Chapitre 2

Le cloud computing et ses services

Le cloud est très à la mode et utilisé par des nombreux acteurs pour décrire des concepts qui peuvent être très différents. Il en perd souvent sa pertinence. Dans ce rapport, il est donc nécessaire de le replacer dans un contexte scientifique.

Dans ce chapitre, nous ferons un rapide tour d'horizon des évolutions qui ont mené au cloud. Nous proposerons ensuite un tour d'horizon de la solution cloud la plus réputée sur le marché : *Amazon EC2*. Ce tour d'horizon nous permettra de mieux comprendre le cloud et le concepts de service qu'il met en avant.

Dans un contexte scientifique, nous apporterons ensuite une définition du cloud et énoncerons les propriétés fondamentales des services qu'il propose. Nous décrirons ensuite le théorème CAP, central à ce mémoire, qui énonce l'impossibilité pour le service base de données de maintenir simultanément consistance, tolérance à la partition et haute disponibilité.

2.1 Contexte

2.1.1 Origine et modèles

Les 20 années précédentes ont vu la démocratisation de l'informatique, d'internet et des réseaux de hauts débits. Le cloud suit cette chute des prix et l'émergence de virtualisation des serveurs d'applications : il consiste en la coopération de ressources informatiques, situées ou non dans le même parc informatique. Cette coopération sous-entend que ces machines vont travailler ensemble dans un but commun via des protocoles et des standards de communication qui sont basés sur ceux d'internet [32].

Le cloud se veut capable de s'auto-gérer, de fournir un certain degré d'automatisme. Tous ces mécanismes sont transparents pour l'utilisateur final qui pense le cloud tel une série de services et fait donc totalement abstraction de tout autre composant. L'utilisateur final voit la solution cloud comme étant dynamique, capable de s'adapter à la charge à laquelle il est soumis, comme une solution qu'il peut payer à la demande.

> "*Pour les entreprises utilisatrices (du grand compte multinational à la PME locale), le Cloud Computing peut se définir comme une approche leur permettant de disposer d'applications, de puissance de calcul, de moyens de stockage, etc. comme autant de « services ». Ceux-ci seront mutualisés, dématérialisés (donc indépendants de toutes contingences matérielles, logicielles et de communication),*

contractualisés (en termes de performances, niveau de sécurité, coûts...), évolutifs (en volume, fonction, caractéristiques...) et en libre-service." [32]

Il existe trois modèles de cloud :

- Le cloud *privé* : les ressources physiques sont entièrement prises en charge par l'entreprise.
- Le cloud *public* : il est externe à l'organisation, géré par un prestataire externe et accessible via internet (telle la solution proposée par Amazon). Les services peuvent donc être hébergés physiquement sur la même machine qu'un autre service extérieur.
- Le cloud *hybride* : il s'agit d'un mélange des deux précédents. Typiquement, lorsqu'une société vient à manquer de ressources physiques, elle peut louer des services à un prestataires de cloud public. Les deux solutions seront alors amenées à partager applications et données via des canaux de communication sécurisés.

L'abstraction faite via la virtualisation peut être extrapolée. On voit ainsi naître le paradigme de *service*. Traditionnellement, on décrivait le cloud tel une architecture comportant 3 couches de services (figure 2.1) :

- *Infrastructure as a Service* (IaaS) : dans ce modèle, le client dispose d'une infrastructure informatique hébergée sur laquelle il a un accès complet (sans restriction).A la différence des services traditionnels, l'infrastructure mise au service du client n'est plus une infrastructure physique (un parc de serveur) mais une infrastructure virtualisée.
- *Platform as a Service* (PaaS) : le fournisseur met à disposition un environnement fonctionnel et performant. Le client ne doit plus qu'y déployer son application.
- *Software as a Service* (SaaS) : ce modèle permet de déporter l'application chez un tiers.

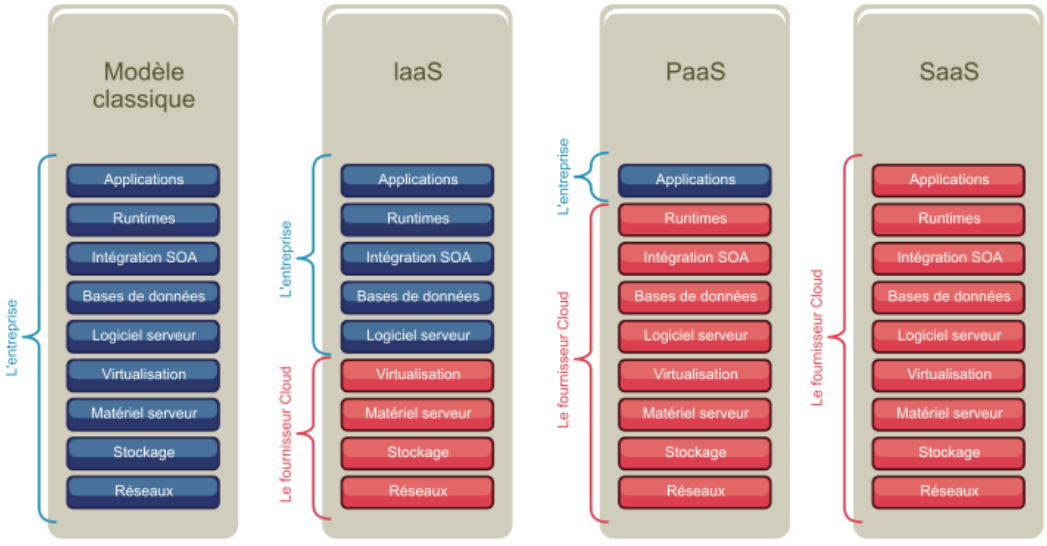

FIGURE 2.1 – Les 3 modèles de cloud[32]

En fait, l'expansion du cloud fait naître le modèle de "XaaS" signifiant "Tout comme un service" (*everything-as-a-service* en anglais). On voit par exemple apparaitre la notion de *Human as a Service*(HuaaS) [15] qui caractérise une couche supérieure à SaaS correspondant à une ressource humaine élastique. Cette intelligence "artificiellement artificielle" peut, par exemple, servir pour des décisions arbitraires comme choisir les vidéos les plus intéressantes à afficher (pour un système comme Youtube). Le service Amazon Mechanical Turk [2] s'inscrit

dans cette couche. Cette classification en couches de services quitte le domaine de ce mémoire mais reste un domaine d'étude fort intéressant.

2.1.2 Caractéristiques spécifiques et capacités

Outre les caractéristiques techniques (sur lesquelles nous reviendrons plus tard), distinguons, parmi les différentes caractéristiques essentielles et relevantes, les non-fonctionnelles et les économiques [37].

Les aspects non-fonctionnels décrivent les propriétés intrinsèques du cloud. Parmi ces aspects nous listons :

- *L'élasticité* : il s'agit d'une des caractéristiques les plus essentielles dans notre vision du cloud. Elle définit la capacité d'une infrastructure donnée à s'adapter de manière dynamique au changement. L'élasticité fait intervenir la capacité à passer à l'échelle mais aussi l'agilité.
- *La capacité à s'adapter* : le cloud doit fournir un ensemble d'automatismes lui permettant de s'auto-gérer. Son administration devra nécessiter le minimum possible d'interventions humaines.
- *La qualité de service* : est un autre aspect essentiel du cloud ; à l'aide de métriques telles que le temps de réponse, le nombre d'opérations à la seconde, le service fournit des garanties à ses utilisateurs. Il n'appartient plus à l'utilisateur de devoir décider quelles ressources déployer mais plutôt de définir des bornes que le service doit satisfaire. Le cloud s'adaptera de manière à assurer ses bornes.
- *La haute disponibilité* : en jouant sur des données répliquées dans des centres de données différents, le cloud doit fournir un service fiable, non sensible à la défaillance d'une instance voire d'un centre de données.

Les aspects économiques du cloud séduisent de plus en plus les sociétés. Parmi ces aspects, nous listons :

- *la réduction des coûts* : son modèle de paiement à l'utilisation (*Pay Per Use* en anglais) signifie que l'utilisateur paie uniquement le service en fonction de son taux d'utilisation (alors qu'il payait par forfait auparavant).
- *Un retour sur l'investissement* : Le paiement à l'utilisation est particulièrement intéressant pour les entreprises de petite taille qui peuvent à présent profiter des avantages d'un service fonctionnel dès le départ. L'idée sous-jacente est donc la suivante : le service deviendra coûteux pour une société dans la mesure où il est fort utilisé, c'est-à-dire à la condition qu'il lui rapporte de l'argent. On passe dès lors de dépenses d'investissement en capital (l'achat de serveurs d'application) aux dépenses d'exploitation (l'achat de ressources consommables).
- *Une démarche écologique* : l'allocation de ressources à la stricte nécessité permet de réduire la consommation énergique des parcs informatiques. Outre l'aspect économique, ces réductions énergétiques permettent de diminuer l'empreinte écologique de la société.

2.1.3 Amazon

Après avoir introduit le cloud, son contexte et ses caractéristiques, faisons un rapide détour par la description d'une solution cloud reconnue pour sa stabilité et le nombre considérable d'applications qu'elle héberge : *Amazon*. Disposant d'une architecture cloud très aboutie, Amazon fournit à ses clients une liste de services ; le plus basique, l'*elastic compute cloud*,

peut être vu de manière abstraite comme une instance dont les ressources pourraient être ajustées à la demande.

Ce n'est qu'en combinant ce service basique à d'autres services (tels que le *CloudWatch*, l' *Auto Scaling*) que le client commencera à disposer d'une architecture qu'il exploitera tel son propre cloud. Afin de mieux comprendre le concept de services et l'interaction existant entre eux, reprenons ci-dessous quelques services proposé par Amazon. Ces services sont de type SaaS ; le client a accès à un service dont l'infrastructure sous-jacente lui est abstraite.

> "*Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers.*" [1]

Amazon Elastic Compute Cloud propose un **service** dont la capacité de calcul est redimensionnable. Le terme "service" met en évidence la volonté de faire abstraction des composants physiques tandis que "redimensionnable" fait référence à la possibilité d'adapter la capacité de calcul de cette instance virtuelle à la demande du client. L'utilisateur du service profitera dès lors pleinement des avantages cloud sans avoir même à considérer l'environnement cloud.

CloudWatch est un service qui fournit une surveillance pour les autres services Amazon. Il fournit une visibilité des ressources (usage CPU, mémoire) utilisées et des performances de ces services. Cette visibilité permettra (lorsqu'on couple *CloudWatch* à un autre service) de déclencher certains triggers forts utiles.

Auto Scaling, couplé au service EC2 et *CloudWatch*, est un service qui permet d'ajuster la capacité de EC2 en fonction de la demande. Ce service définit des conditions et des actions à effectuer lorsqu'elles sont atteintes ; par exemple, on peut décider d'ajouter 3 nouvelles instances EC2 au parc lorsque l'utilisation moyenne de l'unité centrale atteint 70 %. *Auto Scaling* permet aussi de s'assurer d'avoir un parc d'instances saines de taille fixe en décidant donc de remplacer toute instance défaillante par une nouvelle saine.

Elastic Load Balancing (traduit par *Equilibreur de charge élastique*) distribue automatiquement le trafic entrant à travers de multiples instances EC2 appartenant, ou non, à la même zone de disponibilité. *Elastic Load Balancing* détecte aussi les défaillances d'instances (appartenant à son domaine) et redirige le trafic qui leur était destiné (vers d'autres instances saines) jusqu'à ce que les instances défaillantes soient assainies. On peut dès lors créer une tolérance à la défaillance [1] : en plaçant des instances EC2 dans différentes zones de disponibilité, le service *Elastic Load Balancing* peut automatiquement gérer le trafic (le diriger vers des instances saines). Couplé à *Auto Scaling*, on peut s'assurer que le nombre d'instances sous *Elastic Load Balancing* ne soit jamais inférieur à un nombre voulu ou bien s'assurer que le temps de latence d'une application n'excède jamais un seuil fixé par l'administrateur.

Simple Queue Service (SQS) est une file d'attente fiable et capable de monter à l'échelle. Utilisée pour transmettre des messages entre différents services, elle facilite la création d'un flux de travail automatisé.

2.2 Définition du cloud

Dans cette section, nous tenterons une approche plus théorique des concepts énoncés précédemment.

1. Nous définirons le terme de *tolérance à la défaillance* dans la section 2.2.

Le cloud dans la littérature

> "*A cloud (...) is an "elastic execution environment of resources involving multiple stakeholders and providing a metered service at multiple granularities for a specified level of quality (of service)*"
>
> EU Expert Group [37]

> "*Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*"
>
> National Institute of Standards and Technology [29]

Ces deux définitions ont retenu notre attention. Chacune propose des paradigmes fort intéressants même si réellement différents. Essayons de les souligner :

Les experts de l'Union Européenne mettent en avant la notion d'*élasticité* inhérente à notre mémoire. Ils soulignent aussi la nature *hétérogène* de ses acteurs (fournisseurs et clients) et enfin la notion de *qualité des services* fournis. Soulignons que cette notion de qualité de services est un modèle essentiel du cloud qui, malheureusement, quitte le cadre de ce mémoire.

L'institut national des standards et de la technologie met en avant les caractéristiques d'*adaptabilité* et, sans la nommer comme telle, l'élasticité du cloud qu'il définit comme la capacité de passer *rapidement* à l'échelle en nécessitant un minimum d'effort humain.

On remarque que les deux définitions proposent une certaine abstraction en parlant de ressources informatiques (définissant un large panel de composants). Un second point commun entre ces deux définitions, leur haut degré d'abstraction, nous pousse à proposer notre propre définition du cloud (dans la suite de cette section) que nous voudrons plus précise.

Cluster et cloud

Définition 1.

Un **cluster** d'ordinateur est un parc d'ordinateurs interconnectés (on parle souvent de *grappe*) afin de partager leurs ressources dans un but commun.

Définition 2.

Un **cloud** est un modèle d'architecture qui fournit un parc d'instances virtuelles capable de s'auto-gérer et d'adapter ses capacités (modulo son support physique) de manière dynamique et automatique à la charge à laquelle il est soumis.

Le cluster est donc de manière assez basique l'architecture matérielle nécessaire au déploiement d'une solution cloud. La définition du cloud nécessite quant à elle quelques éclaircissements :

- Les ressources que propose le cloud sont virtuelles. On parle donc de ressources logiques (par opposition aux ressources physiques).
- L'architecture est capable de s'auto-gérer. Est comprise dans l'architecture toute la couche de gestion des instances, des défaillances, etc.
- Le cloud est capable de s'adapter à la charge en changeant le nombre de ses instances, ou en modifiant leurs caractéristiques.
- Il est capable de s'adapter de manière dynamique et automatique ; le cloud est capable de détecter lui-même ses propres défaillances et de ré-instancier les modules nécessaires à son bon fonctionnement.
- Il est aussi capable de relever les métriques relevantes, de déterminer s'il procure un service satisfaisant et, s'il ne le fait pas, d'augmenter ses capacités (de s'adapter à la charge).

Il existe naturellement une limite à sa capacité de s'adapter à la charge ; le cloud reposant sur un cluster, il ne pourra excéder les ressources dont il dispose. Remarquons néanmoins qu'un modèle public ou hybride semble dès lors assumer des variations de charge plus importantes. Mais cette capacité est toujours limitée par les ressources physiques.

Cloud et grille de calcul

La définition précédente peut provoquer une confusion entre le cloud et la grille de calcul (*grid* en anglais). Effectivement, leurs architectures sont assez similaires mais ils sont destinés à des fonctions bien différentes. Le cloud, comme nous l'avons vu, est destiné à être capable de monter en charge c'est-à-dire traiter un nombre important de requêtes concurrentes.

La grille de calcul est plutôt destinée à traiter un nombre plus réduit de requêtes. Ces requêtes sont, en règle générale, bien plus complexes et peuvent facilement être divisées en sous-requêtes qui seront adressées à d'autres nœuds. La figure 2.2 nous donne un aperçu visuel de cette différence.

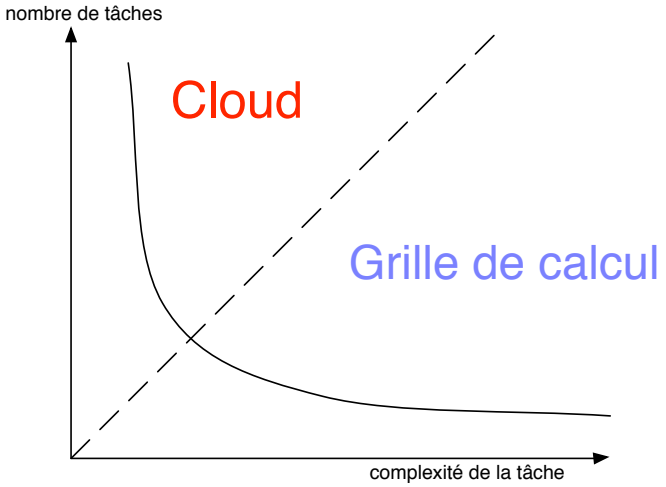

FIGURE 2.2 – Cloud Vs Grille de calcul

Cette différence de vocation fait que le cloud et la grille de calcul se pensent différemment. Néanmoins, on constate que certaines approches pensées pour l'un peuvent se décliner pour l'autre. Récemment, le concept de **distribution-réduction** (*Map-Reduce* en anglais), pensé à l'origine pour la grille de calcul, semble particulièrement bien s'adapter au cloud.

2.3 Propriétés fondamentales des services

Après ce tour d'horizon du cloud, décrivons à présent les propriétés fondamentales des services sur cloud. Notons que le SGBD est un service du cloud. A ce titre, toutes les définitions suivantes sont immédiatement applicables aux bases de données. Nous conseillons donc au lecteur de les lire en pensant au service de base de données.

2.3.1 La haute disponibilité

Définition 3.

Un service est dit **hautement disponible** (*High available* en anglais) si toute requête reçu par un nœud n'étant victime d'aucune défaillance retourne un résultat.

traduit de [26]

La haute disponibilité d'un service caractérise sa capacité à assurer son effective accessibilité durant une période donnée. Le service devra donc pouvoir détecter les points de défaillances et réduire l'impact de leur potentielle défaillance grâce à la mise en place de techniques de redondance et/ou réplication.

2.3.2 Passer à l'échelle

Un des principaux atouts d'une solution cloud est sa **capacité à passer à l'échelle** que nous décrirons plus loin. Définissons d'abord sa capacité à monter à l'échelle :

Définition 4.

La capacité à monter à l'échelle d'un service est sa capacité à pouvoir assumer une production constante lorsque le nombre de requêtes augmente.

Cette définition ne fait intervenir aucune notion de dynamisme ; quel que soit le moyen d'étendre ses capacités, un cloud est capable de monter à l'échelle s'il peut monter en charge jusqu'à sa limite (celle de ses composants physiques). Plus précisément, on distingue deux types de montée à l'échelle : verticale et horizontale.

Définition 5.

Considérant un service, **sa capacité à monter à l'échelle verticalement** est la propriété qui décrit l'évolution apportée à sa capacité de traitement lorsqu'on augmente ses ressources (CPU, mémoire, etc.).

On dira donc, par exemple, qu'un service est capable de monter à l'échelle de manière verticale en terme d'usage de mémoire RAM s'il est capable d'augmenter ses performances lorsqu'on augmente sa mémoire RAM.

Définition 6.

Considérant un service, **sa capacité à monter en charge horizontalement** est la propriété qui décrit l'évolution apportée à sa capacité de traitement lorsqu'on augmente le nombre d'instances.

On dira donc qu'un service est capable de monter en charge (horizontalement) de manière linéaire [2] si une augmentation de $X\%$ de ses ressources augmente ses performances de $X\%$.

On remarque dès à présent que décrire l'augmentation du nombre d'instances en terme de pourcentage est sujet à discussion, nous le discutons à la section 4.2. Supposons, à ce stade, que chaque instance est identique et que le pourcentage se résume donc au rapport du nombre d'instances futur sur le précédent.

La plupart des solutions cloud mettent en avant leurs capacités à monter à l'échelle pour des raisons commerciales. La capacité à **descendre à l'échelle** est souvent négligée mais n'en est pas moins intéressante : pour des enjeux économiques et écologiques, il est très intéressant de pouvoir diminuer ses ressources lorsqu'elles sont sous-exploitées.

Le cloud doit être capable de s'adapter et ceci ne peut se résumer à la capacité à monter à l'échelle. Il faut aussi considérer sa capacité à descendre à l'échelle. L'union de ses deux propriétés est sa capacité à **passer à l'échelle** que nous définissons comme suit :

Définition 7.

La capacité à passer à l'échelle d'un service (*(*Scalability en anglais), est sa capacité à pouvoir assumer la variation (descente ou montée) de la charge à laquelle il est soumis.

2. Dans la suite de ce mémoire, nous nous intéressons particulièrement aux adaptations horizontales, qui sont particulières au cloud. Nous omettrons le terme horizontal et ne préciserons que lorsqu'il s'agira des propriétés verticales.

Par “assumer”, on entend fournir des performances constantes. De manière assez intuitive, on peut comprendre qu'un facteur pouvant caractériser cette capacité à passer à l'échelle pourrait être le rapport entre le gain en performance et l'augmentation des ressources du service nécessaire à ce gain de performance.

L'élasticité

Dans cette section, nous proposerons notre définition de l'élasticité du cloud à titre d'introduction. Le chapitre 4 étudie de façon plus poussée l'élasticité.

Définition 8.

L'**élasticité d'un service** est sa capacité à passer à l'échelle de manière dynamique (c'est-à-dire sans nécessiter sa réinitialisation et sans entraîner des effets collatéraux trop importants).

Par “collatéraux”, on entent des pertes en performance inacceptables. L'élasticité d'un service est une caractérisation de sa capacité à passer à l'échelle ; elle requiert son dynamisme. Le terme “dynamisme” signifie que le système sous-jacent au service doit être assez agile que pour garantir la continuité et la stabilité de ce dernier. Listons les conséquence de ces exigences :

- Le service doit être *continu* ; son passage à l'échelle ne doit pas nécessiter sa réinitialisation.
- Le service doit être *continu* et *stable* ; les modifications internes nécessaires à son passage à l'échelle ne doivent pas entrainer son inaccessibilité (ou des temps de réaction trop lents).

Les paramètres nécessaires à caractériser tout passage à l'échelle ne suffisent plus. Il s'agit aussi de pouvoir caractériser ses “effets collatéraux” ainsi que leurs impacts. Nous discutons de ce sujet en section 4.3. Un service ne peut entrer dans le mouvement cloud qu'à la condition qu'il soit élastique : si son passage à l'échelle n'est pas dynamique, ce dernier ne pourra pas être hautement disponible.

2.4 Le théorème CAP

La spécificité du service base de données est qu'il stocke des données dont on doit pouvoir garantir une certaine intégrité. Cette contrainte est assez bloquante lorsqu'on pense les bases de données comme un service cloud c'est-à-dire capable de monter à l'échelle. Ce problème peut être résumé par une conjecture connue sous le nom de *théorème CAP*[3] ou sous le nom de *théorème de Brewer*[4].

Cette conjecture est énoncée comme suit :

3. Dans la suite de ce mémoire, nous parlerons de “CAP” pour signifier “le théorème CAP”.
4. E. Brewer énonce cette conjecture en 2009. Remarquons que S. Gilbert et N. Lynch démontrent en 2002 cette conjecture pour un modèle asynchrone de systèmes distribués [26].

Conjecture 1.

Parmi les trois propriétés suivantes : consistance, haute disponibilité et tolérance à la partition, tout système de données distribué ne peut respecter, au plus, que deux d'entre elles.

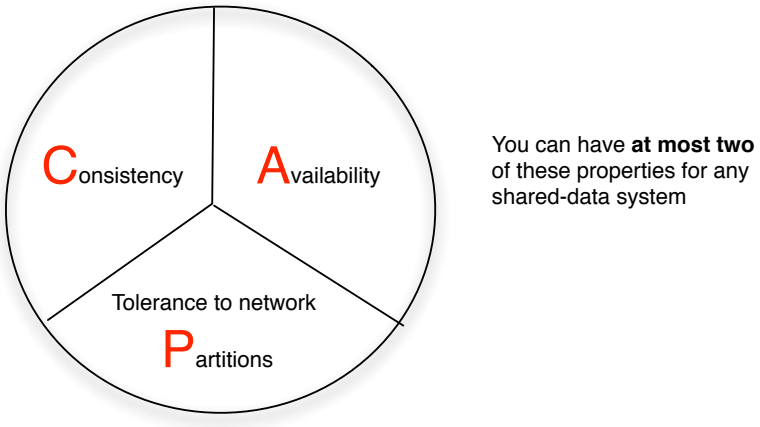

FIGURE 2.3 – CAP (reproduction de [18])

CAP (représenté par la figure 2.3) introduit deux propriétés relatives aux systèmes distribués qui nous sont inconnues. Nous allons maintenant les décrire.

La consistance des données[5] est un concept qui décrit les modes d'accès aux données, leur intégrité et validée. Nous le verrons à la section 3.1.4, il existe plusieurs modèles de consistance des données. La consistance des données, telle que CAP l'entend, décrit le fait que les données auxquelles on accèdera seront toujours à jour.

Pour des raisons de performance (passage à l'échelle) et de sécurité (réplication des données), les systèmes distribués tirent profit du fait de pouvoir s'étirer sur plusieurs instances. Répartir un service sur de nombreuses instances augmente le risque de défaillance d'une partie du service et causer ainsi une partition du service en plusieurs sous-ensembles. Pour expliquer le phénomène de partition, imaginons notre service distribué sur deux boites noires ; un phénomène de partition arrive lorsque le câble reliant ces deux boites noires est débranché ; les deux sous-systèmes ne sont plus capables de communiquer. Un service distribué tolérant à la partition est un système tel qu'un phénomène de partition n'altère pas son bon fonctionnement. Ou de manière plus pragmatique :

5. Il faudrait parler de la cohérence des données (*data consistency* en anglais) mais par abus de langage, nous parlons de la consistance des données

Définition 9.

Dans un service tolérant à la partition, aucun ensemble de défaillance autre que la défaillance de la totalité du réseau ne peut causer des réponses incorrectes du service.

traduit de [26]

Discussion

CAP est un problème inhérent au cloud puisqu'on veut un service base de données hautement disponible (en passant notamment par la bonne gestion du phénomène de partition) capable de stocker de manière cohérente nos données. CAP retient dès lors l'attention d'un bon nombre d'acteurs. A titre d'exemple, citons [42] et [19] qui l'illustrent par des exemples et discutent les défis et enjeux de cette conjecture. Afin de mieux appréhender le phénomène introduit par CAP, nous tenons à résumer l'approche de D. Abadi [16] qui tend à critiquer CAP et en donner une approche plus pragmatique et une compréhension, à nos yeux, plus pertinente.

Soulignons deux problèmes à CAP vus par Abadi [16] :

- Suivant CAP, nous aurons soit des systèmes dits CA (consistants et hautement disponibles), CP (consistants et tolérants à la partition) ou AP (hautement disponibles et tolérants à la partition). Il existe une asymétrie entre la valeur de la disponibilité et de la consistance : un système CP acceptera de sacrifier sa disponibilité mais seulement en cas de partition, tandis qu'un système AP décidera de sacrifier sa consistance tout le temps.
- Le second problème souligné est que, en pratique, la différence entre les systèmes CA et CP est difficile à discerner. En effet, que veut dire "non tolérant à la partition" ? Dans la pratique, toujours selon Adabi, cela signifie que, en cas de partition, le système perdra en disponibilité. Il n'existe donc en pratique que deux types de systèmes : CP/CA et AP.

Il s'agit donc de remplacer CAP par PACELC :

> "*if there is a partition (P) how does the system tradeoff between availability and consistency (A and C); else (E) when the system is running as normal in the absence of partitions, how does the system tradeoff between latency (L) and consistency (C) ?*"
>
> [16]

Dans la pratique, les systèmes AP – qui ont tendance à relâcher la consistance en cas de partition – tireront aussi profit de ce relâchement pour une améliorer leurs performances lorsqu'il n'y a pas de partition du système. Ces systèmes sont donc, dans la théorie d'Abadi des systèmes PA/EL.

Nous tenterons dans la suite de ce mémoire, de classifier les différents SGBDs non seulement en fonction de CAP mais aussi en fonction de PACELC.

Chapitre 3

Etude comparative de différents systèmes de gestion des bases de données

Dans ce chapitre, nous proposerons une analyse des différentes dimensions caractérisant un SGBD sur cloud. Nous ferons un tour d'horizon du courant *SQL* et de ses systèmes distribués (section 3.2). Nous introduirons également le nouveau courant dit *NoSQL* et ses concepts (section 3.3).

Nous analyserons ensuite, de manière plus approfondie, trois différentes solutions *NoSQL* : *MongoDB* [10], *Cassandra* [5] et *Hbase* [6] et étudierons de manière plus abstraite *Voldemort* [12].

Nous résumerons finalement ces analyses et comparerons ces différents systèmes grâce aux dimensions préalablement établies.

3.1 Classification des systèmes

Examinons tout d'abord les différentes décisions architecturales caractérisant un SGBD, à savoir :

- Le modèle de données
- Le choix CAP
- Le choix PACELC
- La réplication synchrone ou asynchrone
- Le modèle de consistance
- Le modèle de requêtes

Remarquons que les quelques autres études comparatives de SGDBs ([20], [17], [40]) nous servant de référence, s'attardent sur leurs licences, leur langages de programmations, etc. Dans le cadre de notre analyse, nous ne pensons pas que ces dimensions apporteraient une valeur ajoutée.

3.1.1 Le modèle de données

Le modèle de données caractérise l'architecture, le schéma logique respecté pour décrire les données stockées ; allant d'un modèle simpliste *clé-valeur* jusqu'au modèle complexe re-

lationnel. Cette dimension aura des répercussions sur les performances et la réputation des SGBDs.

De manière intuitive, on peut comprendre que le modèle le plus basique atteindra de meilleures performances vu qu'il est plus proche des mécanismes de la machine physique. Tandis, qu'un modèle de plus haut niveau, permettant une modélisation plus abstraite, éloignée des composants physiques et plus proche de l'application en verra ses performances réduites mais une utilisation bien plus aisée.

3.1.2 Les choix CAP et PACELC

Comme décrit en section 2.4, chaque système s'inscrit parmi les trois systèmes CA, CP et AP décrit par CAP ou peut se catégoriser via la proposition PACELC d'Abadi.

FIGURE 3.1 – Classification des systèmes grâce à CAP [22]

La catégorisation via CAP est fort utile ; elle permet en seulement deux lettres de comprendre la politique voulue par un système (voir figure 3.1). Toutefois, nous estimons que PACELC semble bien plus descriptif et dès lors mérite d'être repris lors de notre analyse comparative. Remarquons que la classification des systèmes via PACELC ne va pas du tout à l'encontre de CAP mais permet une meilleure compréhension.

3.1.3 La réplication synchrone ou asynchrone

La réplication de données est le processus consistant à maintenir R copies d'un ensemble de données sur d'autres instances de stockage. Ce nombre de copies (R) est appelé *facteur de réplication*. La réplication peut servir à rendre le système plus disponible (en redirigeant le trafic d'une instance défaillante vers sa réplique), à éviter la perte de données et à améliorer les performances (en divisant et redirigeant la charge d'une instance vers ses répliques).

Plusieurs approches sont possibles pour la réplication : la réplication synchrone et la réplication asynchrone.

- La réplication synchrone s'assure que toutes les copies soient à jour, ce qui peut potentiellement entrainer des temps de réponses plus lents lors des mises-à-jour. Remarquons que le mode de réplication synchrone peut être bloquant pour la disponibilité : si une réplique connaît une défaillance, le système pourrait rester bloqué.
- Dans une réplication asynchrone, une instance mère stocke les données actualisées et transmet les données modifiées aux répliques de manière différée. Il existe donc très certainement un gain à l'écriture des données (on doit s'assurer que la modification n'ait eu lieu que sur une seule copie) mais celui-ci implique le risque de perdre des données si l'instance mère connait une défaillance avant d'avoir pu transmettre l'information aux répliques. Ce mode de réplication peut aussi impliquer la nécessité de mécanismes d'assemblage de versions pour assembler différentes versions d'une donnée en une seule valeur mise-à-jour.

Comme nous l'avons déjà mentionné, le mode de réplication a une forte influence sur le comportement du système. Il est donc nécessaire d'entendre le mode de réplication comme une dimension de notre étude comparative.

3.1.4 Le modèle de consistance

En réponse à la montée en charge exceptionnelle à laquelle de nombreux acteurs ont dû faire face, ceux-ci ont créé des systèmes relâchant la consistance transactionnelle pour passer à un autre modèle de consistance. Ces nouveaux modèles ont une influence théorique très importante sur les performances, la disponibilité et la tolérance à la partition (cf. CAP). Mais ce niveau de relâchement ne peut être accepté par toute application (on imagine mal une application bancaire accepter de perdre des transactions faites par ses clients).

Les modèles de consistance proposés par les SGBDs sont donc une caractéristique très significative et importante. Nous pouvons distinguer, en fonction de leur degré de consistance, quatre sous-ensembles de modèles (du plus faible au plus fort) :

- Finalement consistant : garantit que si aucune modification n'est faite sur la donnée, les accès en lecture sur la donnée retournent finalement sa valeur actualisée [42]. Ceci peut notamment se faire par des systèmes de réparation à la lecture (nous verrons dans ce chapitre les techniques mises en place par les différentes SGBDs).
- Consistance forte : s'assure que chaque lecture d'une valeur retourne la valeur actuelle de la valeur (c'est-à-dire la valeur de la dernière écriture réussie et finie).
- Le verrouillage d'entrée signifie que l'on va pouvoir verrouiller une entrée et donc garantir son intégrité.
- Transactionnel : S'assure qu'une transaction respecte les propriétés ACID (voir section 3.2.2)

3.1.5 Le modèle de requêtes

Le modèle de requêtes comprend le langage de communication et les requêtes qu'il est possible de faire sur la BD. Celui des SGBDs relationnels (SGBDRs) traditionnels était riche et plus ou moins similaire pour chaque système. Ceux des nouveaux systèmes sont, en règle générale, bien plus pauvres et diffèrent fortement d'un système à l'autre.

Or la richesse du modèle de requêtes sera un argument prépondérant lors du choix du SGBD. De plus, elle peut aussi avoir son impact sur les performances de celui-ci. Par exemple, un modèle permettant l'utilisation de fonctions Map-Reduce (MR) permettra, en répartissant une charge en sous-charges, de traiter plus aisément des requêtes complexes.

3.2 Les bases de données relationnelles distribuées

Des "bases de données relationnelles extensibles" tel est le terme utilisé par Catell pour décrire les SGBDRs adaptés aux clusters. Les SGBDRs traditionnels ont été développés durant les dernières décennies et sont généralement la meilleure solution sur le marché pour les services ne devant pas passer à l'échelle : ils sont pensés pour tenir sur une seule instance. Dans cette section, nous décrirons leurs adaptations au cluster ainsi que les mécanismes mis en place pour permettre de les distribuer sur plusieurs instances.

3.2.1 Le modèle de données relationnel

Un schéma relationnel est essentiellement un groupe de tables représentant des objets et des relations entre ceux-ci. Ces tables, faites de lignes et colonnes, peuvent être interrogées à l'aide d'un langage d'interrogation structuré (SQL).

Dans un schéma relationnel, les tables sont normalisées dans le but d'éviter les doublons et de consolider les données. Chaque entité ou relation possède une représentation minimaliste qui peut s'étendre grâce à des jointures des tables.

3.2.2 Les propriétés ACID

Les SGBDRs traditionnels reposent sur la notion de transaction et la propriété ACID. La notion de *transaction* dans les BD décrit une interaction qui fait passer une BD d'un état A à un état B tout en satisfaisant les propriétés d'atomicité, de consistance, d'isolation et de durabilité (ACID) :

- *Atomicité* : l'ensemble des opérations est une suite indissociable c'est-à-dire qu'en cas d'échec d'une opération, tout l'ensemble sera annulé (y compris les opérations précédentes).
- *Consistance* : le résultat de l'ensemble des opérations doit être cohérent. Les règles de cohérence varient en fonction de la sémantique des données.
- *Isolation* : lorsque deux transactions ont lieu en même temps, les modifications effectuées par la première ne sont visibles par la seconde que lorsque la première est terminée et validée.
- *Durabilité* : une transaction terminée ne peut être annulée ou recouverte.

3.2.3 MySQL Cluster

Comme premier élément de tour d'horizon, nous nous attardons sur MySQL Cluster [8] car c'est la solution SQL la plus mature. Il s'agit une version de MySQL sortie en 2004 qui remplace le moteur InnoDB par une nouvelle couche distribuée NDB [20]. MySQL Cluster respecte le modèle "ne partageant rien" (*shared nothing* en anglais) et distribue les données sur des nœuds multiples.

Choix CAP et PACELC Les systèmes respectant la propriété ACID sont des systèmes CA (point de vue CAP), c'est-à-dire que pour préserver l'intégrité de leurs données, ils sont prêts à sacrifier la tolérance à la partition. MySQL Cluster a choisi de préserver la consistance au détriment des performances et de la tolérance à la partition. Ces systèmes sont donc PC/LC. En cas de partition, vu son modèle de réplication synchrone, MySQL cluster empêchera les opérations de mises-à-jour.

Réplication Pour éviter de propager la faille d'une instance, les données sont répliquées de façon synchrone[8] c'est-à-dire que lors de l'écriture (insertion ou mise-à-jour) d'une entrée, l'information est propagée sur toutes les répliques et l'opération n'est validée que lorsque toutes les répliques l'ont validée.

Dans le cas où toutes les répliques d'une donnée sont à jour, si une instance subit une défaillance, on peut alors rediriger son trafic sur une réplique et éviter ainsi que cette défaillance soit fatale pour tout le système.

Modèle de consistance MySQL Cluster, vu son respect des propriétés ACID, opte pour une consistance transactionnelle. En pratique, MySQL Cluster ne propose pas de consistance transactionnelle complète mais une consistance transactionnelle *Read Committed* [8]. Lors d'une lecture, la requête voit les données telles qu'elles sont avant que la requêtes ait été effectuée : les modifications validées par les transactions concurrentes sont retournées par la lecture.

Le modèle de requêtes MySQL Cluster garde toute la richesse des versions non distribuées de MySQL. On peut effectuer toutes les requêtes SQL comme si la BD n'était pas distribuée et tous les mécanismes internes sont transparents pour l'utilisateur.

3.2.4 Autres solutions SQL

Dans cette section, nous passerons en revue quelques initiatives SQL entreprises. Nous ne nous attarderons pas sur celles-ci car, même si leurs concepts sont très intéressants, ces solutions ne sont pas encore matures. Il est donc fort difficile de juger de leur efficacité.

De manière théorique, nous suivons l'avis de Catell pour dire que les SGBDRs devraient être capables de passer à l'échelle tant que les applications évitent les opérations traversant plusieurs nœuds et des transactions de grande ampleur [20]. On remarque que cette concession n'affecte pas leur capacité à concurrencer les solutions NoSQL (étudiées en section 3.3) qui traitent aussi difficilement (voire ne sont pas du tout capables de traiter) de telles opérations.

ScaleDB [11] se veut la version MySQL de Oracle RAC. Tout comme MySQL Cluster, il se place sur une version de MySQL en remplaçant *InnoDB*. Son principe est de se reposer sur une architecture à mémoire partagée (*shared memory* en anglais) : tout nœud a accès à tout disque. La distribution des données est automatique au travers des serveurs. ScaleDB n'utilise pas de réplication mais des sauvegardes sont faites sur disque (pour récupérer les données en cas de défaillances).

3.2.5 Les solutions SQL-MR

Map-Reduce

MapReduce est un modèle de programmation (initié par Google) pour manipuler et gérer un large ensemble de données. Le principe est relativement simple ; l'utilisateur définit un schéma de redirection et une fonction de réduction [28]. Un nœud peut alors, en suivant le schéma de redirection (*Map*), diviser une charge en plusieurs sous-charges qu'il soumettra à ses nœuds fils (l'opération peut être récursive). Les nœuds sous-traitants remontent leurs résultats au nœud parent qui combine ces résultats via la fonction de réduction.

SQL-MR

SQL-MR unit un système SQL épuré au concept de MapReduce. Les fonctions de réduction se chargent de toutes les tâches non relationnelles et optimisations liées au domaine [24].

Ces fonctions (qui peuvent être écrites dans de nombreux langages de programmation) sont, par défaut, parallèles. Les requêtes sont donc prises en charge par ces fonctions et distribuées en parallèle sur les nœuds du cluster, ce qui assure au système de supporter la montée en charge. Les fonctions reçoivent des relations (tables) en entrée et retournent des relations. Elles sont adaptatives c'est-à-dire qu'en fonction du contexte, elles déterminent quelles relations retourner.

Les systèmes SQL-MR

Les systèmes SQL-MR les plus reconnus sont *AsterData*, *VoltDB* et *Greenplum*.

VoltDB [13] est une nouvelle solution open-source qui est pensée pour être performante sur chaque nœud et capable de passer à l'échelle (notamment grâce au fait qu'il est orienté colonnes). Les tables sont partitionnées et distribuées sur les différents nœuds de manière automatique (mais l'utilisateur peut en définir les paramètres). Les données peuvent être répliquées (réplication synchrone) et l'application peut distribuer ses requêtes de lecture sur les répliques pour améliorer ses performances.

VoltDB propose des solutions intéressantes (même si elles ne sont pas encore mises en place) : les indexes et la structure des données sont pensées pour s'adapter à une BD qui tiendrait entièrement en mémoire. Cette solution optimise nettement les écritures. VoltDB serait 100 fois plus rapide que MySQL et 13 fois plus rapide que *Cassandra* [34].

3.3 Le mouvement NoSQL

Les systèmes *NoSQL* (Not Only SQL) proposent de surprenantes architectures quittant le chemin relationnel. Le message transmis est que, si on veut une BD capable de passer à l'échelle, on ne peut plus utiliser un SGBDR. Nous examinons dans cette partie les concepts variés qu'ils mettent en place.

Dans cette section, nous donnerons un aperçu des différents modèles de données du mouvement NoSQL (section 3.3.1) et du paradigme *BASE* (section 3.3.2) qu'il propose en réponse aux propriétés ACID des solutions SQL.

Les sections 3.4, 3.5 et 3.6 décrivent de manière approfondie les moyens mis en place par *MongoDB*, *Cassandra* et *HBase*. La section 3.7 donne un aperçu de *Voldemort*.

3.3.1 Les modèles de données

Les différents systèmes NoSQL utilisent des terminologies fort différentes. Cattell [20] différencie comme suit les modèles de structure qu'ils adoptent :

- Un *n-uplet* est un ensemble de paires clé-valeur telles que les noms d'attributs sont définis dans le schéma et les valeurs sont basiques. On parle souvent de modèle *clé-valeur*
- un *document* est un ensemble de paires clé-valeur telles que les valeurs peuvent être composées (d'autres paires clé-valeur) et que les noms d'attributs sont définis dynamiquement pour chaque document.
- Une *entrée extensible* est un hybride entre un n-uplet et un document où des familles d'attributs sont définies dans le schéma tout en permettant de définir de nouveaux attributs à la volée. On parle souvent de modèles orientés *colonnes* voire orientés *familles de colonnes*.
- Un *graphe* qui représente l'information sous forme de graphe. Selon la mise en oeuvre, le graphe peut représenter le schéma et l'architecture de la BD ou peut servir à encoder toute la DB.

Le modèle clé-valeur

Le modèle clé-valeur (figure 3.2) est d'une grande simplicité : il peut être vu comme une large table de hachage persistante. Il est, par conséquent, adapté aux caches et offre, dès lors, de hautes performances en terme d'accès aux informations.

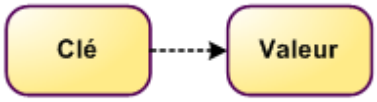

FIGURE 3.2 – BD clé-valeur [30]

Cette modélisation la plus simpliste est justifiée par le constat qu'un bon nombre d'accès aux bases de données se résume à de simples lectures ou écritures à partir d'un index.

Le modèle orienté document

La représentation orientée documents est une extension du modèle clé-valeur.A l'instar de celui-ci, elle associe à chaque clé un document qui contient des données organisées de manière hiérarchique à l'image d'un document XML (figure 3.3).

Le modèle d'entrées extensibles

Aussi connu sur le nom de représentation orientée colonnes (figure 3.4), ce modèle est une deuxième évolution, multidimensionnelle, du stockage clé-valeur. Les données y sont représentées comme des groupes de colonnes.

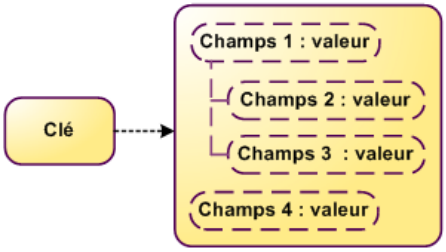

FIGURE 3.3 – BD orientées documents [30]

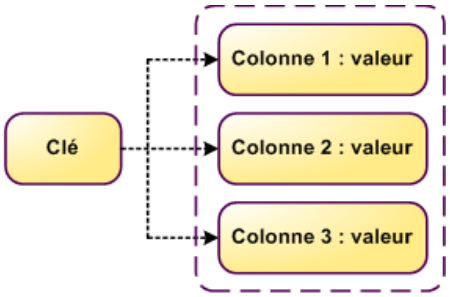

FIGURE 3.4 – BD orientées colonne [30]

Le modèle en graphe

Ce modèle (figure 3.5) permet une modélisation plus aisée d'objets et de leurs interactions. Un exemple typique serait les interactions entre acteurs, réalisateurs, producteurs et films.

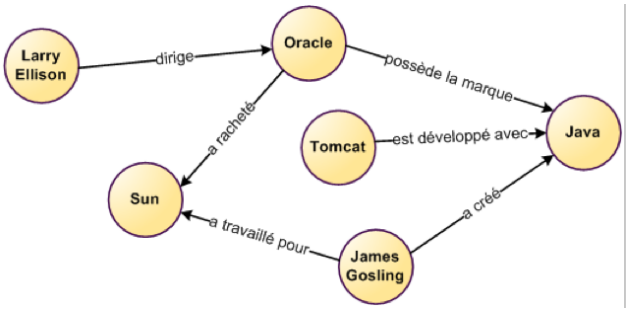

FIGURE 3.5 – BD orientées graphe [30]

3.3.2 BASE en réponse à ACID

Si les solutions SQL avaient tendance à être des systèmes CA, le NoSQL lui s'oriente plutôt vers un modèle AP (voire CP). Le courant veut redéfinir de nouvelles formes de consistance en fournissant, en réponse à la propriété ACID, le paradigme BASE : essentiellement disponible (*Basically Available* en anglais), à état variable dans le temps (*Soft State* en anglais) et fina-

lement consistant (*Eventually Consistant* en anglais). Le principe est d'oublier le 'C' et le 'I' d'ACID pour une dégradation avantageuse de consistance et pour une amélioration des performances. Pour plus de détails, nous dirigeons le lecteur vers la présentation de Brewer [18]. Pour le lecteur surpris de cet affaiblissement de la consistance, nous proposons la lecture de l'article de Volgels [42] qui explique pourquoi ces concessions et comment elles peuvent être vues du point de vue client et serveur.

3.4 MongoDB

MongoDB est un SGBD NoSQL de type document, open source, proposé par la société 10gen. Sa popularité est grandissante, il est notament utilisé par *FourSquare* [35], *SourceForge* ou *GitHub* qui, à eux seuls, lui garantissent sa notoriété.

3.4.1 Modèle de données

Le modèle de données de *MongoDB* est orienté documents. Etudions ce modèle via une approche *bottom-up*.

Documents

Un document, l'unité basique de *MongoDB* , est un ensemble ordonné de paires clé-valeur. Il est identifié par son nom. Pour faire une analogie à SQL, un document peut être vu comme une entrée. A la différence d'une entrée SQL, le nombre de champs d'un document n'est pas défini préalablement. Un exemple d'un simple document est le suivant :

Exemple 1 (MongoDB - Document)**.**

{"nom" : "Degroodt", "prenom" : "Nicolas", "age" : 25}

Les clés sont des strings alors que les valeurs sont typées : dans notre exemple, *Degroodt* et *Nicolas* sont des strings et 25 est un entier.

Collections

Une collection est un ensemble de documents (et peut donc être vu comme une table SQL). Les documents d'une collection peuvent avoir des formes très différentes comme l'illustre l'exemple suivant

Exemple 2 (MongoDB - Collection)**.**

{"nom" : "Degroodt", "prenom" : "Nicolas", "age" : 25}
{"marque" : "apple", "nom" : "MacBook Pro"}

MongoDB ne pose aucune restriction quant aux documents contenus dans une même collection. Pourtant, pour des raisons pratiques, il est intéressant de distinguer des documents différents en les plaçant dans différentes collections (et donc ne pas procéder comme nous l'avons fait dans l'exemple précédent).

A la différence d'une table SQL, le nombre de champs des documents d'une même collection peut varier d'un document à l'autre.

Bases de données

Une base de données est un conteneur pour les collections (tout comme une base de données SQL contient des tables) avec ses propres permissions.

3.4.2 Choix CAP et PACELC

MongoDB a fait le choix CP favorisant la consistance à toute autre considération. *MongoDB* prévoit des mécanismes fort intéressants pour pallier à un bon nombre de défaillances (c'est-à-dire de manière à préserver consistance, disponibilité et tolérance à la partition). Néanmoins, lorsque ces mécanismes ne suffisent plus, le choix est fait de ne pas garantir la disponibilité et de rester tolérant à la partition.

La politique de *MongoDB* est d'être consistant avant tout. Il s'inscrit donc dans un choix PC/EC. Néanmoins, si l'utilisateur désire augmenter les performances du système, il serait possible de libérer la consistance pour une amélioration des temps de réponse.

3.4.3 Vue d'ensemble de l'architecture

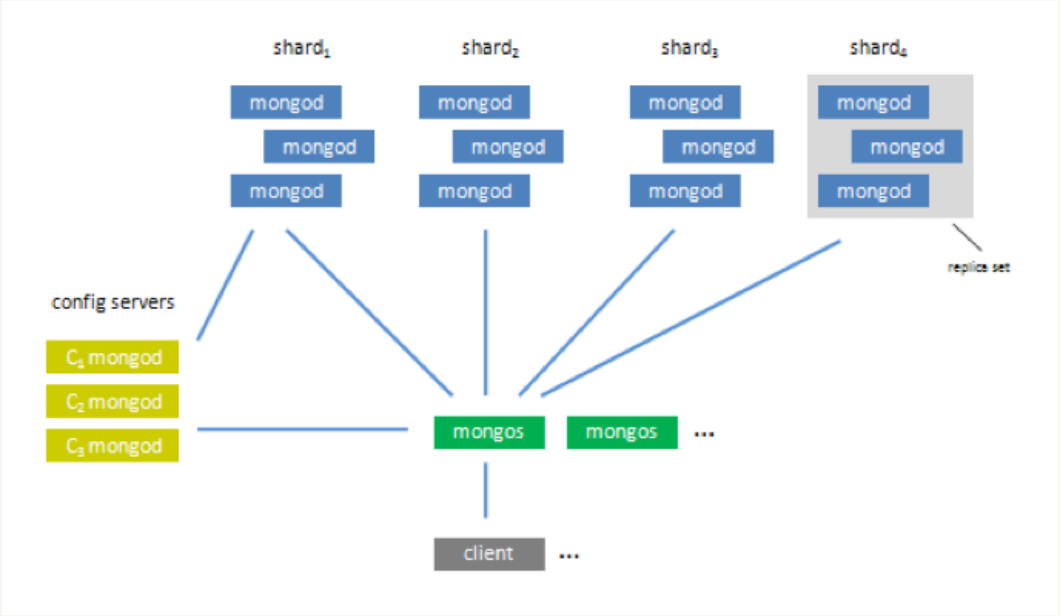

FIGURE 3.6 – L'architecture de *MongoDB* [38]

Comme illustré par l'image 3.6, *MongoDB* se décompose en 3 services différents mais dépendants les uns des autres.

Un shard se compose d'un ou plusieurs serveurs. Via son processus *mongod*, il est en charge de stocker les données.

Les serveurs de configurations stockent les méta-données du système c'est-à-dire les informations basiques relative à chaque *shard* et des indications quant aux données stockées par ceux-ci.

Ce service est indispensable, il est donc nécessaire de l'assurer en lui consacrant deux voire trois instances. L'écriture sur ces services utilise le protocole de validation en deux phases (*2-phase commit* en anglais) pour assurer un stockage fiable [38]. En cas de défaillance d'un service, tous les autres serveurs passent en mode de lecture unique (*read only* en anglais).

Le processus Mongos redirige les requêtes des applications clientes vers les *shards* adéquats et regroupe les résultats avant de les transmettre à l'application cliente. Il n'a pas d'état persistant : son premier état provient des serveurs de configuration. Toute modification dans les serveurs de configuration est propagée immédiatement aux processus mongos. Etant donné que ce service n'a pas d'état persistant, en cas de défaillance de ce service, seules les applications s'y connectant seront affectées, aucun autre service ne sera affecté. Pour répondre à une telle défaillance, il suffit de relancer le service.

MongoDB distribue les données en fonction de ses collections c'est-à-dire que chaque collection peut définir sa propre politique de partionnement de données. Un morceau (*chunk* en anglais) est un ensemble continu de documents d'une collection. Un morceau, défini via un triplet *(collection A, minKey, maxKey)*, contient les données de la *collection A* dont la clé est comprise entre les bornes *minkey* et *maxKey*.

Les serveurs de configuration ont connaissance de chaque morceau et de leur localisation (comme décrit en figure 3.1). La taille d'un morceau ne peut dépasser $200MB$. Lorsqu'il atteint cette limite, il est divisé en deux nouveaux morceaux. Lorsque la limite de stockage d'un *shard* est atteinte, des morceaux lui appartenant émigreront vers un autre *shard*.

Collection	mink	maxkey	location
users	nom : "Degroodt"	nom : "Skhiri"	$shard_1$
users	nom : "Skhiri"	nom : "Zimányi"	$shard_4$

TABLE 3.1 – *MongoDB* - Configuration Morceaux, *Shard*

Actuellement, l'algorithme de répartition des données est très simple et est appliqué de manière automatique ; on ne sait pas le configurer (cette faiblesse est connue et est en cours d'amélioration [21]). L'algorithme déplace les morceaux en fonction de la taille des différents *shards*. Le but est de maintenir les données distribuées de manière uniforme mais aussi de réduire au minimum les transferts de données. Pour qu'un déplacement ait lieu, il faut qu'un *shard* ait au minimum 9 morceaux de plus que le plus impopulaire des *shards*. A partir de ce point, les morceaux vont être répartis de manière uniforme sur les différents *shards*.

3.4.4 Réplication des données

Le premier modèle de réplication proposé par *MongoDB* est le modèle *maître-esclave*. Récemment, *MongoDB* propose une nouvelle approche, la réplication par pairs (*Replicat set* en anglais), qui voit tous les serveurs d'un *shard* comme des pairs qui éliront un maître temporaire. Dans les deux modèles, la réplication est asynchrone.

Réplication maître-esclave

Ce modèle (voir figure 3.7) est le plus répandu, il peut être utilisé comme outil de sauvegarde, de basculement (*failover* en anglais) et peut servir à la montée en charge horizontale (pour des opérations de lecture).

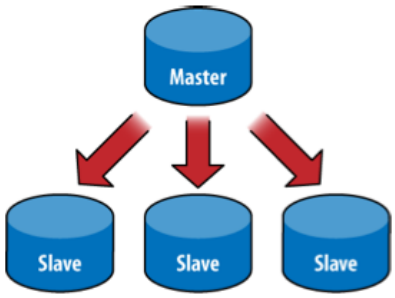

FIGURE 3.7 – Réplication maître-exclave[25]

Les données sont, dans un premier temps, écrites sur le serveur *maître*, l'information est ensuite transmise de manière asynchrone aux serveurs esclaves.

Si un serveur esclave subit une défaillance, il suffit de relancer l'instance et il synchronisera ses données avec le serveur maître via un fichier log (pour ne pas surcharger le réseau avec une migration de données depuis le début). Si un serveur maître subit une défaillance, le système passe en mode lecture unique jusqu'à ce que le serveur maître soit relancé. On voit donc ici que ce système de réplication permet un service de basculement en cas de défaillance. Ces serveurs esclaves possèdent donc une version plus ou moins actualisée des données et peuvent donc être utilisés en lecture par l'application si celle-ci concède un relâchement de la consistance. Les esclaves allègent alors la charge subie par le serveur maître (le système devient alors PA/EL). Une telle utilisation favoriserait fortement les applications à lectures fréquentes.

Réplication par pairs

La réplication par pairs (*Replicat Set* en anglais) est une version améliorée du modèle maître-esclave. Elle offre un balancement automatique.

Comme l'indique son nom, chaque serveur est identique et a un rôle : *primaire* ou *secondaire*. Il n'y a qu'un serveur *primaire* par ensemble de répliques. En cas de défaillance de celui-ci, les membres de l'ensemble procèdent à l'élection d'un nouveau serveur primaire (en fonction de celui qui détient les informations les plus récentes) [1]. Les figures 3.8, 3.9 et 3.10 nous montrent un scénario d'élection.

Ce modèle de réplication donne à *MongoDB* la capacité de s'auto-gérer ; il ne nécessite plus d'intervention humaine pour continuer son bon fonctionnement en cas de défaillance d'une instance de stockage.

1. D'autres arguments peuvent être définis manuellement par l'utilisateur. Ces arguments donnent des valeurs à chaque nœud qui seront prises en compte lors de l'élection du nouveau primaire.

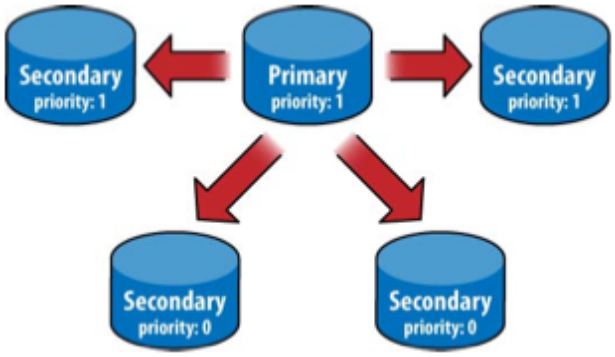

FIGURE 3.8 – Un ensemble de réplicats[25]

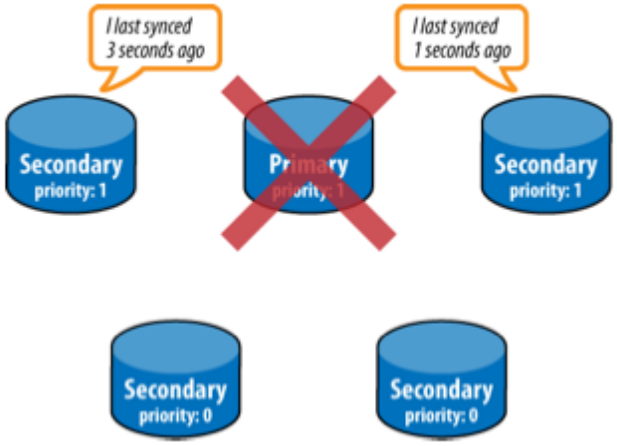

FIGURE 3.9 – Défaillance du serveur primaire, procédure d'élection[25]

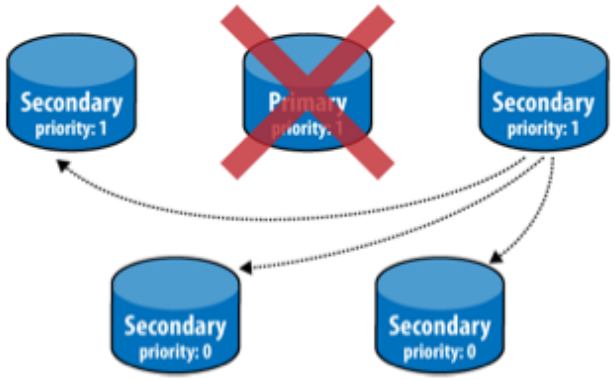

FIGURE 3.10 – Défaillance du serveur primaire, un nouveau serveur est élu[25]

3.4.5 Modèle de consistance

Par défaut, les requêtes sont donc consistantes mais il est possible de modéliser d'autres degrés de consistance [39], notamment via l'utilisation des instances de stockages secondaires (ou esclaves selon le mode de réplication) (voir section 3.4.4).

3.4.6 Modèle de requête

Grâce à son modèle de données en documents, *MongoDB* propose une interface de programmation assez riche (sans doute parmi les plus riches du mouvement NoSQL). A titre d'exemple, nous proposons deux listes non-exhaustives de ses fonctions.

La première liste illustre les différents opérateurs pour filtrer une sélection :

- les opérateurs $<, \leq, >, \geq$
- l'opération *exists* : utilisée pour filtrer les entrées telles qu'il existe ou non un certain attribut
- l'opérateur *in* : qui vérifie qu'une valeur doit être présente dans l'entrée retournée
- des expressions régulières qui peuvent être appliquées aux champs retournés

La seconde liste reprend quelques méthodes proposées par *MongoDB* :

- *COUNT* qui retourne le nombre de documents correspondant aux critères de recherche
- *LIMIT* qui limite le nombre de documents retournés
- *SORT* qui trie les documents retournés
- *SKIP* qui permet de ne pas retourner les n premiers documents

De plus, Mongo propose une mise en oeuvre de MR. Les fonctions doivent être écrites en JavaScript [9].

3.5 Cassandra

Cassandra est un SGBD NoSQL orienté colonnes développé à l'origine par la société Facebook et repris par la suite sous licence Apache. Le but du projet est de faire tourner un système sur des milliers de serveurs [27]. A cette taille, les défaillances (grandes et petites) sont courantes. *Cassandra* propose une nouvelle façon de gérer l'état persistant de ses instances pour pallier à ces problèmes. Cassandra a été développé dans l'optique de diminuer les coûts hardware et de gérer des données avec des fréquentes écritures.

3.5.1 Modèle de données

Cassandra propose un modèle de données orienté colonnes. Nous décrivons ce modèle de données via une approche top-down.

Keyspace

Un *Keyspace* est le point de départ de stockage, l'équivalent d'une base de données dans les SGBDR. Il est conseillé d'avoir un et un seul *Keyspace* par application [33].

Famille de colonnes

Une famille de colonnes est un conteneur pour un ensemble ordonné de colonnes (super ou simples colonnes). Les familles de colonnes étaient statiques et définies manuellement dans

le fichier de configuration dans les versions antérieures à 0.7. Depuis *Cassandra* 0.7, on peut ajouter/modifier/supprimer une famille de colonnes à la volée.

Une famille de colonnes a deux attributs : son nom et son comparateur qui définit le critère permettant d'ordonner ses colonnes (par exemple un critère basé sur l'encodage UTF8).

Les différentes familles de colonnes sont stockées sur différents fichiers. Ce fait est important lors de la modélisation du schéma de données. Pour former une famille de colonnes, on ne pensera plus qu'à la sémantique des données mais aussi à la manière et à la fréquence dont on y a accès. Si deux colonnes ont tendance à être lues ou modifiées en même temps, les mettre dans un même fichier permettra de limiter cette opération en un seul accès fichier.

Colonnes

Une (simple) colonne est l'unité la plus basique du modèle. Une colonne est un triplet : la clé (le nom), la valeur et l'heure (un timestamp) (figure 3.11) de la dernière modification. Ces colonnes ne sont pas définies au départ et peuvent être ajoutées/retirées à la volée, ce qui permet une certaine flexibilité.

FIGURE 3.11 – Cassandra - La structure d'une colonne [33]

Une super colonne (figure 3.12) est une colonne spéciale ; sa valeur est un ensemble de colonnes simples. L'idée est donc de donner un degré de hiérarchie en plus au modèle. (Ce concept de super-colonne est un des principaux aspects qui différencie le modèle de données de *Cassandra* à celui de Dynamo).

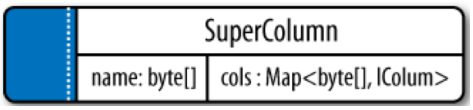

FIGURE 3.12 – Cassandra - La structure d'une super colonne [33]

Remarquons que lorsqu'une famille de colonnes contient des super-colonnes, cette famille devient une *famille de super-colonnes* et ne pourra plus stocker que des super-colonnes.

Résumé

Ce modèle peut donc être vu comme une architecture à 5 niveaux (figure 3.13) : [keyspace][famille de colonne][clé][super colonne][colonne].

3.5.2 Le choix CAP et PACELC

Cassandra a très clairement été développé tel un système AP. Mais notons que ce système donne l'option de pouvoir établir un niveau de consistance différent pour chaque requête : (notamment en utilisant la technique de *quorum*[2]). Toutefois, on remarque que, s'il est possible

2. Cette technique sera décrite à la section 3.5.5

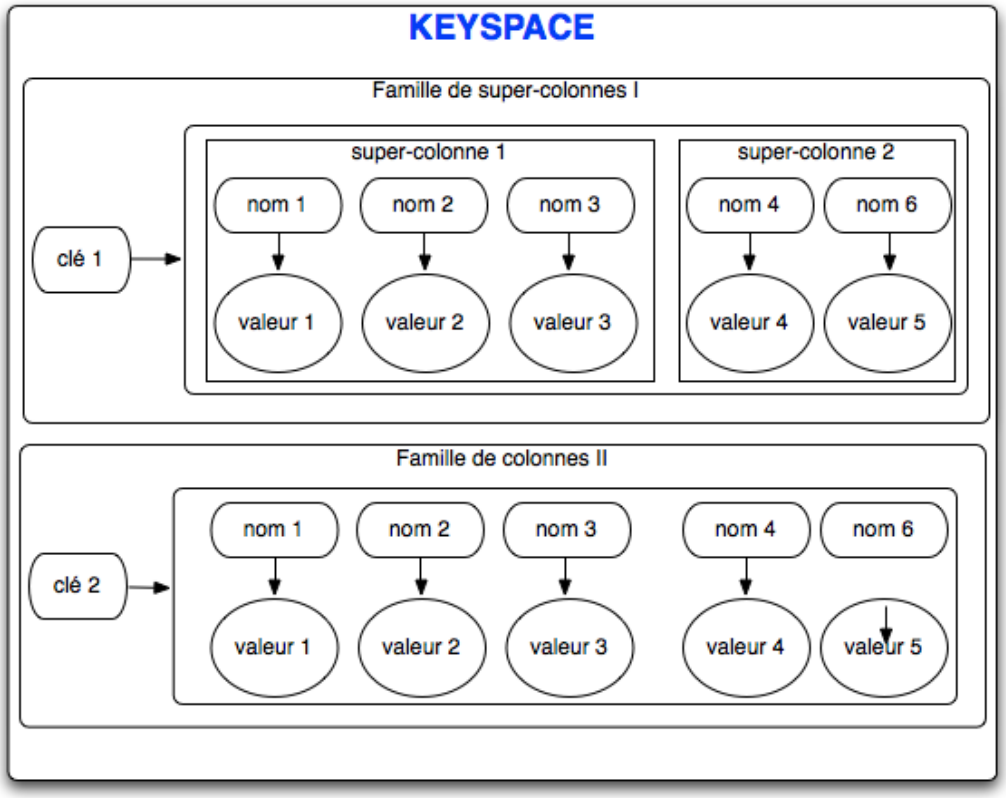

FIGURE 3.13 – Cassandra - Un modèle de données à 5 niveaux

grâce à cette méthode d'avoir la dernière version d'une colonne, avoir la dernière version de toute une entrée reste impossible. Ce choix est fait par *Cassandra* pour garantir la disponibilité et la tolérance à la partition.

Cassandra a été pensé pour favoriser la disponibilité et la tolérance à la partition (système PA/EL). Grâce à son modèle de consistance, à ses lectures réparatrices, *Cassandra* peut laisser le système disponible en cas de partition (les données seront réparées plus tard) ; sans partition, le système utilisera ce modèle pour améliorer les performances.

3.5.3 Vue d'ensemble de l'architecture

Cassandra propose une architecture décentralisée basée sur les solutions Dynamo et Big-Table. Inspiration de Dynamo, chaque nœud de *Cassandra* est égal. Leur gestion des nœuds se base sur le protocole peer-to-peer *Gossip* [33]. Cette décentralisation, par opposition à l'architecture *maître-esclave*, permet d'éviter les points de défaillances isolés et les goulots d'étranglement.

Le *Gossiper* est en charge de s'assurer que chaque information importante concernant l'état des nœuds du cluster est transmise à tous les nœuds du cluster (et aussi aux futurs nœuds et ceux ayant subi une défaillance). Le service *Gossip* est aussi en charge de répartir les clés sur les différents nœuds.

Une des principales fonctionnalités voulues pour *Cassandra* est la capacité de passer à

l'échelle de manière linéaire, ce qui nécessite la capacité de distribuer les données de manière dynamique sur les différentes instances du cloud. Pour distribuer les données au travers des instances du système, *Cassandra* utilise une méthode de hachage cohérent (*Constitent Hashing* en anglais) à l'aide d'une fonction de hachage préservant l'ordre [27]. Dans le hachage cohérent, l'ensemble des instances est considéré comme un anneau (c'est-à-dire que le nœud d'indice (*token* en anglais) d'ordre le plus grand est suivi par le nœud d'indice le plus faible). Lorsqu'un nœud se rajoute à l'anneau, on lui assigne un indice. Cet indice déterminera sa position dans l'anneau.

Chaque donnée est identifiée par des clés. Pour déterminer la position de la donnée dans l'anneau [3], sa clé est donnée en entrée à la fonction de hachage qui détermine alors sa position dans l'anneau. Cette position déterminée, la donnée parcourt alors l'anneau jusqu'à trouver le premier nœud dont l'indice sera supérieur à sa position. Ce nœud sera alors responsable de cette donnée.

Chaque nœud est dès lors responsable de toutes les données dont la position est supérieure à l'indice de son prédécesseur et inférieur au sien. On parle de nœud *coordinateur*. Le principal avantage de cette technique est que l'ajout/retrait d'une instance n'influence que ses prédécesseur et successeur directs.

Pour assigner les indices aux nœuds, *Cassandra* propose différents outils de partitions. En voici une liste non-exhaustive :

- Le *RandomPartitioner* va assigner un indice compris entre 0 et 2^{127}. C'est l'outil de partition par défaut de *Cassandra* . Néanmoins, cet aspect aléatoire peut vite conduire à une répartition non-uniforme des charges et des données.
- Le *OrderPreservingPartitioner* va utiliser les clés des données pour les répartir dans l'anneau. C'est donc aux développeurs de l'application de bien choisir ses clés.

3.5.4 Réplication des données

Cassandra utilise la réplication de données dans le but de proposer la haute disponibilité et durabilité de ses données. Le facteur de réplication N définit le nombre de copies des données. Chaque Keyspace a son facteur de réplication.

Chaque nœud est donc chargé de stocker les données dont il est le coordinateur mais aussi de s'assurer que ses données soient répliquées N-1 fois au travers l'anneau. *Cassandra* propose plusieurs options de réplication. Dans la première, *Rack Unaware*, les $N-1$ copies sont répliquées dans les nœuds successeurs directs du coordinateur. Dans les suivantes, *Rack Aware* et *Datacenter Aware*, en utilisant le système Zookeeper [4], un nœud leader est élu. Le leader dit à chaque nœud du système quel ensemble de données (toujours en fonction de leur clé) il est chargé de répliquer. Le leader fait en sorte (dans la mesure du possible) que chaque nœud ne soit jamais responsable de plus de $N-1$ jeux de données tout en s'assurant que les données aient des copies dans différents centres de calcul ou racks.

Le modèle de réplication entre ces différents nœuds est différent de tout ce que nous avons déjà vu. Il n'est ni synchrone, ni asynchrone. *Cassandra* permet de définir un degré de consistance pour chacune des opérations l'attaquant, définissant ainsi (nous le verrons à la section 3.5.5) le nombre de serveurs devant répondre pour que l'opération soit validée. Résumons le concept : l'application envoie une requête à chaque nœud stockant la donnée concernée et attend un certain nombre de réponses pour valider l'opération. Si ce nombre est

3. Un cluster *Cassandra* peut être vu comme un anneau d'instance ayant un rôle similaire.
4. Le service Zookeeper est décrit de manière plus précise en section 3.6.4

égal au facteur de réplication, alors le modèle pourra être considéré comme synchrone ; dans le cas contraire, la réplication sera asynchrone et un phénomène de réparation aura lieu durant les lectures.

3.5.5 Modèle de consistance

S'inspirant de Dynamo [31], *Cassandra* met en place le concept de "finalement consistant" (*Eventually consistent* en anglais) mais ne se limite pas à celui-ci. Son modèle de consistance est configurable au quorum et permet dès lors une large gamme de consistance.

Modèle au quorum

Définissons les trois variables suivantes :
- **N** : le nombre de répliques
- **R** : le quorum en lecture (le nombre de réponses de lectures à attendre lors d'une requête de lecture)
- **W** : le quorum en écriture (le nombre de confirmations d'écriture à attendre lors d'une requête d'écriture)
- si $W + R > N$, la consistance sera dite *forte*
- si $W + R \leq N$, on dira le système *finalement consistant*. Il se peut que les opérations d'écriture et de lecture ne se recouvrent pas. Des conflits de versions peuvent alors apparaitre. Ces conflits peuvent être résolus à l'aide de vector clocks et un système de gestion de versions.

Niveaux de consistance

Il y a quelques niveaux typiques de consistance que l'on peut décider. Ces niveaux peuvent être différents en fonction de l'opération (écriture ou lecture) [4].

Ecriture : Voyons une liste non-exhaustive de ces différents niveaux (par consistance croissante) pour l'écriture :
- ONE : s'assure que l'opération a été écrite sur au moins un nœud. Si un seul nœud répond, l'opération sera considérée comme réussie.
- QUORUM : le quorum en écriture est assigné à *nombre de répliques*/2 +1. Par exemple, pour un facteur de réplication de 8, *Cassandra* écrira l'information dans 5 nœuds (4+1)
- ALL : tous les nœuds (leur nombre est spécifié par le facteur de réplication) doivent retourner une réponse positive. Si un seul nœud signale une erreur, l'écriture est un échec.

On le comprend, plus la consistance sera forte, plus les performances seront faibles.

Lecture Voyons une liste non-exhaustive de ces différents niveaux (par consistance croissante) pour la lecture :
- ONE : retourne la valeur donnée par le premier nœud.
- QUORUM : le quorum de lecture a la même valeur que celui de l'écriture : *nombre de répliques*/2 +1. Via la gestion des conflits de versions des différentes réponses, on construit la donnée et, considérant le quorum de lecture et d'écriture, on a la garantie d'avoir une réponse consistante.

– ALL : interroge tous les nœuds. Si un seul ne répond pas la requête est considérée comme non validée. Dans le cas contraire, on reconstruit la donnée avec toutes les réponses.

On le comprend, tous les mécanismes de réparation ont lieu à la lecture, ce qui peut entrainer des pertes de performances à la lecture. *Cassandra* a donc été modélisée pour des écritures performantes (aux dépens de la performance en lecture).

3.5.6 Modèle de requêtes

Le modèle de requêtes de *Cassandra* peut être considéré comme assez riche (même si il est moins riche que celui de Mongo). A titre d'exemple, nous proposons une liste non-exhaustive des différentes requêtes possibles[4] [5] :

– *GET* retourne une (super) colonne d'une entrée spécifiée par sa clé.
– *GET_SLICE* retourne un ensemble de (super) colonnes (spécifiées par un *SlidePredicate* [6]) d'une entrée spécifiée par sa clé.
– *MULTIGET_SLICE* retourne un ensemble de (super) colonnes (spécifiées par un *SlicePredicate*) de plusieurs entrées (spécifiées par leurs clés).
– *GET_COUNT* retourne le nombre de colonnes présentes dans une entrée et respectant un *SlicePredicate*.

Si *Cassandra* est couplée à un cluster *Hadoop* (voir section 3.6.1), des fonctions MR peuvent être écrites pour faciliter l'exécution de tâches complexes.

3.6 HBase

HBase est un projet open-source écrit en JAVA Apache dont le but est de fournir un produit similaire au produit propriétaire *Big Table* de Google. Tout comme *BigTable* se déployait au dessus du système de fichiers distribué *Google File System*, *Hbase* se déploie sur un système de fichiers distribué open-source : *The Hadoop Distributed Filesystem* (HDFS) (un projet de la fondation *Apache*) qui se charge des opérations de réplication [7].

3.6.1 The Hadoop Distributed Filesystem

HDFS est conçu autour de l'idée que le mode de traitement de données le plus efficace consiste en une et une seule écriture et plusieurs lectures pour chaque donnée. [43]. Un ensemble de données est donc copié ou généré une fois et analysé à de nombreuses reprises. Il est conçu pour pouvoir se déployer sur des nœuds différents les uns des autres (pouvant appartenir à des clusters différents). Dans ce contexte, les défaillances ne sont pas exceptionnelles. Il est conçu de manière à gérer ces défaillances sans entrainer de trop longs temps d'interruption chez l'utilisateur [43].

L'architectre HDFS repose sur deux composants : le nœud de noms (*namenode* en anglais) et les nœuds de données (*datanodes* en anglais) (figure 3.14) et suit le pattern maître-esclave :
.

5. Rappelons que la consistance de la plupart des requêtes peut être définie individuellement pour la plupart des requêtes
6. Un *SlicePredicate* est l'équivalent d'un prédicat en logique mathématique.
7. HBase peut se déployer sur d'autres systèmes de fichiers distribués mais il est d'usage courant de la coupler à HDFS