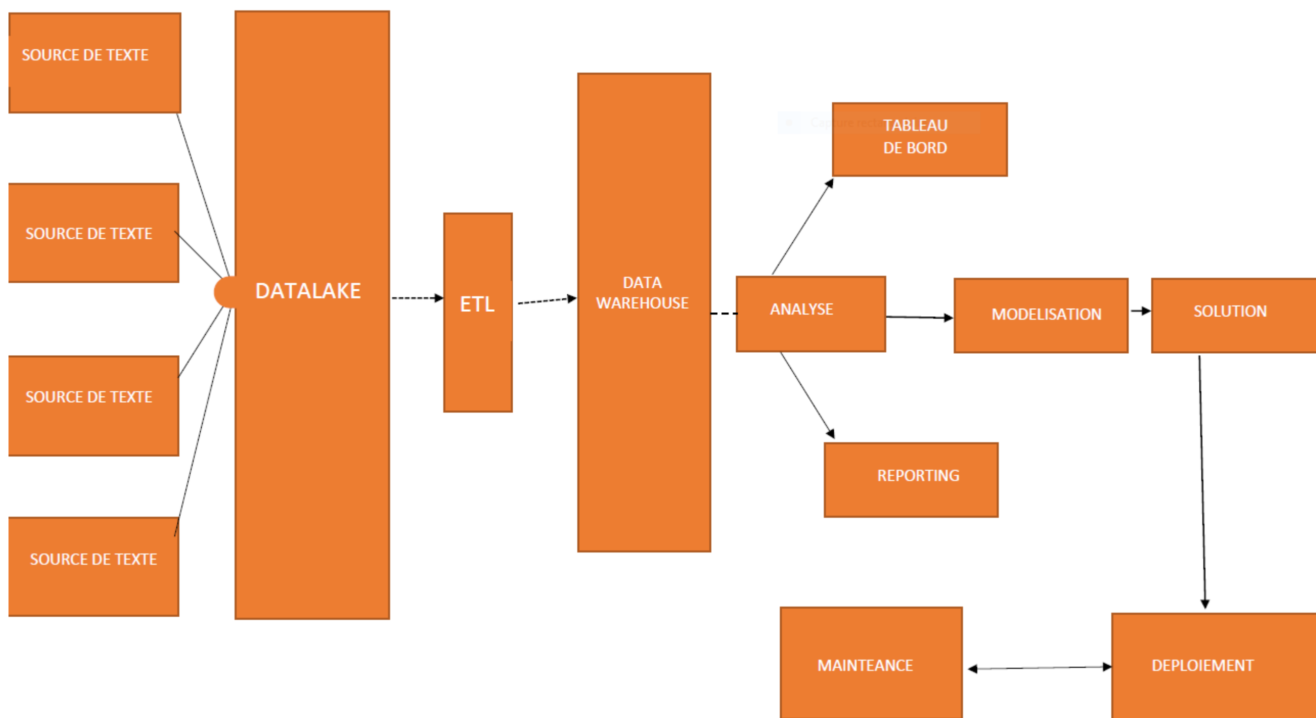


Figure 1.12: Schéma d'un projet en Data Science partant de la source de la donnée et finissant avec le déploiement d'une solution IT.

La figure 1.12 est un schéma représentant les différentes étapes d'un projet en Data Science. Ce schéma part du développement IT, comme vu dans le tableau 1.7, pour arriver à une solution IT orientée business. Ce schéma reprend les 3 parties du Big Data que nous avons vu au début de ce chapitre: la partie sur les données (de sources de données à Datawarehouse), la partie analyse (analyse, tableaux de bords, modélisation) et la partie management (reporting, solution, déploiement et maintenance). Précisons que la sécurité et la gestion des accès ne sont pas représentées dans le schéma car ce n'est pas représentable comme une simple étape, mais elles font bien parties de la

partie management d'une solution Big Data. Nous verrons dans le chapitre 3 que l'architecture Big Data reprend les différentes étapes de cette figure au travers de ses différentes couches.
Pour comprendre le cheminement d'un projet en Data Science, nous allons suivre le cycle de vie de la donnée. La donnée est d'abord brute (non-structurée ou semi-structurée et parfois structurée) et a de multiples origines en fonction du nombre de sources. Après la donnée est traitée à travers une couche d'ingestion, une couche découverte et de nettoyage (par exemple un ETL). Ensuite, la donnée passe par une couche d'intégration pour qu'elle soit stockée et organisée dans des DatawareHouse et/ou dans des systèmes de fichiers distribués tel que Hadoop. Par la suite, on va exploiter ces données à l'aide d'analyses (data mining, machine learning, deep learning, etc.) pour en obtenir des modèles et créer des solutions à partir de ceux-ci.
Rappelons la différence entre le **Front-End** du **Back-End** (Brewster, 2020).

Front-end: représente l'interface avec laquelle l'utilisateur (d'un site internet ou d'une application d'une entreprise) va interagir. Un développeur front-end utilisera typiquement de l'HTML, du Javascript et du CSS.
Back-end: représente la structure qui va permettre le développement de logiciel/application, sur laquelle l'utilisateur va faire des actions. Il s'agit de la partie cachée (que l'on ne voit pas) d'une application ou d'un site web. Le développement d'un back-end impliquera un ou des serveurs, des bases de données, des APIs et un langage de programmation (un pour la manipulation de la base de données ainsi qu'un pour le développement de l'application).
Les deux fonctionnent en complémentarité: sans back-end, un front-end serait statique et sans front-end, le back-end serait juste un ensemble de données sur lesquelles on ne peut pas interagir.
Nous allons le voir, une solution complète de Big Data est composée de tâches front-end et back-end mais il est évident que la plus grande partie du travail d'une solution Big Data est back-end.

Comme nous l'avons précisé plus haut, ce cours se consacre principalement à la structure d'un projet Big Data plutôt qu'aux techniques d'analyse du Big Data et au management du Big Data. Dans le schéma ci-dessus, la structure va se construire à partir des sources de données jusqu'au datawarehouse/système distribué Hadoop. Précisons ici que ce schéma ci-dessus est valable également pour un projet de BI mais les sources de données ainsi que la manière de traiter les données seront différentes que pour un projet de Big Data.

Dans le cadre de ce cours, nous allons nous concentrer principalement sur la partie '*données'* de la solution Big Data (stockage, architecture et traitement des données) et nous présenterons brièvement les différentes techniques d'analyses pour exploiter les données du Big Data ainsi que plusieurs pistes pour le management des données. Ainsi, le chapitre 2 se consacrera à l'étude des nouveaux types de bases de données, adaptées aux nouveaux besoins, appelés NoSQL. Le chapitre 3 portera sur l'architecture particulière du big data : un système distribué Hadoop, sur plusieurs serveurs utilisant les deux technologies de NoSQL et de MapReduce.

2. Bases de données NoSQL

Evolution des bases de données

Les travaux de Meier et Kaufmann (2019) ainsi que de Fowler et Sadalage (2013) nous ont servi de bases sur lesquelles s'appuyer pour le développement et la présentation des bases de données dans le cadre d'une solution Big Data.
Pour bien comprendre l'enjeu derrière les nouveaux types de bases de données, il est important de retracer l'histoire des bases de données et plus particulièrement des bases de données relationnelles.
Commençons par définir ce qu'est une base de données. Dès le début de l'histoire de l'informatique, un besoin de structuration des données est apparu et ce besoin a été traduit en deux parties : le stockage des données et la manipulation de ces données (rechercher des données parmi le stockage ; ordonner les données ; trier les données ; …). C'est pour cela que l'on parle de SGBD : système de gestion de bases de données (abrégé en base de données).
Les données stockées d'une base de données peuvent être structurées, semi-structurées ou non-structurées ; elles peuvent être localisées sur un même support ou être réparties sur différents supports (on parle alors de système distribué).
En fonction du type de SGBD, la base de données peut proposer différentes fonctionnalités (Dubois, 2008):

- **Requêtes CRUD** : create ; retrieve ; update et delete.
- **Filtrer** : appliquer des critères de sélection pour obtenir un sous-ensemble de données.
- **Trier** : utiliser des critères pour placer les données dans un ordre précis.
- **Répliquer** : créer des clones de données pour la sécurité (back-up) et pour l'amélioration de la performance

On peut différencier deux types d'usages généraux pour les bases de données(Stitch, 2020) :

1. Les bases de données opérationnelles, dites **OLTP** (*online transactions processing*), s'occupent de la manipulation des données en temps réel : stockent, modifient et traitent les données des transactions faites en temps réel. Dowling (2000) suggère que « L'accent est mis sur la vitesse de réponse et la capacité de traiter plusieurs opérations simultanément ». *Exemple : opérations bancaires ou achat de biens.*
2. Les bases de données d'analyses (ou décisionnelles), dites **OLAP** (*online analytical processing*), s'occupent d'analyser à l'aide de requêtes complexes sur les données historiques et agrégées, venant de systèmes OLTP et de sources externes, pour obtenir des statistiques ou des prévisions afin d'aboutir à des décisions. Les services OLAP sont typiquement des applications BI ou des applications de data mining. L'accent est mis sur la capacité à faire des requêtes très complexes et sur leur vitesse de réponse. *Exemple : à partir du datawarehouse, trouver le produit qui est le plus vendu parmi les jeunes entre 12 ans et 18 ans.*

Dans les années 1980 et jusqu'aux années 2000, il y a eu une montée en puissance et une dominance des SGBD relationnel/SQL grâce aux caractéristiques intéressantes, présentées par Meier et Kaufmann (2019), qu'offrait ce type de base de données :

- *Langage déclaratif puissant*
- *Schéma et méta-données*
- *Assurance de la consistance*
- *Intégrité référentielle et déclencheurs*
- *Récupération et historique des log*
- *Fonctionnement multi-user et synchronisation*
- *Utilisateurs, rôles et sécurité*
- *Indexation*

Les quatre principaux avantages de bases de données relationnelles sont ceux-ci :

1. **Persistance des données** : capacité pour une base de données de conserver les informations même après avoir éteint la machine ou l'application qui utilisait la base de données.
2. **La concurrence des données** : pour beaucoup d'applications, les utilisateurs regardent (et potentiellement modifient) les mêmes données en mêmes temps. Il faut pouvoir coordonner ces actions pour éviter des problèmes tel que l'exemple connu de la double réservation pour la même chambre d'hôtel. Les bases de données relationnelles (SGBDR) peuvent aider à la coordination en contrôlant l'accès aux données à travers les transactions. D'ailleurs, les transactions ont un double rôle : 1/ celui de mécanisme de _contrôle pour la gestion de la concurrence_ des données et 2/ celui de mécanisme de _gestion des erreurs_ (si une erreur apparait lors du processus de transaction, il est possible de « roll back » la transaction pour corriger l'erreur).
3. **L'intégration des données :** la plupart des applications font partie d'un écosystème plus grand qu'elles-mêmes et celles-ci utilisent la plupart du temps les mêmes sources de données pour réaliser le travail à faire. Les SGBDR autorisent le partage de la même base de données pour différentes applications rendu possible grâce à la concurrence des données qu'offrent les SGBDR ainsi que le langage SQL.
4. **Modèle standardisé** : la plupart des bases de données relationnelles offrent les mêmes types de services dans un modèle et un langage standardisé, ou du moins sont très ressemblants. De cette manière, les développeurs peuvent apprendre le fonctionnement classique des SGBDR et l'appliquer sur la plupart des différents logiciels de type SGBDR.

Souvent, on résume les avantages des bases de données relationnelles en citant *la consistance des données* et la *sécurité (dont la gestion du contrôle d'accès) des données*.
D'ailleurs, les SGDBR ont souvent été utilisés pour l'intégrité et la protection des transactions que l'on retrouve principalement dans des applications bancaires ou dans des logiciels d'assurance. L'inconvénient majeur des SGBDR vient du contrôle de l'intégrité des données, qui demande beaucoup de travail et de puissance et donc prend plus de temps et d'espace de mémoire.
Dans les années 2000, avec la montée d'internet, les quantités de données, la variété des données et la vélocité des données ont explosé (les 3V vus précédemment) et il n'a plus été possible de gérer ces immenses quantités de données avec les bases de données relationnelles.
Là où la puissance du contrôle de la consistance des données était un avantage recherché, il est devenu un frein pour l'efficacité, la performance et la flexibilité du traitement des données (grandes, variées et rapides).

D'une part, il a fallu gérer l'extensibilité des données, c'est-à-dire agrandir les espaces de stockage des données. Pour cela, il y a deux possibilités (figure 2.1) : **1/ l'extension verticale (vertical scaling)** *qui consiste* à *ajouter plus de ressources physiques ou virtuelles, c'est-à-dire plus de puissance (plus de CPU, plus de mémoire, plus de stockage) au serveur*, ou **2/ extension horizontale (horizontal scaling)** (*qui se fait par réplication ou par sharding) qui consiste* à *ajouter plus de serveurs pour avoir une capacité totale plus grande (addition de tous les serveurs)(Yemini, 2019).* Les avantages et inconvénients des deux types d'extensions seront discutés plus loin, mais retenons que l'extension horizontale est préférable pour gérer les immenses quantités de données. Et le problème avec les SGBD relationnel/SQL est que l'architecture relationnelle et SQL n'a pas été conçue pour être répartie sur une multitude d'ordinateurs, ou du moins est très difficile à mettre en pratique.

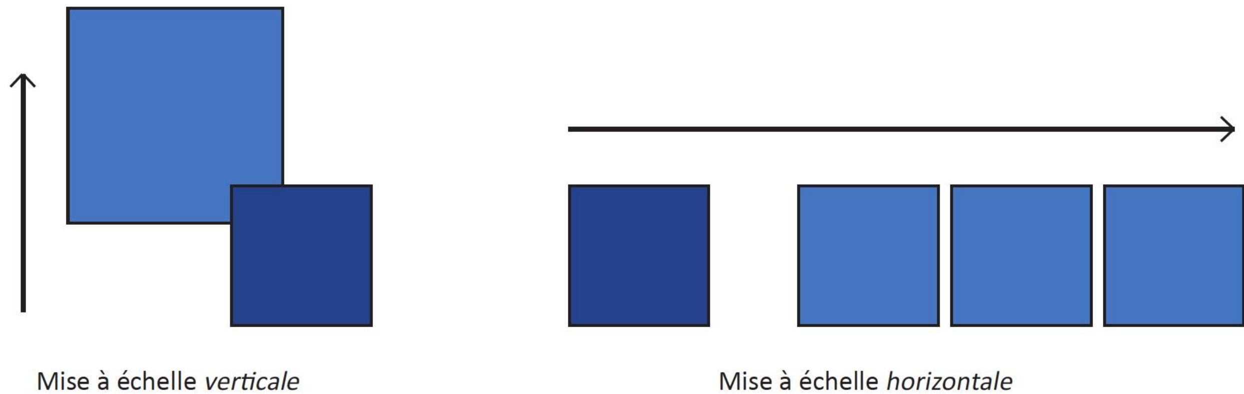


Figure 2.1 Schéma de l'extensibilité verticale et de l'extensibilité horizontale.

D'autre part, les besoins des bases de données avaient évolués : la performance, la disponibilité et la tolérance aux partitions (théorème de CAP, discuté plus loin) sont devenus plus important que la consistance, notamment pour des sites internet comme les réseaux sociaux. Il est aisé de comprendre que les besoins d'une base de données pour le ERP d'une entreprise sont différents des besoins d'une base de données pour un site internet comme Facebook ou Twitter.

Dans les années 2000, les grands acteurs d'internet (Google avec BigTable, Amazon avec Dynamo) ainsi que les communautés open source et de développement du Web ont poussés le développement des système distribués à grande envergure et le développement de logiciels/langages/technologies pouvant répondre aux nouveaux besoins du monde des données (Ali et al., 2019) :

- Système distribué et base de données distribuée ;
- Extensibilité horizontale ;
- Haute disponibilité ;
- Peu de latence ;
- Bon marché

Caractéristiques des bases de données NoSQL

Les bases de données NoSQL ont été développées pour répondre à ces besoins.

Selon le Pr Fowler, le nom « NoSQL » viendrait d'un hashtag de Twitter qui a été utilisé pour définir le nom d'une réunion sur le mouvement des bases de données non-relationnelles, organisé par Johan Oskarsson à San Francisco, et qui est resté depuis.
Il est important de préciser qu'il faut plutôt lire « NoSQL » comme Not Only SQL. Cette précision signifie que même si les bases de données NoSQL n'utilisent pas SQL, chaque type de base de données répond à un type d'objectif précis et il est possible voire très probable qu'une grande application partage différents types de bases de données, SQL et NoSQL, pour répondre aux différentes fonctionnalités de l'application. On parle alors de **Polyglot Persistence**, terme popularisé par le Pr Fowler, qui désigne le fait d'avoir recours à plusieurs types de technologies de stockage de données pour les besoins différents d'une même application.

Il est difficile de définir précisément ce qu'est une base de données NoSQL parce qu'elles n'ont pas toutes les mêmes fonctionnalités ou les mêmes propriétés. Une bonne manière de saisir ce qu'est le NoSQL est de définir des propriétés/caractéristiques communes :

- Le modèle de la base de données est **non-relationnel.**
- Il n'y a **pas de schéma défini** (schema-free).
- **Flexibilité** : « *Compatible avec une large variété de technologies permettant le stockage de données structurées, semi-structurées et non structurées* » (Ali & al., 2019).
- **Tolérance aux partitions** et **extensibilité horizontale**.
- Les données sont **distribuées** sur plusieurs serveurs (extensibilité horizontale et tolérance au fail-over).
- La **réplication des données est facile.**
- **Open-source** (la plupart des bases de données NoSQL le sont malgré que certains services peuvent être commerciaux).
- Le modèle de consistance est **BASE** sauf pour les bases de données orientées graphe qui est ACID. (Voir *infra* pour une présentation du concept de modèle de consistance).

Donnée structurée : les données sont définies et formatées pour un usage qui est déjà connu afin de faciliter leur stockage et leur utilisation future. On peut faire des requêtes sur ce type de données avec le langage SQL (Structured Query Language). Généralement, les données structurées sont stockées sous forme de tableaux avec des colonnes et des lignes. Les données structurées nécessitent moins d'espaces de stockage que les données non-structurées.
Exemple : les données d'une commande faite sur un site d'E-commerce. Les données seront considérées comme structurées si elles sont stockées sous forme de tableaux de chiffres sur lesquels on peut faire des requêtes pour obtenir des informations.

Données non-structurées : Weglarz (2004) les définit comme « toutes données stockées dans un format non-structuré à un niveau atomique ». Les données nécessitent un traitement afin d'en extraire de la connaissance ou de produire de la valeur. Elles ne sont pas définies ou formatées et peuvent avoir n'importe quel type ou forme. Les données non-structurées sont plus compliquées à gérer et à protéger que les données structurées. Rappelons que nous avons montré dans la figure 1.2 que les données non-structurées représentent plus de 80% du volume total des données générées par les entreprises.
Exemples (Tondak, 2021) : PDFs, documents Word, posts de réseaux sociaux, livres, fichiers vidéo/audio, emails, images, données de capteurs, BSON documents, applications logs (informations à propos des downtime, de la maintenance, des mises-à-jour, etc), open-data (ex: open-data sur la météo), …

Données semi-structurées : c'est un mélange de données structurées et non structurées. Les données ne sont pas assez définies et formatées pour être considéré comme structurées mais qui ont tout de même une certaine forme de structure. Ce concept de données est assez flou.
Exemple : JSON documents.

L'utilisation de bases de données NoSQL va être intéressante pour le stockage, la gestion et la manipulation de très grandes quantités de données et la variété des données. Le choix des NoSQL est particulièrement judicieux pour les applications qui nécessitent un traitement des données en temps réel ou presque en temps réel. De plus, lorsque le modèle de stockage ainsi que la quantité de données et les types de données sont incertains ou évoluent au fil du temps, il est opportun d'avoir recours à des bases de données NoSQL.

Il existe 4 grandes familles de bases de données NoSQL :

1. Bases de données clé-valeur (*orienté agrégat*)
2. Bases de données orientées colonne (*orienté agrégat*)
3. Bases de données orientées document (*orienté agrégat*)
4. Bases de données orientées graphe (*aggregate-ignorant*)

On peut diviser ces 4 grandes familles en 2 : les bases de données clé-valeur ; document et colonnes qui sont **aggregate-oriented,** terme inventé et popularisé par les travaux de Evans E. et Evans J. (2004); et les bases de données orientées graphe qui sont **aggregate-ignorant**.
Fowler et Sadalage (2013) reprennent ce concept d'agrégats et le définissent comme suit : « […] est une collection d'objets/de données liés avec laquelle on interagit comme une seule unité ». Afin de préciser cette définition, nous avons défini l'agrégat comme un regroupement de données liées que l'on veut considérer comme une unité unique et sur lequel on fait des opérations récurrentes sur l'ensemble des données du regroupement.
Fowler et Sadalage (2013) présentent également les avantages de l'utilisation d'agrégats pour les bases de données :

- Permet d'augmenter la performance des requêtes.
- Dans un système de données distribué, un agrégat est groupe de données *logique* pour la réplication et le sharding.
- Résout le problème d'*impédance mismatch* (l'impedance mismatch, c'est un concept représentant la différence qui peut arriver entre le modèle relationnel et la structure des données en mémoire).

Ci-dessous sont décrits les avantages et désavantages généraux des bases de données NoSQL. Il est important de préciser que ceux-ci peuvent varier en nature ou en profondeur en fonction du type de base de données NoSQL.

Avantages et désavantages de NoSQL

Avantages des NoSQL :

- **Performance élevée** ;
- **Haute disponibilité** (pas de down-time) ;
- **Tolérance au partitionnement** ;
- **Extensibilité horizontale** et **facilité de réplication** ;
- **Pas de contrainte schématique** (ou du moins flexibilité du schéma). Avec un modèle basé sur un schéma, il faut savoir à l'avance ce que l'on doit stocker, ce qui n'est pas toujours facile. Sans schéma, on peut pratiquement stocker tout type de données que l'on veut. De plus, cela permet de manipuler des données non-uniformes (structurées, non structurées ou semi structurées) ensemble sans se préoccuper de leur type. Enfin, cela implique qu'il n'y a pas de contrainte pour insérer une nouvelle donnée ou modifier une donnée.
- **Facilité d'implémentation** et de design ;
- **Compatible** avec la plupart des langages de programmation ;
- Limitation du problème d'impedance *mismatch ;*
- Base de données **robuste et résiliente** car il n'y a pas de « single-point-of-failure » (pas de panne ni de perte de données).

Désavantages des NoSQL :

- La **consistance** des transactions ;
- **Capacité des requêtes limitées** (low-level query language en comparaison avec le SQL) ;
- **Immaturité**. Etant donné que les bases de données NoSQL sont relativement une nouvelle technologie, les SGDB et les outils liés à ceux-ci ne sont pas autant développés que ceux des SGBD relationnel. De plus, la popularité, le consensus de connaissances parmi la communauté scientifique et les compétences requises pour l'utilisation de solution Big Data ne sont pas encore répandues comme le sont les bases de données relationnelles. En conséquence, l'apprentissage des compétences Big Data n'est pas aisé et, en plus, les entreprises peuvent avoir plus de mal à accepter ce nouveau paradigme dans leurs pratiques et cultures ;
- Le **manque de standardisation** (lié également à l'immaturité).

SQL	NoSQL
Langage standardisé et structuré : SQL	Pas de langage déclaratif
Bases de données relationnelles	Bases de données clé-valeur ; colonnes ; document et graphe
Schéma prédéfini	Pas de schéma ou schéma dynamique
Méthodologie waterfall	Méthodologie agile
Uniquement des données structurées	Tout type de données
Modélisation rigide	Modélisation souple
Extensibilité verticale	Extensibilité verticale et horizontale
Possibilité de requètes complexes	Limite dans la complexité des requètes
Logiciels privés	Open-source et logiciels privés
Consistance forte	Consistance plus souple
Limitations de la sécurité et de la gestion des accès	Bonne sécurité et gestion des accès

Tableau 2.1 Tableau de comparaison entre SQL et NoSQL.

Consistance des données (ACID vs BASE) :

Lors du développement d'une base de données, le choix du modèle de consistance est important car il permet de s'orienter vers un type de SGBD adéquat pour l'application. Une des grandes différences entre les bases de données SQL et NoSQL est la consistance, le premier type préférant une consistance forte et le second préférant une consistance plus flexible en échange de plus d'une plus grande disponibilité ou d'une architecture distribuée.
Fowler et Sadalage (2013) considèrent et présentent deux formes de consistance :

1. **La consistance de lecture.**
Prenons un exemple où deux clients, Guillaume et Adrien, veulent acheter un ticket sur un site internet pour aller voir un concert et il ne reste qu'un ticket pour ce concert spécifique.
Guillaume arrive le premier sur le site internet, met le ticket dans son panier mais il pense que le ticket est un peu cher et hésite à l'acheter. En parallèle, après que Guillaume soit arrivé sur le site internet, Adrien a vu le ticket sur le site internet et l'a directement acheté. Finalement, Guillaume se décide d'acheter le ticket et lorsqu'il procède à l'achat, il découvre qu'il n'y a plus de ticket disponible. On a ce qu'on appelle un « read-write conflict » (Fowler et Sadalage, 2013).

Dans une base de données relationnelle, on utilise la notion d'atomicité de transaction pour éviter ces problèmes « read-write ». Lorsqu'un utilisateur procède à une écriture sur des données, le SGBD va s'assurer que toutes ses opérations d'écritures vont être enveloppées dans une même transaction. Ainsi, le système garanti qu'un autre utilisateur a soit accès aux données avant le début de la transaction, soit accès aux données après que la transaction soit terminée.

2. **La consistance de mise à jour**.
Prenons un exemple où deux employés d'une entreprise, Arthur et Zoé, veulent mettre à jour une donnée (la date de création de l'entreprise sur le site internet de l'entreprise) *simultanément*. Cependant, ils utilisent chacun un format différent pour les dates : Arthur utilise la notation américaine (Mois, Jour, Année) et Zoé la notation anglaise (Jour, Mois, Année). Étant donné qu'ils veulent mettre à jour la même donnée au même moment, on aura ce qu'on appelle un « write-write conflict ». Sans système de contrôle de la concurrence, le SGBD va mettre à jour par ordre alphabétique : d'abord Arthur, puis Zoé et donc on perdrait la mise à jour des données faite par Arthur et la modification de donnée finale sera celle de Zoé. Cependant, avec un système de contrôle de la concurrence, on peut décider comment gérer ce type de conflit. Deux approches sont possibles :

- *Approche pessimiste (Fowler et Sadalage, 2013)* : on empêche les conflits d'arriver. Par exemple, lorsqu'un utilisateur modifie une donnée, l'accès à cette donnée est bloqué. Dans notre exemple, Zoé n'aurait pas eu accès à la donnée, ou du moins seulement après la mise à jour de Arthur. Cette approche entraine des problèmes de disponibilité (dans notre cas pour l'utilisateur Zoé) et elle peut être un inconvénient dans certains cas.
- *Approche optimiste (Fowler et Sadalage, 2013)* : on laisse les conflits arriver, on les détecte et on applique un processus de résolution. Un exemple d'approche optimiste est l'utilisation de **version-stamp**. Chaque fois qu'un utilisateur a accès à la base de données, il reçoit un version-stamp de la base de données. Dans notre exemple, Arthur et Zoé auront reçu tous les deux la version v.001 (vu qu'ils ont accédé simultanément). Avant la mise à jour de Arthur, le SGDB va comparer la version de Arthur (v.001) et la version actuelle de la base de données (v.001), il va autoriser la mise à jour et il va ensuite actualiser la version de la base de données : v.002. Lorsque Zoé va vouloir faire sa mise à jour, le SGDB va comparer la version de Zoé (v.001) avec la version actuelle (v.002) et va refuser la mise à jour (ou du moins recommencer le processus de mise à jour avec la base de données actualisée) car les deux versions ne sont pas identiques.

Ces deux approches fonctionnent lorsque la base de données ne possède qu'un serveur. Dans le cas où il y a plusieurs copies de données similaires sur des serveurs différents (système distribué maitre-esclave ou peer-to-peer), la gestion de la concurrence nécessite d'autres mécanismes, on parle alors de consistance séquentielle (tous les nœuds appliquent les opérations dans le même ordre).
Cet exemple permet de percevoir le compromis fondamental entre *sécurité et consistance* de la donnée (éviter les « write-write conflicts », qui entraine un ralentissement des mises-à-jour) avec *performance* de la base de données (opération rapide et disponible).
Nous détaillerons ce compromis sécurité-performance, lié à la distribution des données sur différents serveurs, lorsque nous parlerons du théorème du CAP.

Consistance logique et consistance de réplication

Fowler et Sadalage (2013) suggèrent qu'il y a deux types de consistance : logique et de réplication. La principale différence est d'être sur un seul serveur (ou base de données distribué en sharding) : consistance logique ou alors sur un cluster de serveurs : consistance de réplication.
Comme nous l'avons vu dans l'introduction, pour un cluster de serveurs, il y a deux possibilités : le sharding (les données sont réparties sur plusieurs serveurs mais il n'y a pas de copie de données : chaque donnée n'existe qu'à un seul endroit) ou alors la réplication (les données sont réparties à travers plusieurs serveurs et elles existent plusieurs fois sur plusieurs serveurs en même temps). Le sharding correspond plus ou moins à la consistance logique (plus ou moins car les données sont réparties sur plusieurs serveurs mais pourraient être considérées comme l'étant sur un serveur central) et la réplication correspond à la consistance de réplication. Retenons que lorsque les données sont distribuées sur plusieurs serveurs en réplication, il y a des nouveaux types d'inconsistance qui apparaissent. Nous en parlerons plus en détail avec le théorème du CAP.

Il existe deux **grands modèles de consistance** : le **modèle ACID**, qui favorise plutôt une *consistance forte*, et le **modèle BASE**, qui favorise plutôt une *consistance plus flexible* au profit d'une plus grande disponibilité.

Modèle de consistance « ACID » :
Atomic : chaque opération lors d'une transaction a réussi ou chaque opération est annulée.
Consistent : à la fin d'une transaction, la base de données est cohérente.
Isolated : les transactions ne sont pas en conflit les unes avec les autres.
Durable : les résultats de l'application d'une transaction sont permanents (même en présence d'erreurs).
Les propriétés ACID signifient qu'une fois qu'une transaction est complète, les données sont cohérentes et stables sur le disque.
On veut plutôt un modèle ACID lorsque la fiabilité et la consistance des données sont essentiels.

Dans d'autres cas, il se peut que le modèle ACID soit un modèle trop concerné par la sécurité des données par rapport à la réalité des besoins de la base de données.
Dernièrement, avec l'apparition du nouveau paradigme des données (les 3V) ainsi que les bases de données NoSQL, le modèle ACID est devenu moins à la mode parce les priorités pour les bases de données se sont déplacées : les exigences de cohérence immédiate et de précision des données ont été relâchées au profit d'autres avantages : comme l'extensibilité, la résilience, la disponibilité et la performance.

Modèle de consistance « BASE » :
Basic Availability : la base de données doit fonctionner la plupart du temps. Certains serveurs de la base de données peuvent échouer mais le reste de la base de données peut continuer de fonctionner.
Soft-state : le stockage ne doit pas être cohérent en écriture, et les différentes répliques (clones) n'ont pas à être mutuellement cohérentes en permanence.
Eventual Consistency : le stockage est cohérent à un moment ultérieur. Cela veut dire qu'il se peut que certaines copies de données soient inconsistantes pour une courte durée, le temps que la mise à jour des données se fasse sur tous les serveurs.
Dans l'exemple plus bas du théorème de CAP de la chambre d'hôtel, si Arthur réserve la chambre d'hôtel, il est possible que Zoé puisse voir encore une chambre disponible le temps que la propagation de la mise à jour se soit faite sur tous les serveurs. Et à un moment donné, la propagation se sera faite partout et Zoé verra que la chambre d'hôtel est indisponible. Le temps entre la réservation effective d'Arthur sur un serveur et la mise à jour sur tous les serveurs s'appelle la *inconsistency window* (Fowler et Sadalage, 2013).

Ainsi, les propriétés de BASE sont beaucoup plus souples que celles d'ACID mais il n'existe pas uniquement l'un ou l'autre modèle, il faut considérer ces concepts comme un compromis entre les deux modèles, qui va dépendre des besoins de l'application liés à la base de données.
Une base de données se basant sur le modèle BASE va plutôt faire valoir la disponibilité à la cohérence permanente, et à l'inverse un SGBD ACID va faire valoir la consistance à la disponibilité.
Les bases de données orientées agrégats n'ont pas besoin d'autant de transactions ni d'autant de consistance de données que dans le cas des bases de données relationnelles ou graphes (aggregate-ignorante). Ainsi, en général, les bases de données NoSQL clé-valeurs, colonnes et documents suivent plutôt le modèle de consistance BASE.

Le Théorème du CAP

Ce théorème a été proposé par E.Brewer (2000) et celui-ci propose :
« Un système distribué ne peut garantir en même temps les 3 propriétés suivantes :
Consistency (=cohérence, consistance) : tous les nœuds du système voient exactement les mêmes données en même temps.
Availability (= disponibilité) : toutes les requêtes reçoivent une réponse (même si un ou plusieurs nœuds ont échoué).
PartitionTolerance (= tolérance aux partitions) : le système continue de fonctionner malgré un nombre quelconque de pannes de communications entre les nœuds. ».

Un système avec un seul serveur (nœud) n'est pas concerné par le théorème car il ne possède pas de partition. Un single-server système peut donc concilier consistance et disponibilité.
Fowler et Sadalage (2013) nous expliquent que, théoriquement, un cluster de nœuds pourrait être consistant et disponible mais cela signifierait que si une partition apparaissait dans le cluster, tous les nœuds devraient s'arrêter afin qu'aucune requête ne puisse se faire sur aucun nœud de manière à maintenir la cohérence. Pour avoir un cluster cohérent et disponible, il faut s'assurer qu'il n'y ait de partitions que très rarement et de manière complète. Fowler et Sadalage (2013) expliquent qu'en pratique ce n'est pas possible car il s'agirait d'une contrainte très chère et complexe.

Ainsi, en pratique, un système distribué a de grandes chances de subir des partitions, Fowler et Sadalage (2013) proposent donc de voir le théorème de Brewer sous une autre perspective.
Étant donné que, en pratique, un système distribué doit être tolérant aux partitions, le compromis n'existe réellement qu'entre la consistance et la disponibilité, comme le suggère la figure 2.2.
Précisons à nouveau que ce n'est pas un choix binaire, il est possible de faire un compromis entre les deux propriétés. De plus, la plupart du temps, il ne s'agit pas d'un compromis entre consistance et disponibilité mais plutôt d'un compromis entre consistance et temps de réponse (délai). Effectivement, au plus la consistance est recherchée, au plus le nombre de nœuds impliqués devra être grand et donc au plus le temps de réponse sera grand. Fowler et Sadalage (2013) proposent de considérer la disponibilité comme le délai maximum que l'on peut tolérer ; au-dessus de ce délai, la donnée sera considérée comme non disponible.

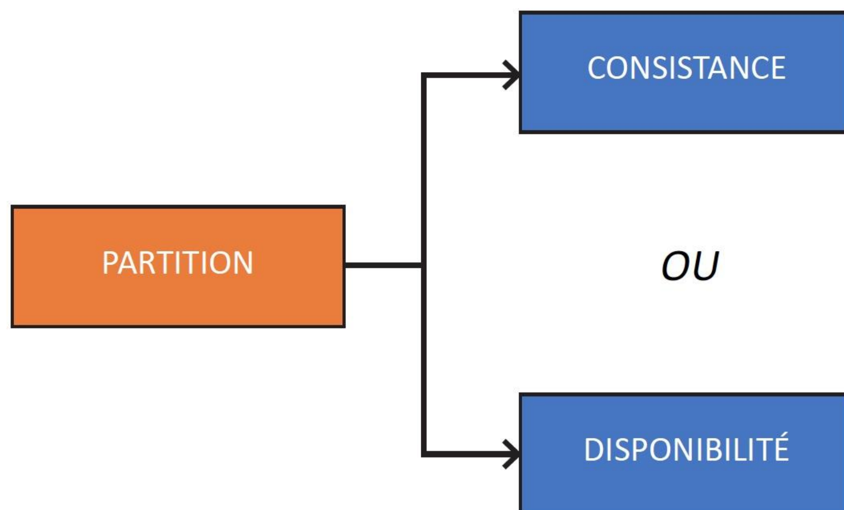


Figure 2.2 : Théorème du CAP pour les systèmes distribués sous une autre perspective.

Pour illustrer le théorème du cap, prenons l'exemple de Fowler et Sadalage (2013). Considérons Arthur et Zoé, qui vivent chacun sur un continent différent, Arthur en Amérique du Nord et Zoé en Europe. Ils veulent chacun réserver, simultanément, la même dernière chambre disponible d'un hôtel en Inde. Zoé interagit avec un serveur en Europe, Arthur avec un serveur en Amérique. Dans cette première situation, les deux serveurs peuvent communiquer ensemble (pas de partitions entre les serveurs de la base de données) peuvent donc décider qui aura la chambre et en fonction de la décision, l'un des serveurs va accepter et confirmer la réservation et l'autre va refuser la réservation.
Dans l'exemple représenté sur la figure 2.3 (Fowler et Sadalage, 2013), les serveurs décident d'accepter la demande de réservation d'Arthur et de refuser celle de Zoé.
Cet type situation représente la majorité des cas d'utilisations.

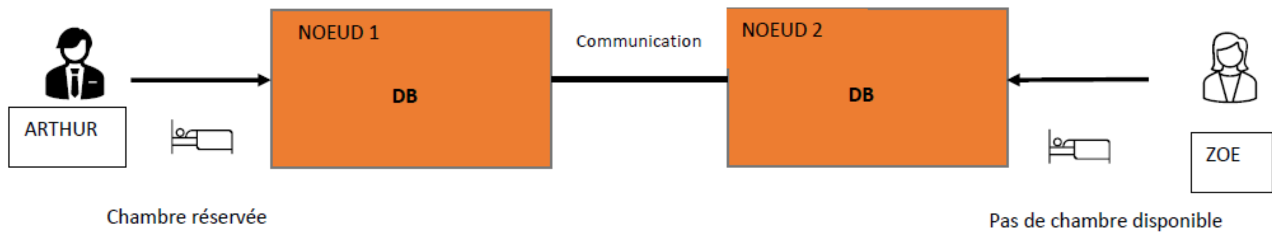


Figure 2.3 : Schéma d'un exemple de système distribué où les nœuds peuvent communiquer entre eux.

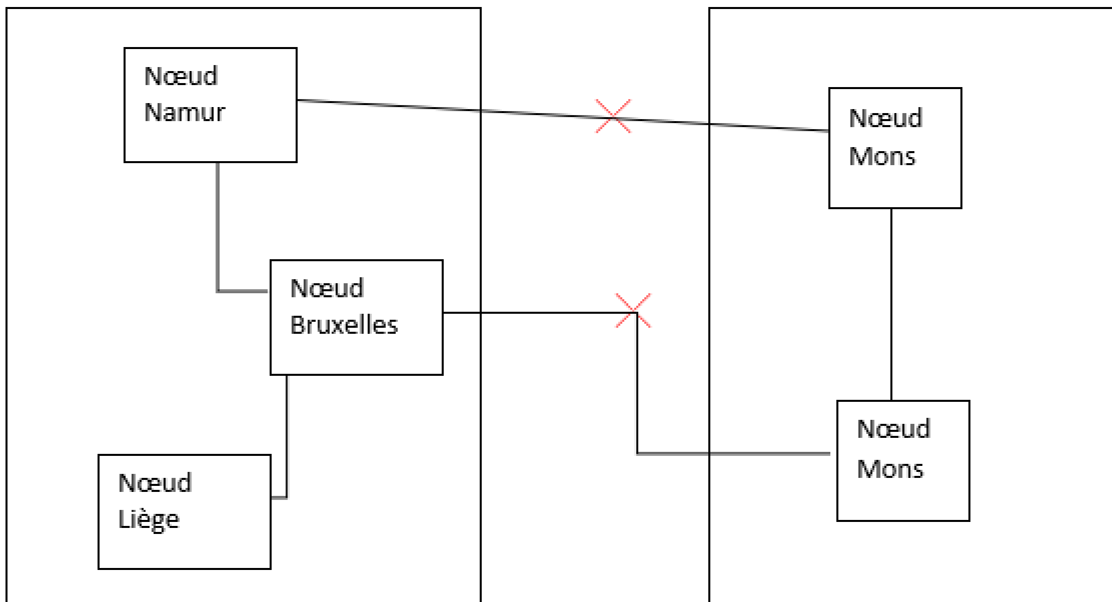


Figure 2.4 : Exemple de système distribué avec deux partitions.

Maintenant, reprenons le même exemple, à la différence que, dû à un problème, les deux serveurs ne peuvent pas communiquer ensemble (le système distribué, tolérant aux partitions, fait face à des partitions comme sur l'exemple de la figure 2.4). Lorsque Arthur et Zoé font leur demande de réservation simultanée, les serveurs peuvent réagir de deux manières possibles, comme nous pouvons le voir sur la figure 2.5 (Fowler et Sadalage, 2013) :

a) Étant donné qu'il n'y a pas de communications, aucune opération n'est possible. La consistance des données est le plus important que la disponibilité. Dans notre exemple, ni Arthur et ni Zoé ne peuvent effectuer de réservations.

b) Bien qu'il n'y ait pas de communication et qu'il y a un risque d'inconsistance, la disponibilité des serveurs pour les utilisateurs est plus importante que le risque d'inconsistance. Arthur et Zoé peuvent tous les deux effectuer une réservation, ce qui entraine une double réservation pour l'unique chambre qui restait.

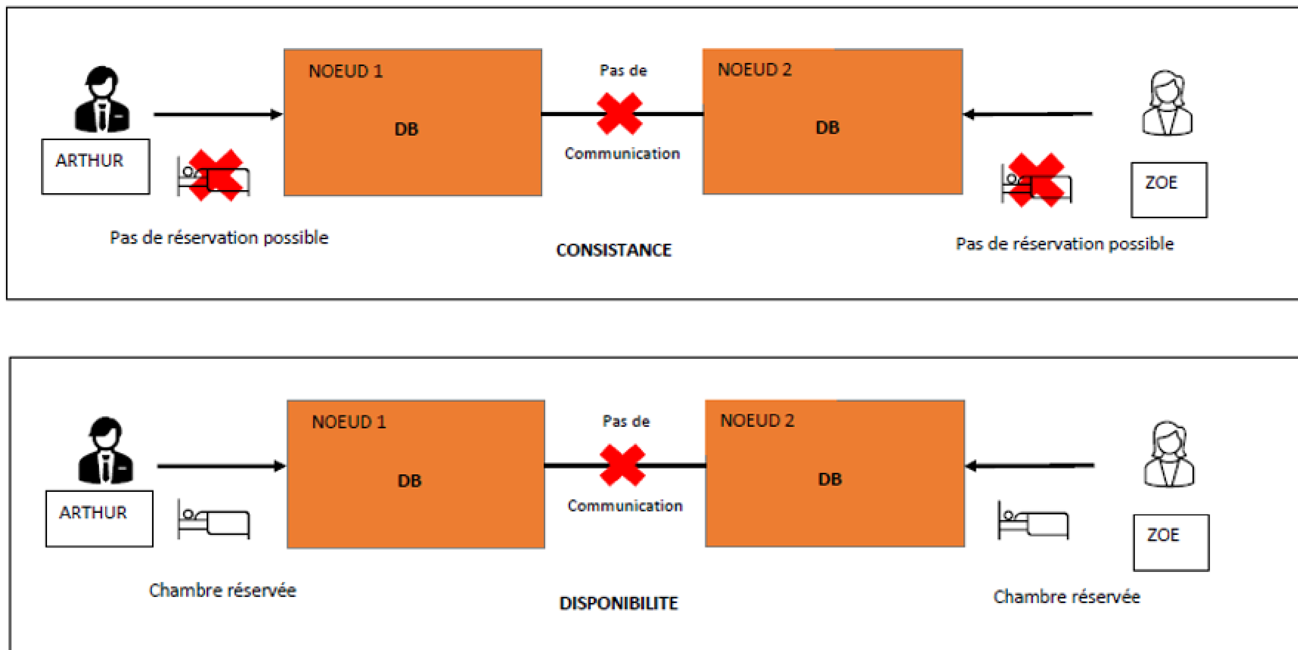


Figure 2.5 : Exemples de système distribué avec des partitions. Le premier exemple représente le cas où la consistance est choisie. Le deuxième exemple représente le cas où la disponibilité est choisie.

Les 2 possibilités montrent le compromis qu'il y a à faire entre la consistance des données et la disponibilité des données lorsque notre système distribué est tolérant aux partitions. **Ce choix se décide en accord avec les business rules** (les experts, pas uniquement les développeurs) de l'application. Est-ce que l'application veut privilégier un service toujours cohérent ou est-ce que l'application veut privilégier un service toujours disponible ? Il est important de bien comprendre que ce choix, qui parait uniquement technique, est crucial et doit être en accord avec la vision « business » de l'application.
La question commerciale serait : *est-ce que le manque à gagner de ne pas permettre la réservation de chambre lors de problème de communication entre les serveurs est supérieur ou inférieur à la réputation (ou la cohérence) de l'hôtel* ? S'il est supérieur, alors il faut privilégier la disponibilité et si il est inférieur, alors il faut privilégier la cohérence.

Par exemple, c'est ce dilemme de consistance contre disponibilité qui a poussé Amazon a développé DynamoDB (base de données NoSQL) ayant comme vision que la disponibilité (toujours avoir le panier d'achat disponible…) devait être privilégiée à la consistance (… quitte à avoir des risques d'erreurs et d'incohérence dans le panier d'un utilisateur, par exemple avoir 2 fois le même item). Dans cette application en particulier, la disponibilité est effectivement plus importante car il est moins dérangeant pour l'utilisateur de supprimer le deuxième item de son panier que de ne pas avoir son panier d'achat disponible.

Extensibilité horizontale et extensibilité verticale

Extensibilité Horizontale	Extensibilité Verticale
Cluster de serveurs	Serveur central
Plus de stockage et de puissance grâce à l'ajout de nouveaux serveurs	Plus de stockage et de puissance grâce à de plus grands disque de mémoire et CPU
Pas de limite de stockage ou de puissance	Limite physique du meilleur CPU et du plus grand disque qui existe
Besoin de nouvelles méthodes de stockage et de calculs	Méthodes de stockage et de calculs sont identiques
La distribution des données impliquent de nouveaux challenges : consistance, partitionnement, complexité du management etc.	Facilité d'implémentation et de gestion
Coût initiaux élevés	Coût initiaux bas
Coût d'extension bas grâce à l'utilisation de serveurs low-cost, et facile (simple ajout de noeuds)	Coût d'extension à cause d'utilisation de serveurs plus qualitatifs et plus chers
Robustesse et resilience et tolérance aux défaillances	Single-point-of-failure
Haute disponibilité	Possibilité de downtime

Tableau 2.2 : Comparaison entre extensibilité horizontale et extensibilité verticale

Le tableau 2.2 compare les deux possibilités d'extension : horizontale et verticale. Nous comprenons l'intérêt d'avoir recours à l'extensibilité horizontale pour le Big Data étant donné que celle-ci permet une extensibilité, théoriquement illimitée, une haute disponibilité, de la tolérance aux défaillances et de la résilience. De plus, nous verrons plus tard qu'un système distribué permet d'obtenir de meilleures performances grâce aux calculs parallèles (Massive Parallel Processing, MPP) qui est une méthode qui divise et répartit la charge de calcul à travers différents nœuds pour obtenir une réponse plus rapide.

3.Base de données orientée clé-valeur

3.1. Introduction

Le modèle d'une base de données clé-valeur est basé sur la méthode de stockage clé-valeur pour stocker des données. **Cette méthode clé-valeur est l'une des plus simples qui existent** et la plupart des langages de programmation possèdent une fonctionnalité de stockage de données de type clé-valeur. Par exemple, le dictionnaire dans le langage python est un stockage clé-valeur.

On stocke les données sous forme de **paires clé-valeur** dont **chaque clé représente un identifiant unique qui est associée à une valeur.** La particularité d'une base de données clé-valeur vient du fait que la valeur peut être de tout type de données : des *données simples* (texte, chiffre,…) aux *données plus complexes* telles que des objets, des listes ou *même des fichiers*. Les clés peuvent prendre n'importe quel type de données ou de nom, tant qu'on peut leur appliquer une fonction de hachage. Cependant, elles doivent être choisie de manière stratégique car il s'agit de la seule manière de récupérer les données qui lui sont associées.
Les requêtes se font uniquement sur les clés et ne peuvent pas se faire sur les valeurs.
Chaque instance de la base de données correspond à une paire clé-valeur.
Le SGBD utilise la clé, et lui applique une fonction de hachage, détaillé plus bas, pour obtenir un nombre, qui correspond à un endroit précis et retrouvable dans lequel la valeur liée à la clé sera stockée. Ensuite, comme l'expliquent Meier et Kaufmann (2019), on utilise à nouveau les clés à la manière d'un identifiant unique en SGBDR, pour faire des opérations et des requêtes sur la base de données.
En principe, une base de données clé-valeur ne répond qu'aux requêtes CRUD. Cette simplicité de fonctionnalité implique que ces bases de données n'ont pas besoin et ne possèdent pas de langage de requêtes qui leur est propre.
Une base de données clé-valeur ne nécessite aucun schéma prédéfini ni de métadonnées pour encadrer les données pour le stockage. Cette absence de schéma permet de *stocker n'importe quel type de donnée* ainsi que de créer ou faire des requêtes sur n'importe quelle donnée *à tout moment*.
Les deux grands avantages de ne pas avoir de schéma défini sont la **simplicité d'utilisation** et la **grande performance des requêtes simples**.

3.2 Fonctionnement

Pour expliquer comment fonctionne ce type de base de données, il est nécessaire de comprendre le fonctionnement d'une table de hachage. En effet, en pratique, une base de données clé-valeur est une table de hachage très large. Nous nous sommes principalement basé sur le cours de Demaine et Devadas (2011) et le cours de Gonnage et Nebra (2017) pour présenter le fonctionnement des tables de hachage.

Les tables de hachage permettent d'accéder efficacement à un type abstrait de données « associative array », composé d'une clé liée à une valeur. Il est possible d'accéder rapidement à la valeur d'une clé en connaissant celle-ci. Le fonctionnement est simple : la clé est *hashée* à l'aide d'une fonction de hachage et transformée en un entier. Ensuite, cet entier est utilisé comme indice pour indiquer dans quel slot de la liste des slots de la table de hachage, la valeur liée à la clé (et donc à l'indice) est stockée.

Idéalement, une fonction de hachage ne devrait pas permettre que deux clés différentes partagent le même index, sinon deux clés devraient partager le même slot et lorsqu'un *lookup* serait fait sur cet indice, deux valeurs seraient retournées. Pour se rapprocher de cet idéal, une fonction de hachage devrait avoir une distribution uniforme des valeurs, c'est-à-dire que chaque clé passant par la fonction de hachage doit avoir la même probabilité d'obtenir un entier que n'importe quel autre entier de la

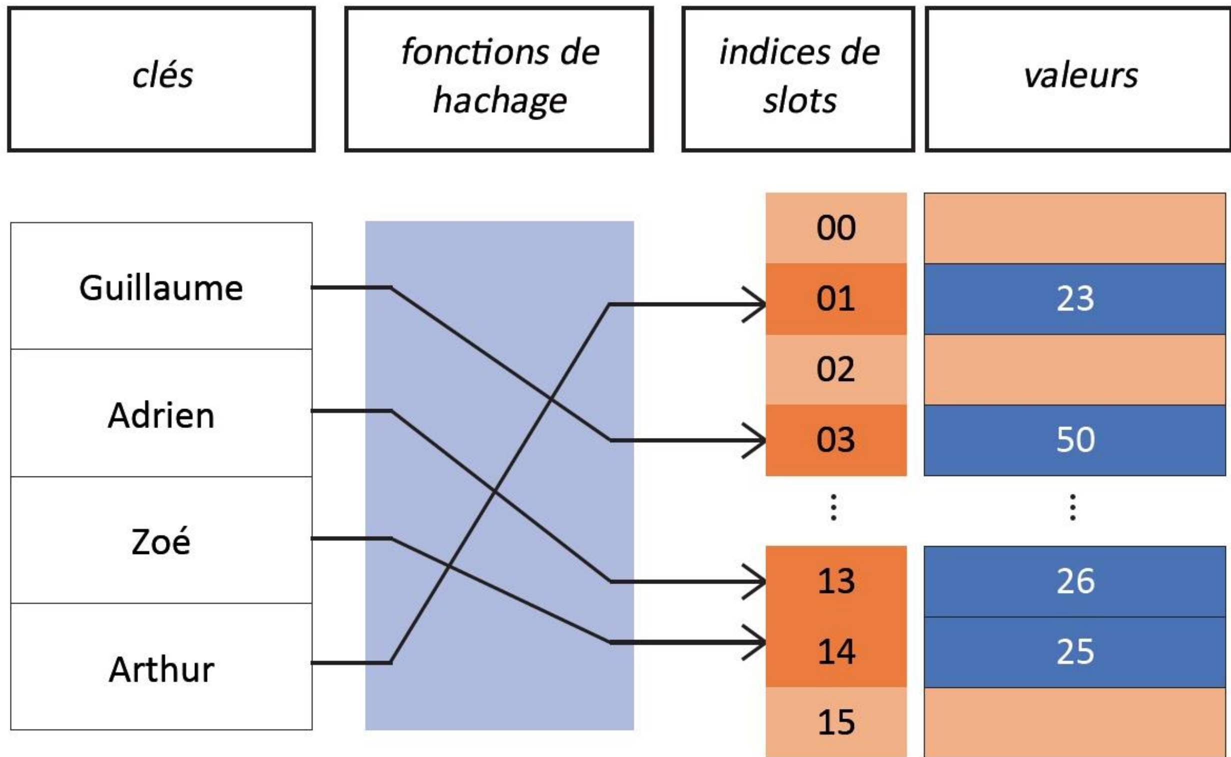


table de hachage. Le contraire signifierait que certains entiers seraient plus communs que d'autres, ce qui entrainerait plus de chances que deux clés partagent le même index.

Figure 3.1 : Fonctionnement du stockage clé-valeur.

En pratique avec de très grands volumes, c'est-à-dire un très grand nombre de clés, il arrive que différentes clés partagent le même entier après transformation avec la fonction de hachage, c'est-à-dire que différentes clés partagent le même slot dans la table de hachage. On parle alors de collision (hash-collision).

Il y a collision si $h(c1) = h(c2)$ et $c1 \neq c2$,

avec $h()$ = fonction de hachage ;
$c1$ = clé 1 ;
$c2$ = clé 2

Figure 3.2 : Formule mathématique d'une collision.

Pour résoudre ces problèmes de collisions, il existe deux types de techniques: le *chainage* ou l'*adressage ouvert*.

Le chainage :
Cette solution consiste à créer une liste chainée à l'emplacement de la collision (Gonnage et Nebra, 2017). Lorsque l'on hache une clé et que l'on voit que l'emplacement correspondant à l'indice est déjà utilisé, on crée un pointeur depuis le tableau vers cette liste chainée, qui comprend, dans l'ordre d'arrivée, les différentes paires de clé-valeurs qui ont fait une collision.

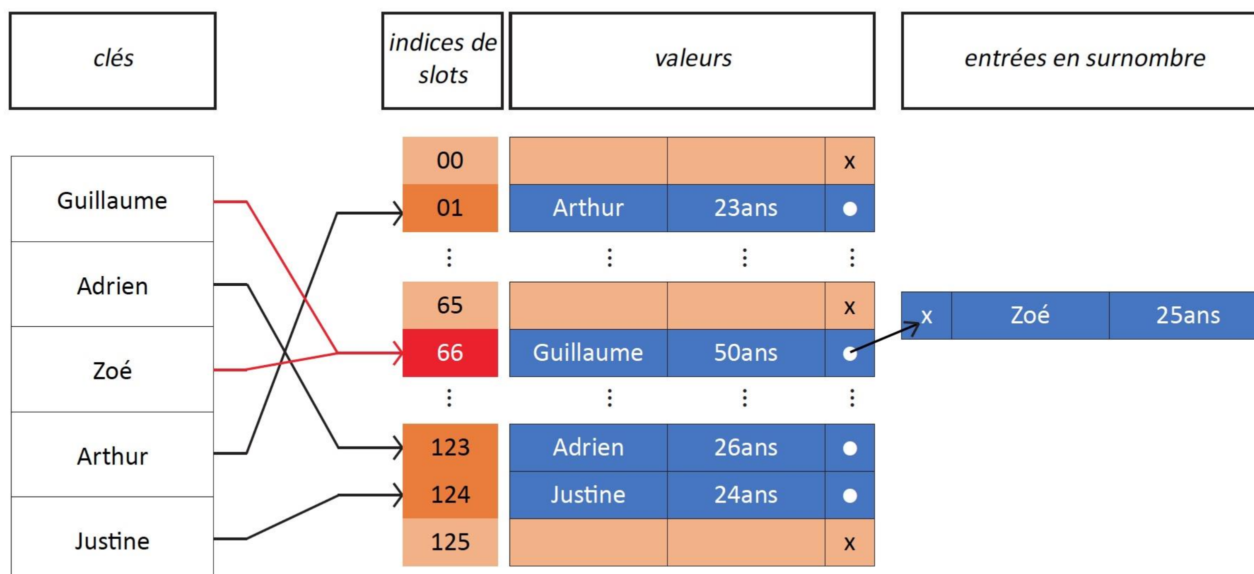


Figure 3.3 : Schéma d'une résolution de collision entre la clé Zoé et la clé Guillaume avec l'approche chainage.

La figure 3.3 illustre la méthode de résolution du chainage : Guillaume et Zoé obtiennent le même entier, 66, lors de la transformation avec la fonction de hachage, et donc partagent le même indice sur la liste des slots. Pour résoudre cela avec le chainage, la paire clé-valeur Guillaume-50 ans va être stockée dans le slot 152 (étant donné que c'est la première paire à être arrivée sur le slot) et un pointeur renverra la paire de clé Zoé-25 ans vers une liste chainée, liée à ce pointeur de cet indice précis.

Le **chainage devient un problème lorsque la liste chainée devient trop longue** et qu'il faut parcourir des centaines d'éléments (paires de clé-valeurs) pour accéder à la valeur recherchée. La performance des requêtes *search* diminue fortement avec la longueur de la chaine : **O(n)** avec *O la performance de la requête* et *n la longueur de la liste chainée*. La requête la moins performante se fera lors d'opération *search*. En effet, si l'opération se fait sur une chaine, il faudra parcourir l'entièreté de la chaine pour obtenir ce que l'on cherche. Ainsi, le temps maximum d'exécution d'une opération est égal à : 1 + la longueur de la plus grande chaine (c'est-à-dire n, le nombre d'éléments). Cela correspond au pire des cas lorsque la fonction de hachage renvoie toujours au même indice. Grace à la randomisation, une bonne fonction de hachage va distribuer *de manière uniforme* les éléments à stocker de manière à bien repartir toutes les clés dans notre tableau et ne pas créer de longue chaine (Demaine et Devadas, 2011).
Il est important de préciser que dans une liste chainée, on stocke la paire clé-valeur et pas uniquement les valeurs associées à ces clés. Ceci permet de retrouver, parmi la liste chainée, quelle valeur est liée à quelle clé. Dans l'exemple de la figure 3.3, lorsqu'on recherche la valeur de la clé '*Guillaume*', on utilise la fonction de hachage sur la clé *'Guillaume'* pour obtenir l'indice

correspondant. Ensuite la chaine de l'indice correspondant est parcourue jusqu'à la paire qui correspond à la clé '*Guillaume*' et la valeur de cette clé est récupérée.

L'avantage de cette technique de résolution est que la table de hachage ne devient que linéairement plus lente que le load factor (facteur de compression, correspond au nombre de paires de clés valeurs qui occupent une place dans la liste des slots (K) divisé par le nombre de place total de slots(n)) augmente.
La figure 3.4 ci-dessous compare la performance des requêtes (indirectement car au plus il y aura d'erreur de lookup au plus cela prendra du temps et au plus la performance sera impactée) entre la résolution de collision chainage et la résolution de collision open-dressing, avec l'approche linear probing. On voit que la performance du chainage diminue linéairement avec le pourcentage de remplissage de la table de hachage alors que pour l'adressage ouvert, la performance diminue exponentiellement avec le pourcentage de remplissage de la table de hachage. Nous pouvons en conclure qu'il est plus intéressant d'utiliser de la résolution d'adressage ouvert si le load factor est bas mais il est plus intéressant d'utiliser la résolution par chainage si le load factor est haut.

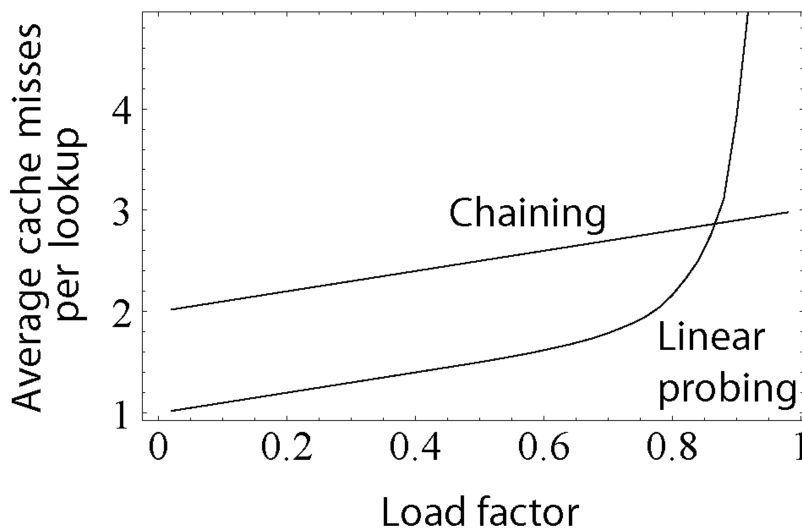


Figure 3.4 : Graphique comparant l'évolution de nombre d'erreur lors de la recherche d'un slot de deux méthodes de résolution de collisions (chainage et adressage ouvert) en fonction du load.

Précisons qu'en plus de pouvoir choisir la technique de résolution pour avoir de meilleures performances, il est également possible d'agir sur le load factor en agrandissant la table de hachage (ce qui est logique car Load factor = K/n, et si n augmente alors le facteur de compression diminue).

L'adressage ouvert :
Contrairement au chainage, tous les éléments sont stockés dans la table de hachage elle-même (et non dans un pointeur vers une liste chainée). Pour ce type de résolution, la table de hachage peut manquer de places pour stocker de nouvelles paires clés-valeurs et donc cela implique qu'il faut avoir une attention particulière à la taille de la table de hachage (n) et au nombre d'éléments présents dans la table de hachage (K) : **n doit toujours être plus grande que K** et le ratio (le facteur de compression) doit être pris en considération afin de garder une performance correcte de requêtes.
L'idée est que l'on va calculer l'indice d'une clé et si le slot correspondant à l'indice est occupé, nous cherchons un autre slot. Si ce nouveau slot est libre, la clé sera stockée au nouvel indice correspondant. Dans le cas contraire nous cherchons à nouveau un autre slot et ainsi de suite jusqu'à trouver un slot libre. C'est pour cette raison que la taille de la table de hachage doit toujours être plus grande que le nombre d'éléments présents dans celle-ci. Sinon, le SGBD ne pourrait jamais trouver de slot libre pour la nouvelle paire clé-valeur.

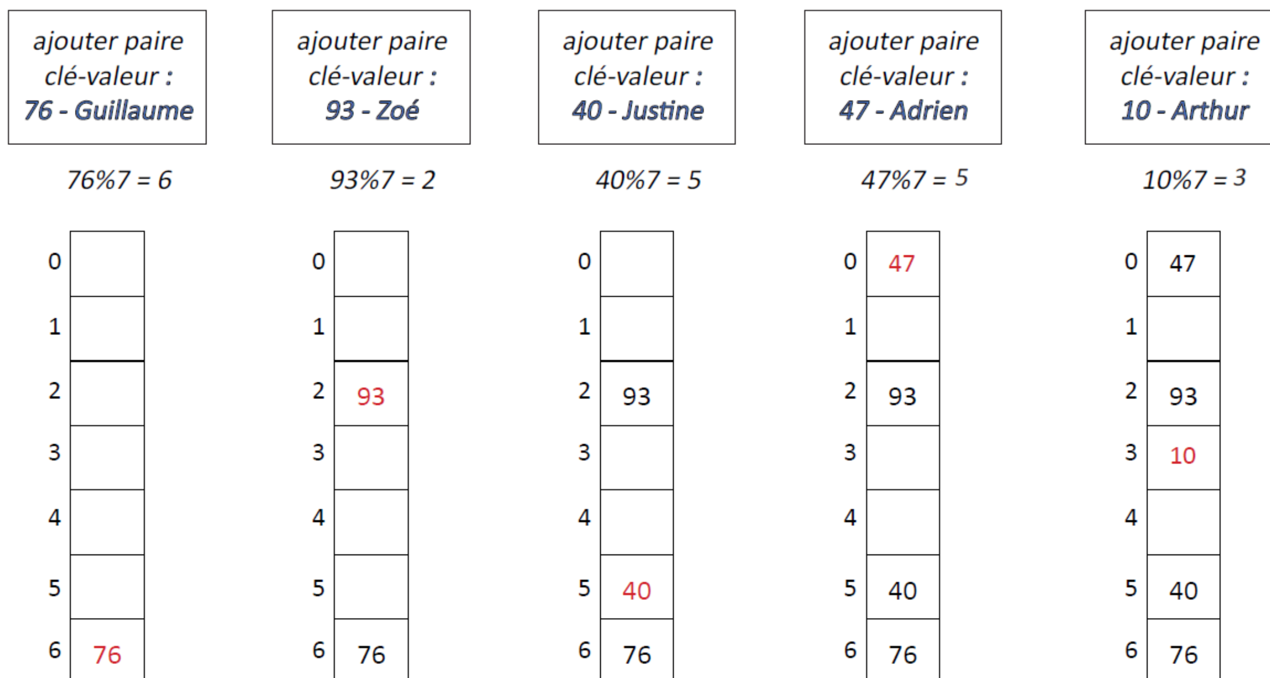


Figure 3.5 : Exemple d'un processus de résolution de collision avec l'adressage ouvert, suivant l'approche linear probing (GeeksForGeeks, 2021 a).

La figure 3.5 est un exemple de processus de résolution de collision avec adressage ouvert suivant l'approche linear probing. Supposons une base de données clé-valeurs, et nous voulons ajouter différentes paires de clé-valeur qui ont pour clé un chiffre et pour valeur un nombre. La fonction de hachage de la table de hachage est : le *numéro de la clé modulo 7*.
Pour la première paire, la clé est 76, nous faisons 76 modulo 7 et nous obtenons 6. Ainsi, pour la paire 76-Guillaume, l'indice du slot est 6 et cette paire est stockée dans le slot numéro 6 de la table de hachage.
Pour la seconde paire, nous faisons le même procédé et cette paire est stockée dans le slot numéro 2.
Pour la troisième paire, nous faisons le même procédé et cette paire est stockée dans le slot numéro 5.
Pour la quatrième paire, nous faisons le même procédé et cette paire obtient comme indice le chiffre 5. Ce slot est déjà occupé par la paire 40-Justine. Avec la méthode de résolution adressage ouvert, nous regardons si le slot suivant est libre (le slot numéro 6) : il ne l'est pas. Nous regardons donc le slot suivant, le slot numéro 1 (lorsque nous arrivons à la fin de la table de hachage, nous recommençons à partir du début de la table de hachage) : il est libre et nous y stockons la paire 47-Adrien.
Il existe différentes approches pour la recherche d'un nouveau slot :
- Le hachage linéaire (linear probing), comme nous venons de faire, consiste à regarder le slot suivant. Cette méthode est très simple mais s'il y a beaucoup de collisions, trouver une case libre peut prendre beaucoup de temps.
- Le double hachage. Si le slot est occupé, une autre fonction de hachage, de type multiplication, est utilisé pour trouver le slot suivant.
- Le hachage quadratique. Si le slot est occupé, une autre fonction de hachage, de type quadratique, est utilisé pour trouver le slot suivant.

L'adressage ouvert est plus performant que le chainage lorsque le facteur de compression est petit, comme on peut le voir sur la figure 3.4. Ceci est tout à fait logique car cela signifie qu'il y a beaucoup

plus de slots libres que de slots occupés et donc la probabilité de trouver un slot libre est plus grande. Cependant, lorsque celui-ci augmente et se rapproche de 1, l'adressage ouvert peut devenir très lent, il faut parcourir pratiquement toute la table de hachage pour trouver un slot libre. Pour avoir une bonne performance minimale, on conseille de garder le facteur de compression en dessous d'une certaine valeur lorsqu'on utilise l'adressage ouvert comme technique de résolution de collision.
Pour pallier ce problème de performance au fur et à mesure que le facteur de compression se rapproche de 1, on peut agrandir la taille de la table de hachage, ce qui va diminuer logiquement le load factor. Pour cela, il suffit de créer une nouvelle table de hachage plus grande et d'y copier les données de l'ancienne table de hachage (NewbeDev, 2021).
Il faut préciser que la suppression d'une paire clé-valeur dans une table de hachage qui suit la méthode d'adressage ouvert est difficile et ne peut pas se faire simplement en vidant le slot concerné de la table de hachage. Pour mieux le comprendre, considérons clés, c1 et c2 partageant le même indice avec la fonction de hachage avec c1 étant la première clé arrivée sur le slot et c2 la seconde. Etant donné que le slot de l'indice de c2 sera déjà occupé c1, le mécanisme d'adressage ouvert recherchera un autre slot vide où stocker c2. L'indice de la paire clé-valeur c2 va dépendre de l'indice de la paire clé-valeur c1 (étant donné que la recherche d'un slot libre se base sur le premier slot occupé). Si le slot correspondant à l'indice de c1 devient vide, la recherche de l'indice de la clé c2 sera différente et il ne sera plus possible de retrouver l'indice de la clé c2. Ainsi, pour supprimer une paire, il faudrait alors vider le slot de l'indice mais garder les chemins liés à l'indice de la clé x.

Chainage	Adressage ouvert
Facilité d'implémentation	L'implémentation nécessite plus de calculs
Il est toujours possible de rajouter de nouveaux éléments dans la table de hachage	La table de hachage peut se remplir au maximum et ne plus accepeter de nouveaux éléments
Moins sensible à la fonction de hachage et au facteur de compression	Plus sensible à la fonction de hachage et au facteur de compression
Utilisé lorsque le nombre d'éléments et la fréquence d'ajout et de suppression des éléments est inconnu	Utilisé lorsque le nombre d'éléments et la fréquence d'ajout/suppresion est connu
La performance de la mise en cache est mauvaise (certaines données sont stockées sous formes de listes, donc lecture plus lente)	La performance de la mise en cache est bonne (tout est stocké dans la même table)
Gaspillage d'espaces (certains slots de la table d'hachage peuvent ne jamais être utilisé)	Les indices des slots jamais transformé par la fonction de hachage peuvent être utilisés
Utilisation d'espaces supplémentaire avec les chaines listées	Utilisation de l'espace de stockage de la table de hachage

Tableau 3.1 : Tableau de comparaison entre l'adressage ouvert et le chainage (GeeksForGeeks, 2021 b).

Retenons que le point crucial d'une table de hachage est d'avoir une bonne fonction de hachage qui répartit **aléatoirement et uniformément** les clés dans l'index du tableau, de manière à **produire le moins de collisions possible**.

Enfin, pour la récupération de la valeur d'une clé, il suffira de hacher le nom de cette clé avec la fonction de hachage pour obtenir l'indice du slot de la table de hachage dans lequel la valeur est stockée. Si la table de hachage est bien dimensionnée, une opération *lookup* sur un élément est indépendante du nombre d'éléments présents dans la table de hachage.

3.3 Utilité et usage

L'intérêt de ce type de base de données est l'efficacité et la simplicité.
Ce type de SGBD peut gérer de très grands volumes de données et ainsi fournir un service pour des milliers d'utilisateurs simultanément, et ce avec une implémentation très simple. Sawant et Shah (2013) expliquent qu'elles sont particulièrement intéressantes pour les requêtes simples de lecture parce que les données n'ont pas de relation entre elles.
Précisons que la seule condition pour avoir recours à ce type de base de données est de connaitre les clés. *Si l'on ne sait pas où chercher*, c'est-à-dire que l'on ne connait pas les clés de la base de données, *ce type de base de données peut s'avérer inadéquat*. En effet, il n'est pas possible de faire des requêtes spécifiques sur une partie des données comme on pourrait le faire avec un « where » dans une base de données relationnelle.
Les bases de données clé-valeur sont également utilisées pour leur méthode de stockage. On les utilise pour le ***stockage en mémoire vive pour une mise en cache rapide*** qui permet d'accélérer la performance des applications comparé aux SGBD utilisant du stockage ***sur disque*** (Hazelcast, 2019). La mémoire en cache est une mémoire plus petite qui se situe entre les bases de données et les applications (du moins là où les données sont calculées) et qui a la particularité d'avoir des taux de réponses très rapide grâce au fait de stocker des données récemment utilisées ou souvent utilisés au plus près de l'endroit où se fait le calcul (GridGain, 2021). Lire des données sur une mémoire cache est plus rapide que lire des données sur un disque.
De manière générale, les bases de données key-value store sont utilisés pour les applications assez simple (requêtes non complexes) qui doivent stocker des données de tous types de manière temporaire.

L'exemple d'application la plus souvent présentée est un site internet de E-commerce avec des sessions d'utilisateurs et des paniers d'utilisateurs. La session de l'utilisateur sera la clé de la paire clé-valeur, et servira d'identification unique, et les éléments du panier de l'utilisateur seront les valeurs (la liste de valeurs) liées à cette clé.

Afin de mieux comprendre comment les données sont stockées sur un ordinateur, reprenons les explications de Emarcore et Kendrick (2016) à propos des types de mémoires d'ordinateurs. Pour fonctionner, un ordinateur a besoin de deux types de mémoires :

- **La mémoire vive (in-memory ou RAM)** : pour faire fonctionner un programme et conserver les données que l'on a besoin lors de l'utilisation de l'ordinateur. Les données de la mémoire vive sont supprimées lorsque l'ordinateur est éteint ou lorsque l'ordinateur échoue.
- **La mémoire de stockage (on disk)** : pour stocker les données même lorsque l'ordinateur est éteint ou échoue mais également pour stocker les programmes installés. Il y a de nouveaux 2 types de mémoires de stockage :
  1. Les disques durs mécaniques (HHD)
  2. Les disques à mémoire flash (SSD), qui sont plus rapides que les HHD mais qui sont plus chers (bien que le marché des SSD devient de plus en plus démocratique).

La mémoire vive est plus performante que la mémoire de stockage, cependant la mémoire de stockage a des capacités de stockage bien plus grand que la mémoire vive.
Il existe aussi la mémoire virtuelle qui est un espace sur la mémoire de stockage qui a été dédiée pour agir comme une mémoire vive.

3.4 Avantages

- **Simplicité et facilité d'implémentation**. La plupart des développeurs peuvent être rapidement et facilement familier avec une base de données clé-valeur étant donné que la plupart des langages de programmation proposent ce type de structure de données (via des hash ou des dictionnaires). De plus, le concept même de base de données en clé-valeur est facile à comprendre pour un « *développeur qui ne code pas* » également.
- **Performant et rapide** : Si la table de hachage est bien dimensionnée, une opération *lookup* sur un élément est indépendante du nombre d'éléments présents dans la table de hachage. Sawant et Shah (2013) s'accordent pour dire que la structure de ces bases de données permet d'avoir de bonne performance sur des opérations de lecture très grande.
- **Flexibilité sur les types de données** (tout type de donnée possible).
- **Extensibilité horizontale possible**, contrairement aux bases de données relationnelles qui ne permettent que l'extensibilité verticale (en pratique l'extensibilité horizontale est possible mais très compliquée à mettre en place). Comme vu dans l'introduction, à l'ère du big data, l'extensibilité horizontale est indispensable et essentielle.
- **Hautement divisible** : grâce à la simplicité schématique, la distribution (réplication et sharding) de la base de données est plutôt facile à mettre en place.
- **Fiabilité**, grâce à *la redondance intégrée*. « *En informatique, la redondance désigne le fait que des données identiques soient disponibles dans différentes bases ou gisements de données. La redondance volontaire trouve son intérêt dans la protection, la sauvegarde, la sécurité et la cohérence des données en cas de défaillance d'un des serveurs hébergeant les données* »(Talend, 2019).
- **Coût de stockage plus petit**. La plupart du temps, les bases de données key-value utilisent moins d'espace de données que les SGBDR, pour stocker la même quantité de données (Foote, 2020) ce qui mène souvent à de meilleures performances.

3.5 Désavantages

- Le **manque de schéma** peut entrainer des problèmes de désordre et de gestion en data management. De plus, Nayak, Poriya et Poojary (2013) relèvent que le manque de schéma *ne permet pas de faire obtenir des vues customisées* (*materialized views*) sur les données.
- **Simplicité relative**. Selon le type de SGBD choisi, la gestion des structures de données et des requêtes peut être plus ou moins efficiente. Typiquement, la base de données clé-valeur la plus basique *ne permettra pas de filtrer ou contrôler les données retournées* d'une requête.
- Il n'y a **pas de langage de requêtes standard**, ce qui implique qu'il n'est pas possible de transposer des requêtes d'un SGBDR. Cependant, il est tout de même possible de manipuler SQL ou d'autres moyens à l'aide de quelques ajustements (Raoul, 2021).
- Les valeurs ne peuvent **pas être filtrées par défaut**.

3.6 Applications

Comme nous l'avons vu plus haut, les key-value stores sont particulièrement intéressants pour les applications qui ont besoin de stocker des données de manière temporaire. C'est pour cette raison qu'elles conviennent très bien pour tout type de sites internet qui ont recours à des sessions d'utilisateurs dans leur service.
Amazon Web Service illustre l'utilisation d'une base de données clé-valeur, tel que leur produit DynamoDB, au travers de deux exemples (Amazon Web Services, 2018) :

1. **Magasins de sessions** : L'application ouvre une session lorsque l'utilisateur se connecte et ferme la session lorsque l'utilisateur se déconnecte ou qu'elle expire. Le temps de la session, toutes les données de l'utilisateur liées à cette session (informations sur le profil de l'utilisateur, la boite de messagerie de l'utilisateur, des promotions ciblées, etc.) sont stockées dans une base de données clé-valeur où la session d'utilisateur possède un identifiant unique qui est la clé et dont la valeur est l'ensemble des données liées à cette session. Selon AWS, en général, les frais généraux par page de ce type de SGBD sont inférieurs à ceux associés à des SGBDR.

Cet exemple de use-case montre bien l'utilité de pouvoir stocker et accéder *rapidement* à des données stockées *temporairement*.

2. **Panier d'achat :** Il y a certaines périodes de l'année (par exemple : *le Black Friday, ou à plus petit échelle les jours de week-ends*) où un site Web d'e-commerce reçoit énormément de commande (des milliards en quelques secondes) en dans un court laps de temps. La base de données doit pouvoir être disponible pour l'ensemble de ces commandes. Techniquement parlant, cela signifie que la base de données doit avoir une haute disponibilité pour pouvoir répondre à des milliards de requêtes en même temps. Avec leur extensibilité et la gestion de la distribution des données sur plusieurs serveurs (nécessaire et obligatoire pour pouvoir garder une performance suffisante pour répondre à des milliards de requêtes), les bases de données clé-valeur sont adéquates pour répondre à ce type de besoin.
   Cet exemple de use-case montre bien l'utilité d'avoir recours à une base de données clé-valeur lorsqu'il s'agit de *garder de bonnes performances lors de grands pics de trafic* (des milliards de requêtes simples en quelques secondes).
3. **Panier d'achat** (Raoul, 2021) : Pour les transactions financières lors du paiement d'une commande sur un site Web d'e-commerce, il est plus judicieux d'utiliser un SGBDR pour les raisons de consistance et de sécurité vues précédemment. Cependant, il y a généralement plus de cas où le client a un panier mais ne procède pas directement au paiement que de cas où le client paie directement la commande de son panier. Les paniers d'achats, ne nécessitant pas la consistance et la sécurité d'une transaction d'achat, ne doivent pas être stocké dans un SGBDR mais plutôt dans une base de données clé-valeur car celle-ci sera plus efficaces (le stockage en paire clé-valeur sera rapide pour enregistrer et obtenir des données) et fiable (grâce à la redondance des données).
   Cet exemple de use-case montre l'intérêt d'utiliser une base de données clé-valeur pour avoir des *performances rapides sur des données moins sensibles* à la consistance ou la sécurité.

3.7 Acteurs principaux

Attention, il est important de préciser que parmi les différents SGBD clé-valeur, tous ne sont pas pareils et peuvent se différencier sur le modèle de consistance ou encore sur le modèle de stockage (en mémoire ou sur disque).
Selon le classement fourni par DB Engines, les deux bases de données orientées clé-valeur les plus cotées sont Redis (open-source) et DynamoDB (Amazon service) et en troisième place CosmosDB (Azure service, propriété de Microsoft, qui est une base de données multi-modèles).

REDIS

Redis (qui signifie Remote Dictionary Serve) est une base de données NoSQL clé-valeur **open-source, en mémoire** et **distribué**. Redis a été écrit en langage C et fonctionne dans la plupart des systèmes d'exploitation tels que BSD, Linux, OS X. Redis supporte la réplication maitre-esclave. Redis offre de très grandes performances qui se traduisent par la possibilité de faire des millions de requêtes par seconde avec des temps de réponse inférieurs à la milliseconde. Ces hautes performances fait de Redis un SGBD très utilisé pour les applications en temps réel dans les domaines tels que les jeux, l'IoT, les réseaux sociaux, les services financiers ou encore l'industrie de la santé. Redis est un choix populaire pour sa mise en cache, sa gestion des sessions, l'analyse en temps réel et le streaming multimédia. Il prend en charge de nombreuses structures de données telles que les chaînes de caractères, les hachages, les listes, les ensembles, les bitmaps, les hyperlogs et les index géospatiaux (Namuag, 2021).

Il y a de nombreuses grandes compagnies qui utilisent Redis : Twitter, GitHub, Pinterest, Craigslist, StackOverFlow, Vodafone, TripAdvisor, Nokia, etc.

DYNAMODB

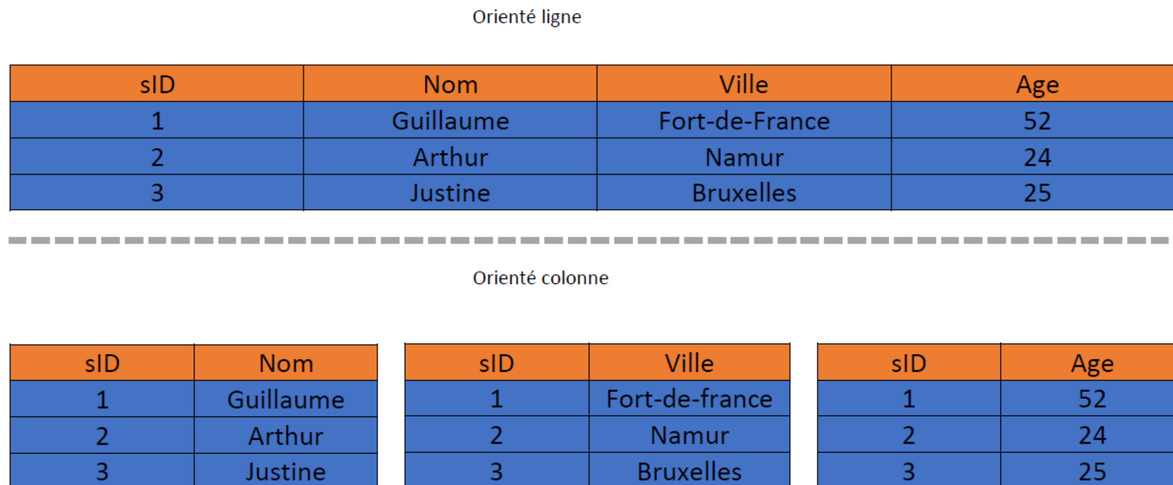
Amazon décrit DynamoDB comme « […] une base de données clé-valeur NoSQL entièrement gérée et sans serveur, conçue pour exécuter des applications hautes performances à n'importe quelle échelle. DynamoDB offre une sécurité intégrée, des sauvegardes continues, une réplication multi-régions automatisée, une mise en cache en mémoire et des outils d'exportation de données.
C'est un service de base de données NoSQL rapide et flexible pour des performances de latence de l'ordre de quelques millisecondes à n'importe quelle échelle.» (Amazon Web Services, 2018).

Pour DynamoDB, il y a également de grandes compagnies qui utilisent cette base de données : Netflix, Zoom, Dropbox, Disney, Amazon, The NewYorkTimes, Lyft, BitPanda, etc.

4.Base de données orientée colonnes

4.1 Introduction

Historiquement, les données ont toujours été stockées dans des tables relationnelles orientées sur le modèle de lignes. Fowler et Sadalage (2013) expliquent qu'utiliser des lignes comme unité de stockage est particulièrement intéressant pour les performances d'écriture. Avec ce modèle, les applications doivent lire les données de gauche à droite à la manière d'un livre.


Orienté ligne

sID	Nom	Ville	Age
1	Guillaume	Fort-de-France	52
2	Arthur	Namur	24
3	Justine	Bruxelles	25

Orienté colonne

sID	Nom
1	Guillaume
2	Arthur
3	Justine

sID	Ville
1	Fort-de-france
2	Namur
3	Bruxelles

sID	Age
1	52
2	24
3	25

Figure 4.1 : Comparaison schématique entre stockage en ligne et stockage en colonne (Bog et al., 2008).

Comme vu précédemment, la technologie a transformé le domaine de la donnée et les données ont commencé à affluer en masse, et la notion des 3V du big data a émergée. Avec l'énorme quantité de données qui est apparue, ce modèle de lignes a commencé à avoir des problèmes de performance de lecture lorsqu'il s'agissait de passer en revue des milliards de données à travers des lignes (Kumar, 2021).
Un autre modèle de stockage de données s'est alors distingué : le modèle colonne, lorsque l'on utilise des colonnes comme unité de stockage, utilisable en SQL et en NoSQL. Fowler et Sadalage (2013) font remarquer que le stockage en ligne améliore les performances en écriture et que le stockage en colonne lui est très efficace lorsque les opérations d'écriture sont rares et que les opérations de lecture sont fréquentes et se font sur des colonnes entières. Meier et Kaufmann (2019) suggèrent que les opérations de lectures sont plus efficaces lorsque les données sont stockées en colonnes plutôt qu'en lignes. Ils expliquent qu'en pratique, la lecture de toutes les colonnes d'une instance en particulier se fait très rarement et qu'en réalité les opérations de lectures les plus fréquentes se font sur des groupes de colonnes. Pour cette raison, stocker les données en colonne permet d'accéder au contenu des groupes de colonnes plus rapidement que si les données étaient stockées en lignes.

Comme nous l'avons vu plus haut, les bases de données clé-valeur ont de bonnes performances pour traiter de grandes quantités de données cependant leur structure rudimentaire peut manquer d'un schéma pour mieux traiter les données. Les bases de données orientées colonnes améliorent l'idée des bases de données clé-valeur en leur ajoutant une structure, *flexible*, basée sur le modèle de colonnes. Dans ces bases de données, les données ne sont pas stockées dans des tables relationnelles mais dans des espaces structurés multi-dimensionnels. Dans les bases de données colonnes, les **familles de colonnes** (Column-Family) sont les *seules contraintes schématiques obligatoires*. Il est important que préciser que ces **contraintes schématiques sont beaucoup plus flexibles** que celles des bases de données relationnelles.

L'idée derrière le stockage en colonne est double. Premièrement, l'amélioration de la performance des requêtes en faisant les requêtes sur les colonnes directement plutôt que de passer en revue chaque colonne de chaque ligne. Deuxièmement, le stockage en colonne permet de compresser les données. Raichand et Aggarwal (2013) expliquent que la compression améliore la performance des

requêtes analytiques (permettant au CPU de faire des opérations directement sur les données compressées) et améliore l'optimisation du stockage des données. Ces techniques de compressions ont révolutionné la manière de concevoir le stockage en mémoire. En effet, grâce à celles-ci, on a pu stocker de grandes bases de données multi-TB dans un stockage en mémoire alors qu'auparavant ces tailles de données de stockage étaient uniquement stockées sur disque (Kumar, 2021).

Prenons par exemple une base de données répertoriant des bières avec 10.000 bières (10.000 lignes) avec 50 attributs (50 colonnes). Nous demandons à la base de données d'extraire le nom des bières de toute les bières blondes.
Le modèle traditionnel avec les lignes passe en revue et retourne chaque colonne de chaque ligne correspondant aux valeurs des deux colonnes de la requête, *Nom de la bière* et *Type*. Les données sont lues de gauche à droite ligne après ligne. Le résultat retourné comprend donc les valeurs des deux colonnes concernées mais également toutes les autres valeurs des 48 autres colonnes de chaque ligne retournée. Retourner un plus grand nombre de données que nécessaires peut devenir très contraignants pour les bases de données avec de très grands volumes.
Le nouveau modèle avec les colonnes passe en revue les deux colonnes concernées et retourne uniquement les colonnes des lignes correspondant aux valeurs des deux colonnes *Nom de la bière* et *Type*. Les données sont lues de haut en bas colonne après colonne. Chaque instance d'une colonne est liée avec un rowID de manière à savoir relier les colonnes d'une même instance ensemble.

Pour résumer, avec ce modèle, il est possible de lire une seule colonne directement pour avoir une information sur une entité sans à avoir à lire toutes les informations de cette entité. De cette manière, la performance, grâce au stockage et la recherche en colonne, va être améliorée grandement.
Techniquement parlant, la différence entre les deux modèles est au niveau de l'I/O (entrées/sorties, ensemble des informations échangées entre une machine et les différents périphériques à laquelle elle est connectée). Pour les modèles en ligne, le I/O sera moins précis (on lit des informations non pertinentes par rapport à la requête) et donc plus grand que pour les modèles en colonne, ce qui explique pourquoi la performance sera meilleure pour le second modèle.

Une base de données NoSQL orientée colonne stocke les données dans des colonnes flexibles qui peuvent être distribué sur plusieurs serveurs et qui utilise une cartographie multidimensionnelle à travers les lignes, les colonnes et le timestamp. Plus simplement, une base de données orientée colonne peut être interprétée comme une base de données clé-valeur à deux dimensions.

4.2 Fonctionnement

Les bases de données orientées colonne utilisent le concept de Keyspace. Ce concept est comparable à un schéma dans le modèle relationnel et est utilisé pour donner une représentation de la structure de la base de données.
Un keyspace (figure 4.2) contient toutes les familles de colonnes (figure 2.16), comparable à des tables dans le modèle relationnel, et chacune des familles contient des lignes (figure 2.17) qui contiennent elles-mêmes des colonnes. Les rowkeys du schéma de la figure 2.16 sont id_42, id_2456, id_2434. Ce sont des identifiants uniques et doivent être choisi de manière à rester unique.

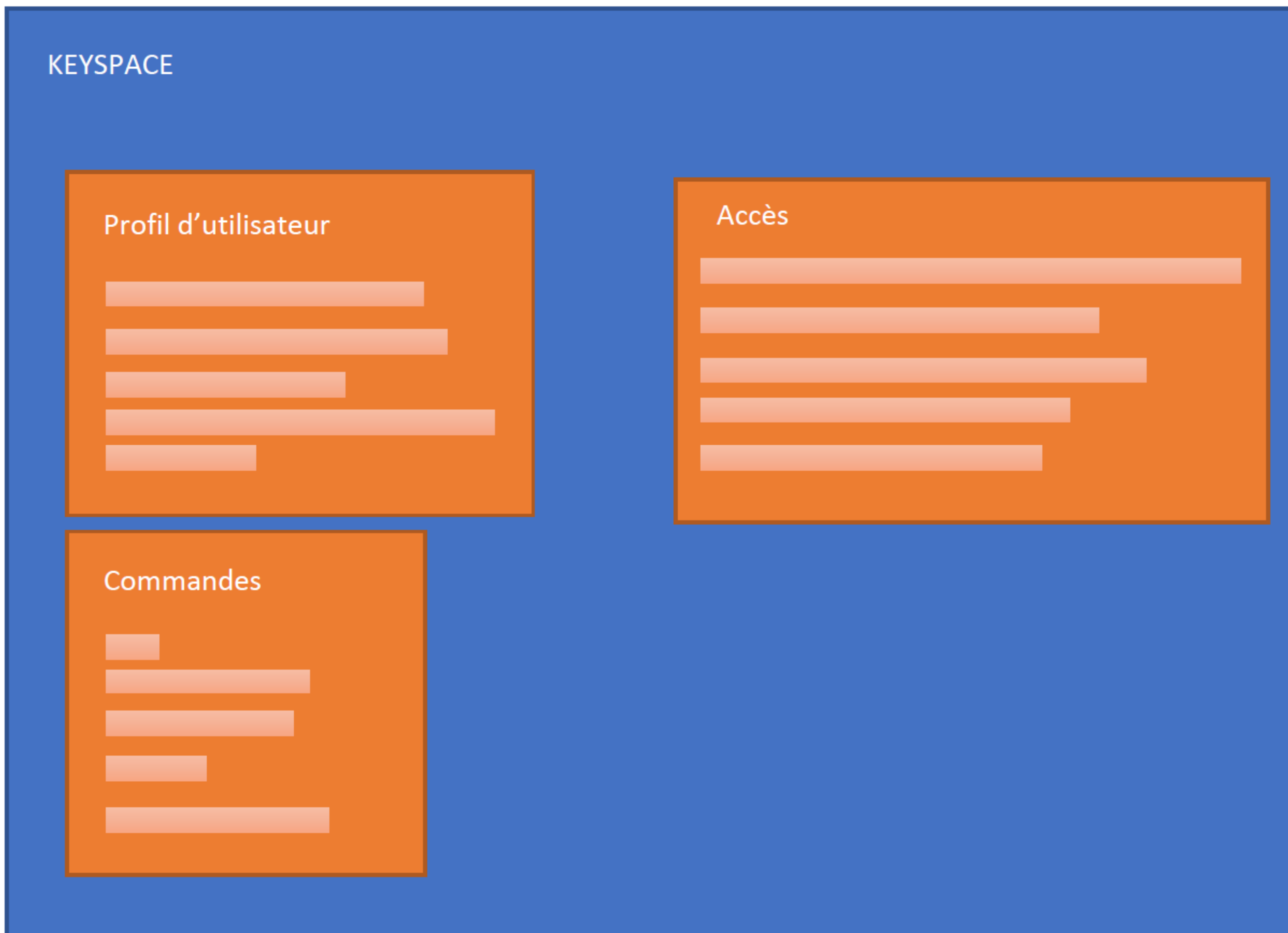


Figure 4.2 : Schéma d'un keyspace avec 3 familles de colonnes (Dehdouh et al. 1999).

Lorsqu'on descend en granularité dans le schéma, on trouve les lignes. Chaque ligne possède un « rowkey » (=rowID) qui lui sert d'identifiant unique et une ou plusieurs colonnes. Chaque colonne d'une même ligne est liée à ce rowkey spécifiquement. L'ajout d'un timestamp pour le stockage d'une donnée est utilisé pour déterminer la version la plus récente de la donnée, que l'on utilise pour la vérification de la consistance des données. En effet, la réplication de données dans une base de données distribuées peut mener à différentes versions sur différents nœuds, et avoir un timestamp des données permet d'assurer la consistance (Ian, 2016).

Au sein d'une même famille de colonnes, il peut y avoir plusieurs lignes et chaque ligne peut avoir **son propre format** (*nombre de colonnes indépendants*, *nom des colonnes indépendants*, *types des colonnes indépendants*, etc). Chaque valeur est accompagnée d'un timestamp. L'exemple sur la famille de colonne Profil d'Utilisateur de la figure 4.3 illustre la liberté de formats parmi les différentes lignes. En effet, le nombre de colonnes est indépendants (id_2456 possède 3 colonnes et les deux autres lignes en possèdent 4) et le type du contenu des colonnes est indépendants (le type de la colonne 'genre' est un 'string' et celle de la colonne âge un 'int'). La seule obligation est de posséder un rowID unique. Dans une même famille de colonnes, il est possible d'ajouter dynamiquement de nouvelles colonnes. Le timestamp de l'exemple de la figure 4.3 illustre que l'ajout de nouvelles colonnes, après que la ligne a été créée, est faisable : le id_2434 possède deux timestamp différents pour la colonne 'nom' et la colonne 'profession'.
C'est pour ces raisons que ces bases de données sont considérées comme très flexible. De plus, l'absence de schéma obligatoire parmi les lignes d'une même famille permet d'éviter le gaspillage d'espaces de stockage. En effet, dans un SGBDR chaque ligne doit avoir une valeur pour chaque colonne et donc une ligne peut se voir attribuer la valeur NULL ou par défaut. Alors que pour les bases de données orientées colonnes il n'est pas obligatoire que chaque ligne ait une valeur pour chaque colonne, ce qui signifie qu'il n'y a pas d'espace de stockage gaspillé inutilement. Bien qu'il n'y ait pas d'obligation de schéma explicit, il y a tout de même un schéma implicite à l'intérieur des familles de colonnes afin de rassembler les données liées ensemble et ainsi pouvoir faire des

requêtes directement sur un ensemble de données liées, qui correspondent aux agrégats dont nous avons parlé plus tôt. Les schémas implicites sont les colonnes qui sont liées et qui sont rassemblés dans des familles de colonnes. Nous l'avons vu lors de l'introduction : les bases de données colonnes sont orientées-agrégats. L'exemple de la figure 4.3 en est une illustration, la famille de colonne Accès pourrait avoir comme colonnes : mots de passe ; nom d'utilisateurs ; contenu autorisé ; etc.

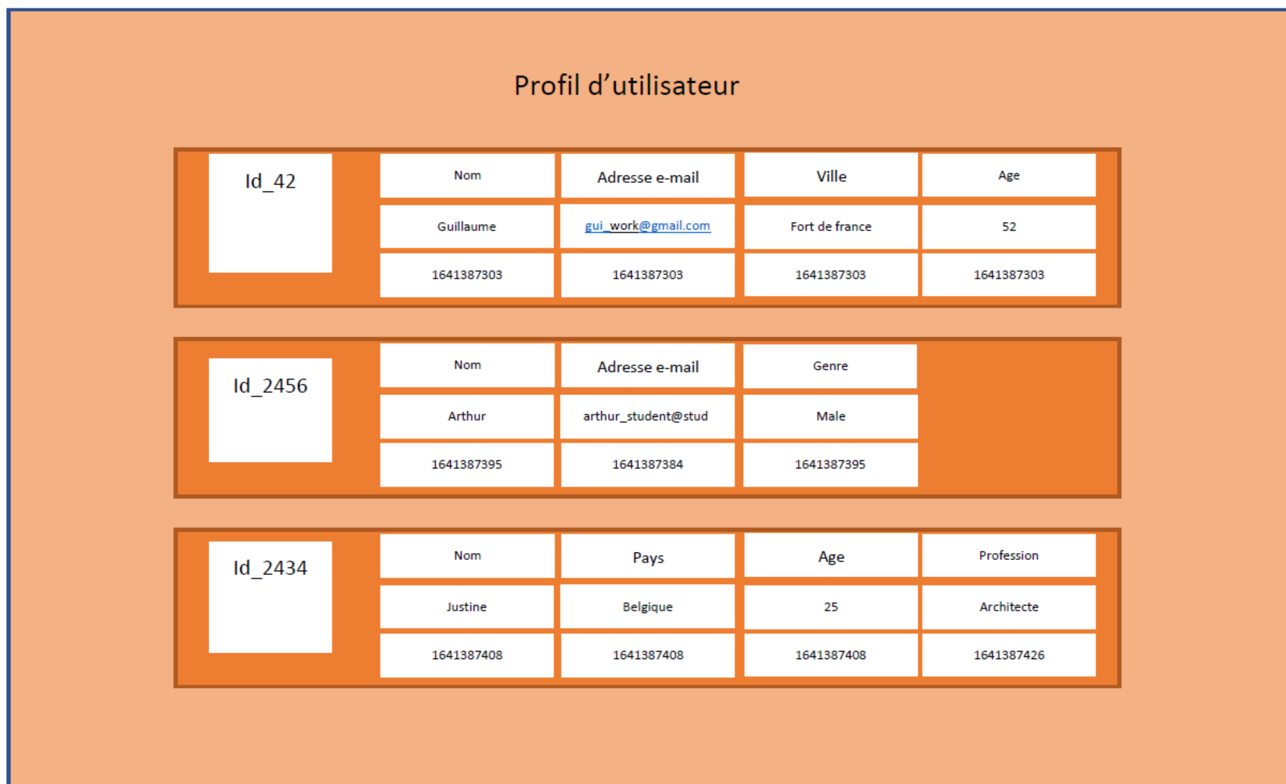


Figure 4.3 : Schéma d'une famille de colonne, Profil d'utilisateur, qui comprend les colonnes liées au profil d'un utilisateur. Trois lignes sont représentées (Dehdouh et al. 2015).

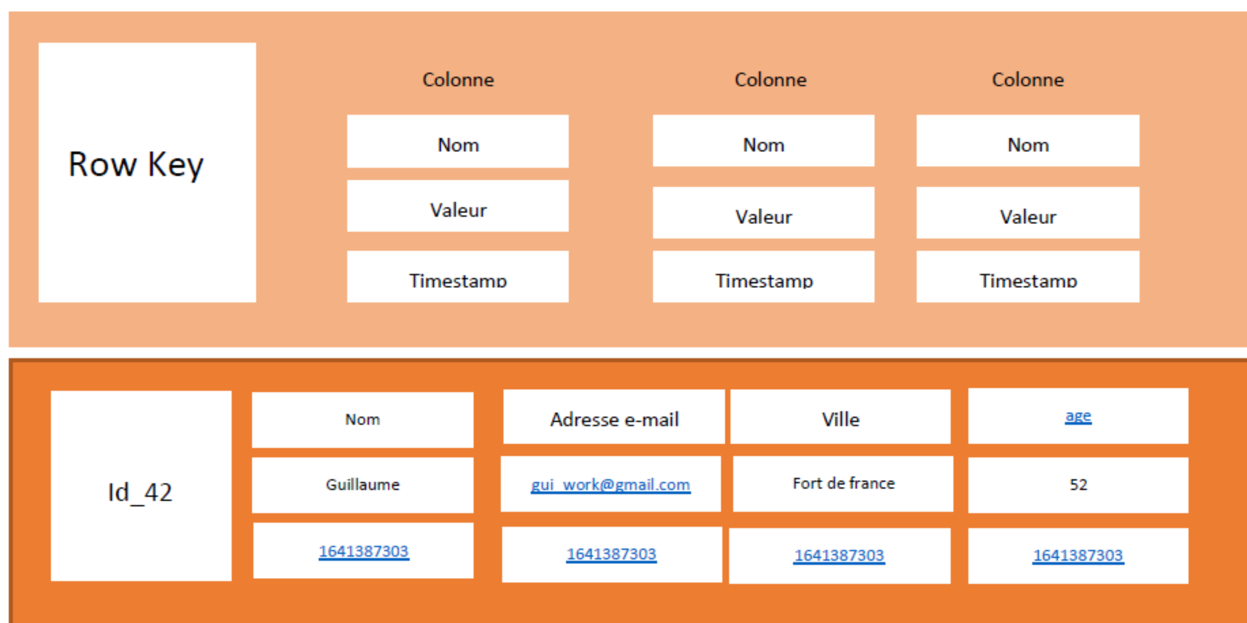


Figure 4.4 : Schéma théorique (au-dessus) d'une ligne d'une famille de colonne avec le schéma d'un exemple concret (en-dessous) d'une ligne d'une famille de colonne (Dehdouh et al. 2015).

Pour comprendre le fonctionnement de ces bases de données, Fowler et Sadalage (2013) proposent de voir ce modèle comme une structure d'agrégats à deux niveaux :

1. Le premier niveau est le rowkey, pour choisir l'agrégat d'intérêt. Ce premier niveau pourrait correspondre à une base de données clé-valeur : une clé lié à des données (tout type de données possibles).
2. Le deuxième niveau est représenté par les colonnes, que l'on choisit pour avoir des informations plus précises sur l'agrégat qui nous intéresse. Le deuxième niveau a un double intérêt car il permet d'avoir des informations précises sur une ligne d'une colonne en particulier ou permet d'avoir des informations sur une ligne à propos d'un agrégat (ensemble de colonnes).

Chaque colonne doit faire partie d'une famille de colonnes unique et la colonne fonctionne comme une unité de stockage sur laquelle on peut avoir accès pour faire des opérations.
On peut avoir accès aux instances avec les rowkey, ensuite on peut avoir un accès plus précis de l'instance en utilisant les clés des colonnes. Le seul schéma fixe de la base de données est le schéma référant aux familles des colonnes. Meier et Kaufmann (2019) expliquent que pour des architectures distribuées ou fragmentées, il est préférable de stocker au même endroit physique les données d'une même famille, pour optimiser le temps de réponse.

Dans leur travail sur la compression des données, Raichand et Aggarwal (2013) nous montrent que la compression des données permet de réduire le coût I/O (input/output) mais cela est contrebalancé avec les coûts induits de compression et de décompression. Boncz, Manegold et Kersten (1999) suggèrent que le compromis entre I/O et coûts de compression et décompression penche en faveur de la compression lorsque la vitesse du CPU augmente. Raichand et Aggarwal (2013) détaillent différentes techniques de compression dont les populaires techniques *Compression par dictionnaire* et la *RLE (Run-Length-Encoding*). La compression des données est la méthode permettant de transformer une suite de bits G, qui est grande, en une suite de bits P, qui est plus petite que G, en restituant les mêmes informations sans pertes ou avec peu de pertes d'informations. La décompression des données est la méthode inverse qui permet de récupérer une suite de bits G, qui est grande, à partir d'une suite de bits P, qui est plus petit, par l'utilisation d'un algorithme de décompression.
La **compression RLE** est populaire et efficace. Le principe est simple : toute suite de bits ou de caractères identiques est remplacée par un triplet : valeur - position de départ - longueur. La valeur correspond au caractère identique qui se suit. La position de départ correspond à la position où la suite commence. La longueur correspond au nombre d'occurence du caractère identique.

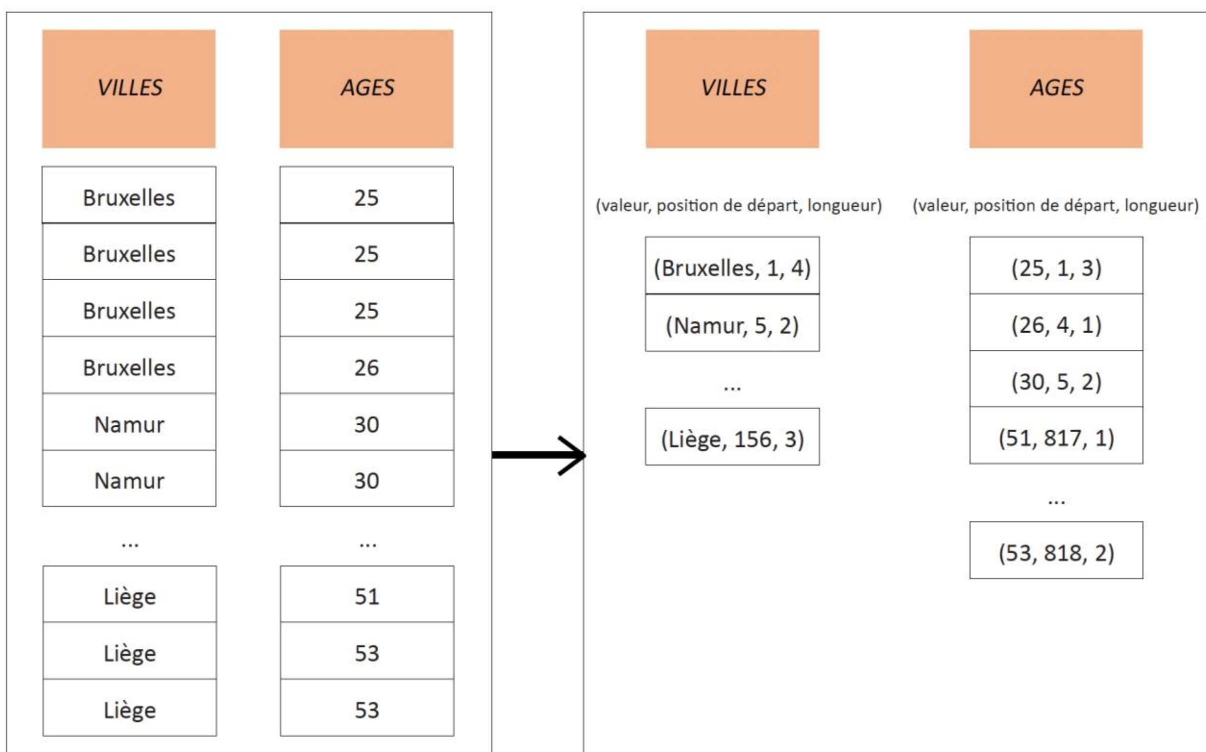


Figure 4.5 : Exemple de compression RLE pour la colonne Age et la colonne Villes dans une base de données (Cleve et Faulkner, 2021).

4.3 Utilité et usage

Ce type de base de données est principalement utilisé des applications de BI et d'analyses (OLAP) : pour du datawarehousing, pour du CRM, etc. (Ian, 2016).
On les utilise pour leur efficacité et leur rapidité dans les tâches analytiques à grande échelle.
Rappelons que pour les taches de transactions financières et l'OLTP, une approche orienté ligne, et donc une base de données relationnelle est le bon choix. Pour les taches d'analyse de données avec des agrégats, il faut plutôt se tourner vers les bases de données orientées colonnes car leur structure est optimisée pour une récupération rapide des données. Résumons comment ce type de SGBD a de meilleure performance en opérations de lecture :

1. La compression de données permet la réduction des besoins en E/S (entrées-sorties).
2. Les bases de données orientées colonne sont particulièrement adéquate pour la compression parce que les données sont stockées en colonnes et les données d'une même colonne sont homogènes et similaires. On comprend facilement comment une compression de type RLE s'appliquent particulièrement bien sur une colonne avec des données similaires. *Par exemple, dans une application d'E-commerce, on peut réduire la quantité de données à lire lorsqu'on a des centaines de produits avec le même nom.*
3. De par la possibilité de compresser facilement les données, les bases de données orientées colonnes favorisent la réduction du E/S et donc permettent d'augmenter le débit, qui se traduit par de meilleures performances de requêtes.

4.4 Les bases de données orientées colonnes sont-elles NoSQL ?

Il n'y a pas de consensus parmi les scientifiques à propos du fait que les bases de données colonnes soient NoSQL ou non. Certains vont promouvoir ces bases de données comme étant NoSQL en pointant leurs différences avec les bases de données relationnelles. D'autres, comme Steve Sarsfield, considèrent que la plupart du temps les bases de données orientées colonnes sont conciliants avec le SQL et le modèle ACID (contrairement à la plupart des bases de données NoSQL) et que donc elles ne seraient pas NoSQL (Ian, 2016). Notre position est de considérer les bases de données orientées colonnes comme NoSQL étant donné qu'elles ont la plupart des caractéristiques : flexibilité, extensibilité, performance sur de grands volumes, réplication, etc.

4.5 Type de stockage

La particularité des bases de données colonnes est qu'elles stockent **les données physiquement (sur le disque dur) en colonnes**, et cela permet d'améliorer la performance des requêtes.
Pour comprendre cela (GeeksForGeeks, 2021c) il faut savoir que les données sont stockées sur des plateaux magnétiques et que pour récupérer une donnée, il y a une *tête* qui se déplace sur les plateaux magnétiques des disques durs pour répondre aux opérations de lecture ou d'écriture : *au plus la tête doit se déplacer physiquement* sur les plateaux magnétiques, *au plus ça prend du temps pour répondre aux opérations*. Ainsi, organiser efficacement la manière dont les données sont stockées sur les plateaux magnétiques permet d'améliorer la performance des requêtes.
Dans une base de données avec **un modèle en ligne**, les données seront stockées physiquement en ligne où **chaque colonne d'une ligne est stocké dans un bloc** qui vient juste après la colonne qui précédait. Cela va être intéressant pour les applications de *transactions* car potentiellement, lors de ce type d'opérations, *on va modifier chaque colonne de chaque ligne les unes à la suite de l'autres* et donc il y a du sens de les stocker physiquement les unes à la suite de l'autre sur le disque dur.

Dans une base de données avec **un modèle en colonne**, les données seront stockées physiquement en colonne, où **chaque bloc contient le contenu d'une colonne** (ou du moins la suite du contenu d'une colonne). L'avantage avec ce type de stockage est qu'on a accès à des données qui sont liées et qui sont *déjà groupées ensemble physiquement sur le disque*. De plus, ce type de stockage évite de devoir récolter des informations qui n'étaient pas pertinentes pour l'opération.
A nouveau, la performance des requêtes est également améliorée grâce à la compression des données qui est facilitée par le stockage en colonne et qui permet de réduire la quantité de données à lire sur le disque. Le fait de stocker les données en colonne permet de facilité la compression des données.

4.6 Avantages

- **Facilité de compression des données** : optimisation des ressource utilisés sur le disque pour stocker de grandes quantités de données, c'est-à-dire réduit le coût de stockage (Kumar, 2021) **.**
- **Bonne performance** sur des requêtes impliquant de très grandes quantités de données et nécessitant un temps de réponse rapide, grâce aux données regroupés en agrégat et le stockage en colonne (réduction des coûts I/O lors de l'exécution des requêtes) (Kumar, 2021).
- **Extensibilité horizontale**, qui peut être pratiquement infinie en ajoutant simplement de nouvelles machines dans le cluster. L'extensibilité horizontale implique également que ces bases de données fonctionnent bien pour les **MPP (massive parallel processing)**, ce qui va aussi améliorer la performance des requêtes.
- **Flexibilité** : pas de schéma fixe et pas de symétrie obligatoire entre les différentes colonnes. De plus facilité d'ajout de nouvelles colonnes sans perturber la base de données. C'est un attribut très important pour les développeurs. La plupart du temps, après la conception de la base de données, il faut rajouter un ou des attributs (c'est-à-dire une nouvelle colonne) pour le développement d'un nouveau service ou simplement l'amélioration de la base de données. Avec le modèle relationnel, comme nous l'avons vu, toutes les lignes devront avoir une valeur pour ce nouvel attribut, et cela entraine un gaspillage de l'espace de stockage.
- **Accès très rapide aux éléments pertinents** de la base de données : on peut faire des requêtes directement sur les colonnes qui sont pertinentes.
- **Haute disponibilité** grâce à l'architecture distribuée de ces bases de données, souligné par Meier et Kaufmann (2019).
- **Lecture des données plus rapide que avec le modèle en lignes** :
  En utilisant du SQL avec un modèle en ligne, le SGBD doit analyser toute la base de données de la première ligne à la deuxième ligne et passe en revue toutes les colonnes même celles qui ne sont pas concernées par la requête, il y a un gaspillage de temps.
  En utilisant un modèle en colonne, le SGBD peut sélectionner la colonne ou les colonnes concernées par la requête et va uniquement analyser celles-ci sans se préoccuper des autres colonnes. Ainsi, il y a un gain de temps entre les deux modèles de stockage de données (Baez, 2021).
- **Gain d'espaces grâce à la compression**

4.7 Désavantages

- **Les requêtes d'écriture et de mise à jour sont lentes**. Pour cette raison, on évite d'utiliser ce type de bases de données pour faire de l'OLTP.
- **La manipulation de lignes entières est lente**. En effet, il faut aller chercher chaque information dans chaque colonne de la ligne concernée. Généralement, on les utilise lors d'application où l'on veut faire des requêtes sur des agrégats de données et non sur une seule instance de la base de données (Kumar, 2021).

4.8 Applications

Sawant et Shah (2013) présentent un cas d'utilisation concret d'une base de données orientées colonnes. Supposons qu'une organisation veuille un moteur de recherche très performant qui puisse à la fois proposer des outils de reporting mais également qui puisse gérer un grand volume de données qui grandit continuellement. D'un point de vue strictement orienté base de données, une base de données NoSQL orientée colonnes telle que Cassandra serait adéquate, car ce type de SGBD fournirait un accès aux données très rapide et permettrait de faire du traitement en temps réel. De plus, ce type de SGBD est flexible (adapté pour une quantité de données qui évolue) et extensible horizontalement (adapté pour les très grands volumes de données). D'un point de vue plus global, la solution architecturale du problème de l'organisation serait un stockage des données sur Hadoop (pour supporter la distribution du NoSQL Cassandra), l'utilisation du SGBD Cassandra et enfin une solution orienté Machine Learning pour les outils de reporting.
De manière plus générale, les applications typiques de ces SGBD sont :

- Les applications liées aux données de Log (intéressant grâce à la présence des timestamp) ; les données de IoT ; les données de type time-series.
- Nayak et al. (2013), comme d'autres dans la littérature, déclare que les bases de données orientées colonnes sont particulièrement adéquates pour les analyses en temps réel.
- Fowler et Sadalage (2013) nous disent que les applications de blog, de management de contenu, etc. Les familles de colonnes permettent de stocker les données liées (commentaire, tags, catégories, liens, etc.) ensemble et ainsi créer des familles de colonnes en lien avec les besoins. Par exemple, si l'application de blog se concentre plutôt sur les utilisateurs que les contenus des blogs, faire une famille de colonne lié aux utilisateurs.

4.9 Acteurs principaux

Il existe différents acteurs pour les bases de données orientée colonnes. La base de données orientée colonnes la plus populaire est Cassandra et pour cette raison nous allons bien détailler les fonctionnalités et caractéristiques de cette base de données en particulier.
Ensuite, nous détaillons très brièvement un autre acteur populaire pour les bases de données orientés colonne, appelé Hbase.

CASSANDRA:

La présentation du logiciel Cassandra se base sur les informations de Apache Cassandra (2021), de leur site internet. Développé par Facebook à l'origine et ensuite devenue open-source sous le Apache Incubator en 2009, Cassandra est une base de données distribuée NoSQL. Ses principaux avantages sont: la haute disponibilité, sa performance, sa résilience (ne possède pas de single point-of-failure, c'est-à-dire ne peut pas tomber en panne ou ne peut pas perdre de données) et son extensibilité horizontale qui lui permet de traiter des quantités immenses de données .
Cassandra se dit distribuée dans le sens où elle fonctionne sur plusieurs machines tout en apparaissant unifiée comme une seule machine auprès des utilisateurs. La communication entre les machines se fait par un protocole *gossip,* qui est une communication peer-to-peer, c'est-à-dire sans maitre. Comme nous l'avons vu plus tôt, une architecture sans-maitre fournit de la robustesse et de la résilience à la base de données. Effectivement, tous les nœuds sont équivalents et donc l'échec d'un des nœuds (=machines) n'entraine pas la perte de données. Ainsi, le fait d'être distribuée lui apporte deux avantages : une bonne performance lorsque l'application reçoit beaucoup de requêtes et une protection contre la perte de données en cas d'échec d'hardware d'un data center.
En plus de la distribution des données sur un cluster de machine, Cassandra est également populaire pour sa facilité d'extension horizontale de la base de données, et ce sans temps d'arrêt et de manière dynamique. En cas de besoin d'augmentation de la performance (en stockage ou en rapidité) de la

base de données, il suffit d'ajouter de nouvelles machines au cluster de machines de la base de données. A l'inverse, il est également possible de réduire le nombre de nœuds du cluster en cas de surcapacité par rapport aux besoins réels.
Cassandra assure la répartition des données automatiquement en utilisant des partitions et des tokens et une communication *gossip*.
Cassandra assure également la réplication des données sur plusieurs nœuds grâce à la notion de *replication factor*, RF, qui détermine le nombre de copies que l'on veut sur la base de données pour chaque donnée. A nouveau, la réplication est assurée automatiquement par Cassandra. Précisons à nouveau que la réplication permet non seulement d'assurer une robustesse et une résilience des données mais également permet une amélioration de la performance à deux niveaux :

- Pouvoir répartir la charge de travail des requêtes sur plusieurs nœuds différents ;
- Pouvoir garder une performance constante entre différents data centers répartis à travers le monde. Une requête d'écriture faite en Europe va se répliquer dans les différents data centers d'Asie et d'Amérique, et ainsi il n'y aura pas de diminution de performance pour les utilisateurs de ces continents-là par rapport à une requête lié à un data center Européen.

Par défaut, Cassandra est une base de données AP, Available et PartitionTolerant (disponible et tolérant à la partition), par rapport au théorème du CAP. Comme cela a été précisé plus haut, le **théorème du CAP est un compromis** entre *disponibilité* et *consistance* et **non un choix binaire**.
Cassandra possède une fonctionnalité qui permet de choisir son niveau de consistance (consistency level : CL). Le CL correspond au nombre de nœuds minimum qui doivent accuser réception d'une requête de lecture ou écriture auprès du nœud Coordinateur avant que l'opération ne soit considérée comme réussie. Cassandra préconise de choisir son CL en accord avec le RF. Le fait de pouvoir contrôler la réplication et le niveau de consistance à un si bas niveau de granularité fait de **Cassandra une base de données très flexible et ajustable**.

De nombreuses entreprises, en plus de Facebook auparavant, utilisent Cassandra : Uber, Instagram, Apple, Spotify, Netflix, Ebay, Twitter, etc.

Hapache HBASE

Créée en 2006, Hbase est une base de données open-source, distribuée et non-relationnelle qui s'est inspiré des publications de BigTable (développé par Google) (Apache Hbase, 2019). Elle fonctionne sur le système de gestion de fichier de Hadoop, HDFS. Théoriquement, il n'est pas établi que Hbase ne puisse pas fonctionner sur un autre système de fichiers distribués. Hbase utilise également le modèle de calcul MapReduce pour répondre aux opérations. Hbase est notamment utilisé pour fournir l'accès en temps réel des données, pour les opérations de lecture ou d'écriture. Hbase est extensible horizontalement, c'est-à-dire qu'il supporte l'ajout de nouvelles machines au cluster et comme pour Cassandre, cela permet d'améliorer la performance de requêtes et également cela supporte la gestion d'immenses quantités de données.

Comme pour Cassandra, de nombreuses entreprises utilisent Hbase : Facebook, Pinterest, Vinted, Tumblr, jet.com, etc.

5.Base de données orientée documents

5.1 Introduction

Les bases de données orientées documents sont très similaires aux bases de données clé-valeur, et pourraient même être considérées comme une sous-classe de celles-ci : elles utilisent également un index pour associer des clés avec des 'documents'. Fowler et Sadalage (2013) soulèvent que la grande différence entre clé-valeur et document est dans l'opacité de la donnée stockée. Dans le système de stockage des bases de données clé-valeur, les données sont stockées de manière opaque : la seule donnée que l'on sait est qu'une clé est associée à un contenu de données. Dans le système de stockage des bases de données documents, le contenu des données liées à une clé est examinable et l'on peut faire des requêtes par rapport au contenu lié à la clé. L'utilisation du terme document peut prêter à confusion dans le sens où le contenu des documents est du multimédia ou des données non-structurées. Cependant, Meier et Kaufmann (2019) rappellent qu'un document est un fichier avec des données structurées dont la structure est une liste de paires : attribut-valeur. De plus, chaque document possède un attribut « _id » qui joue le rôle d'identifiant unique. Il arrive qu'un SGBD document inclut l'attribut « _rev » pour indiquer la version du document et ainsi faciliter le contrôle de consistance sur multiples machines.
Une base de données orientée documents est flexible et évolue en fonction des besoins de changement de l'application sans engendrer des temps d'arrêt à l'application. En effet, si un ou des documents doivent être mis à jour, seuls les documents concernés vont être affectés, il ne faudra pas mettre à jour le schéma et donc l'application ne devra pas s'arrêter (Foote, 2019). Comme pour les bases de données orientées colonnes, il n'y a pas de schéma explicite. Cependant le bon sens commun veut que, lors de la conception d'un SGBD documents, un certain schéma implicite soit respecté afin de pouvoir effectuer des opérations qui ont du sens et non juste des opérations pour récupérer un flow de données sans sens. Ces SGBD sont extensibles horizontalement grâce à leur architecture qui permet de distribuer les données sur différentes machines.

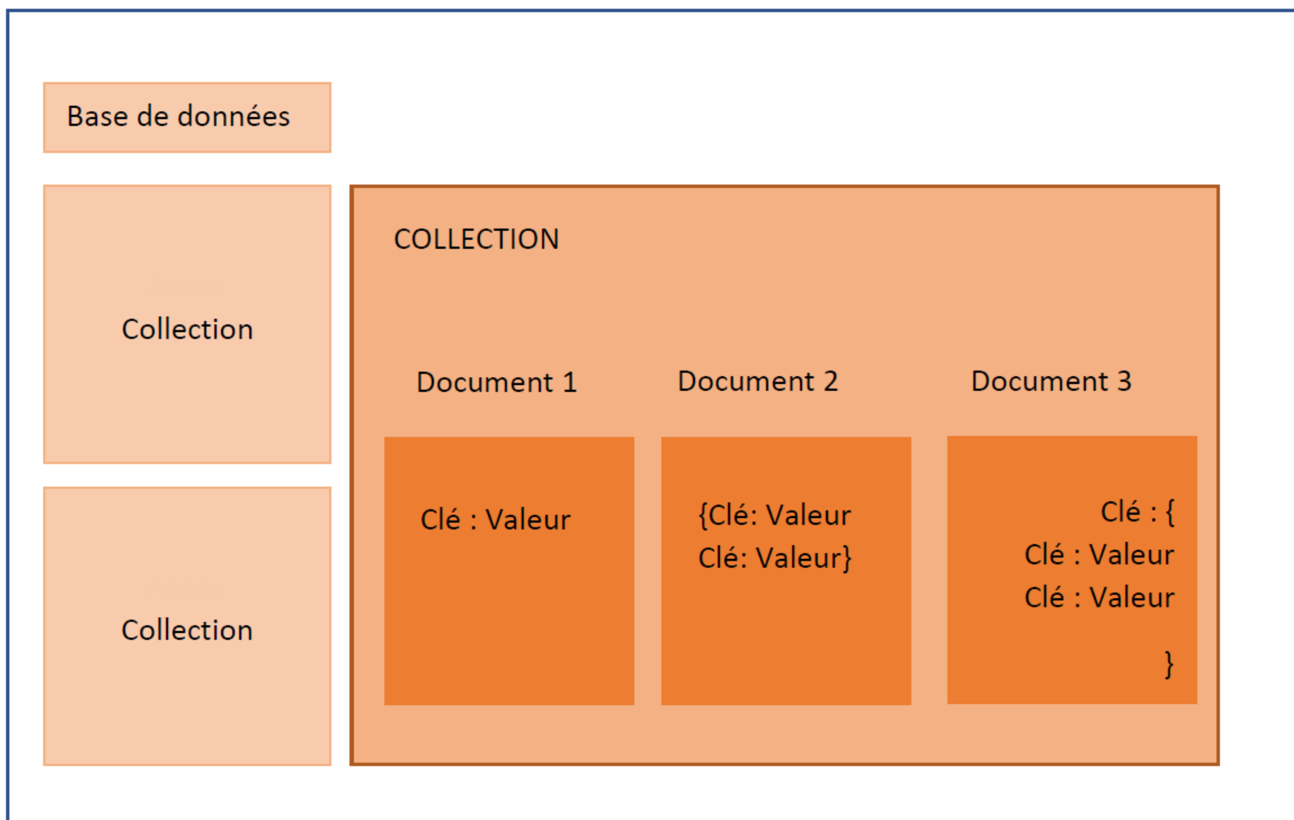


Figure 5.1 : Schéma d'une base de données orientée document. (Hosting Data, 2021)

Ainsi pour résumer, ces bases de données combinent l'absence de schéma, c'est-à-dire de la flexibilité, avec la possibilité de structurer des données stockées. Les clés des documents,

correspondant à l'attribut « _id » sont utilisés pour l'identification et il est possible d'intégrer des documents dans des documents. (Meier et Kaufmann, 2019).

5.2 Fonctionnement

Chaque base de données orientée documents possède sa propre implémentation pour stocker et gérer les données. Cependant, l'encodage suit un format standard parmi les SGBD documents et l'encapsulation et l'encodage des données se fait : en XML, ou en YAML, ou en JSON ou également en BSON. Pour le reste, il n'y a pas de schéma obligatoire, dans une même base de données, il peut y avoir différents types de documents, eux-mêmes ayant des attributs différents et/ou des attributs optionnels. Dans ces SGBD, il n'existe pas d'attributs vides. De plus, dans une même base de données, deux documents peuvent être encodés suivant deux systèmes de codages différents. *Par exemple, on peut avoir un document encodé en JSON et un autre en XML*. Ces bases de données peuvent gérer des données structurées, semi-structurées ou non structurées. Une des fonctionnalités des bases de données documents est de pouvoir intégrer des documents à l'intérieur d'un document. Une autre fonctionnalité classique de ces SGBD est le concept de *Collection*. En comparaison, une collection d'un SGBD document serait une table dans un SGBDR. La grande différence est que la collection est beaucoup plus flexible et que des documents au sein d'une même collection ne doivent pas avoir le même schéma obligatoirement, bien qu'un schéma implicite fait du sens pour avoir une certaine structure entre les documents d'une même collection. Cette notion de Collection est très intéressante car elle permet d'avoir des sortes de hiérarchies à la manière de la programmation orientée objet.
L'absence de schéma obligatoire, contrairement aux bases de données relationnelles, permet d'ajouter des nouvelles données sans devoir tenir de contraintes obligeant à faire correspondre la nouvelle donnée avec le reste. On peut donc ajouter de nouveaux attributs à un document sans devoir définir ceux-ci ou sans devoir changer le contenu du document (Fowler et Sadalage, 2013).
Conceptuellement, les documents doivent être considérés comme des unités uniques et complètes, et il est conseillé d'éviter de diviser un document en plusieurs parties. Comme pour les bases de données clé-valeur, ce type de base de données supporte les requêtes CRUD. Les requêtes sont faites sur les documents via les clés qui servent d'identifiant unique à un document. Une clé peut être un simple ID, un string, un chemin ou un URI. Comme dit plus haut, la plus grande différence avec les bases de données clé-valeur est l'opacité des données dans la base de données. Dans le modèle clé-valeur, les données dans la base de données sont opaques et sont pratiquement des black-box. Dans le modèle orienté documents, grâce à certaines API ou certains langages de requêtes, il est possible de faire des requêtes sur le contenu des documents. En effet, en utilisant les métadonnées du document, il est possible de classifier le contenu et donc de faire des requêtes plus précises, plus affinées sur le contenu des documents (Desk, 2021)

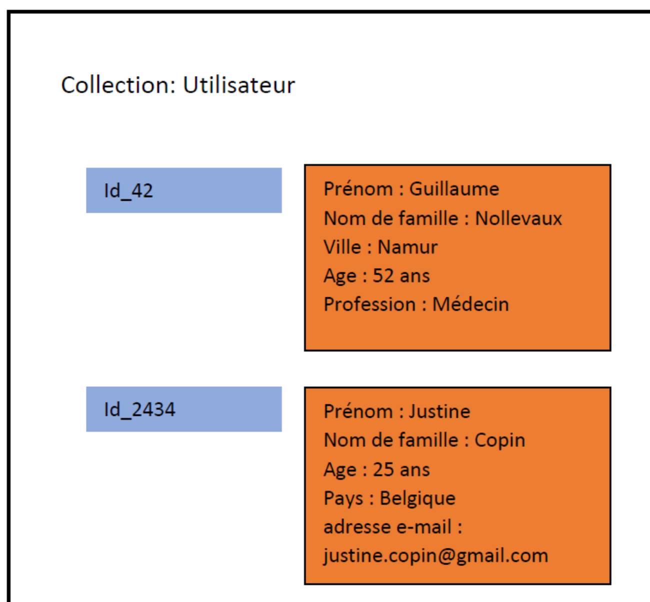


Figure 5.2 : Exemple d'une collection, Utilisateur, comportant de deux documents.

La figure 5.2 illustre bien la possibilité pour différents types de documents de la même base de données d'avoir des schémas différents. Cette vérité s'applique également pour des documents de la même collection. Comme nous l'avons vu plus tôt, le théorème du CAP nous dit que parmi la Consistance, la Disponibilité et la Tolérance aux partitions, on ne peut en choisir que deux. Les bases de données orientées documents améliorent la disponibilité en utilisant la réplication, rendue possible grâce à une architecture distribuée de type maitre-esclave. Il y a deux types de nœuds, le nœud-maitre et les nœuds esclaves. L'utilisation de données de réplication a plusieurs utilités : la redondance des données, les opérations de lecture à grande échelle, la possibilité de faire des maintenances sans temps d'arrêt et la récupération de données après des défaillances (Fowler et Sadalage, 2013).
Certains softwares comme MongoDB permettent de choisir et configurer le niveau de consistance désirée pour sa base de données. Par exemple, avec MongoDB, le niveau de consistance peut être défini à l'aide des « replica sets », qui correspond au nombre minimum de machines sur lesquelles l'opérations d'écriture a été transmise pour considérer que l'opération est réussie. Prenons un exemple d'une base de données avec un cluster de 15 machines (également appelé nœuds), définissons que le niveau de consistance d'une opération d'écriture doit être fait à la majorité. Dans ce cas, il faudra que la réplication de l'opération d'écriture se fasse sur un minimum de 8 nœuds pour être considéré comme une opération réussie. Ainsi, lorsque le niveau de consistance est augmenté, le nombre de réplications qui doit se faire augmente, et donc les performances des opérations d'écriture sont ralenties (Fowler et Sadalage, 2013).

Dans son article, Keith Footh (2019), explique l'intérêt d'utiliser JSON pour le format de stockage des documents et le concept REST en combinaison. Commençons par définir JSON et REST :
« **JSON** (JavaScript Object Notation) est un format léger d'échange de données. Il est facile à lire ou à écrire pour des humains. Il est aisément analysable par des machines. Il est basé sur un sous-ensemble du langage de programmation JavaScript. C'est un format texte complètement indépendant de tout langage, mais les conventions qu'il utilise seront familières à tout programmeur habitué aux langages descendant du C. Ces propriétés font de JSON un langage d'échange de données idéal »(Json, 2002). Ce qu'il faut retenir c'est que *JSON est non structuré, flexible* et, surtout, c'est **lisible par les humains** et **par les machines**.
« **REST** (Representational State Transfer = style architectural permettant de construire des applications (Web, IntraNet, Webservice). Il s'agit d'un ensemble de conventions et de bonnes pratiques à respecter et non une technologie à part entière» (Hachet, 2019) On utilise ce style architectural pour **communiquer sur le web entre différents machines ou systèmes**. L'intérêt est la flexibilité : l'implémentation du coté client peut changer sans affecter la communication avec le coté serveur et vice-versa.
Cette particularité est un grand avantage car, à partir du moment où le format de communication est décidé et accepté à l'avance, le coté client et le coté serveur peuvent rester séparés et modulable, c'est-à-dire que l'interface utilisateur et la base de données peuvent être séparés. La séparation de ces deux composants est un double avantage(Foote, 2019):

1. La facilité de l'extensibilité. On peut augmenter le nombre de machines indépendamment du coté client ou du coté serveur.
2. Chaque composant peut évoluer de son côté indépendamment de l'autre composant.

JSON comme REST sont tous les deux supportés par la plupart des langages de programmation.

5.3 Utilité et usage

Ce type de SGBD a été initialement développé pour les applications Web, ce qui explique pourquoi il s'intègre facilement avec des langages tels que JavaScript ou HTTP.
De manière générale, ce type de base de données convient très bien pour les applications de gestion de contenu et également pour les applications pour la gestion de profil d'utilisateurs (Nayak et al., 2013). Elles sont également très efficaces pour le stockage de données d'inventaires ou de catalogues.

Il n'est pas nécessaire de savoir à l'avance quels seront les types de données stockés, et donc particulièrement intéressant lors du développement d'une base de données où les données qui seront stockées ne sont pas connues précisément.
De manière générale, l'utilisation de ces SGBD se fait pour les applications de données Log, les applications de gestion de contenu (blog management par exemple), les applications d'analyse en temps réel et les applications d'E-commerce (Fowler et Sadalage, 2013).

5.4 Avantages

- **Extensibilité horizontale**, ce qui implique deux avantages : permet de *stocker de très grandes quantités* de données sans limites (Tableau 2.2 : comparaison entre les deux types d'extensibilité), permet d'*améliorer la performance* (répartition des calculs sur plusieurs machines) et améliore la *robustesse* et la *résilience* de la base de données.
- **Flexibilité** : ces bases de données peuvent gérer tous types de données (structurées ou non structurées). Particulièrement adéquat lorsqu'on ne sait pas quel type de données sera stocké. De plus, cela permet la possibilité de modifier des documents sans impacter le reste des documents de la base de données.
- **Haute disponibilité** : le schéma de la base de données peut être manipuler sans entrainer de temps d'arrêt.
- **Séparation de la couche utilisateurs avec la couche base de données.** (Foote, 2019).
- **Possibilité de faire des requêtes plus affinées que pour les bases de données clé-valeurs.**

5.5 Désavantages

- **Limitation dans le contrôle de la consistance** (Foote, 2019). Le manque de schéma strict et de contraintes d'intégrité peut entrainer un problème de consistance entre deux données, par exemple avoir des informations contradictoires entre deux données.
- **Problème d'atomicité**. Si l'on veut faire un changement qui implique une donnée présente dans deux collections différentes, il faut faire deux requêtes différentes, ce qui est contraire à l'atomicité.
- **Limitation de la complexité des requêtes.**

5.6 Applications

- *Les plates-formes E-commerce* : ce type de plateforme a un besoin de flexibilité pour les produits, avec leurs nombreux attributs, et leurs commandes. De plus, ces plateformes évoluent rapidement et ont donc besoin d'une base de données qui peut évoluer constamment et facilement et ce sans faire de grandes migrations de données (Fowler et Sadalage, 2013). La gestion de plusieurs produits avec des centaines d'attributs serait moins performante avec une base de données relationnelle. En effet, ce type de gestion implique des opérations concentrées sur des requêtes sur des colonnes plutôt que sur des lignes. Avec un SGBD documents, chaque produit peut contenir tous ses attributs en description sur une seule page, ce qui rend la gestion plus facile et les opérations plus performantes.
- *Plateforme de blog ou système de management de contenu* : l'absence de schéma et le fait de travailler avec du JSON (habituellement), fait que les bases de données orientées documents sont plutôt efficace pour gérer les commentaires d'utilisateurs, la publication de websites, l'enregistrement de nouveaux utilisateurs, la modification du profil d'un utilisateur, etc. (Fowler et Sadalage, 2013).
- Les applications proposant des fonctionnalités de *Profil en ligne*. La gestion des profils d'utilisateurs doit permettre la flexibilité. En effet, les utilisateurs fournissent différents types d'informations et font évoluer les informations de leur profil avec le temps (par exemple : un changement d'adresse ou de numéro de téléphone). Les bases de données orientées documents répondent à ces besoins de flexibilité (les documents n'ont pas de schéma fixe et peuvent stocker différents attributs entre différents documents), de gestion efficace du stockage (les données stockées pour le profil d'un utilisateur sont uniquement celles qui lui sont spécifiques et pas celles qui lui sont attribuées par défaut ou avec la valeur NULL) et d'évolution (les documents peuvent facilement modifier les attributs d'un utilisateur).

5.7 Acteurs principaux

Comme il existe beaucoup de bases de données orientées documents, il peut être compliqué de savoir laquelle utiliser, c'est pourquoi il est utile de se tourner vers la littérature pour comparer. Il existe notamment de sites internet pertinents pour la comparaison de SGBD documents tel que Pat Research (Desk, 2021).
MongoDB est la base de données orientée document la plus populaire. Couchbase Server, CouchDB ou MarkLogic sont d'autres des bases de données orientés documents populaires.

MongoDB

La présentation de MongoDb se base sur le cours de Cleve et Faulkner (2021). C'est un système de gestion de bases de données distribué et ne nécessitant pas de schéma prédéfini des données. Il a été développé par l'entreprise MongoDB et est écrit en C++.
Les documents sont stockés sous le format BSON et les documents sont stockés dans des collections. En comparaison avec un SGBDR, un document dans MongoDb est l'équivalent d'une ligne en SGBDR et une collection est l'équivalent d'une table en SGBDR. Comme expliqué plus haut pour les bases de données orientés documents, les documents ont une clé unique (UUID) et ne doivent pas suivre la même structure, même s'ils font partie de la même collection.
La répartition des données suit l'approche *maitre-esclave*, qui permet de faciliter la consistance des données entre les nœuds mais qui implique également la possibilité de perdre des données en cas d'échec du maitre (les données qui n'ont pas été répliqué aux restes des serveurs esclaves avant l'échec du serveur maitre). MongoDB fournit de nombreuses fonctionnalités telles que la disponibilité, la tolérance aux défaillance, l'auto-réplication, la possibilité d'agrégation, etc.). Ce SGBD propose également l'ajout d'index au sein des documents, ce qui permet d'améliorer les performances des opérations de lectures.
MongoDB est utilisé par de nombreuses grandes entreprises dont Craigslit, Ebay, The Guardian, New-York Times, Toyota or Forbes (Nayak, Poriya et Poojary, 2013).

6.Base de données orientée graphe

6.1 Introduction

Les premières bases de données orientées graphe ont vu le jour dans les années 1980. L'intérêt pour ce type de base de données s'est décuplé avec l'arrivée de l'internet et le développement des réseaux sociaux, ce qui a eu pour conséquence une production de plus en plus importante de bases de données graphes (Meier et Kaufmann, 2019).
Cette base de données NoSQL diffère des autres NoSQL. Les bases de données NoSQL ont été développées dans l'idée de les faire fonctionner de manière distribuée sur un cluster de machines et de rassembler les données similaires, appelées agrégats, sur des mêmes machines : il s'agit d'un traitement de très large quantité de données avec peu de connections entre elles. Les SGBD graphes ont été conçus pour traiter des données avec beaucoup de connections entre elles (Fowler et Sadalage, 2013). Les bases de données graphes sont spécialisées dans les requêtes qui étudient les relations complexes entre les nœuds tels que les réseaux sociaux, les préférences de produits (exemple : recommandations basées sur des achats précédents), etc.
De plus, le sharding des données d'une base de données graphe est plus complexe que pour les autres NoSQL. Le sharding des données d'une base de données graphe trouve sa complexité dans le fait que les relations entre les nœuds doivent être pris en compte, ce qui relève du « domain specific knowledge » (Meier et Kaufmann, 2019). Il faut à la fois stocker les nœuds qui sont liés sur le même serveur pour garder de bonnes performances et à la fois ne pas stocker trop de nœuds sur un même serveur pour éviter une charge trop lourde et trop encombrée sur le serveur. Les autres NoSQL au contraire, dans leur conception même, évitent les relations entre les instances afin de pouvoir stocker les données sur différentes machines sans devoir rompre une relation, ou du moins sans devoir y prêter une attention particulière.

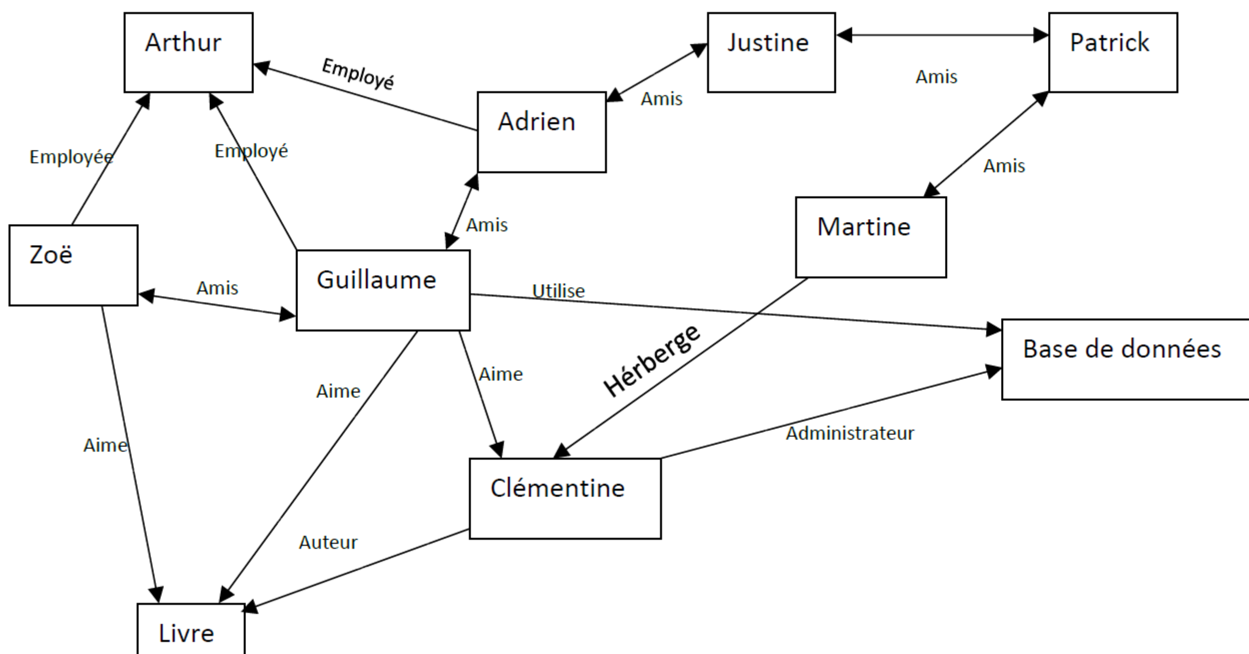


Figure 6.1 : Exemple d'une structure d'un graphe (Fowler et Sadalage, 2013).

6.2 Fonctionnement

Les bases de données orientés graphe (BOG) s'inspirent de la théorie des graphes et consistent en un ensemble de **nœuds**, qui représentent des entités (concepts ou objets) et de **liens**, qui représentent les relations entre les entités. Les nœuds ainsi que les liens peuvent avoir des

propriétés ainsi qu'un label. Les propriétés sont une fonctionnalité très intéressante car cela permet d'ajouter de l'intelligence aux liens entre les entités. Par exemple, dans le cadre d'un réseau social, on pourrait avoir comme attribut : date de leur premier jour d'amitié ; nombre de nœuds qui sépare deux nœuds en particulier, etc. (Fowler et Sadalage, 2013).
Les liens sont directionnels ou peuvent être bidirectionnels et il n'y a pas de limites entre le nombre de liens entre deux nœuds ou de limites du nombre de liens qu'un nœud peut avoir avec d'autres liens.
Les BOG peuvent être manipulées à l'aide d'un langage informatique et faire des requêtes ou de traitements sur les liens ou sur les entités. Le langage le plus connu est Cypher, qui est le langage utilisé pour la base de données Neo4j.
Mathématiquement parlant, les BOG sont un ensemble de *property graphes.* Un *property graphe* est composé de nœuds et de liens dirigés (relations) entre les nœuds.
Leurs particularités (Meier et Kaufmann, 2019) par rapport à un graphe normal sont :

- Lien dirigé avec un nœud de début et un nœud de fin
- Les nœuds et les liens peuvent contenir des propriétés et/ou peuvent être caractérisé par un label. Ces propriétés et labels stockent la valeur réelle des données.

Comparé à un SGDB relationnel, les nœuds sont l'équivalent d'une table, un nœud en particulier est l'équivalent d'une ligne et un lien est l'équivalent d'une relation. Contrairement à un SGBDR où chaque modification ou ajout dans la base de données doit respecter des contraintes d'intégrité, pour les BOG il n'y a pas de schéma défini, ce qui signifie qu'un nœud peut être ajouté ou modifié à la base de données sans qu'il soit défini au préalable ou sans faire de changement sur le reste de la base de données (Fowler et Sadalage, 2013). Et c'est le système de gestion de la BOG qui gère l'intégrité référentielle des liens entre les nœuds afin d'assurer la consistance des données (Meier et Kaufmann, 2019).
La grande différence avec les SGBDR est que ceux-ci doivent utiliser des *foreigns keys* et des *joints* pour investiguer sur les relations entre des entités. Le coût de ce type de requête est élevé et la performance peut rapidement baisser lorsque les entités sont fortement connectées, c'est-à-dire beaucoup de liens entre les entités. Pour les bases de données graphes, les requêtes sur les relations des entités sont faciles et peu coûteuses, ce qui explique la bonne performance des BOG pour ce type de requêtes. Ainsi, le coût pour faire des requêtes sur les relations d'un nœud est constant et est indépendant du volume total de la base de données (Meier et Kaufmann, 2019).

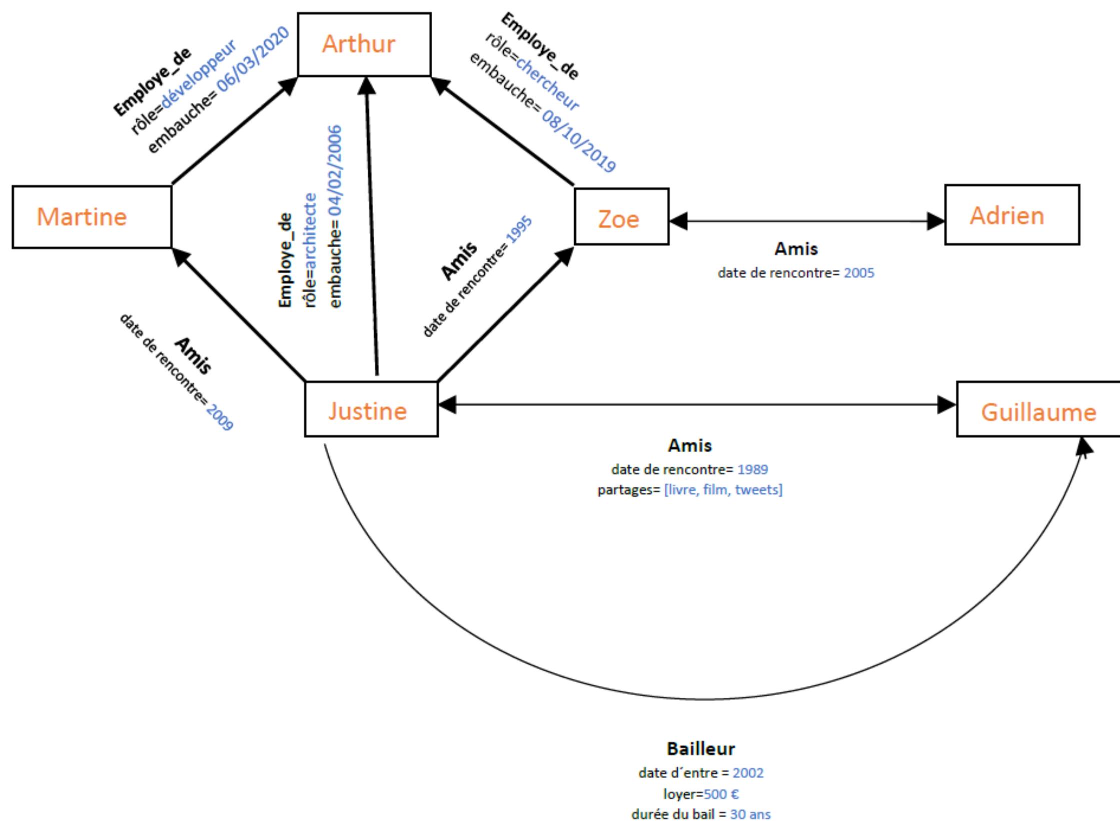


Figure 6.3 : Exemple d'une base de données orientées graphe de 6 entités et 8 relations au total (Neo4j, 2021)

La figure 6.3 illustrent les différentes caractéristiques des graphes :
- Relations directionnelle ou bidirectionnelle.
- Les relations peuvent avoir un label et des propriétés.
- Il n'y a pas de limites de relations entre les entités

Labels et propriétés des nœuds n'ont pas inclus dans le graphe afin de ne pas encombrer le schéma. Précisons donc que les entités peuvent avoir un label et des propriétés comme pour les relations.

6.3 Utilité et usage

L'objectif d'un BOG est de pouvoir faire des requêtes efficaces et performantes qui traversent le graphe rapidement et d'étudier les relations entre les entités. L'idée principal derrière les BOG est la connexion entre les données. Les BOG sont utilisés lorsque les données sont organisées en réseaux et de manière plus générale, lorsque les relations entre les instances sont plus importantes que les instances individuelles en elles-mêmes (Meier et Kaufmann, 2019).

Les BOG sont basés sur la théorie des graphes. Ainsi, en fonction du software et à partir de la théorie des graphes, le SGBD peut proposer des algorithmes qui permettent de calculer :
- La **connectivité** : lorsque chaque nœud d'un graphe possède au moins un lien avec tous les autres nœuds du graphe.
- Le **chemin le plus court entre deux nœuds**, avec l'algorithme de Dijkstra. Cet algorithme va être utilisé notamment pour les applications géographiques et les recherches d'itinéraire. Ou alors dans le cadre d'un réseau social, on pourrait l'utiliser pour trouver les relations qui sépare deux entités.
- Le **voisin le plus près** : si les liens sont pondérés, c'est le nœud le plus proche en termes de distance ou de temps. Exemple : où est la pompe à essence la plus proche ? Dans une ville où il y aurait plusieurs pompes à essence, un utilisateur pourrait vouloir faire le plein dans la station la plus proche.

En fonction de l'utilisation et des besoins de l'application, il est possible d'utiliser d'autres notions et propriété de la théorie des graphes comme dans le software Neo4j (Neo4j, 2021) :

- **Betweeness** Centrality
- **Closeness** Centrality
- **Degree** Centrality
- **Clique** (Triangle Count)

De plus, d'autres notions liées aux graphes peuvent être intéressantes pour l'analyse des relations entre les membres d'une communauté ou pour des applications en supply chain.
Star : Dans un graphe, une entité est une « star » au sein d'un groupe lorsque cette entité reçoit un grand nombre de liens dirigé vers elle.
Isolate : Dans un graphe, une entité est « isolée » au sein d'un groupe lorsque cette entité ne reçoit aucun ou très peu de lien, en comparaison avec le reste du groupe, dirigés vers elle.
Dans une application d'analyse de processus dans une usine, repérer les produits « star » ou les produits « isolate » peut aider la prise de décisions.

Les BOG peuvent également être utilisé pour analyser les relations entre les membres d'une communauté. Dans ce contexte, un réseau social est une communauté ou un groupe d'utilisateurs du web ayant des relations. Chaque réseau social et chaque individu possède une réputation. La recherche sociale empirique peut utiliser la base de données orienté graphe pour essayer de comprendre les relations entre les membres d'un réseau social, par exemple avec l'utilisation de notions telles que « Star » ou « Isolate ». La théorie des graphes, avec ses diverses métriques et indices, est un outil précieux pour les analyses sociologiques de réseaux.

6.4 Avantages

- **Structure de stockage flexible et efficace .**
- **Grande performance** des opérations de lecture. Les requêtes traversent la base de données à travers le réseau, ce qui les rend plus rapides qu'un SGDB relationnel.
- Conforme aux propriétés **ACID**.
- **Support de retour en arrière** (Nayak et al. 2013).
- **Représentation** de la base de données **significative**. La visualisation d'un graphe permet une compréhension rapide et facile de ce qu'il contient.
- **Extensibilité horizontale**. Par leur nature, les BOG sont conçues pour traiter facilement de grands volumes de données.
- Haute performance sur les **requêtes de relations complexes**. Les BOG peuvent faire des requêtes peu couteuses sur des relations complexes sans devoir faire des jointures ou indexer des tables.
- **Rapide**. Il est possible de traverser la base de données de manière très rapide, et ce indépendamment de la taille de la base de données. En effet, il est possible de faire des requêtes ciblées sur des portions de graphe pour ne pas parcourir l'entièreté des données (et donc également des données non pertinentes).
- **Flexible**. Il est possible d'ajouter et de supprimer des nœuds du graphe facilement, sans préoccupation sur le schéma général du graphe ou des autres entités du graphe.
- Un **arbre peut être considéré comme un sous-graphe** (graphe spécial sans aucun cycle). Les propriétés d'un arbre peuvent être impliquées, notamment la possibilité d'avoir ses propres indexes. Les arbres peuvent être considérés comme une certaine forme d'agrégats structurels.

6.5 Désavantages

Le désavantage principal des bases de données orientées graphe est la **difficulté de faire du sharding**. La gestion de la distribution des données à travers les serveurs est bien plus complexe que pour les autres bases de données NoSQL.

6.6 Gestion des accès

Neo4j et Cypher ne proposent pas de fonctionnalité incluant la gestion des permissions et des accès. En effet, l'accès est accordé à chaque utilisateur pour l'ensemble de la base de données. Il est techniquement possible de programmer des règles de permission précise sur Neo4j, cependant cela nécessite des compétences de programmation précises et ne peut aucunement rivaliser avec la simplicité de programmation des règles de permission de langage SQL (Chen et Lee, 2019).

6.7 Applications

Les bases de données graphes peuvent être utilisées pour un grand nombre d'applications diverses. Nayak et al. (2013) ainsi que Meier et Kaufmann (2019) ont répertoriées celles-ci :

- Applications en réseaux ou du Cloud. *Exemple : Réseau sociaux.*
- Applications de recommandation. *Exemple : recommandation d'articles similaires après la sélection d'un article en particulier pour un site d'E-commerce.*
- Applications d'analyses d'infrastructures de réseaux. *Exemple : réseaux d'eau ou d'électricité ou réseaux internet.*
- Applications de navigation. *Exemple : trouver le chemin le plus court vers un lieu.*

6.8 Acteurs principaux

Les deux bases de données graphes les plus connues sont Neo4j et JanusGraph (Chen et Lee, 2019).

Neo4j

Neo4j est une base de données orientée graphe, open-source et développé en Java. Neo4j supporte la plupart des systèmes d'exploitation les plus utilisés (Linux, OS X, Windows,…) et possède plusieurs langage de requêtes, dont Cypher qui est le plus populaire. Cypher est comparable à SQL mais est adapté aux bases de données graphes. Ce langage permet d'effectuer des requêtes sur les données ainsi que du traitement sur les données comme le fait SQL. Neo4j supporte les transactions ACID et est considérée comme une des BOG les plus performantes du marché.
Neo4j est utilisé par de nombreuses grandes entreprises telles que : NASA, AirBnb, Lyft, Ebay, Catterpillar, etc.

JanusGraph

JanusGraph est une base de données graphe, extensible et optimisée pour stocker et manipuler des graphes contenant des centaines de millions de nœuds et de liens distribués sur des clusters de machines (JanusGraph,2017), Ce SGBD est open-source et a été développé et lancé en 2017 sous Linux Foundation. JanusGraph supporte moins de langage de programmation (seulement Python, Clojure et Java) mais se distingue de par sa capacité à être fortement extensible. Cette base de données est conforme au modèle de consistance ACID. En plus du traitement OLTP, ce SGBD supporte également OLAP pour des analyses grâce à l'intégration de Apache Spark.

Janus est utilisé par Crédit Agricole, RedHat, Ebay, Netflix, Uber, etc.

7. Comparaison et choix de la base de données

Comparaison entre les bases de données

Le fait que certaines très grosses entreprises comme Netflix, Ebay ou Facebook ont été citées plusieurs fois dans différents types de bases de données NoSQL démontre la réalité du concept de Polyglot Perstistence. En fonction du besoin recherché pour l'application, pour une partie de l'application ou pour une partie d'une solution informatique, différentes bases de données NoSQL et SQL peuvent être utilisées. La figure 7.1, inspirée de Meier et Kaufmann (2019), reprend les 4 grandes familles de NoSQL avec les applications typiques de chacune. Lorsque la famille de NoSQL est choisie, il faut choisir parmi tous les différents SGBD qui existent parmi cette famille. Précisons que les SGBD les plus populaires ne sont pas spécialement les plus adéquats pour chaque application. Pour cela, nous conseillons de recourir à des benchmarks pour comparer et choisir le SGBD le plus adéquat. Furht et Villanustre (2016) en ont répertoriés plusieurs évaluant les performances de ces solutions utilisées pour le cloud computing et le big data : PigMix, GridMix, Terasort, GreySort et YCSB (Yahoo Cloud Serving Benchmarking).
Précisons que ces études comparatives comparent la performance selon certains critères et qu'il faut tenir compte des critères choisis pour l'évaluation lorsque ces benchmarks sont utilisés. Par exemple, la plupart des benchmarks cités plus haut se basent sur le critère « computation time ». Ci-dessous figurent différents critères populaires utilisés pour évaluer la performance d'un SGBD:

- Computation time
- Le nombre d'opérations par seconde et le délai de réponse pour des opérations de lecture et d'écriture
- La taille maximum des données
- etc.

Cependant, il existe d'autres types de critères, moins fonctionnels, qu'il faut prendre en compte pour le choix d'un SGBD tels que : la tolérance aux défaillances, le hardware, le coût, la consommation électrique, etc. (Furht et Villanustre, 2016).

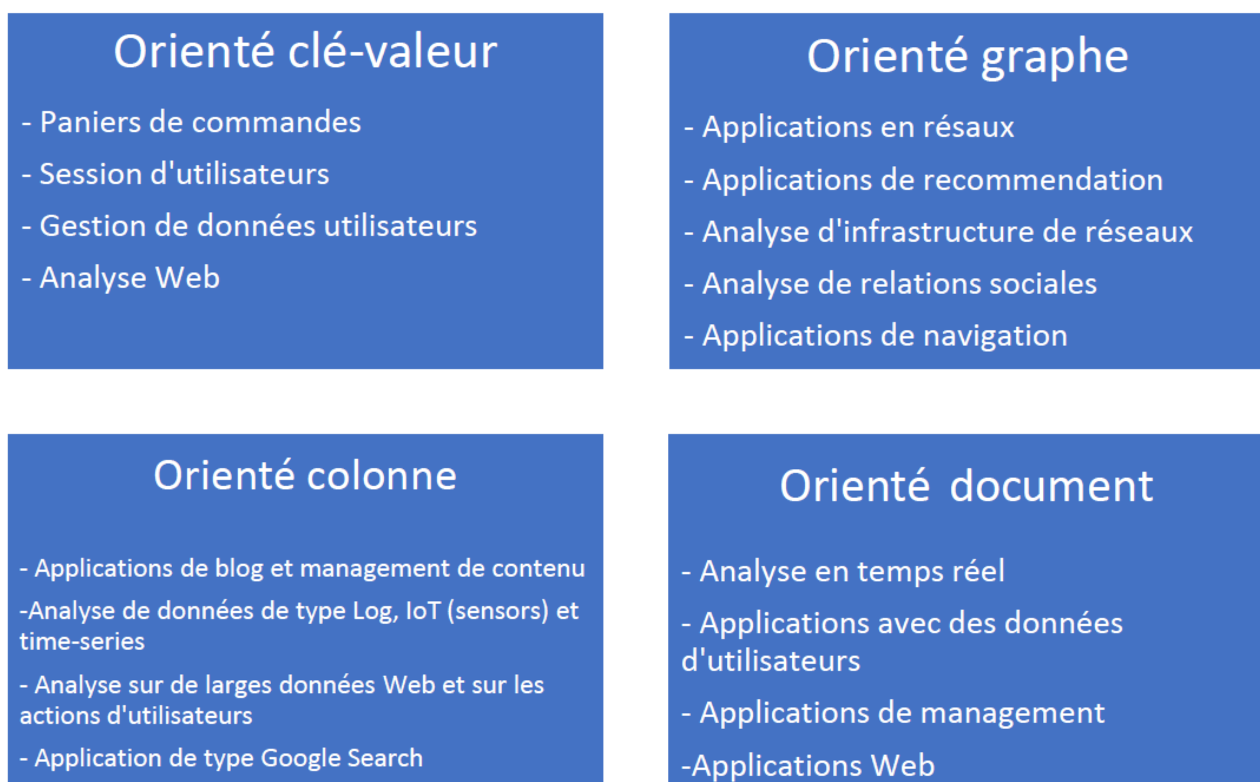


Figure 7.1 : Applications typiques des 4 familles de bases de données NoSQL (Meier et Kaufmann, 2019).

Clé-valeur ou Documents ?

Les bases de données clé-valeur et documents sont deux SGBD qui sont similaires. D'ailleurs, d'une certaine manière, les SGBD documents sont une sous-classe des SGBD clé-valeur. Comme nous l'avons vu, la différence se marque par **l'opacité du** contenu du composant '*valeur*' de la paire clé-valeur (Fowler et Sadalage, 2013). Pour savoir quel SGBD choisir, il faut garder en mémoire que *l'opacité du contenu permet de stocker n'importe quel type de données*, avec comme seule limite la taille du contenu mais qu'en contrepartie, *les opérations sur les données sont moins précises*. Pour les SGBD document, le contenu est *moins opaque et il est limité dans des structures* et dans certains types, cependant cette limite se récupère *en flexibilité dans les opérations sur les données*. En effet, les requêtes peuvent se faire sur des agrégats et sur le contenu des documents et donc faire des requêtes plus précises et plus complexes (Fowler et Sadalage, 2013).

Ben Scofield est un software ingénieur et il a comparé les différents types de bases de données selon quatre critères : la performance, l'extensibilité, la flexibilité et la complexité. Les trois premiers critères sont des critères essentiels pour le choix d'une base de données qui traite du Big Data (George S, 2013).

Data Model	Performance	Extensibilité	Flexibilité	Complexité
Modèle Relationnel	Variable	Variable	Basse	Modérée
Modèle Clé-valeur	Haute	Haute	Haute	Très basse
Modèle Colonne	Haute	Haute	Modéré	Basse
Modèle Document	Haute	Variable-Haute	Haute	Basse
Modèle Graphe	Variable	Variable	Basse	Modérée

Figure 7.2 : Comparaison des types de SGBD selon la performance, l'extensibilité, la flexibilité et la complexité (George, 2013).

Choix du type de base de données

Avant de proposer différents critères pour choisir adéquatement la base de données, il nous a paru important de différencier trois notions (Fowler et Sadalage, 2013) qui peuvent être confondues mais dont la différence est significative.

1. Le **Data Model** : c'est le modèle qui va spécifier comment les données sont organisées. Par exemple : modèle relationnel, modèle orienté clé-valeur, modèle orienté graphe, etc. Ce modèle décrit comment nous interagissons avec la base de données.
2. Le **Query Model** : c'est le modèle qui va spécifier comment les données vont être retrouvées et modifiées. Par exemple: langage SQL, langage Cypher, MapReduce, etc.
3. Le **Storage Model** : c'est le modèle qui va spécifier comment la base de données va être stockée physiquement sur le hardware et comment les données en interne vont être manipulées.

Les deux premiers modèles sont généralement pris en charge par le SGBD, au moins pour le Data Model et en partie pour le Query Model. Dans le chapitre suivant, sur l'architecture, nous détaillerons plus en précision les considérations à propos du Storage Model.
Les SGBD, SQL ou NoSQL, combinent le Data Model et le Query Model dans leur solution proposée. Cependant, les différents systèmes NoSQL peuvent avoir des combinaisons de Data Model et Query Model différentes, ce qui va induire des considérations architecturales différentes (Marcus, 2018). Par exemple, utiliser un SGBDR qui fonctionne sur un seul serveur implique des considérations architecturales différentes qu'un SGBD key-value qui fonctionne sur un cluster de serveurs distribués.

Chen et Lee (2019) proposent deux principes à suivre pour le choix d'une base de données adéquate :

1. Comprendre les objectifs et les challenges de la base de données qu'il faut développer.
2. Les développeurs et les ingénieurs de la donnée doivent réfléchir s'il faut garder une base de données SQL ou passer à une base de données NoSQL. Ce choix est basé sur les fonctionnalités recherchées de la base de données ainsi que sur les formats de données que l'entreprise utilise. Par exemple, si les données utilisées sont fortement non-structurées, le choix d'une base de données NoSQL est évident.

Fowler et Sadalage (2013) considèrent 2 critères pour choisir une base de données : la productivité des programmeurs et *la performance de l'accès aux données*. Comme nous l'avons vu plus haut, le critère de performance peut varier fortement. Par exemple, une base de données orientée agrégats va avoir de très bonnes performances de lecture ainsi qu'une facilité d'extension horizontale ou encore une base de données orientée graphe va avoir de très bonnes performances sur des données hautement connectées entre elles (Fowler et Sadalage, 2013).
Lourenço et al. (2015) évaluent les bases de données NoSQL selon plusieurs attributs de qualité : « la disponibilité, les performances de lecture et d'écriture, le recovery time, la fiabilité, la résilience, la robustesse, la consistance, l'extensibilité et la maintenabilité ».
Chen et Lee (2019) ont également répertoriés différents articles, dont celle de Lourenço et al. (2015), étudiant les critères de qualité d'une base de données NoSQL. Les critères les plus repris sont : l'extensibilité (en système de fichiers et en hardware), la flexibilité du schéma, la disponibilité du schéma, les performances de lecture et d'écriture, la fiabilité et la résilience, la technologie de distribution (en système de fichiers et en hardware), etc.

Selon nos recherches, les critères peuvent se diviser en 3 catégories : les critères des besoins fonctionnels, non-fonctionnels et les critères liés aux caractéristiques de la donnée. 0000

- Les **besoins ,fonctionnels** : fréquence des interrogations ? opération de lecture?; opération d'écriture?; disparité des utilisateurs de la base de données?; taille totale de la base de données?; besoins transactionnels?; etc.
- Les **besoins non-fonctionnels :** disponibilité ? consistance ? extensibilité horizontale ? évolution de la structure avec le temps ? flexibilité ?; etc.
- **Comment sont les données** : structurées ? non structurées ? variété ? haute vélocité ? etc.

En fonction de ceux-ci, il faut choisir le type de Data model, le type de Query model et le type de Storage model. Enfin, lorsque les Data, Query et Storage model sont connus, il faut **choisir la base de données appropriée.**

8. Architecture

Pour transformer les données Big Data en connaissances et créer de la valeur, une architecture hard-ware adéquate est nécessaire pour stocker les données et celle-ci doit fonctionner en symbiose avec les systèmes NoSQL et SQL utilisés, ainsi que les techniques d'analyses propres au Big Data. Le plus souvent, les plateformes de Big Data ont recours au *système de distribution* **Hadoop** pour le stockage et à **MapReduce** pour le traitement des données. Concevoir une architecture adéquate pour une solution Big Data est complexe et requiert d'avoir une vision architecturale globale ainsi qu'une compréhension approfondie des différents concepts expliqués au chapitre précédent.
Les bases de données NoSQL sont capables de gérer les données du Big Data et ont la particularité d'être très flexible. Le contrecoup de cette flexibilité est que la gestion des bases de données n'est plus encapsulée dans le SGBD mais elle devient la responsabilité des concepteurs d'applications, c'est-à-dire celle des data ingénieurs. L'architecture doit également être pensée différemment que pour les données traditionnelles, en petites quantités et structurées. L'architecture d'une solution Big Data est composée de différentes infrastructures, de systèmes de gestion de bases de données (NoSQl et/ou SQL), d'outils d'analyse et également des applications de visualisations. Rappelons que trois types de modèles doivent être pris en considération : le Data Model, le Query Model et le Storage Model, ce dernier correspondant plus à l'infrastructure physique de l'architecture (Sawant et Shah, 2013).

Une bonne conception de l'architecture d'une solution Big Data nécessite la considération de *deux types de besoins* : les **besoins commerciaux** (orientés business), qui répondent à un objectif économique, et les **besoins non-fonctionnels** tels que la consistance, la sécurité, la disponibilité, la performance, l'extensibilité, etc. Intégrer ces besoins non-fonctionnels est essentiel lors de la conception de l'architecture.
Les nouvelles méthodes et les nouveaux outils du Big Data créent de **nouveaux défis**, qui sont mis en avant notamment dans les besoins non-fonctionnels (BNF). *Par exemple, le fait d'avoir une architecture distribuée comparé à une architecture centralisée transforme la manière de concevoir le modèle de consistance (apparition du compromis ACID vs BASE et du théorème du CAP avec l'apparition des nouveaux types de bases de données : les NoSQL*).
Il n'existe pas de système qui puisse à la fois répondre à 100% à tous les BNF et également répondre à toutes les autres exigences du Big Data. Ainsi chaque architecture du Big Data devra être pensée et conceptualisée, pour chaque couche, en cohérence avec les priorités de la solution Big Data tout en essayant d'optimiser les BNF à l'aide de compromis entre les différents besoins.
A partir de nos recherches, (dont Chan (2013), Sawant et Shah (2013), Aye K. N. (2015), Bjeladinovic, Marjanovic et Barbarogic (2020) , Borthakur (2008), Demchenko, De Laat et Membrey (2014)) et parmi toutes les architectures de Big Data proposés que nous avons étudié, nous avons redéfini l'architecture du big data, défini les deux grands objectifs ainsi que 3 couches principales. L'architecture du big data est *un système distribué pouvant gérer des données du 5V, assurant l'extensibilité horizontale et étant hautement performant et hautement tolérant aux défaillances*. Elle répond à 2 **grands objectifs** : le stockage des données et le traitement des données. Ces deux objectifs peuvent être encadrés dans une architecture de **3 grandes couches** : la couche de stockage/infrastructure, la couche d'analyse et la couche interface/visualisation.
La première couche sert à **réceptionner, ingérer et stocker les données** (dans des bases de données NoSQL ou SQL, dans des datawarehouse, …). En fonction du type de projet, cette couche peut être subdivisée en plusieurs couches. Cette couche doit pouvoir stocker les données d'une manière physique, cela correspond au Storage Model, c'est-à-dire qu'il faut développer une *infrastructure hard-ware adéquate* (Hadoop, HDFS, distribution), mais également avec des *logiciels et techniques* capables de stocker les données avec cette infrastructure, cela correspond au Data Model (bases de données NoSQL).
La seconde couche sert à **analyser les données pour créer de la valeur**. Il existe différentes techniques d'analyse telles que Data Mining, Machine Learning, Text Mining, …
La dernière couche sert à **rendre disponible visuellement et manipulable les résultats obtenus** dans la seconde couche par les utilisateurs.

Précisons que chaque couche est reliée aux autres couches et doit être développée en cohérence avec le reste de l'architecture.
Dans l'exemple d'architecture Big Data de figure 3-1, inspiré de Sawant et Shah (2013), nous pouvons voir que les 3 couches décrites ci-dessous ont été subdivisées en plusieurs sous-couches. De plus, remarquons la présence d'autres couches, étant moins orientées besoins commerciaux, qui sont conçues pour encadrer l'architecture en générale : la couche de management, la couche de monitoring et la couche de sécurité. Les deux couches d'Analytics et de visualisation correspondant au deuxième défi principal du Big Data (décrit dans l'introduction), sont deux couches complexes et nécessitent un cours uniquement destiné à leur fonctionnement. Elles sont brièvement détaillées dans ce chapitre.
Précisons que la conception de l'architecture ainsi que les présentations des différentes couches de ce chapitre sont basées du travail de Sawant et Shah (2013) mais que d'autres sources ont également été utilisées pour d'autres concepts.

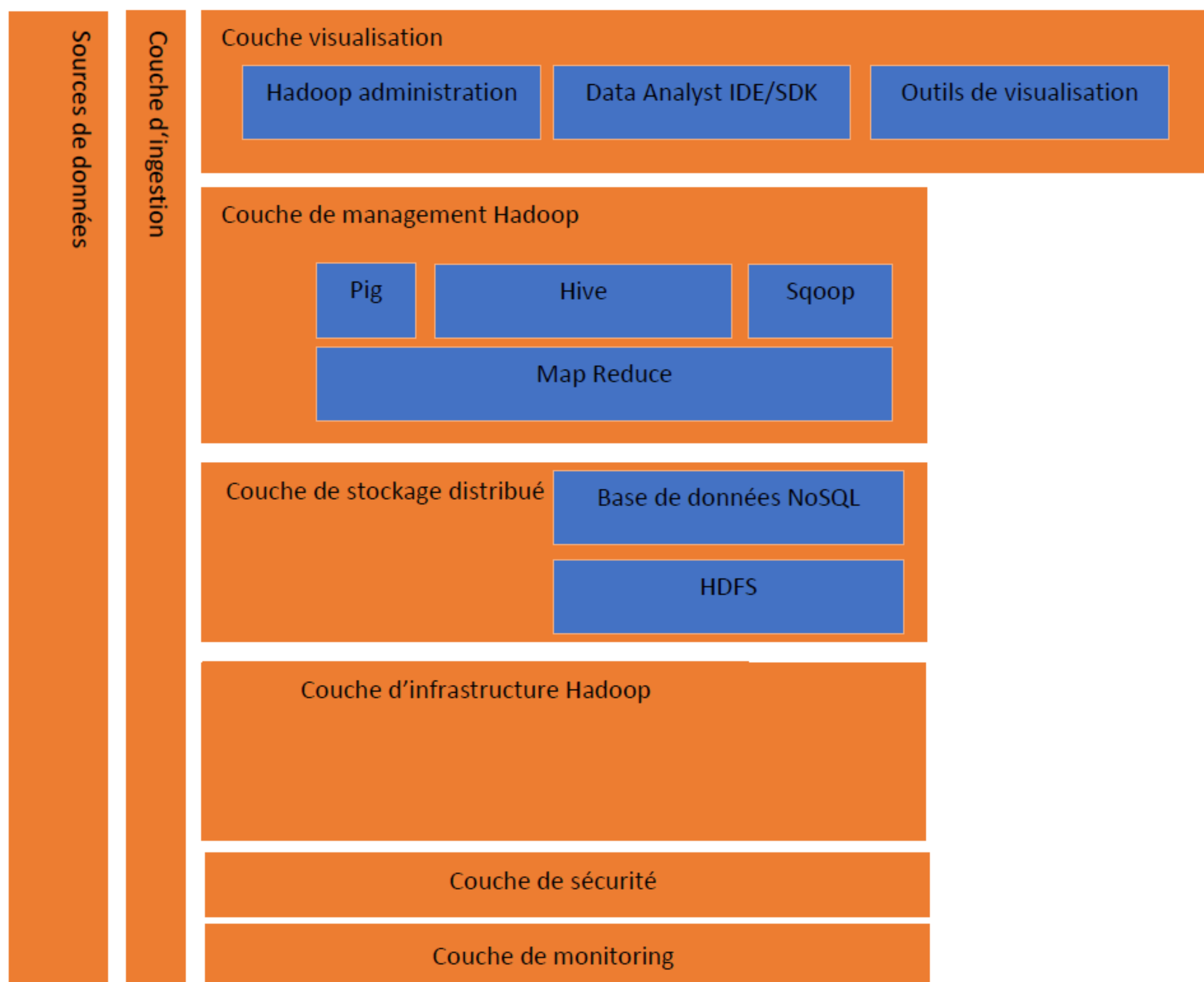


Figure 8.1: Architecture d'un solution Big Data (Sawant et Shah, 2013).

8.1. Couche des sources de distribution et d'ingestion

Les sources de données sont souvent multiples et fournissent deux types de données : structurées et non-structurées. Avant de pouvoir stocker les données dans un datawarehouse ou dans des bases de données NoSQL, les données doivent passer par la *couche ingestion* afin de séparer le bruit des données pertinentes.

Dans cette couche, représentée par la figure 3.2, les données passent par différents composants de l'identification des données, en passant par la filtration, la validation, la réduction du bruit, la transformation, la compression et l'intégration des données pour finalement être stockées dans les bases de données de la *couche de stockage* de l'architecture. Il est possible d'adapter la manière de concevoir cette couche en fonction des besoins non-fonctionnels (performance, disponibilité, …).
Cette couche doit répondre à plusieurs défis (Sawant et Shah, 2013) :

- Être capable d'ingérer des données venant de **sources multiples** et **sous différents formats** ;
- Être capable d'**agir en médiateur** entre des données venant de *sources multiples* et répondant à des protocoles différents, et les *différentes couches de l'architecture* ;
- Être capable d'ingérer des données et de les **transmettre à des destinations différentes** : la couche de stockage Hadoop (voir plus loin) ou les datawarehouse;
- Être capable d'**ingérer des données en continu** pour des analyses en temps réel.
- Être capable de faire **co-exister** dans l'espace de stockage des données sous le **bon format** pour l'analyse et des **données brutes** (devant être rapidement converti sous le bon format sur demande, *Just-In-Time*)

Le chapitre 3 du livre de Sawant et Shah (2013) détaille en profondeur différents modèles répondant à ces défis.

Il existe de nombreux outils et logiciels, commerciaux ou open-source, simplifiant les tâches de la couche ingestion tels que : Sqoop, Flume, Storm, Chukwa, Apache Kafka, Apache S4, …

Retenons pour cette couche qu'elle permet de séparer le *bruit des données brutes* et les *informations pertinentes des données brutes*, avant leur insertion dans la couche supérieure de l'architecture : ***la couche de stockage***. Cet affinement des données réduit la taille du stockage des données, qui entraine une réduction de la latence de l'accès aux données pour les couches supérieures.

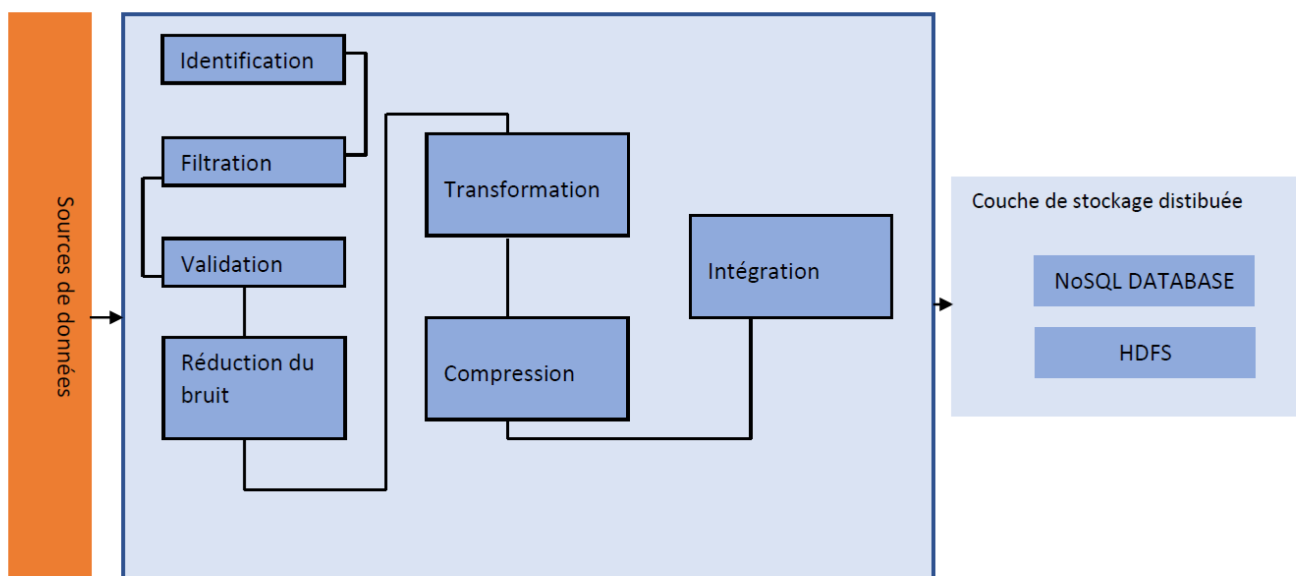


Figure 8.2 : Couche d'ingestion des données (Sawant et Shah, 2013).

8.2. Couche de stockage distribué

La manière de stocker les données va dépendre de deux choses : la manière dont la couche de stockage va *recevoir* les données à stocker, et la manière dont la couche d'analyse des données va *récupérer* les données à analyser. Cela illustre la raison pour laquelle l'architecture d'une solution Big Data doit être pensée comme un ensemble et non comme l'addition de différentes couches indépendantes.
Le nouveau paradigme des données Big Data implique de nouveaux modèles et de nouvelles techniques pour stocker les données (Sawant et Shah, 2013) :

- Utilisation de **bases de données NoSQL** plutôt que les *SGBDR traditionnels ;*

- Utilisation de **HDFS** plutôt que des *datawarehouses classiques*. HDFS ne remplace pas nécessairement et complètement les datawarehouse classiques, HDFS peut également avoir un rôle d'intermédiaire entre la couche d'ingestion et le datawarehouse, avec l'utilisation parallèle d'outils tel que Sqoop.
- **Modèle polyglot** : utilisation de différents modèles et outils de stockage au sein de la même architecture mais pour répondre à des besoins différents. Par exemple, combiner des bases de données NoSQL pour gérer des données devant être hautement disponible et nécessitant plutôt un environnement distribué avec des données recourant principalement à des transactions devant être hautement consistantes. Un exemple concret serait une plateforme de commerce en ligne : pouvoir assurer la disponibilité des paniers d'achats avec une base de données NoSQL, et assurer la consistance du paiement de la commande avec un SGBDR.
- **Stockage de données distribuées** plutôt qu'un stockage de donnée centralisé. En effet, la très grande quantité de données nécessite une extensibilité horizontale. Cette extensibilité horizontale, traduit par de grands clusters de machines, induit de nouveaux besoins non-fonctionnels à prendre en compte tels que la robustesse et la résilience des données.

Le chapitre 4 du livre de Sawant et Shah (2013) détaille en profondeur différents modèles répondant à ces challenges.

La plateforme la plus populaire pour la couche de stockage est Hadoop. C'est un environnement open-source permettant de stocker d'énormes volumes de données (terabytes ou petabytes) de manière distribuée sur des machines à faible coût.
Le fonctionnement ainsi que les objectifs de Hadoop sont définis sur leur site internet mais sont également disponibles grâce au travail de Borthakur (2008). Le site internet de Hadoop ainsi que le travail de Borthakur ont été utilisés pour la présentation de Hadoop ci-dessous.
Hadoop possède deux composants principaux (Aye K. N., 2015) : HDFS (Hadoop Distributed File System), qui est un système de fichier distribué massivement extensible et MapReduce : « modèle de programmation pour traiter de très larges data sets, qui utilise le calcul parallèle, de manière automatisée, sur un large cluster de machines à faible coût pour diviser la charge de travail parmi celles-ci et améliorer les performances » (Dean et Ghemawat, 2008).
La particularité du premier composant de Hadoop est qu'il est hautement tolérant aux pannes et a été conçu pour être déployé sur des machines à faible coût. « Fault-tolerant » signifie qu'un des objectif principal de HDFS est d'être résilient et fiable même en présence d'erreurs. Il existe trois types d'erreurs : NameNode error ; DataNode error et Network Partition error (Dean et Ghemawat, 2008). HDFS a été conçu en utilisant le langage Java et chaque machine supportant ce langage peut faire fonctionner HDFS, ce qui rend HDFS utilisable sur la plupart des machines, dont celles à faible coût. Les nœuds du cluster communiquent par le biais de protocoles TCP.

HDFS

L'architecture HDFS, développé par Apache, est décrit par Borthakur (2008) et est détaillé ci-dessous.
La conception de HDFS est fondée sur certaines hypothèses et objectifs :

1. *Hypothèse 1* : **Les défaillances des machines sont la norme plutôt que l'exception.** En effet, étant donné le très grand nombre de machines dans un cluster de HDFS, les chances d'avoir à un moment donné une ou des défaillances sur une ou de plusieurs machines du cluster sont très élevées. Pour cette raison, la détection des défaillances et la récupération des serveurs défaillant est un des objectifs principaux de HDFS.
2. *Hypothèse 2 :* **Accès continu aux données**. Les applications qui utilisent HDFS ont besoin d'un accès continu à l'ensemble de leurs données. HDFS a été conçu pour un traitement par lot (batch processing) et l'accent est plutôt mis sur un débit élevé de l'accès aux données.
3. *Hypothèse 3 :* Les applications qui utilisent HDFS ont de **très large dataset** *(gigabytes ou terabytes).*
4. *Hypothèse 4:* Les applications qui utilisent HDFS ont besoin d'un accès « **write-one-read-many** ». Cette hypothèse permet de simplifier le contrôle de la consistance des données et améliorer l'accès et la disponibilité des données. Précisons que l'ajout de données est supporté mais que la modification des données doit être réfléchie et occasionnelle.

5. *Hypothèse 5 :* « **Moving Computation is cheaper than Moving Data** ». Si le calcul (computation) et les données sont situés sur deux machines différentes, l'un ou l'autre doit être déplacé vers l'autre machine afin que le calcul sur les données puisse être exécuté. Il est plus facile de déplacer la partie (le calcul ou les données) de la plus petite en taille vers la machine ayant la partie la plus grande taille, pour diminuer la congestion du réseau des nœuds. Dans le monde du Big Data, la taille du calcul est bien plus petite que la taille des données, HDFS propose donc logiquement des interfaces pour déplacer le calcul vers les machines où les données sont situées.
6. *Hypothèse 6 :* **HDFS assure la portabilité** de son architecture à travers différentes plateformes hétérogènes.

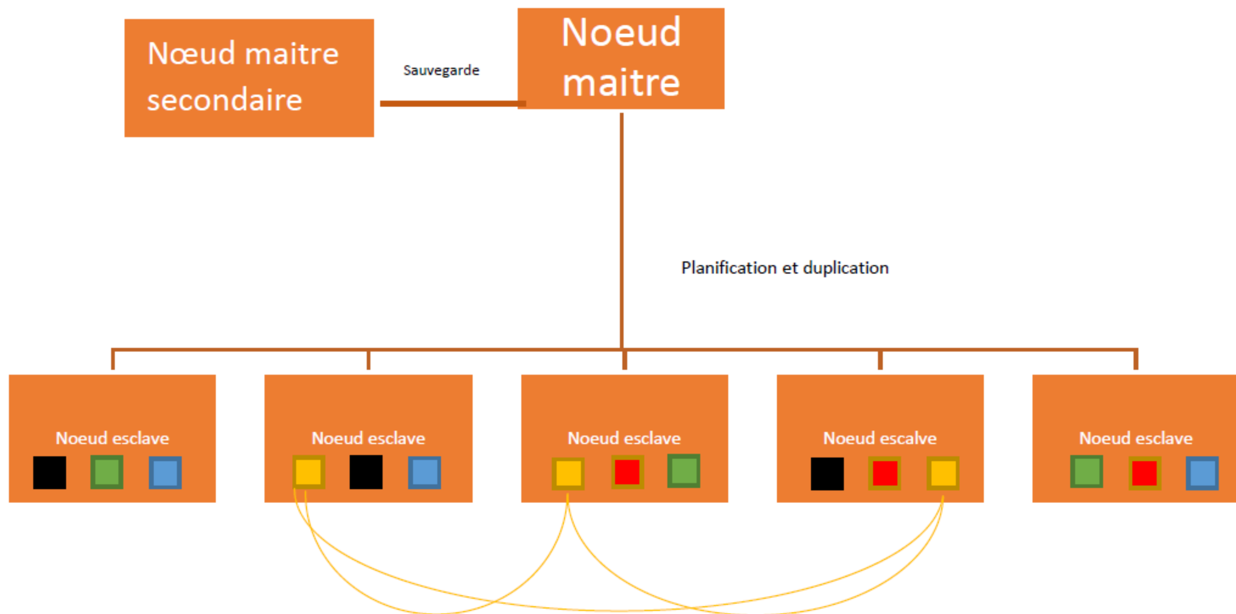


Figure 8.3 Schéma de l'architecture HDFS avec la représentation de réplications de différents data blocs dans différents nœuds (Wikipedia contributors, 2020).

Le modèle de distribution des données de HDFS est *maitre-esclave*, avec deux types de nœuds : **un unique NameNode**, qui est le *nœud-maitre*, et **les nœuds DataNodes**, qui sont les *nœuds esclaves*. Le nœud-maitre s'occupe de deux tâches : **gérer les opérations sur les métadonnées** du système de fichiers, telles que *ouvrir, fermer et renommer des fichiers ou dossiers* ; et **gérer la répartition des blocs** sur les nœuds esclaves. Plus généralement, ce nœud sert à simplifier l'architecture d'un système distribué de machines, en étant le dépositaire de toutes les métadonnées de HDFS. Le système a été conçu de telle sorte que le contenu des données ne passe jamais par le nœud-maitre. Il est possible de spécifier le nombre de réplications qu'un certain fichier doit maintenir. Ce nombre est appelé ***replication factor (RF)***, et il est connu et géré par le NameNode. Avoir un RF plus grand assure une meilleure fiabilité des données mais en contrepartie, plus de mémoire de stockage est utilisée.

Les Datanodes ont également deux tâches : **gérer les opérations d'écriture et de lecture** ainsi que **gérer la création, la suppression et la réplication de block** sous les instructions du nœud-maitre. Généralement, une machine du cluster est dédiée spécialement pour le Namenode et chaque machine restante du cluster fait fonctionner un seul DataNode.

L'organisation des fichiers sous HDFS suit une architecture hiérarchique traditionnelle où l'on peut créer des dossiers et stocker des fichiers dedans, supprimer ou créer des fichiers, déplacer un fichier d'un dossier à un autre ou encore renommer un fichier. HDFS supporte également la *fonctionnalité de permission des accès*. Cette organisation de fichier est appelée « **File Name Space** ».

Les données sont stockées dans des fichiers et chaque fichier stocke les données sous forme de « séquence de blocs », et chaque bloc de la séquence possède la même taille (sauf le dernier bloc qui peut avoir une taille différente étant donné que la taille totale du fichier ne correspond pas

forcément à un nombre exact de bloc). Un bloc typique utilisé dans HDFS aura une taille de 64 Mb, et un fichier sera divisé en plusieurs morceaux, les blocs, de 64Mb et réparti sur différents DataNodes. En comparaison avec le système de fichier Linux, la taille des blocs est d'environ 4Kb. Deux raisons expliquent la grande taille des blocs de HDFS (GeeksforGeeks, 2021):

- Avoir de grand blocs permet de minimiser les opérations de recherche
- Chaque bloc possède des métadonnées qui lui sont propres. Avoir de petits blocs implique avoir un plus grand nombre de blocs au total, ce qui implique stocker de grandes quantités de métadonnées, ce qui causerait un encombrement du trafic du réseau entre les nœuds.

Précisons que la taille du bloc et le réplication factor peuvent être configurés pour chaque fichier. HDFS se préoccupe uniquement de la taille des données et ne se préoccupe pas du type de donnée qui est stockée. Un grand avantage de stocker les données sous forme de bloc est que la taille des fichiers n'est plus limitée par la taille du disque étant donné que le fichier est réparti sous différents blocs dans différents disques (Aye K. N., 2015).

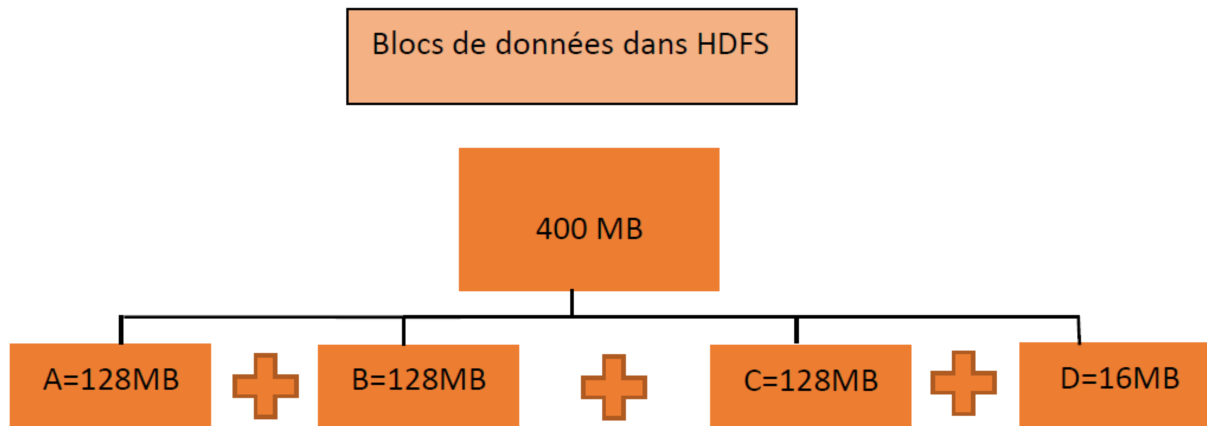


Figure 8.4 Schéma de la division d'un fichier de 400MB sous HDFS en 4 blocs de 128MB, 128MB, 128MB et 16 MB. (GeeksForGeeks, 2021).

Prenons l'exemple du schéma de la figure 5, nous avons un fichier de 400Mb. Les blocs pour ce fichier en particulier, sont de 128Mb. Les données de ce fichier seront stockées sous forme de séquences de blocs de 128Mb (sauf pour le dernier qui ne fera que 16Mb). Ensuite, ces différents blocs peuvent être répartis sur plusieurs nœuds et est répliqués plusieurs fois.

Un des principaux intérêts de HDFS est de pouvoir gérer de grandes quantités de données avec résilience, et cela grâce à la réplication des données. Les blocs d'un fichier sont répliqués, en fonction du réplication factor, sur différents DataNodes.

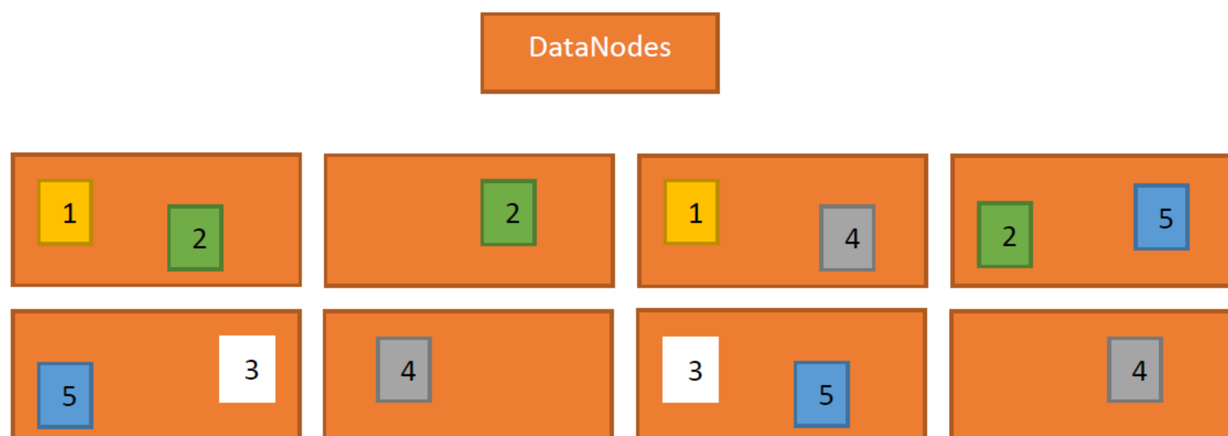


Figure 8.5 : Schéma de la réplication de 5 blocs de de données différents sur différents nœuds.

Sur la figure 8.5, nous pouvons voir différents blocs répartis sur différents DataNodes. Les blocs n'ont pas tous le même réplication factor : le bloc 1 possède un réplication factor de 2; bloc 2 : RF = 3; bloc 3 : RF = 2; bloc 4 : RF = 3; bloc 5 : RF = 3.

En plus du nombre de réplications que l'on souhaite, HDFS permet de configurer le placement des réplications à travers les nœuds pour améliorer la résilience (par exemple, éviter que plusieurs nœuds possédant les mêmes réplications échouent en même temps) et améliorer la performance (lorsque le cluster de machines est très grand, les machines sont réparties sur des *racks,* et une bonne configuration du placement des réplications permet d'améliorer la communication entre les nœuds, et donc la performance). Précisons qu'une bonne configuration des placements est complexe et nécessite de l'expérience. Il existe différentes politiques de placement des réplications, qui sont détaillés dans le travail de Borthakur (2008).
Le rééquilibrage des données (data rebalancing), présenté par Aye K. N. (2015), est la capacité à rééquilibrer automatiquement la répartition des données du système distribué après l'ajout ou la suppression d'un serveur ou le crash d'un serveur. Dans HDFS, un cluster sera considéré comme équilibré si aucun DataNode n'est surutilisé ou sous-utilisé (l'utilisation d'un nœud correspond au pourcentage de l'espace utilisé). Périodiquement, un algorithme itératif va rééquilibrer l'équilibre de l'ensemble cluster plutôt que de réduire le nombre de DataNode déséquilibré. De cette manière, le rééquilibrage permet de diminuer la congestion du réseau entre les nœuds.

Pour résumer, l'architecture HDFS fonctionne avec le modèle maitre-esclave, avec un NameNode qui s'occupe de la gestion des réplications et du FileSystemSpace et des DataNodes qui stockent les données et qui s'occupent des opérations d'écriture et de lecture. Les données sont stockées dans des fichiers, qui eux répartissent les données dans une séquence de blocs de même taille. Les différents blocks sont répartis sur les différents DataNodes et le NameNode s'occupe de la répartition à travers les différents DataNodes. Il est possible de configurer le nombre de réplications des blocs ainsi que le placement des blocks pour améliorer la fiabilité des données et la performance. Enfin, il existe des algorithmes de rééquilibrage pour rééquilibrer automatiquement la répartition des données.

MapReduce

Le deuxième composant de Hadoop est MapReduce. Nous nous sommes basés sur le travail de Fowler et Sadalage (2013) et Dean et Ghemawat (2008) pour présenter MapReduce.
L'arrivée des très grandes quantités de données a entrainé un nouveau paradigme qui est la distribution du stockage des données sur différentes machines (nœuds). Le recours à l'utilisation de clusters change aussi la manière dont les données doivent être calculées et traitées. Avoir un cluster de machines à l'avantage de répartir la charge de travail sur les différentes machines, et donc d'augmenter la performance du traitement des données. Cependant, la répartition de la charge de travail entre les différents nœuds doit être optimisée pour ne pas encombrer le trafic du réseau et augmenter le temps de latence de l'accès aux données. L'optimisation de cette répartition doit être réfléchie de manière à faire le maximum de traitement possible que l'on peut sur nœud pour diminuer la quantité de données à transférer entre les nœuds. MapReduce est un patron d'architecture permettant les calculs parallèles sur une architecture distribuée. L'idée de MapReduce est d'organiser le traitement de manière à profiter du grand nombre de machines tout en utilisant au maximum la capacité de traitement de chaque machine réquisitionnée.
Le fonctionnement de MapReduce consiste en deux étapes (Furht et Villanustre (2016) ; Dean et Ghemawat (2008)).
Une première phase, « Map », où le problème (ou la charge de travail) est divisé en plusieurs tâches et ces tâches sont réparties sur plusieurs machines. Après que les tâches ont été accomplies sur chacune des machines, la seconde étape, « Reduce », consiste à rassembler/combiner les résultats de chaque machine en un résultat/output final du problème.
MapReduce fonctionne grâce à deux types de nœuds : un nœud, NameNode, qui va diviser le travail en plusieurs tâches, allouer les tâches aux différentes machines, s'occuper du rétablissement des nœuds qui ont échoués et s'occuper de l'étape Reduce et des nœuds qui vont accomplir les tâches, DataNode.

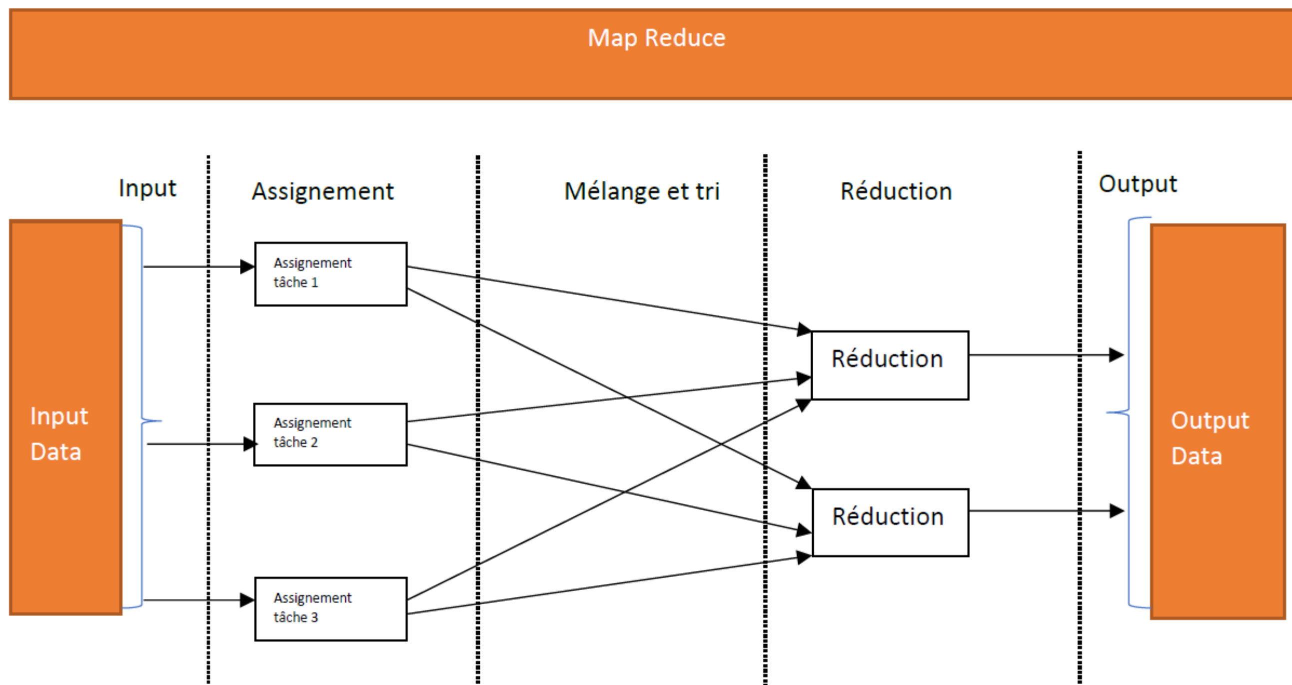


Figure 8.6 : Schéma du fonctionnement de MapReduce en deux parties : Assignement (Map) et Réduction (Reduce) (Furht et Villanustre, 2016).

Dans un traitement MapReduce, il y a 4 entités indépendantes qui interagissent ensemble (Aye K. N., 2015) :

- Le client, qui soumet le travail (MapReduce Job).
- Le jobtracker, qui est le nœud qui coordonne le déroulement du traitement MapReduce.
- Les tasktrackers, qui sont les nœuds qui effectue les tâches réparties par le jobtracker
- Un système de fichier distribué, qui est utilisé pour partager les fichiers entre les entités.

MapReduce n'est pas la seule option pour traiter les données Big Data rapidement. Il existe, entre autres, une alternative appelé **Spark**, qui est *open-source*, qui peut fonctionner avec HDFS. Contrairement à MapReduce, qui fonctionne sur du stockage sur disque, Spark fonctionne sur du stockage « in-memory » (Sawant et Shah, 2013). Cette caractéristique fait que le traitement des données avec Stark est jusqu'à 100 fois plus rapide qu'avec MapReduce sur des petites charges de travail (IBM, 2021).
Dryiad (plateforme distribuée) ; Shark (système d'analyse de données) ; Stratospehre (système de traitement de données massivement parallèle MPP) et Storm (système distribué de calcul en temps réel) sont d'autres alternatives à MapReduce.
Les systèmes de gestion des bases de données qui sont supportés par HDFS peuvent être de type relationnels ou de type NoSQL. Il faut choisir le type de bases de données NoSQL le plus adéquat par rapport à son utilisation. Rappelons que l'utilisation d'un type de bases de données NoSQL n'exclu pas l'utilisation d'un autre type de NoSQL ni d'une base de données relationnelle.

8.3. Couche d'infrastructure Hadoop

Pour pouvoir supporter la couche de stockage, composée de HDFS et de bases de données NoSQL et SQL, les données doivent être stockée physiquement et c'est la couche infrastructure qui s'occupe de cette tâche. La particularité de cette couche, c'est qu'il faut pouvoir gérer de très grandes quantités de données, non prévisibles, de tout type, à tout moment et à toute vitesse, ce qui diffère des données traditionnelles. L'extensibilité du stockage physique des données est indispensable et comme nous l'avons vu dans le chapitre 2, l'extensibilité horizontale est la seule possibilité pour gérer les Big Data (l'extensibilité verticale ayant des limites physiques ainsi que budgétaires).
C'est pour cette raison que cette couche est fondée sur un modèle informatique distribué : les données sont réparties physiquement sur plusieurs machines (également appelés nœuds) situées à différents endroits (suivant le principe d'extensibilité horizontale), et communiquent entre elles par réseau. Elles sont gérées par le système de fichier distribué Hadoop. Chaque nœud possède les données et les fonctionnalités de calcul requis pour effectuer des opérations sur les données. Ce type d'infrastructure s'appelle « share-nothing ».
Plus précisément, l'approche share-nothing signifie que chaque machine possède un processeur, une mémoire locale et des disques durs, ce qui fait que les nœuds ne doivent pas communiquer entre eux pour fonctionner seul. Elle permet de traiter les données de manière performante et avec extensibilité (Furht et Villanustre, 2016). L'utilisation de ce type d'approche est particulièrement intéressant pour les taches qui peuvent être subdivisées en plusieurs taches plus petites et de manière indépendante.

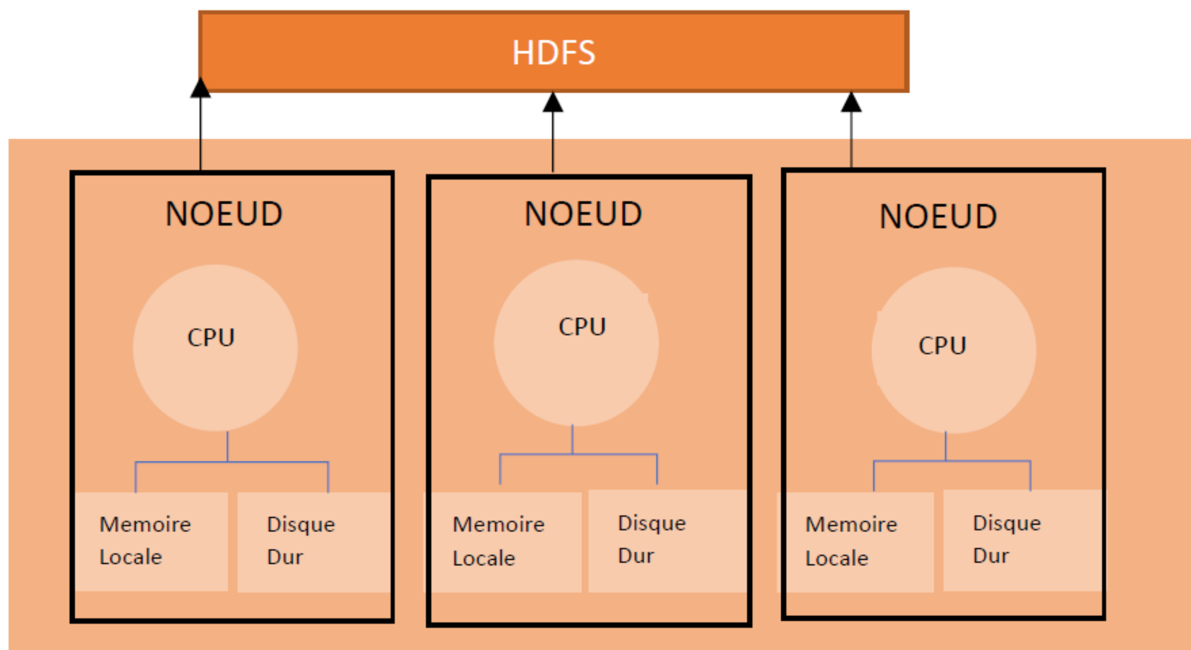


Figure 8.7: Architecture de l'approche Share-Nothing (Furht et Villanustre, 2016).

8.4. Couche de management Hadoop

Cette couche s'occupe d'accéder et manipuler les données mais elle s'occupe également du management des couches inférieures (couche infrastructure et couche de stockage) (Sawant et Shah, 2013).
Hadoop et MapReduce représentent les deux nouvelles techniques qui ont rendu la technologie Big Data réalisable. Hadoop s'occupe de la gestion du stockage des données, l'infrastructure physique et conceptuelle (avec HDFS) du stockage. Elle permet le stockage des données de manière distribuée sur plusieurs machines différentes. MapReduce permet le calcul parallèle sur différentes machines, améliorant la performance du calcul sur des très grands volumes de données, traduite par des opérations plus rapides (Aye K. N. ,2015)
La combinaison de ces deux techniques solutionne l'un des plus grands challenges du Big Data :
le stockage et la manipulation des données Big Data de manière efficace, peu couteuse et presque en temps réel.

Cette couche de l'architecture fournit les outils et les langages de requêtes pour accéder aux bases de données NoSQL, utilisant le système de fichiers HDFS. Ces outils sont décrit par Sawant et Shah (2013) et sont repris plus haut pour MapReduce, et repris ci-dessous pour Hive, PigLatin , Sqoop et ZooKeeper.

Apache **Hive** est une solution de datawarehousing opensource qui fonctionne sur Hadoop. Il possède un langage SQL (Hive QL) qui permet de faire des requêtes sur un cluster Hadoop. Ce langage est très similaire à SQL et les développeurs familiers avec SQL peuvent comprendre facilement Have QL.
Apache **PigLatin** est un langage de script optimisé pour traiter de grandes quantités de données avec un minimum de lignes de code. Ce langage est utilisé avec MapReduce et permet de manipuler les données en parallèle avec HDFS.
Sqoop est un outil particulièrement intéressant pour traiter conjointement des données NoSQL avec des données de bases de données relationnelles. Cet outil permet d'extraire/importer des bases de données relationnelles ou des colonnes spécifiques dans le HDFS. C'est une fonctionnalité indispensable de pouvoir traiter des données relationnelles avec des données NoSQL étant donné que la plupart des entreprises travaillent encore principalement avec des données relationnelles.
ZooKeeper est un outil qui assure la coordination entre les nœuds ainsi que la gestion de l'échec des machines.

8.5. Couche d'analyse

Avec les nouveaux types de données du 5V, de nouveaux problèmes apparaissent : la confidentialité, le stockage distribué, la tolérance aux défaillances, la qualité des données, etc. Comme c'était le cas pour d'autres couches de l'architecture, les méthodes d'analyse des données du Big Data doivent être re-conceptualisées. Précisons à nouveau que l'adoption de nouvelles méthodes n'implique pas forcément l'obsolescence des méthodes traditionnelles d'analyses. Le descriptif de cette couche est basé sur le travail de Furht et Villanustre (2016) et doit être considéré comme une introduction à ce domaine qu'est l'analyse des données.
Comme l'ont fait, Furht et Villanustre (2016), nous divisons l'analyse des données (qu'ils appellent également Knowledge Discovery in Database, KDD) en trois parties : **la partie input**, **la partie analyse** et la **partie output**. Cette division de l'analyse des données nous permet de bien saisir les interrelations des différentes couches de l'architecture. En effet, la partie input se déroule lors des couches que l'on a vues dont on a parlé précédemment ; la partie analyse à la partie présente et la partie output se déroule notamment dans la couche visualisation.

L'objectif principal de la **première partie** est de réduire la complexité des données pour améliorer la performance des calculs. En effet, avoir moins de données à traiter améliore la performance ainsi que la précision des résultats de l'analyse (Furht et Villanustre, 2016). Les 5V du Big Data ont pour conséquence une complexification de cette première partie : les données sont volumineuses, rapides, incomplètes, variées, hétérogènes, non structurées, etc.
L'objectif principal de la **deuxième partie** est de retrouver les informations pertinentes parmi les données ainsi que de transformer les données en connaissances (de la prédiction, des généralités cachées dans la masse de données, …). Il existe 4 grandes types d'analyses (Furht et Villanustre 2016) et chacune peuvent utiliser différentes approches algorithmiques :

1. La classification : consiste à utiliser un set de donnés déjà classifié (supervised learning), et en faire sortir un modèle qui permet de classifier des données qui ne sont pas classifiées. Il s'agit de généralisation. Exemple : u*tiliser un set de données sur des photos de fleurs étant déjà classifiées avec une race botanique. Et ensuite en faire sortir un modèle qui permet de classifier une photo de fleur dont on ne connait pas la race botanique, à une certaine race botanique.*
2. Le clustering : consiste à utiliser un set de données, non classifié (unsupervised learning), et séparer ce groupe de données en différents sous-groupes de données similaires. Exemple : *utiliser un set de données de différents pays du monde, lui appliquer un algorithme et récupérer ce set de données sub-divisés en différents groupes de pays qui sont similaires (ou du moins qui sont censés appartenir au même groupe).*
3. Les règles d'association et le séquential pattern ne font pas de classification parmi les données mais permettent plutôt d'étudier les relations entre les données et en faire sortir des *patterns*.

L'objectif principal de la troisième partie consiste à rendre utilisable les résultats de l'analyse de la deuxième partie. Furht et Villanustre (2016) suggèrent que cet objectif se base sur l'évaluation (mesurer les résultats) et l'interprétation (comment interpréter les résultats et comment les faire visualiser). Précisons que la visualisation en elle-même peut également apporter de la valeur aux données, grâce au visuel et sans le contenu des données. Cette partie est détaillée plus bas dans la *couche de visualisation*.

Dans leur travail, Furht et Villansutre (2016) observent que les méthodes traditionnelles d'analyse de données ont quelques limites :

- Elles sont non-adaptées aux systèmes distribués et à l'extensibilité. Elles sont conçues pour fonctionner sur un serveur central unique (le problème du volume des 5V).
- Elles sont non dynamiques, elles ne peuvent pas analyser les données en temps réel (le problème de la vélocité)
- Elles sont conçues pour traiter des données respectant un certain format, étant normalisées et stockées dans une structure rigide (le problème de variété).

Par exemple, les méthodes traditionnelles d'algorithmes de clustering ne sont plus adaptées pour les données actuelles. En effet, ces méthodes requerraient d'avoir le même format sous une certaine structure pour pouvoir en retirer de l'information (Furht et Villanustre, 2016). La différence fondamentale entre les méthodes traditionnelles et les nouvelles méthodes sont la distribution : le fait d'appliquer des méthodes d'analyses sur un serveur central ou sur un système de serveurs distribué implique beaucoup de nouveaux problèmes et de nouvelles contraintes à respecter (pouvoir faire appliquer les algorithmes sur une architecture MapReduce), au profit d'avantages tels que la disponibilité, le volume et la flexibilité, entre autres.
En plus des nouveaux défis du Big Data, Furht et Villanustre (2016) nous rappellent qu'il ne faut pas oublier les autres problèmes des méthodes traditionnelles telles que le bruit des données, les outliers, les problèmes de sécurité et d'autorisation, les problèmes de confidentialités, etc.
Furht et Villanustre (2016) présentent en plus des différentes techniques d'analyses vues précédemment, une technique d'apprentissage moins populaire mais qui a déjà fait ses preuves : le Transfer Learning. Le principe du Transfert Learning est d'améliorer un algorithme d'un certain domaine en lui fournissant les connaissances d'un autre algorithme d'un domaine relatif. Pour mieux comprendre cela, prenons l'exemple de deux personnes voulant apprendre à jouer un instrument de musique : le piano. Aucune des deux personnes n'a déjà joué du piano mais la première personne fait déjà de la guitare depuis deux ans. Il est aisé à comprendre que la première personne apprendra plus facilement et mieux à jouer du piano grâce à ses connaissances sur la musique déjà acquises (Furht et Villanustre, 2016). Les applications de cette technique sont nombreuses et sont souvent utilisés dans du NLP, dans les classifications d'images et de vidéos, etc.

Furht et Villanustre (2016) présentent différentes techniques d'analyse de données :

- ANN : artificial neural networks
- Analyse prédictive
- Machine Learning
- Signal Processing
- Ensemble learning
- Calcul linguistique
- Algortihme de recherche et de tri
- Text mining
- Data Mining
- Deep Learning
- etc.

Le Deep Learning est une technique particulièrement intéressante pour un usage sur le Big Data. Cette technique, contrairement au Machine Learning qui est un algorithme supervisé et adapté aux données tabulaires et relationnelles (la cible est connue à l'avance), permet de faire ressortir des features et des caractéristiques d'un set de données sur lequel nous n'avions pas de connaissances. L'avantage pour les Big Data est qu'il est adapté pour les grands volumes de données qui sont variés, moins structurés et plus complexes.

Enfin, comme nous l'avons vu, la distribution des données nécessite une parallèlisation du traitement. Furht et Villanustre (2016) observent 4 problèmes liés à cette parallelisation : « le choix de l'algorithme, la stratégie de décomposition de la donnée, la répartition de la charge de travail entre les nœuds (load balancing) et la communication entre les nœuds ».
Rappelons qu'une bonne architecture doit être conçue de manière à pouvoir combiner les résultats des analyses sur des données non-structurées avec les analyses sur des données structurées afin d'obtenir un résultat complet basé sur tous les types de données.

8.6. Couche de visualisation

La couche de visualisation a été présentée en dernière afin de présenter les couches en suivant le déplacement de la donnée (des sources de données aux connaissances extraites et à leur visualisation). Précisons que les couches de monitoring et de sécurité sont présentées par après car elles ne font pas partie directement du processus de création de valeur.
Les données commencent leur périple à partir des sources de données, passent par les différentes couches d'ingestion, de stockage et d'infrastructure et ensuite elles arrivent à la couche d'analyse. Ce traitement des données permet de passer de données brutes à des informations pertinentes. Enfin, l'étape finale consiste à transformer ces informations pertinentes en visualisations perceptibles et compréhensibles par l'œil humain.
L'étude des techniques de visualisation est complexe et nécessite un travail dédié uniquement à celle-ci pour en connaitre entièrement les fondements. Afin de ne pas sortir du cadre de ce cours mais tout de même donner un petit aperçu de cette discipline, voici une brève introduction basée sur le travail de Furht et Villanustre (2016) ainsi que de Sawant et Shah (2013).
Selon le travail de ces premiers, l'objectif principal de la visualisation est d'améliorer les informations diffusées de manière qu'un utilisateur puisse comprendre et interagir avec de grandes quantités de données.
Les méthodes et les outils utilisés traditionnellement pour la visualisation des connaissances extraites de l'analyse ont dû évoluer pour pouvoir tirer pleinement profit des grands volumes variés de données ainsi que pour gérer la visualisation des connaissances extraites de l'analyse des données Big Data. Ces nouveaux outils et méthodes doivent être capable de lire des données sur un cluster de machines sous Hadoop, des données de « in-memory », ou encore d'afficher les données avec une perspective panoramique.
Avec les méthodes traditionnelles, la connaissance était principalement visualisée sous forme de tableau, graphique ou schéma. Les connaissances de l'analyse Big Data nécessite de nouvelles techniques pour pouvoir visualiser les données sous une vision plus globale. Les grandes quantités de données permettent d'obtenir des connaissances globales et faire des généralités, et la visualisation de ceux-ci nécessitent une vue plus aérienne, et non granulaire, pour pouvoir comprendre les schémas qui s'en dégagent (Sawant et Shah, 2013).

Il existe différentes méthodes (Sawant et Shah, 2013) pour améliorer la qualité de la visualisation du Big Data, tels que « MashUp View Pattern » (pour améliorer la performance de MapReduce) ; « Compression Pattern » (pour améliorer la performance des opérations sans faire de l'agrégation ou du MashUp); « Zoning Pattern » (pour faciliter la localisation des données); « FirstGlim Patter » (pour voir l'essentiel des données en un seul endroit)… et ces méthodes sont bien détaillées dans le travail de Sawant et Shah (2013).
Il existe de nombreux outils commerciaux pour la visualisation du Big Data tels que :
QlikView, Tableau, SAS Visual Analytic, MicroStrategy, etc.

Ces outils sont particulièrement intéressants pour la visualisation d'analyse en temps réel et ont été conçus pour avoir une bonne intégration avec l'écosystème Hadoop. Furht et Villanustre (2016) suggèrent que les outils de visualisation doivent répondre à trois exigences :

- « Efficacité (démontrer exactement l'information contenue dans les données)
- Expressivité (garder en tête les capacités cognitives du système visuel humains)
- Pertinence (rapport coût-bénéfice de la visualisation) » (Furht et Villanustre, 2016)

8.7. Couche de sécurité

Les systèmes SGBDR traditionnels existent depuis longtemps et ont vu leurs mécanismes de sécurité se développer et évoluer au fil du temps, tels que le contrôle d'accès, le chiffrement des données, la gestion des autorisations, la gestion de la confidentialité, les audits, …
Pour les systèmes de gestion de données Big Data, les mécanismes de sécurité ne sont pas encore complètement aboutis et les systèmes possèdent toujours des vulnérabilités et des risques. En effet, lors du développement d'Hadoop, les préoccupations étaient concentrées sur la gestion des 5V des données et les questions liées à la sécurité ont été mis de côté. Pour cette raison, de nombreux services tiers à Hadoop ont proposés des outils permettant de palier à ces problèmes de sécurité, tels que Cloudera Sentry, Kerberos, etc. (Sawant et Shah, 2013).
Etant donné l'importance et la valeur que possèdent les données, il est essentiel d'avoir une couche permettant de protéger les données : accès, manipulation, partage, autorisation, … Il est bien connu de la théorie sur la sécurité informatique que celle-ci doit être élaborée dans la conception même de l'architecture de la solution informatique, et non comme une couche ou une fonctionnalité à développer indépendamment du reste des couches.
Le Big Data est particulièrement à risque pour ce type de problème à cause de sa nature même :

- de très grandes quantités de données —> la gestion du transfert et du partage des données est Dlus complexe. De même l'archive des données peut être plus compliquée à gérer et on peut omettre de supprimer des données qui doivent être archivées.
- L'architecture distribuée —> un plus grand nombre de machines impliqués dans la solution informatique signifie un plus grand nombre de vulnérabilités. De plus, cela signifie plus de communication entre les différentes machines et les différents services, ce qui signifie également plus de vulnérabilités également.

Il y existe 3 mesures principales (Sawant et Shah, 2013) pour assurer la sécurité des données lors de la communication entre les différentes couches :

- La gestion de la **confidentialité** : les données doivent être chiffrés lors du transfert entre deux couches.
- La gestion de l'**authentification** : seuls les utilisateurs authentifiés peuvent avoir accès aux données.
- La gestion de l'**autorisation** : spécifier et définir les droits d'accès aux données pour chaque type d'utilisateur.

8.8. Couche de monitoring

Etant donné le grand nombre de machines distribuées dans l'architecture d'une solution Big Data (ainsi que le grand nombre de sources de données), il est essentiel de pouvoir contrôler l'état de fonctionnement général de l'architecture pour plusieurs raisons :

- Eviter que celle-ci échoue ;
- Améliorer la performance de celle-ci.

Afin de contrôler au mieux l'état de fonctionnement, il faut fournir des outils de visualisation de l'ensemble de l'état des machines ainsi que de l'état des communications entre elles. Contrôler au

mieux ce monitoring permet d'améliorer la performance générale de la solution Big Data (Sawant et Shah, 2013).

Le troisième grand challenge du Big Data, qui concerne le management du Big Data est également complexe et nécessiterait un cours dédié spécialement pour présenter les techniques et méthodes de ce challenge. De manière générale, nous pouvons considérer deux types de résolutions de problèmes. Une résolution basée sur la technique, correspondant aux deux dernières couches présentées (gestion de la sécurité, développement de monitoring, gestion du contrôle d'accès, gestion de la protection des données, mises-en place d'une culture de la donnée au sein de l'entreprise, etc.). Et une résolution qui est d'avantage basée sur la législation et l'éthique de ces nouvelles technologies, (utilité du RGPD, réflexion sur la dangerosité de faire des algorithmes et leurs responsabilités, la confidentialité des données, …). La plupart de ces résolutions se basent sur des politiques et des règles établies par des institutions (Furht et Villanustre, 2016). (Par exemple, *la norme ISO27001 pour la sécurité de l'information*).

8.9. Cloud Computing

La conception d'une architecture pour une solution Big Data est composé de différentes couches nécessitant différents types de compétences pour être conçues ainsi que différents types de matériels hardware. L'architecture implique du calcul distribué, des clusters de machines stockant les données, des réseaux permettant la communication entre les données, ainsi qu'une gestion de la sécurité de toutes les machines, de tout le réseau, de tout les calculs et de toutes les données. Gérer l'intégralité de ces différents composants et aspects est complexe et n'est pas accessible pour toutes les entreprises.
Le Cloud Computing (Rani et Ranjan, 2014) est une technologie qui permet aux entreprises de sous-traiter le développement ou la gestion de certaines parties de leur solution informatique. Cette technologie utilise Internet pour fournir des services (en hardware et software) de location de serveurs, d'espaces de stockage, de réseaux et de logiciels. Il y a plusieurs raisons qui expliquent le recours au Cloud Computing, entre autres : minimiser les coûts d'infrastructure informatique, ne pas devoir s'occuper de la gestion de la sécurité (en hardware ou software en fonction du service fourni), obtenir de la flexibilité (extensibilité horizontale notamment) grâce au service à la demande, obtenir une plus grande puissance de calcul et/ou capacité de stockage. De plus, cette technologie a démocratisé certaines fonctionnalités (de calcul, de stockage ou de logiciel), qui n'étaient réservées qu'aux grandes entreprises pouvant s'offrir de grandes infrastructures avec de grands coûts de développement, et qui maintenant sont accessibles même aux PME ou encore aux particuliers.

Cette technologie se caractérise par 4 aspects (Rani et Rajan, 2014) :

- **Hardware et maintenance**: il n'est pas nécessaire de se préoccuper de l'aspect hardware et de la maintenance, ce qui réduit les coûts (jusqu'à une certain échelle).
- **Service à la demande**. Les services fournis peuvent être configuré en fonction des besoins de l'utilisateur.
- **Mise à jour**. Le fournisseur de service est responsable de la mise à jour de l'hardware et du software. L'utilisateur ne doit pas s'en préoccuper.
- **Extensibilité**. Les services fournis par le Cloud Computing sont très extensibles.

On distingue en général 3 types de services de Cloud Computing (OVH Cloud, 2015) :

IaaS : Infrastructure as a Service, les clients ont accès à des machines virtuelles qu'ils peuvent utiliser pour faire fonctionner un système d'exploitation et des applications. Ces machines virtuelles fonctionnent comme une machine normale, et les utilisateurs peuvent s'occuper du stockage, du réseau, de la mémoire, du CPU, etc. En comparaison avec les différentes couches de l'architecture décrite plus haut, un service IaaS s'occupe de gérer la partie de stockage ainsi qu'une partie (ou l'entièreté, en fonction du prestataire de service) de l'ingestion et de l'intégration des données.
C'est le niveau de service le plus bas, et permet principalement de minimiser les coûts d'infrastructure hardware.
PaaS : Platform as a Service, les clients ont accès à une plateforme sur lesquels ils peuvent concevoir, développer et faire fonctionner des applications. PaaS permet aux développeurs de créer et déployer une application sans à avoir à se soucier de la mémoire, du stockage et de la quantité de processeurs que l'application utilisera. Ce type de service offre un niveau d'abstraction supplémentaire que le IaaS.
SaaS : Software as a Service, les clients ont accès à un logiciel avec pour seule exigence d'avoir une machine et d'avoir un accès à internet. Pour ce type de service, le niveau d'abstraction est encore plus grand que celui de PaaS. Les avantages sont le gain de temps, la prédictibilité et la simplicité. Idéal par exemple pour une PME qui veut sous-traiter la gestion du CRM pour ne pas à avoir à investir dans de grand et couteux développement informatique.

Chaque type de service a ses avantages et désavantages. Le choix d'un type de service dépend des besoins propres du client du Cloud Computing. On peut les utiliser en même temps et indépendamment afin de répondre aux besoins des différentes couches de la solution Big Data. Il existe d'autres types de services, plus spécialisés tels que DaaS (Data as a Service), NaaS (Network as a Service), STaaS (Storage as a Service), …

Les acteurs commerciaux les plus connus du Cloud Computing sont Amazon, Microsoft, Google, HP, Intel, IBM, SalesForces, etc.
Amazon Web Service propose différents types de services : fournir des infrastructures (IaaS), fournir des plateformes de développement (PaaS) ou encore des logiciels directement utilisables pour le Big Data (SaaS) tel que Amazon Azure pour l'analyse de données.
Il existe de nombreuses options disponibles répondant aux besoins d'un système d'information traitant du Big Data. Ces options peuvent être complètes et haut niveaux (SaaS) ou peuvent être plus bas niveaux (IaaS). Les grands acteurs commerciaux proposant ces options sont, notamment, Amazon, SAP, Oracle, IBM (Fowler et Sadalage, 2013).

Grâce au Cloud Computing, à la démocratisation de l'hardware et des logiciels opensource, l'accès au Big Data Processing s'est élargi au plus grand nombre et n'est plus l'unique fait des grandes entreprises.

9.Conclusion

L'objectif de ce cours était de fournir une présentation globale du Big Data, pour connaître les implications, commerciales et techniques, d'une telle technologie. Nous estimons que la compréhension du Big Data doit inclure les deux aspects de celui-ci : une compréhension des techniques informatiques utilisées mais également une compréhension des besoins orientés business et des besoins techniques. Ces deux aspects sont intimement liés et l'un trouve son origine dans l'autre. Pour cette raison, nous avons tenté de montrer l'imbrication des deux aspects entre eux tout au long de ce cours.
Les avancées techniques et technologiques ont transformé le paradigme des données et de nouveaux besoins sont apparus : le stockage d'immenses quantités de données (Volume), le traitement de données fortement variées (Variété) et la rapidité de ce stockage et ce traitement (Vélocité). C'est pour cette raison qu'on parle des 3V du Big Data :

- Le Volume des données : passage de données en Kb à Tb/Eb/Pb;
- La Variété : passage de données structurées et uniformes à des données non-structurées et très variées ;
- La Vélocité : passage de données stables dans le temps et peu dynamiques à des données très dynamiques, évoluant très rapidement (en temps réel ou presque en temps réel).

Cependant les méthodes et les outils traditionnelles de la BI ne pouvaient pas répondre à ces besoins, pour des raisons structurelles et de business. Différencions les raisons structurelles qui sont liées à la technicité (hardware utilisé, SGBD utilisé, système de fichiers utilisés, etc.) et les raisons de business qui sont liées aux objectifs des fonctionnalités (assurer la consistance dans une transaction, etc.). Pour pouvoir répondre à ces nouveaux besoins, des méthodes et des outils ont vu le jour : les systèmes distribués (Hadoop), les bases de données NoSQL, les calculs parallèles, l'extensibilité horizontale des outils et des framework, etc. A partir de ce nouveau paradigme de données, nous avons défini l'objectif du Big Data: pouvoir stocker, traiter et analyser les données du 3V pour en extraire des connaissances qui apportent de la valeur à une entreprise.
Etant donné que la distribution des systèmes a été une différence majeure entre la BI et le Big Data, nous avons présenté différents modèles de distribution : single-server, sharding et réplication (maitre-esclave et peer-to-peer). Ces différents modèles se différencient par leur niveau de consistance et leur niveau de résilience et robustesse (compromis entre consistance et résilience). La distribution des données possède les trois grands avantages de permettre la manipulation (stockage et traitement) de grandes quantités de données, l'amélioration des performances ainsi que l'amélioration de différents besoins non-fonctionnels (disponibilité, résilience, flexibilité, etc.) Mais ceci se fait au détriment d'autres besoins non-fonctionnels telle que la consistance des données. Nous avons présenté les deux grands modèles de consistance : ACID et BASE ainsi qu'un des grands compromis du Big Data qui est le théorème du CAP. Enfin, nous avons également différencié l'extension horizontale de l'extension verticale et nous avons démontré l'importance de l'extension horizontale pour des grands volumes de données, ce qui explique, entre autres, l'utilisation de système distribué pour les données Big Data. La gestion de ces nouveaux types de données implique l'utilisation de nouvelles méthodes et outils tout le long du cycle de la donnée. Ainsi pour répondre aux différents besoins décrits plus haut, une certaine architecture commune pour le Big Data a été admise : système de serveurs distribué (Hadoop) sur lesquels fonctionnent des bases de données (NoSQL) et sur lesquels fonctionne également le calcul parallèle (MapReduce).
Nous avons vu que le cycle de la donnée est composé de trois parties : la gestion de la donnée elle-même (stockage, manipulation et traitement) , les analyses appliqués sur la donnée et enfin, le management de la donnée qui encadre ces deux premières parties (gestion de la sécurité, contrôle des accès, monitoring, etc.). Ce cours s'est principalement concentré sur la première partie du cycle de la donnée.
Les bases de données relationnelles n'étant plus adaptées à ce nouveau paradigme de données, un nouveau type de base de données a émergé : le NoSQL. Nous avons présenté les quatre grandes familles de bases de données NoSQL : clé-valeur, colonnes, document et graphe ; les implications de celles-ci (modèle de consistance ACID vs BASE, théorème du CAP, extensibilité horizontale, etc.). Pour chaque famille NoSQL, le fonctionnement et les caractéristiques ainsi que les applications typiques ont été présentés. Chacune possède des avantages et des désavantages et sont utilisés pour différents types de fonctionnalités ou d'applications. Pour cette raison, nous avons parlé du

concept de Polyglot Persistance qui désigne le fait d'avoir recours à plusieurs types de technologies de stockage de données pour les besoins différents d'une même application. Enfin, un tableau de comparaison des différents Data Model ainsi que des critères pour le choix d'une base de données ont été proposés.

L'architecture du Big Data consiste en un système distribué sur un très large clusters de machines qui repose sur une infrastructure physique distribuée, un système de fichiers distribués (HDFS) , des bases de données distribuées (NoSQL) et un système de traitement de la donnée distribuée (calculs parallèles : MapReduce). L'architecture se divise en plusieurs couches et sous-couches : nous avons définis les 7 grandes couches principales : la couche d'ingestion, la couche d'infrastructure, la couche de stockage, la couche de visualisation, la couche d'analyse et de management ainsi que la couche de sécurité et de monitoring. Il est important de saisir les différentes couches de cette architecture : une couche d'infrastructure (Hadoop) qui permet de stocker les données physiquement sur les disques des différentes machines, une couche d'infrastructure conceptuelle (HDFS) et un système de gestion de bases de données (NoSQL) adapté à ces deux couches d'infrastructures.
Enfin, nous avons présenté le concept de Cloud Computing qui est une technologie qui permet aux entreprises de sous-traiter le développement ou la gestion de certaines parties de leur solution informatique. Cette technologie utilise Internet pour fournir des services (en hardware et software) de location de serveurs, d'espaces de stockage, de réseaux et de logiciels.

Pour résumer, les données du Big Data sont stockées dans des bases de données NoSQL (distribuées, extensibles, flexibles, consistance BASE et résilientes), qui repose sur une infrastructure conceptuelle HDFS (système de fichiers distribués qui divise les fichiers en bloc de données et qui les réparti dans les différentes machines du système), qui, elle-même, repose sur l'infrastructure physique d'Hadoop. Enfin, le traitement de la donnée (manipulations, opérations, calculs, etc.) de ce système distribué de serveurs et de ces données réparties et répliquées sur plusieurs serveurs est géré par les SGBD NoSQL et en utilisant le calcul parallèle MapReduce. Map consiste en la division de la charge de calculs en sous-tâches et la réparation de ces taches parmi les serveurs du cluster et Reduce consiste à un rassemblement de chaque résultat des sous-tâches en un résultat final.

Références

Akerkar, R. (Ed.). (2013). *Big data computing*. Crc Press.

Ali, W., Shafique, M. U., Majeed, M. A., & Raza, A. (2019). Comparison between SQL and NoSQL databases and their relationship with big data analytics. *Asian Journal of Research in Computer Science*, 1-10.

Amazon Web Service. (2018). Qu'est-ce qu'Amazon DynamoDB, Amazon Web Services, Inc. https://aws.amazon.com/fr/nosql/key-value/

Apache Cassandra. (2021). Apache Cassandra | Apache Cassandra Documentation. https://cassandra.apache.org/_/cassandra-basics.html

Apache Hbase. (2019). Apache HBase – Apache HBaseTM Home. https://hbase.apache.org

Aye, A Platform for Big Data Analytics on Distributed Scale- out Storage System A Platform for Big Data Analytics on Distributed Scale-out Storage System Kyar Nyo Aye University of Computer Studies, Yangon A thesis submitted to the University of Computer Studi,no. November, 2015.

Baez, E. (2021, 27 juillet). Column Store vs. Row Store : Their Differences & How to Choose. Scalyr. https://www.scalyr.com/blog/column-store-vs-row-store/

Bjeladinovic, S., Marjanovic, Z., & Babarogic, S. (2020). A proposal of architecture for integration and uniform use of hybrid SQL/NoSQL database components. *Journal of Systems and Software*, *168*, 110633.

Boncz, P. A., Manegold, S., & Kersten, M. L. (1999, September). Database architecture optimized for the new bottleneck: Memory access. In *VLDB* (Vol. 99, pp. 54-65).

Borthakur, D. (2008). HDFS architecture guide. *Hadoop apache project*, *53*(1-13), 2.

Brewer, E. A. (2000, July). Towards robust distributed systems. In *PODC* (Vol. 7, No. 10.1145, pp. 343477-343502).

Brewster, C. (s. d.). *Front-End vs. Back-End : What's the Difference? (2020)* trio.dev. https://trio.dev/blog/front-end-vs-back-end

Chan, J. O. (2013). An architecture for big data analytics. *Communications of the IIMA*, *13*(2), 1.

Chen, H., Chiang, R. H., & Storey, V. C. (2012). Business intelligence and analytics: From big data to big impact. *MIS quarterly*, 1165-1188.

Chen, J. K., & Lee, W. Z. (2019). An Introduction of NoSQL Databases based on their categories and application industries. *algorithms*, *12*(5), 106.

Cleve, A. et Faulkner, S. (2021). Big Data : ingénierie et traitement. Cours à l'Université de Namur.

Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.

Debortoli, S., Müller, O., & vom Brocke, J. (2014). Comparing business intelligence and big data skills. *Business & Information Systems Engineering*, *6*(5), 289-300.

Dehdouh, K., Bentayeb, F., Boussaid, O., & Kabachi, N. (2015). Using the column oriented NoSQL model for implementing big data warehouses. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)* (p. 469). The Steering
Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).

Demaine, E., & Devadas, S. (2011, octobre). Introduction to Algorithms. MIT OpenCourseWare. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/

Demchenko, Y., De Laat, C., & Membrey, P. (2014, May). Defining architecture components of the Big Data Ecosystem. In *2014 International conference on collaboration technologies and systems (CTS)* (pp. 104-112). IEEE.

Desk, B. (2021, 19 décembre). Top 12 NoSQL Document Databases. PAT RESEARCH : B2B Reviews, Buying Guides & Best Practices. https://www.predictiveanalyticstoday.com/top-nosql-document-databases/

Dowling, N. (2000). Database Design and Management Using Access. Cengage Learning EMEA.

DuBois, P. (2008). *MySQL*. Pearson Education.

Durcevic, S. (2020, 21 octobre). *18 Examples Of Big Data Analytics In Healthcare That Can Save People*. datapine.com. https://www.datapine.com/blog/big-data-examples-in-healthcare/

Ermacore, S., & Kendrick, N. (2016). Choisissez les composants de stockage et de mémoire. OpenClassrooms. https://openclassrooms.com/fr/courses/7210326-montez-un-pc/7402460-choisissez-les-composants-de-stockage-et-de-memoire

Evans, E., & Evans, E. J. (2004). Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional

Foote, K. D. (2019, 18 octobre). Fundamentals of Document Databases. DATAVERSITY. https://www.dataversity.net/fundamentals-of-document-databases/

Foote, K. D. (2020, 27 janvier). Understanding Key-Value Databases. DATAVERSITY. http://www.dataversity.net/understanding-key-value-databases/#

Furht, B., & Villanustre, F. (2016). Introduction to big data. In *Big data technologies and applications* (pp. 3-11). Springer, Cham.

Gaudiaut, T. (2021, 19 octobre). *Le Big Bang du Big Data*. fr.statista.com https://fr.statista.com/infographie/17800/big-data-evolution-volume-donnees-numeriques-genere-dans-le-monde/

GeeksforGeeks. (2021b, 22 décembre). Hashing | Set 3 (Open Addressing). https://www.geeksforgeeks.org/hashing-set-3-open-addressing/

GeeksforGeeks. (2021, 9 mars). *Hadoop - File Blocks and Replication Factor*. https://www.geeksforgeeks.org/hadoop-file-blocks-and-replication-factor/

GeeksforGeeks. (2021a,18 août). Implementing own Hash Table with Open Addressing Linear Probing. https://www.geeksforgeeks.org/implementing-hash-table-open-addressing-linear-probing-cpp/

GeeksforGeeks. (2021c, 22 décembre). Hashing | Set 3 (Open Addressing). https://www.stitchdata.com/columnardatabase/

George, S. (2013). NoSQL–NOT ONLY SQL. International Journal of Enterprise Computing and Business Systems, 2(2).

Greg, T. (2021, 16 janvier). Le théorème CAP : Bien choisir sa base de données - DataScientest.com. Formation Data Science | DataScientest.com. https://datascientest.com/theoreme-cap

GridGain. (2021). In-Memory Cache. GridGain Systems. https://www.gridgain.com/resources/glossary/in-memory-computing-platform/in-memory-cache

Hachet, N. (2019, 13 décembre). L'architecture REST expliquée en 5 règles. Blog de Nicolas

Hachet. https://blog.nicolashachet.com/developpement-php/larchitecture-rest-expliquee-en-5-regles/

Hazelcast. (2019, 9 novembre). Key-Value Stores Explained. Advantages & Use Cases. https://hazelcast.com/glossary/key-value-store/

Hosting Data. (2021, 10 décembre). Complete NoSQL Database Guide for Beginners. https://hostingdata.co.uk/how-to-use-nosql-databases-guide/

Ian, I. (2016, 23 juin). What is a Column Store Database ? | Database.Guide. Database Guide. https://database.guide/what-is-a-column-store-database/

IBM. (2021, 15 juillet). *Hadoop vs. Spark : What' ; s the Difference?* https://www.ibm.com/cloud/blog/hadoop-vs-spark

Ilieva, G., Yankova, T., & Klisarova, S. (2015, September). Cloud business intelligence: contemporary learning opportunities in MIS training. In *CEUR Workshop Proceedings of the 2015 Balkan Conference in Informatics* (Vol. 1427, pp. 25-32).

JanusGraph. (2017). JanusGraph. https://janusgraph.org

Jeremy, R. (2021, 28 mai). Introduction à la Business Intelligence - DataScientest.com. Formation Data Science | DataScientest.com. https://datascientest.com/business-intelligence

Json. (2002). Json. https://www.json.org/json-fr.html

Kambau, R. A., & Hasibuan, Z. A. (2017, September). Unified concept-based multimedia information retrieval technique. In *2017 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)* (pp. 1-8). IEEE.

Kituta Ezéchiel, K., Agarwal, R., & Kant, S. (2019, janvier). *A systematic review on Distributed Databases Systems and their techniques.* www.researchgate.net.

Kumar, D. (2021, 23 décembre). What is Columnar Database ? - A Comprehensive Guide 101.

Learn | Hevo. https://hevodata.com/learn/columnar-databases/

Lecomte, S. (2021, 5 janvier). La différence entre Business Intelligence (BI) et Big Data. Alphalyr. https://alphalyr.fr/difference-bi-business-intelligence-big-data/

Linden I. (2021). Business Intelligence. Cours à l'Université de Namur.

Lourenço, J. R., Cabral, B., Carreiro, P., Vieira, M., & Bernardino, J. (2015). Choosing the right NoSQL database for the job: a quality attribute evaluation. *Journal of Big Data*, *2*(1), 1-26.

Marcus, A. (2018). The Architecture of Open Source Applications : The NoSQL Ecosystem. Aosabook. http://www.aosabook.org/en/nosql.html

Margot, P. (2018, 2 juin). NoSQL : Tout comprendre sur les bases de données non relationnelles. Formation Data Science | DataScientest.com. https://datascientest.com/nosql

Marr, B. (2018, 21 mai). *How Much Data Do We Create Every Day ? The Mind-Blowing Stats Everyone Should Read.* Bernardmarr.com. https://bernardmarr.com/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/

McCormick, S. (2021, 2 avril). *Sharding a SQL Server database.* blog.pythian.com. https://blog.pythian.com/sharding-sql-server-database/

Meier, A., & Kaufmann, M. (2019). *SQL & NoSQL databases*. Springer Fachmedien Wiesbaden

Mlitz, K. (2021, 22 janvier). *Big data market size revenue forecast worldwide from 2011 to 2027*. statista.com https://www.statista.com/statistics/254266/global-big-data-market-forecast/

Muskan. (2021, 27 juin). *5 Applications of Big Data in Law Industry.* analyticssteps.com, https://www.analyticssteps.com/blogs/5-applications-big-data-law-industry

Namuag, P. (2021, 20 août). Redis vs DynamoDB (a comparison). Severalnines. https://severalnines.com/database-blog/redis-vs-dynamodb-comparison

Nayak, A., Poriya, A., & Poojary, D. (2013). Type of NOSQL databases and its comparison with relational databases. *International Journal of Applied Information Systems*, 5(4), 16-19.

Nebra, M., & Gonnage, R. (2017). Stockez et retrouvez des données grâce aux tables de hachage. OpenClassrooms. https://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c/19978-stockez-et-retrouvez-des-donnees-grace-aux-tables-de-hachage

Neo4j. (2021). Betweenness Centrality - Neo4j Graph Data Science. Neo4j Graph Database Platform. https://neo4j.com/docs/graph-data-science/current/algorithms/betweenness-centrality/

NewbeDEV. (2021). Chained Hash Tables vs. Open-Addressed Hash Tables. https://newbedev.com/chained-hash-tables-vs-open-addressed-hash-tables

OVH Cloud. (2015). *IaaS, PaaS et SaaS*. OVHcloud. https://www.ovhcloud.com/fr-tn/public-cloud/cloud-computing/iaas-paas-saas/

Petrov, C. (2021, 4 décembre). *25+ Impressive Big Data Statistics for 2021*. Techjury. https://techjury.net/blog/big-data-statistics/#gref

Petrov, C. (2022, 4 janvier). 25+ Impressive Big Data Statistics for 2021. TechJury. https://techjury.net/blog/big-data-statistics/#gref

Raichand, P., & Aggarwal, R. R. (2013). A short survey of data compression techniques for column oriented databases. *Current Trends in Information Technology*, *3*(2), 1-6.

Rani, D., & Ranjan, R. K. (2014). A comparative study of SaaS, PaaS and IaaS in cloud computing. *International Journal of Advanced Research in Computer Science and Software Engineering*, *4*(6).

Raoul, A. W. E. G. (2021, 20 août). Base de données NoSQL : l'essentiel sur le modèle clé-valeur. LeMagIT. https://www.lemagit.fr/conseil/NoSQL-lessentiel-sur-le-modele-cle-valeur

Roser, M. (2013, 11 mai). Technological Progress. Our World in Data. https://ourworldindata.org/technological-progress

Sadalage, P. J., & Fowler, M. (2013). *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education.

Sawant, N., & Shah, H. (2013). Big data application architecture. In *Big data Application Architecture Q & A* (pp. 9-28). Apress, Berkeley, CA

Stitch. (2020). OLTP and OLAP : a practical comparison | Stitch resource. https://www.stitchdata.com/resources/oltp-vs-olap/

Talend. (2019). Redondance des données : définition et guide. Talend - A Leader in Data Integration & Data Integrity. https://www.talend.com/fr/resources/guide-redondance-donnees/

Thiessen, M. (2012, 12 novembre). *How Obama trumped Romney with big data.* washingtonpost.com. https://www.washingtonpost.com/opinions/marc-thiessen-how-obama-trumped-romney-with-big-%20data/2012/11/12/6fa599da-2cd4-11e2-89d4-040c9330702a_story.html

Tinnefeld, C., Krueger, J., Schaffner, J., & Bog, A. (2008, Septembre). A database engine for flexible real-time available-to-promise. In *2008 IEEE Symposium on Advanced Management of Information for Globalized Enterprises (AMIGE)* (pp. 1-5). IEEE

Tondak, A. (2021, 4 septembre). Structured Vs Unstructured Data Vs Semi-Structured Data. Cloud Training Program. https://k21academy.com/microsoft-azure/dp-900/structured-data-vs-unstructured-data-vs-semi-structured-data/

Weglarz, G. (2004). Two worlds of data unstructured and structured. *Information Management*, *14*(9), 19.

Whet, M. (2012, 13 novembre). Mitt Romney's Big Data Fail and Its Ideological Roots. Chicago Magazine. https://www.chicagomag.com/city-life/november-2012/mittromneys-big-data-fail-and-its-ideological-roots/

Wikipedia contributors. (2018, 28 décembre). Traitement massivement parallèle. Wikidepia. https://fr.wikipedia.org/wiki/Traitement_massivement_parallele

Wikipedia contributors. (2020, 23 décembre). *Hadoop*. Wikipedia. https://fr.wikipedia.org/wiki/Hadoop

Wikipedia contributors. (2021, 24 décembre). Cloud computing. Wikidepia. https://fr.wikipedia.org/wiki/Cloud_computing

Wikipedia contributors. (2021a, décembre 15). Loi de Moore. Wikidepia. https://fr.wikipedia.org/wiki/Loi_de_Moore

Xiaoqian, S. (2019, 25 janvier). Introduction to Big Data. Sun's Landscape. http://landscapesun.com/bigdata.html

Yemini, B. (2019, 30 octobre). Vertical Scaling in a Horizontal World. Turbonomic. https://blog.turbonomic.com/vertical-scaling-in-a-horizontal-world