View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by RERO DOC Digital Library

h e g

NoSQL

Etat de l’art et benchmark

Not Only SQL

Travail de Bachelor réalisé en vue de l’obtention du Bachelor HES

par :

Adriano Girolamo PIAZZA

Conseiller au travail de Bachelor :

David BILLARD, Professeur HES

Genève, 9 octobre 2013

Haute École de Gestion de Genève (HEG-GE)

Filière Informatique de Gestion

Hes·SO GENÈVE
Haute Ecole Spécialisée de Suisse occidentale

Déclaration

Ce travail de Bachelor est réalisé dans le cadre de l'examen final de la Haute école de gestion de Genève, en vue de l'obtention du titre de Bachelor en informatique de gestion. L'étudiant accepte, le cas échéant, la clause de confidentialité. L'utilisation des conclusions et recommandations formulées dans le travail de Bachelor, sans préjuger de leur valeur, n'engage ni la responsabilité de l'auteur, ni celle du conseiller au travail de Bachelor, du juré et de la HEG.

« J'atteste avoir réalisé seul présent travail, sans avoir utilisé des sources autres que celles citées dans la bibliographie. »

Fait à Carouge, le 9 septembre 2013

Adriano Girolamo Piazza

Remerciements

Tout d'abord, je tiens à remercier Monsieur David BILLARD pour m'avoir suivi tout au long de ce travail de Bachelor, ainsi que les différents points sur lesquels il m'a conseillé.

Je souhaite également remercier ma famille ainsi que la famille POLI qui m'ont encouragé à reprendre mes études et m'ont apporté un très grand soutien.

Pour finir je tiens à remercier mes amis pour leur relecture de ce document, ainsi que les corrections orthographiques qu'ils ont pu amener.

Résumé

Les bases de données NoSQL, qui signifie « not only SQL » sont des types de base de données qui ont commencé à émerger depuis 2009, pour répondre aux nouveaux besoins. Le nom peut sembler comme une opposition aux bases de données SQL, sa fonction première n'étant pas de remplacer les bases de données relationnelles, mais de proposer une alternative.

La première partie de ce travail consistera à expliquer ce que sont les bases de données de type NoSQL, l'histoire du NoSQL, dans quel but ce que genre de base de données a vu le jour.

Nous allons également voir de quelle manière ces type de base de données fonctionnent, les technologies qu'elles utilisent, sur quelles fondements elles s'appuient, ainsi que les avantages et désavantages qu'une base de données de type NoSql peut avoir en comparaison à une base de données relationnel standard. Nous allons également voir un bref aperçu du marché actuel.

La dernière partie sera une synthèse d'un travail pratique reprenant la partie Bdd effectuée lors du projet de Génie logiciel et de le reproduire sur une base de données NoSql. Ce rapport expliquera les différents résultats obtenus ainsi qu'une conclusion sur l'applicabilité du NoSql plutôt que le SQL standard pour ce type de projet.

Table des matières

Déclaration i
Remerciements ii
Résumé iii
Table des matières iv
Liste des figures vii
1. Introduction 1
2. L'émergence du NoSQL 2
2.1 Pourquoi l'alternative du NoSQL ? 2
3. Les différences entre le NoSQL et le SQL 2
3.1 Les principes du relationnel 3
3.1.1 Le modèle de données 3
3.1.2 La gestion de la valeur NULL 3
3.2 Les transactions 4
3.2.1 Les propriétés ACID 4
3.3 Cohérence contre disponibilité 5
3.3.1 Le théorème de CAP 5
3.4 Différents types de base de données NoSQL 7
3.4.1 Les bases de données clé-valeur 7
3.4.2 Les bases de données orientées colonnes 8
3.4.3 Les bases de données orientées document 10
3.4.4 Les bases de données orientées graphe 11
4. Le marché 13
4.1 DynamoDB (orientée clé-valeur) 13
4.2 Cassandra (orientée colonne) 14
4.3 MongoDB (orientée document) 15
4.4 Neo4j (orientée graphe) 15
5. Caractéristiques techniques 16
5.1 La distribution de données 16
5.1.1 Distribution sur un schéma maître esclave. 16
5.1.2 Distribution sur un schéma multi-maitre 18
5.2 MapReduce 23
5.2.1 Principe de MapReduce 23
5.2.2 Programmation fonctionnelle 24
5.2.3 Fonctionnement de MapReduce 24
5.2.4 Les avantages 26

5.2.5 Les inconvénients 26
5.2.6 Hadoop 27
5.3 La gestion de la cohérence des données 31
5.3.1 Anti – Entropie 31
6. Les avantages / désavantages du NoSQL 33
6.1 Les avantages 33
6.1.1 Plus évolutif....................33
6.1.2 Plus flexible....................34
6.1.3 Plus économique....................34
6.1.4 Plus simple....................34
6.1.5 Le Cloud Computing 34
6.2 Les désavantages 35
6.2.1 Fonctionnalités réduites 35
6.2.2 Normalisation et Open Source 35
6.2.3 Performances et évolutivité au détriment de la cohérence 35
6.2.4 Manque général de maturité 35
6.3 Conclusion 36
7. Benchmark 36
7.1 Présentation du sujet....................36
7.2 Le modèle de données Cassandra....................36
7.2.1 La colonne 37
7.2.2 La ligne 37
7.2.3 Famille de colonnes 38
7.2.4 Le keyspace....................38
7.2.5 Les différents types d'attributs....................38
7.3 Mise en place de la base de données EasyLocation sur Cassandra.......39
7.3.1 Création de la base de données....................39
7.3.2 Gestion des utilisateurs....................40
7.3.3 La modélisation par la dénormalisation 42
7.3.4 Modèle de données « Easy_location »....................42
7.3.5 L'indexation....................45
7.3.6 Utilisation de Cassandra-cli pour une meilleure approche....................46
7.3.7 La création de familles de colonnes 46
7.3.8 Insertion de données....................48
7.4 Conclusion 50
Bibliographie 53
Webographie 54
Annexe 1 : Script CQL de création des utilisateurs et D'attribution des permissions 58
Annexe 2 : Modèle de données relationnelle Easy_Location....................59

Annexe 3: Modèle de données dénormalisé……….. 60

Annexe 4: Création des familles de colonnes avec le client « Cassandra-cl » 61

Liste des figures

Figure 1 : Gestion de la valeur NULL 4
Figure 2 : Théorème de CAP……………………...6
Figure 3 : Représentation clé-valeur…………………………………………………7
Figure 4 : Comparaison schéma relationnel vs colonne………………………………. 9
Figure 5 : Représentation d'une famille de colonnes………………………………………9
Figure 6 : Base de données orientée document…………………………………..10
Figure 7 : Représentation des graphes……………………………………………12
Figure 8 : Logo DynamoDB………………………………………………………13
Figure 9 : Logo Cassandra………………………………………………………..14
Figure 10 : Logo mongoDB…..15
Figure 11 : Logo Neo4j……………………………………………………………15
Figure 12 : Distribution sur schéma maître-esclave………………………………17
Figure 13 : Illustration de problème de cohérence lors de la consultation……………..18
Figure 14 : Protocole de bavardage…………………………………………………...19
Figure 15 : Distribution par hachage sur la clé primaire…………………………20
Figure 16 : Répartition sur hachage consistant……………………………………21
Figure 17 : Ajout d'un nouveau nœud………………………………………………22
Figure 18 : Schéma MapReduce……………………………………………………25
Figure 19 : Exemple comptage de mot……………………………………………..26
Figure 20 : Hadoop Distributed FileSystem………………………………………..28
Figure 21 : Architecture d'un cluster Hadoop……………………………………...29
Figure 22: Fonctionnement de MapReduce dans Hadoop…………………………30
Figure 23 : Représentation du HashTree sur Cassandra…………………………...32
Figure 24 : Représentation de conflit………………………………………………33
Figure 25 : Description d'une colonne……………………………………………..37
Figure 26 : Description d'une ligne…………………………………………………37
Figure 27 : Description d'une famille de colonnes…………………………………..38
Figure 28 : Différents types proposés par Cassandra……………………………39
Figure 29 : Modèle conceptuel de données…………………………………………..43
Figure 30 : Modélisation par la dénormalisation…………………………………..44
Figure 31 : Interrogation table client………………………………………………..49
Figure 32 : Interrogation table client………………………………………………..50
Figure 33 : Modèle de données SQL Easy Location………………………………...59
Figure 34 : Modèle de données dénormalisé………………………………………...60

1. Introduction

Qu'est-ce qu'une base de données ? A quoi cela peut-il servir ? Une base de données est un dispositif servant à stocker des données de différents types telles que des lettres, des chiffres, des dates… de manière structuré. Aujourd'hui, pratiquement tous les programmes informatiques utilisent une base de donnée afin de pouvoir consulter, ajouter, modifier des informations diverses. La gestion de ces données, ainsi que l'accès se fait à travers une suite de programme qu'on appelle SGBD (Système de gestion de base de données).

Le SQL qui est aujourd'hui le langage le plus utilisé par les systèmes informatiques, fut développée par IBM en 1970, sous le nom SEQUEL (Structured English Query Language), par la suite renommé SQL en 1975 pour des raisons de propriété intellectuelle, soit de marque déposée. Il sera finalement normalisé internationalement par ISO en 1987.

Bien que le terme soit entendu pour la première fois en 1988 par Carlo Strozzi qui présentait ça comme un modèle de base de donnée plus léger et sans interface SQL, c'est en 2009, lors de la rencontre « meetup » NoSql de San Francisco, qu'il prend un essor important. Durant cette conférence, il y sera présenté des solutions de projet telles que Voldemort, Cassandra Project, Dynomite, HBase, Hypertable, CouchDb, MongoDb. Ce « meetup » sera considérée comme l'inauguration de la communauté des développeurs de logiciel NoSql.

2. L'émergence du NoSQL

Avec l'augmentation de la bande passante sur internet, ainsi que la diminution des coûts des matériels informatiques, de nouvelles possibilités ont vu le jour dans le domaine de l'informatique distribuée. Le passage au 21ème siècle par la révolution du WEB 2.0 a vu le volume de données de certaines entreprises augmenter considérablement. Ces données proviennent, essentiellement, de réseaux sociaux, base de données médicales, indicateurs économiques,etc. L'informatisation croissante de traitement en tout genre a eu pour conséquence une augmentation exponentielle de ce volume de données qui se compte désormais en pétaoctets, les anglo-saxon l'ont nommé le Big Data. La gestion de ces volumes de données est donc devenue un problème que les bases de données relationnelles n'ont plus été en mesure de gérer. Le NoSQL regroupe donc de nombreuses bases de données qui ne se reposent donc plus sur une logique de représentation relationnelle. Il n'est toutefois pas simple de définir explicitement ce qu'est une base de données NoSql étant donné qu'aucune norme n'a encore été instaurée.

2.1 Pourquoi l'alternative du NoSQL ?

Le besoin fondamental auquel répond le NoSQL est la performance. Afin de résoudre les problèmes liés au « Big data », les développeurs de sociétés telles que Google et Amazone ont procédé à des compromis sur les propriétés ACID des SGBDR. Ces compromis sur la notion relationnelle ont permis aux SGBDR de se libérer de leurs freins à la scalabilité horizontale. Un autre aspect important du NoSql est qu'il répond au théorème de CAP qui est plus adapté pour les systèmes distribués.

L'autre avantage du NoSQL est d'ordre financier. Servant à manipuler de gros volumes de données, il est donc destinés aux grandes entreprises. Dès 2010, le NoSql a commencé à s'étendre vers les plus petites entreprises. Majoritairement des start-up n'ayant pas les moyens d'acquérir des licences Oracle qui ont donc développé leurs propres SGBD en imitant les produits Google et Amazon.

3. Les différences entre le NoSQL et le SQL

Pour commencer, il est important de savoir que le SQL n'est pas un modèle relationnel en soit, mais un langage de manipulation de données conçu autour du modèle relationnel. Les bases de données NoSql n'ont pas pour but de s'éloigner de ce langage mais du modèle relationnel. Nous allons voir les facteurs différenciateurs.

3.1 Les principes du relationnel

3.1.1 Le modèle de données

Le modèle relationnel basé est sur un modèle mathématique qui est celui de la notion des ensembles. Chaque ensemble ici est représenté par une table, et ces attributs sont quant à eux représentés par des colonnes. L'un des principes fondamentaux est justement cette notion de relation entre tables à l'aide de cardinalités, clés primaires et clé étrangères, ceci implique au préalable une étude minutieuse sur la modélisation du schéma de la base de données. Comme par exemple les tables dont le système aura besoin, ses attributs, les relations possibles entre différentes tables, etc. Une fois ce modèle mis en place dans un SGBDR, il est difficile de changer la structure de celui-ci et ceci pose par conséquent des problèmes lors de sa réingénierie.

Le mouvement NoSql lui est plus pragmatique, basé sur des besoins de stockage de données ainsi qu'une liaison plus forte avec les différents langages clients. La plupart des moteurs de base de données NoSQL n'utilisent pas de schémas prédéfinis à l'avance, d'où l'appellation « schema-less », ce qui signifie sans schéma. Ainsi le moteur de la base de données n'effectue pas de vérification et n'impose pas de contrainte de schéma, les données étant organisées du coté client du code. Toutefois le principe « schema-less » est théorique et ne se vérifie pas entièrement en pratique. Le maintien d'une structure de données homogènes est important, pour des questions d'indexation, de recherche par critère ou tout simplement pour des raisons de logique.

3.1.2 La gestion de la valeur NULL

En SQL, lors de la définition d'une table, il est possible de décider si un attribut peut contenir une valeur ou non lors de l'enregistrement d'un tuple en définissant la colonne à « Null » ou « Not Null ». Ceci est utile pour contraindre l'utilisateur à renseigner sur certains attributs que l'administrateur de la base de données considère comme indispensables. Cela dit, dû au fait de sa représentation en deux dimensions (ligne, colonne) et de sa rigidité dans sa structure, il est indispensable de signaler l'absence de valeur à l'aide d'un marqueur NULL, ce qui coûte en espace mémoire. Un autre souci du NULL est sa gestion dans le langage SQL. Etant donné que le NULL n'est pas une valeur, il ne peut donc pas être comparable. En voici une illustration avec la clause « WHERE »:

Figure 1

Gestion de la valeur NULL

Produit	
Type	Prix
Viande	10
Chocolat	20
Salade	20
Pain	NULL
Fromage	10

Test	Résultat
Select * From Produit WHERE Prix <> 10	Chocolat, Salade
Select * From Produit WHERE Prix <> 20	Viande, Fromage
Select * From Produit WHERE (Prix <> 20) OR (Prix IS NULL)	Viande, Fromage, Pain

L’exemple ci-dessus démontre qu’il faut explicitement citer la valeur NULL dans la requête. En effet, le test sur une valeur NULL ne retourne pas TRUE ou FALSE mais la valeur UNKNOW. Dans nos 2 premiers tests il va donc sélectionner les produits qui répondront TRUE à leur requête respective. Dans le troisième test il faut explicitement signaler une valeur NULL possible pour que la valeur Null soit prise en considération. Il faut donc gérer une logique à 3 états possibles.

Dans des bases de données NoSQL la gestion du NULL n’existe pas. Etant donné que dans des bases de type document ou clé-valeur la notion de schéma n’existe pas, si l’on veut exprimer qu’un attribut n’a pas de valeur il suffit de ne pas le mettre. Même chose dans les bases de type colonne, les colonnes étant dynamiques, elles ne seront présentes qu’en cas de nécessité.

3.2 Les transactions

Une transaction est une suite d’opérations faisant passer l’état du système d’un point A (antérieur à la transaction) à un point B (postérieur à la transaction). Dans les SGBDR une transaction doit respecter les propriétés ACID afin que celle-ci soit validée. Le respect de ces propriétés dans un environnement distribué est pénible et il existe un risque de diminution des performances proportionnelle aux nombres de serveurs. Les moteurs NoSQL font l’impasse sur ces propriétés, leur but étant d’augmenter les performances par l’ajout de serveurs (« scalabilité horizontale »).

3.2.1 Les propriétés ACID

Les propriétés ACID (atomicité, cohérence, isolation, durabilité) sont un ensemble de propriétés garantissant la fiabilité d’une transaction informatique telle qu’un virement bancaire par exemple.

- Atomicité : cette propriété assure qu’une transaction soit effectuée complètement, ou pas du tout. Quelle que soit la situation, par exemple lors d’une panne d’électricité, du disque dur ou de l’ordinateur, si une partie de la

transaction n'a pu être effectuée, il faudra effacer toutes les étapes de celle-ci et remettre les données dans l'état où elles étaient avant la transaction.

- Cohérence : la cohérence assure que chaque transaction préserve la cohérence de données, en transformant un état cohérant en un autre état de même données cohérent. La cohérence exige que les données concernées par une potentielle transaction soient protégés sémantiquement.
- Isolation : l'isolation assure que chaque transaction voit l'état du système comme si elle était la seule à manipuler la base de données. En d'autres termes, si pendant la transaction *T1*, la transaction *T2* s'exécute au même moment, *T1* ne doit pas la voir, tout comme *T2* ne doit pas voir *T1.*
- Durabilité : La durabilité assure qu'une transaction confirmée le demeure définitivement, peu importe les différents imprévus (panne d'électricité, panne d'ordinateur etc). Pour ce faire, un journal contenant toutes les transactions est créé, afin que celles-ci puissent être correctement terminées lorsque le système est à nouveau disponible.

3.3 Cohérence contre disponibilité

La cohérence de données ou la disponibilité de celles-ci, que choisir? C'est l'un des sujets les plus sensibles entre les deux mondes que constituent le relationnel et le NoSQL. Dans un système de gestion de base de données relationnelle, les différents utilisateurs voient la base de données dans un état cohérent, les données en cours de modification par un utilisateur n'étant pas visibles aux autres utilisateurs, on dit qu'un verrou a été posé. Le maintien de cette cohérence de données dans une base contenue dans un seul serveur (ou autrement dit architecture centralisée) est tout à fait envisageable (les SGBRD ont longuement fait leurs preuves), cela devient cependant problématique dans le contexte d'une architecture distribuée.

Les fondateurs du NoSQL tel que Google, Amazon, Facebook etc., dont les besoins en disponibilité priment sur la cohérence des données, ont décidés de privilégier celle-ci au détriment de la cohérence. L'opposition de ces deux propriétés a été présentée par Eric Brewer dans ce qu'on appelle aujourd'hui le théorème de CAP.

3.3.1 Le théorème de CAP

Le théorème de cap ou théorème de Brewer a été énoncé pour la première fois en tant que conjecture par le chercheur en informatique Eric Brewer. En 2002, deux chercheurs au MIT, Seth Gilbert et Nancy Lynch, ont formellement démontré la vérifiabilité de la conjecture de Brewer afin d'en faire un théorème établi. Ce théorème énonce qu'une

base de données d'un système distribué ne peut garantir les trois contraintes suivantes en même temps, à savoir :

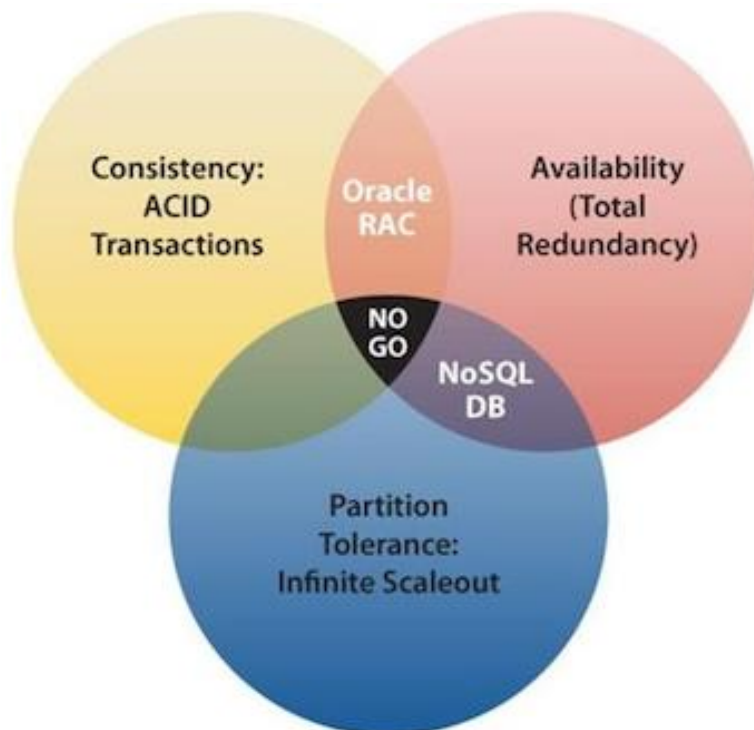
Cohérence (Consistency) : tous les nœuds (serveurs) du système voient exactement les mêmes données au même moment.

Disponibilité (Availability) : garantie que toutes les requêtes reçoivent une réponse.

Résistance au morcellement (Partition Tolerence) : le système doit être en mesure de répondre de manière correcte à toutes requêtes dans toutes circonstances sauf en cas d'une panne générale du réseau. Dans le cas d'un morcellement en sous-réseaux, chacun de ces sous-réseaux doit pouvoir fonctionner de manière autonome.

Figure 2

Théorème de CAP



Le théorème de CAP dit que seules deux de ces trois contraintes peuvent être respectées à un instant *T* car elles se trouvent en opposition. La plupart des bases de données NoSql se concentrent plus sur la résistance au morcellement, afin de garantir une disponibilité en tout temps et par conséquent abandonnent la cohérence.

3.4 Différents types de base de données NoSQL

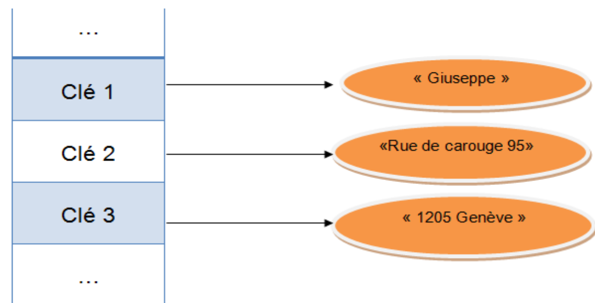
Les bases de données NoSql sont donc une catégorie de base de données qui n'est plus fondée sur l'architecture classique des bases relationnelles, et non pas un type à part entière. Quatre grandes catégories se distinguent parmi celles-ci.

3.4.1 Les bases de données clé-valeur

La base de données de type clé-valeur est considérée comme la plus élémentaire. Son principe est très simple, chaque valeur stockée est associée à une clé unique. C'est uniquement par cette clé qu'il sera possible d'exécuter des requêtes sur la valeur.

Figure 3

Représentation clé-valeur 1



La structure de l'objet stocké est libre et donc à la charge du développeur de l'application. Un avantage considérable de ce type de base de données est qu'il est très facilement extensibles, on parle donc de scalabilité horizontale. En effet dans le cas où le volume de données augmente, la charge est facilement répartissable entre les différents serveurs en redéfinissant tout simplement les intervalles des clés entre chaque serveur.

En ce qui concerne le besoin de la scalabilité verticale, celle-ci est très fortement réduite par la simplicité des requêtes qui se résument à PUT, GET, DELETE, UPDATE. Ce type de base de données affiche donc de très bonnes performances en lecture/écriture et tendent à diminuer le nombre de requêtes effectuées, leur choix étant restreint.

Néanmoins, les requêtes ne peuvent être exécutées uniquement sur les clés à cause de la simplicité du principe. Permettant un stockage de données sans schéma, le client est contraint à récupérer tout le contenu du blob et ne peut obtenir un résultat de requêtes plus fin tel que des résultats de requêtes SQL pourraient permettre.

Ce type de base de données orienté clé-valeur implique donc des cas d'utilisation très spécifiques, à cause de leur simplicité de modèle et de la fine palette de choix de requêtes proposées. Ces systèmes sont donc principalement utilisés comme dépôt de données à condition que les types de requêtes nécessitées soient très simples. On les retrouve comme système de stockage de cache ou de sessions distribuées, particulièrement là où l'intégrité des données est non significative. Aujourd'hui, les

solutions les plus connues ayant adoptées le système de couple clé-valeur sont Voldemort (LinkedIn), Redis et Riak.

3.4.2 Les bases de données orientées colonnes

Les bases de données orientées colonnes ont été conçues par les géants du web afin de faire face à la gestion et au traitement de gros volumes de données s'amplifiant rapidement de jours en jours. Elles intègrent souvent un système de requêtes minimalistes proche du SQL.

Bien qu'elles soient abondamment utilisées, il n'existe pas encore de méthode officielle ni de règles définies pour qu'une base de données orientée colonnes soit qualifiée de qualité ou non.

Le principe d'une base de données colonnes consiste dans leur stockage par colonne et non par ligne. C'est-à-dire que dans une base de données orientée ligne (SGBD classique) on stocke les données de façon à favoriser les lignes en regroupant toutes les colonnes d'une même ligne ensemble. Les bases de données orientée colonnes quant à elles vont stocker les données de façon à ce que toute les données d'une même colonne soient stockées ensemble. Ces bases peuvent évoluer avec le temps, que ce soit en nombre de lignes ou en nombre de colonnes. Autrement dit, et contrairement à une base de données relationnelle ou les colonnes sont statiques et présentes pour chaque ligne, celles des bases de données orientée colonnes sont dite dynamiques et présentes donc uniquement en cas de nécessité. De plus le stockage d'un « null » est 0. Prenons l'exemple de l'enregistrement d'un client nécessitant son nom, prénom et adresse. Dans une base de données relationnelle il faut donc au préalable créer le schéma (la table) qui permettra de stocker ces informations. Si un client n'a pas d'adresse, la rigidité du modèle relationnel nécessite un marqueur NULL, signifiant une absence de valeur qui coûtera toutefois de la place en mémoire. Dans une base de données orientée colonne, la colonne adresse n'existera tout simplement pas. Les bases de données colonnes ont été pensées pour pouvoir stocker plusieurs millions de colonnes et se relèvent donc parfaites pour gérer le stockage dit « one-to-many »

Figure 4

Comparaison schéma relationnel vs colonne

ID	Prénom	Opérateur	Localité
1	Alain		Meyrin
2	Adriano	Swisscom	
3	Sébastien		
4			Carouge
5			Onex

Schéma: données d'une base de données relationnelle

Prénom	Opérateur	Localité
Alain(1)	Swisscom(2)	Meyrin(1)
Adriano(2)		Carouge(4)
Sébastien(3)		Onex(5)

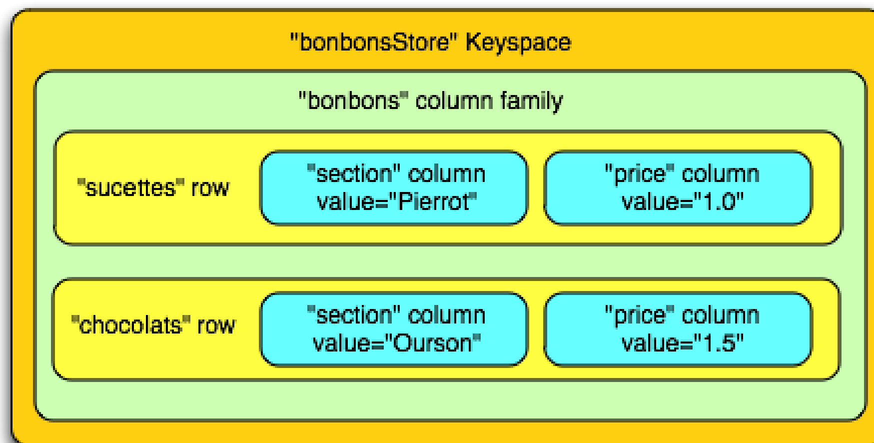
Schéma: données d'une base de données orientée colonnes

Ces deux schémas démontrent que le stockage de données en colonne permet de gagner un espace considérable, spécialement lors des stockages des tuples incomplets. En effet, la valeur « null » n'est pas stockée dans ce type de base de données.

Dans des bases de données telles que Cassandra ou HBase il existe quelques concepts supplémentaires qui sont pour commencer les familles de colonnes, qui est un regroupement logique de lignes. Dans le monde relationnel ceci équivaudrait en quelque sorte à une table. Cassandra offre une extension au modèle de base en ajoutant une dimension supplémentaire appelée « Super colonnes » contenant elle-même d'autres colonnes.

Figure 5

Représentation d'une famille de colonnes



Les bases de données orientées colonnes comportent des avantages considérables. Leur capacité de stockage est accrue, en partie grâce à l'espace économisé sur les valeurs « null »*, comme déjà vu plus haut. Leur compression est significative pour autant que les données des colonnes se ressemblent fortement. Afin d'éviter la décompression à chaque interrogation des données, de plus en plus de SGBD orientées

colonnes ont intégré une technologie permettant d'accéder directement aux données sans les décompresser au préalable.

Elles assurent également une rapidité remarquable d'accès aux données. Si l'on souhaite récupérer un gros volume de donnés d'un attribut spécifique d'un enregistrement, il est préférable de récupérer les informations de toute cette colonne plutôt que de parcourir les lignes une à une et de prendre uniquement cette information.

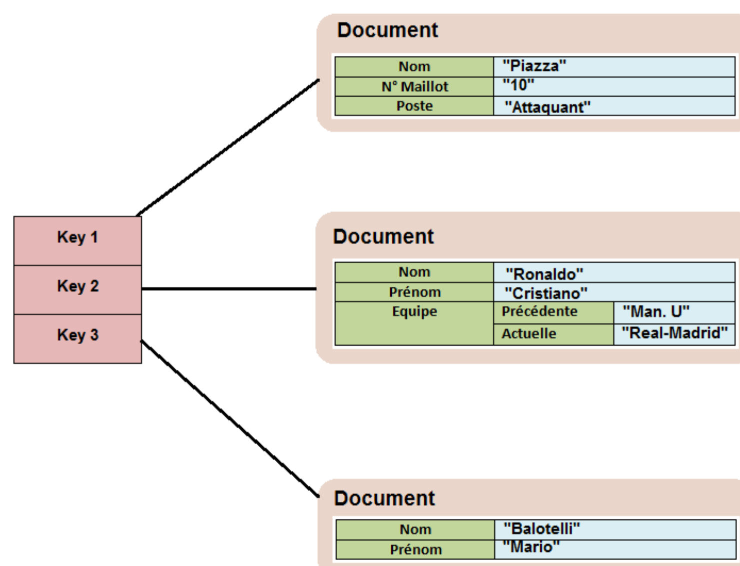
Cependant, l'utilisation de ces bases de données n'est réellement qu'efficace dans un contexte « BigData », pour les grands volumes de données de même type, et dont les données se ressemblent. De plus, le système de requêtes minimalistes comme cité plus haut a son prix et les différentes solutions possibles pour interroger les données sont très limitées.

3.4.3 Les bases de données orientées document

Les bases de données document sont une évolution des bases de données de type clé-valeur. Ici les clés ne sont plus associées à des valeurs sous forme de bloc binaire mais à un document dont le format n'est pas imposé. Il peut être de plusieurs types différents comme par exemple du JSON ou du XML, pour autant que la base de données soit en mesure de manipuler le format choisit afin de permettre des traitements sur les documents. Dans le cas contraire, cela équivaudrait à une base de données clé-valeur. Bien que les documents soient structurés, ce type de base est appelée « schemaless ». Il n'est donc pas nécessaire de définir les champs d'un document.

Figure 6

Base de données orientée document



Comme l'image peut nous le montrer, chacun des documents ci-dessus a une structure différente, ils peuvent donc être très hétérogènes. Un document peut contenir des chaînes de caractères comme illustré sur cette image, mais également des valeurs numériques ainsi que d'autres documents.

L'avantage des bases de données document est de pouvoir récupérer un ensemble d'informations structurées hiérarchiquement depuis une clé. Une opération similaire dans le monde relationnelle équivaudrait à plusieurs jointures de table.

Les bases de données document sont par conséquents très convoitées par les applications Web diffusant des pages entières obtenues par un ensemble de jointures. Ces applications peuvent également mettre en cache des informations sous une forme intelligible. Pour la majorité d'entre elles, la base peut être directement mise sur la mémoire RAM permettant un gain de performance considérable.

Un point négatif concernant ces bases de données document concerne la duplication des informations. Les données étant regroupées par document, il se pourrait que des problèmes d'intégrité surgissent, certaines données étant nécessairement dupliquées sur plusieurs documents. De plus, par la très grande flexibilité de leur schéma (à priori un atout), il est très facile de commettre une erreur lors de l'insertion de données. Les bases de données relationnelles offrent une plus grande garantie de cohérence des données, car dans un cas comme celui-ci, le serveur refusera d'exécuter la requête si le schéma de la base de données n'est pas respecté.

3.4.4 Les bases de données orientées graphe

Bien que les bases de données de type clé-valeur, colonne, ou document tirent leur principal avantage de la performance du traitement de données, les bases de données orientées graphe permettent de résoudre des problèmes très complexes qu'une base de données relationnelle serait incapable de faire. Les réseaux sociaux (Facebook, Twitter,etc), où des millions d'utilisateurs sont reliés de différentes manières, constituent un bon exemple : amis, fans, famille etc. Le défi ici n'est pas le nombre d'élément à gérer, mais le nombre de relations qu'il peut y avoir entre tous ces éléments. En effet, il y a potentiellement n^2 relations à stocker pour n éléments. Même s'il existe des solutions comme les jointures, les bases de données relationnelles se confrontent très vite à des problèmes de performances ainsi que des problèmes de complexité dans l'élaboration des requêtes. L'approche par graphes devient donc inévitable pour les réseaux sociaux tels que Facebook ou Twitter.

Les bases de données graphe sont également utilisées pour la gestion d'importantes structures informatiques. Elles permettent également l'élaboration de liens entre les divers intérêts que pourraient avoir un internaute, afin de pouvoir lui proposer des produits susceptibles de l'intéresser. Ainsi, les pubs s'affichant sur Facebook sont très souvent en relation avec les recherches effectués sur Google, et les propositions d'achats de sites de vente en ligne tels qu'Ebay et Amazon sont en relation avec des achats déjà effectués (par exemple : les personnes ayant acheté ce produit ont également acheté …).

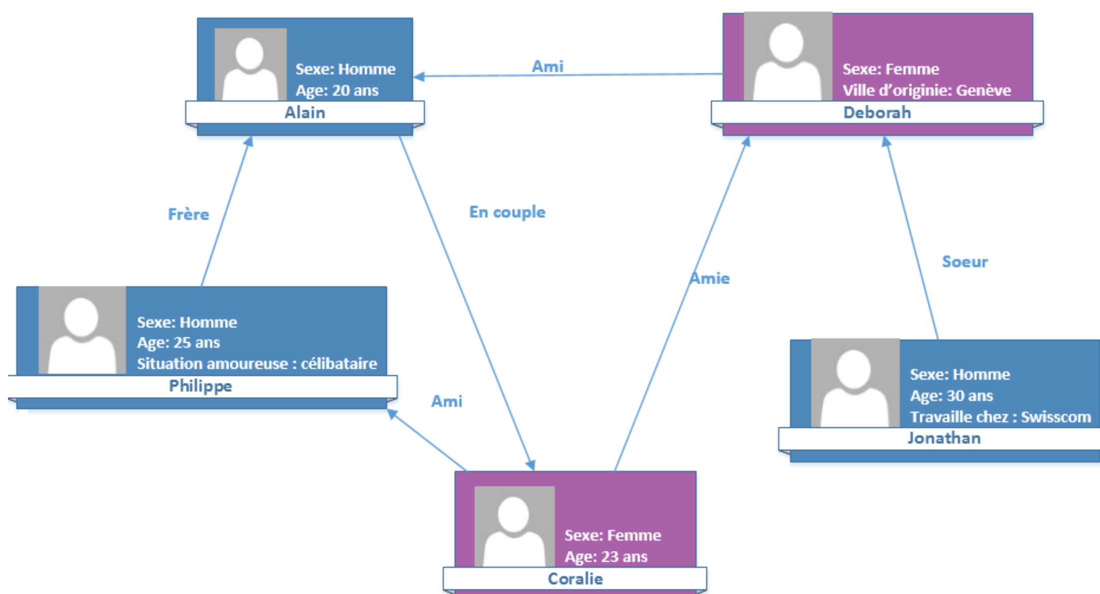
Comme son nom l'indique, ces bases de données reposent sur la théorie des graphes, avec trois éléments à retenir :

- Un objet (dans le contexte de Facebook nous allons dire que c'est un utilisateur) sera appelé un Nœud.
- Deux objets peuvent être reliés entre eux (comme une relation d'amitié).
- Chaque objet peut avoir un certain nombre d'attributs (statut social, prénom, nom etc.)

Les données sont donc stockées sur chaque nœud, lui-même organisé par des relations. A partir de là, il sera nettement plus aisé d'effectuer des opérations qui auraient été très complexes et lourdes dans un univers relationnel.

Figure 7

Représentation des graphes



L'implémentation d'une base de données graphe peut varier selon les besoins, ou les propriétés à mettre en avant suivant l'utilisation. Certaines se basent sur des orientations colonne, d'autres sur l'enregistrement clé-valeur ou sur une combinaison de plusieurs d'entre elles. HyperGraphDB, Neo4j, FlockDB, BigData sont quelques noms de bases de données orientées graphe.

Le point positif de ce type de bases est qu'elles sont parfaitement adaptées à la gestion des données relationnelles même dans un contexte de « BigData ». De plus, leur architecture étant modelable, elles peuvent être adaptées selon les besoins rencontrés. Cependant, cette architecture est très limitée dans d'autre cas plus classiques car très spécifique aux graphes.

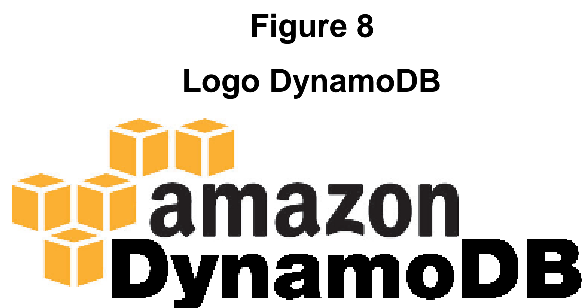
4. Le marché

Principalement lancé par les géants du Web (Facebook, LinkedIn, Google etc.) pour trouver une solution à leurs besoins, On décompte aujourd'hui 150 bases de données NoSQL de tous types (Clé-valeur, Colonne, Document, Graphe). La majorité de ces bases de données se trouve sous licence Open Source. Etant donné le nombre relativement élevé de ces bases de données, je ne vais parler que des plus populaires.

4.1 DynamoDB (orientée clé-valeur)

Figure 8

Logo DynamoDB



Conçu par Amazon, DynamoDB est un service de base de données NoSQL de type clé-valeur. Parce qu'il est un service basé sur le cloud, il permet aux clients l'utilisant de s'affranchir de lourdes tâches administratives que peuvent représenter l'exploitation et le dimensionnement d'un cluster de base de données distribué hautement disponible. DynamoDB propose également une grande quantité de fonctionnalités permettant le développement de celle-ci de manière rapide et efficace. Afin de garantir un haut niveau de durabilité ainsi que de disponibilité, les données sont stockées sur des disques SSD sur trois zones de disponibilités. Cette nouvelle génération de mémoire, proche de celles qu'utilisent les clés USB, constitue un atout très important en faveur DynamoDB. Elle propose des temps d'accès bien plus courts qu'un accès disque classique.

Un autre aspect important d'Amazon DynamoDB est son intégration à Amazon Elastic MapReduce (Amazon EMR), un puissant outil qui permet entre autres d'effectuer des

analyses complexes sur de très gros volume de données. Nous reviendrons plus en détail sur le paradigme MapReduce plus bas.

La souscription aux services de DynamoDB permet aux entreprises d'éviter des coûts d'infrastructure et de maintenance considérables, et permet ainsi aux entreprises de se focaliser sur leur core-business.

4.2 Cassandra (orientée colonne)

Figure 9
Logo Cassandra

Initialement développé par Facebook afin de répondre à des besoins concernant son service de messagerie, Cassandra a été libéré en Open-source et a été adopté par plusieurs autres grands du Web tel que Digg.com ou Twitter. Il est aujourd'hui l'un des principaux projets de la fondation Apache, une organisation à but non lucratif développant des logiciels open-source. Cassandra est un système de gestion de base de données NoSQL orientée colonne et écrit en java. Il permet la gestion massive de données réparties sur plusieurs serveurs, assurant ainsi une haute disponibilité des données. Voici une liste des principales caractéristiques de Cassandra.

- **Haute tolérance aux pannes :** les données d'un nœud sont automatiquement répliquées sur différents nœuds. De cette manière, les données qu'il contient sont également disponibles sur d'autres nœuds si l'un des nœuds venait à être hors service. Conçu sur le principe qu'une panne n'est pas une exception mais une normalité, il est simple de remplacer un nœud qui est tombé sans rendre le service indisponible.

- **Modèle de données riche :** Cassandra propose une modèle de données basé sur BigTable (par Google) de type clé-valeur. Il permet de développer de nombreux cas d'utilisation dans l'univers Web.

- **Elastique :** Que ce soit en écriture ou en lecture, les performances augmentent de façon linéaire lorsqu'un serveur est ajouté au cluster. Cassandra assure également qu'il n'y aura pas d'indisponibilité du système, ni aucune interruption au niveau des applications.

4.3 MongoDB (orientée document)

Figure 10

Logo mongoDB

Développé depuis 2007 par 10gen (une société de logiciel), MongoDB est un système de gestion de base de données orientée document. Ecrit en C++ et distribué sous licence AGPL(licence libre), elle est très adapté aux applications web. MongoDB a été adopté par plusieurs grands noms de l'informatique, tels que Foursquare, SAP, ou bien même GitHub.

Manipulant des documents au format BSON (Binary JSON), un dérivé de JSON binaire plus axé sur la performance, l'utilisation d'un pilote est donc nécessaire pour communiquer avec une base MongoDB. Fonctionnant comme une architecture distribuée centralisée, il réplique les données sur plusieurs serveurs avec le principe de maître-esclave, permettant ainsi une plus grande tolérance aux pannes. La répartition et la duplication de document est faite de sorte que les documents les plus demandés soient sur le même serveur et que celui-ci soit dupliqué un nombre de fois suffisant.

Par sa simplicité d'utilisation du point de vue de développement client, ainsi que ces performances remarquables, MongoDB est l'une de base de données orientées document la plus utilisé.

4.4 Neo4j (orientée graphe)

Figure 11

Logo Neo4j

Neo4j est une base de données orientée graphe écrite en Java. Développé par Neo Technology, elle a vu le jour pour la première fois en 2007. Celle-ci est open-source et est distribuée sous une double licence GPLv3 et AGPLv3. Son usage fait intuitivement sens dans le contexte des réseaux sociaux comme par exemple Viadeo dans le cadre de son moteur de recommandation de contact. Neo4j est également présent dans les télécoms (Teleno, Deutsche Telekom, SFR) avec un but de gestion de catalogues permettant la modélisation d'une large palette de combinaisons entre forfait téléphonique, option éligible, etc. Il s'est avéré être également très utile pour la détection de fraudes.

Une grande particularité de Neo4j est qu'il supporte les transactions dites ACID. Il est doté d'un moteur de graphe extrêmement performant et possède toutes les caractéristiques d'une base de données de production. Outre les points forts qui sont la haute disponibilité des données et une très grande scalabilité, il est important de noter que Neo4j est en production depuis 2003 sans jamais avoir subi la moindre interruption ce qui témoigne de sa très grande robustesse.

5. Caractéristiques techniques

Dans ce chapitre je vais aborder les caractéristiques techniques principales sur lesquelles se repose le NoSQL.

5.1 La distribution de données

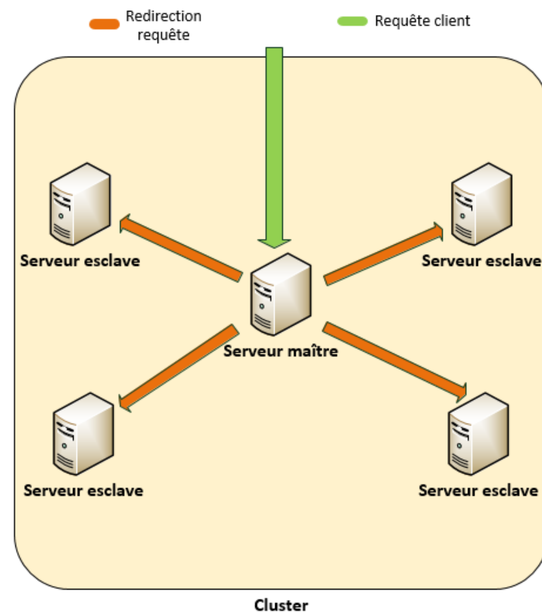
Comme déjà dit précédemment, les moteurs NoSQL sont majoritairement conçus pour être utilisés sur une architecture distribuée, afin de pouvoir gérer la montée de charge et de volumétrie de données, rappelons-nous du principe de la scalabilité horizontale. Les moteurs NoSQL utilisent deux manières différentes pour distribuer les traitements et les données sur leurs divers nœuds. La distribution de données avec maître et celle sans.

5.1.1 Distribution sur un schéma maître esclave.

La distribution de données sur un schéma dit maître esclave consiste à avoir un seul serveur dit maître dans un cluster, les autres serveurs étant esclave. Les requêtes des clients arrivent directement sur le serveur maître, pour ensuite être redirigées sur les serveurs « esclaves » qui contiennent l'information de la requête.

L'une des vulnérabilités de cette configuration consiste dans l'exposition à un point unique de défaillance (SPOF – Single Point of Failure). Ainsi, une malfonction du serveur « maître » entraîne la panne du cluster. En effet, si le serveur tombe en panne, il ne pourra plus réceptionner les requêtes clientes et les redistribuer aux serveurs « esclave » concernés.

Figure 12 Distribution sur schéma maître-esclave



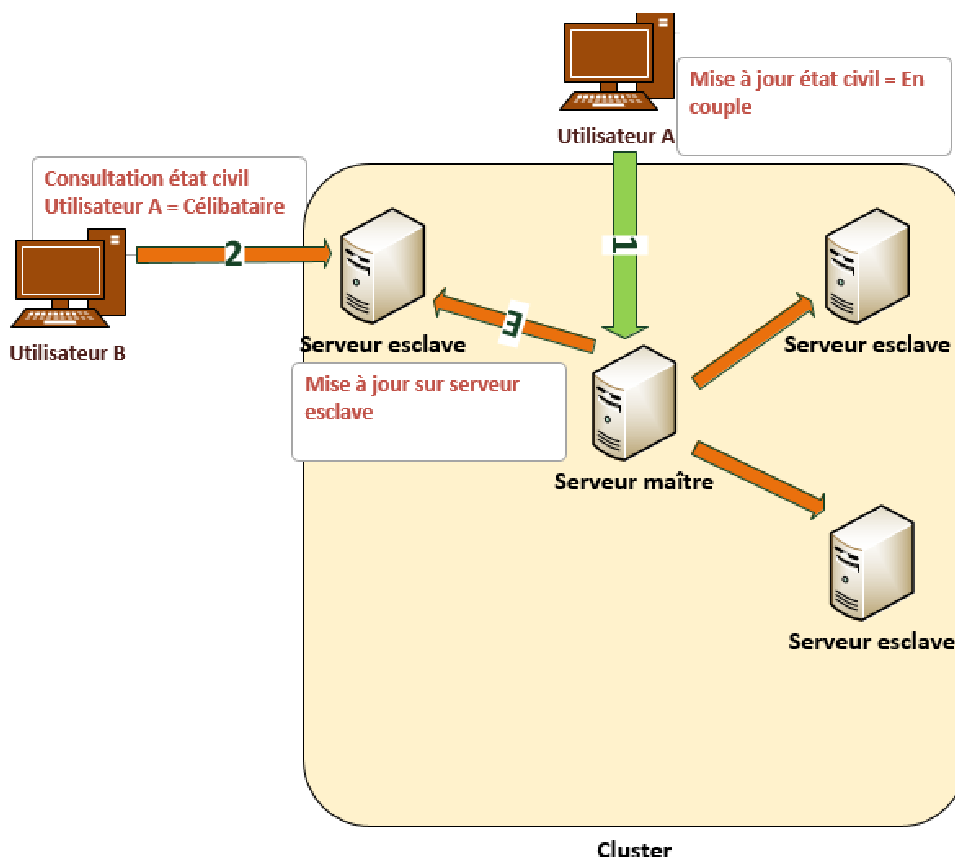
Un point important ici est la façon dont sont répliquées les données dans un schéma maître-esclave. Une réplication de données consiste à faire une copie des données entre serveurs afin de garantir non seulement la disponibilité de celles-ci mais aussi de se protéger de la perte de données.

Cela fonctionne ainsi : Les écritures se font uniquement sur le serveur « maître », le serveur lui va se charger de faire les réplications de données sur les serveurs « esclaves ». Le processus de réplication est géré automatiquement par le serveur et sa fréquence peut être configurable par l'administrateur. La lecture de données peut être faite aussi bien sur le maître que sur l'esclave, avec néanmoins un risque de produire des problèmes de cohérence de données si une lecture est faite sur un esclave n'ayant pas reçu la dernière version des données du maître.

Prenons un exemple, soit deux utilisateurs Facebook l'utilisateur A et l'utilisateur B. L'utilisateur A met à jour son état civil en passant de « célibataire » à « en couple », l'utilisateur B au même moment va consulter l'état civil de l'utilisateur A indiquant « célibataire ». Ceci s'explique par le fait que le serveur sur lequel les informations ont été consultées se trouvait être un serveur « esclave » n'ayant pas encore reçu le réplica de la base de donnée à jour du serveur « maître ». En effet, les réplications de données ne se font pas en temps réel.

Figure 13

Illustration de problème de cohérence lors de la consultation



5.1.2 Distribution sur un schéma multi-maitre

La distribution de données sur une architecture multi-maître part du principe que la panne est une normalité et que chaque serveur du cluster a exactement la même importance et que chaque serveur peut fournir un service complet en écriture ou en lecture. Il est important d'aborder différents point cruciaux afin qu'une architecture de ce type puisse fonctionner correctement, tel que la redirection de requête aux bons serveurs, savoir quels serveurs composent le cluster, ainsi que les techniques pour distribuer au mieux les données sur les différentes machines du cluster.

5.1.2.1 Le protocole de bavardage (Gossip Protocole)

Le protocole de bavardage est bien connu dans le milieu du réseau, il est utilisé dans le but d'informer tout le monde dans un réseau sans recourir à un gros système centralisé. Le principe de ce protocole est simple : toutes les paires du réseau, à savoir les nœuds du cluster, transmettent l'information à d'autres paires choisies au hasard parmi les nœuds connus, qui refont le même processus. Il est essentiel que chaque nœud maintienne un historique (information envoyée, destinataire de l'information, etc.) pour

un fonctionnement optimal, afin de ne pas perdre de temps à retransmettre des messages au même nœud par exemple.

Les communications se font périodiquement afin de ne pas saturer le réseau par la communication des différents nœuds et impliquent un ralentissement au niveau de la distribution des données. Voici une petite illustration du fonctionnement de ce protocole

Figure 14

Protocole de bavardage

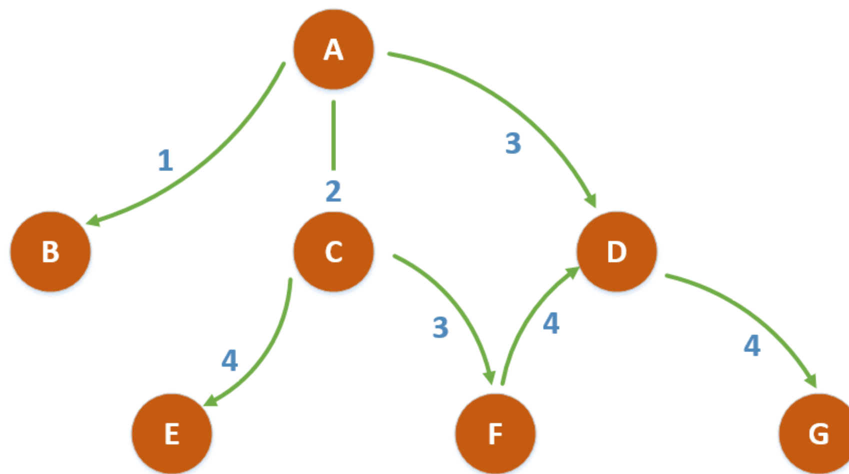


Figure-7: Gossip protocol

Une information a été mise à jour dans le nœud A. Parmi les autres nœuds dont le nœud A a connaissance (ici B,C,D) il va en choisir un de façon aléatoire afin de communiquer l'information, ici la communication numéro 1 avec le nœud B. Il est important de noter qu'une communication va dans les deux sens et que si le nœud B doit communiquer quelque chose au nœud A il le fera lors de cette communication. Ensuite chaque nœud en contactera un autre à intervalle régulier. Le A contacte C. Ensuite A contacte D et C contacte F. Ensuite C contacte E et F contacte D et D contacte G.

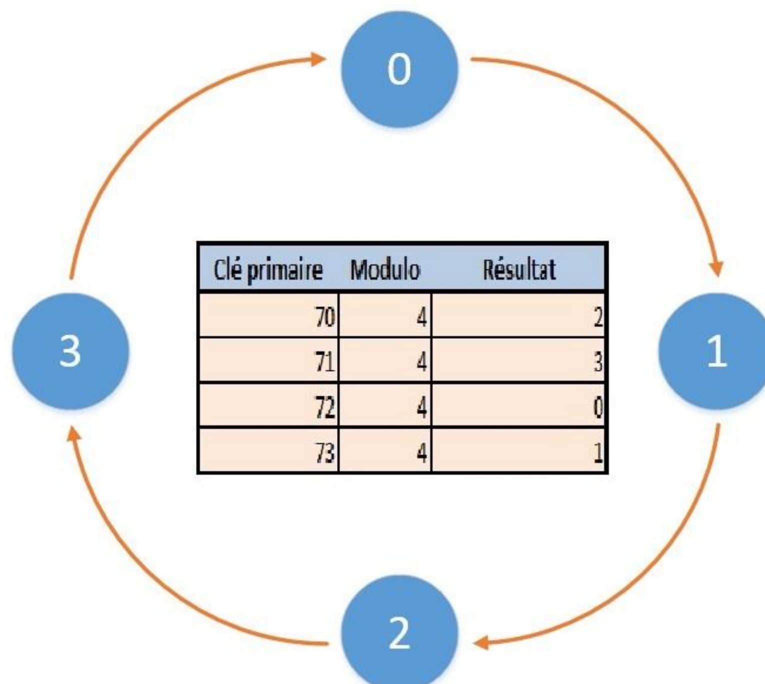
Ce protocole est aussi utilisé pour l'ajout d'un nouveau nœud dans un cluster, au-delà de l'échange des données. Après son ajout, le nœud publie sa présence et après plusieurs échanges, grâce à un historique des membres du cluster, tous les nœuds seront avertis. Il permet également l'échange d'informations de partitions entre nœuds, permettant ainsi à chaque nœud de savoir sur quel nœud il doit rediriger une requête pour laquelle il ne contient pas d'information.

La très grande tolérance aux pannes constitue un des points fort de ce protocole. En effet si un nœud tombe, l'information qu'il était censé retransmettre sera connue par un autre nœud. Prenons un exemple avec l'image ci-dessus : le nœud G reçoit l'information par le nœud D et si le nœud D devait tomber en panne nous pourrions imaginer une connexion entre le nœud F et le nœud G pour recevoir l'information. Dans un souci de lisibilité, les communications possibles entre les différents nœuds sur l'image ci-dessus ont été simplifiées et ne sont donc pas exhaustives.

5.1.2.2 Le hachage consistant

Le partitionnement de données constitue un point important dans une architecture distribuée décentralisée. Il faut en effet pouvoir distribuer les données de la meilleure façon possible entre les nœuds. L'algorithme de hachage consistant a été conçu pour répondre à des problèmes que l'on peut rencontrer lors de répartition des données dans une architecture évolutive. Afin de démontrer les problèmes ci-dessus, nous allons prendre l'exemple d'une distribution des données classique avec un hachage sur la clé primaire.

Figure 15
Distribution par hachage sur la clé primaire



Clé primaire	Modulo	Résultat
70	4	2
71	4	3
72	4	0
73	4	1

Voici une représentation simplifiée d'une répartition utilisant le hachage sur la clé primaire. Cette solution semble adaptée pour autant que la composition d'un cluster reste ainsi figée et n'évolue pas. Malheureusement la réalité des choses dans un

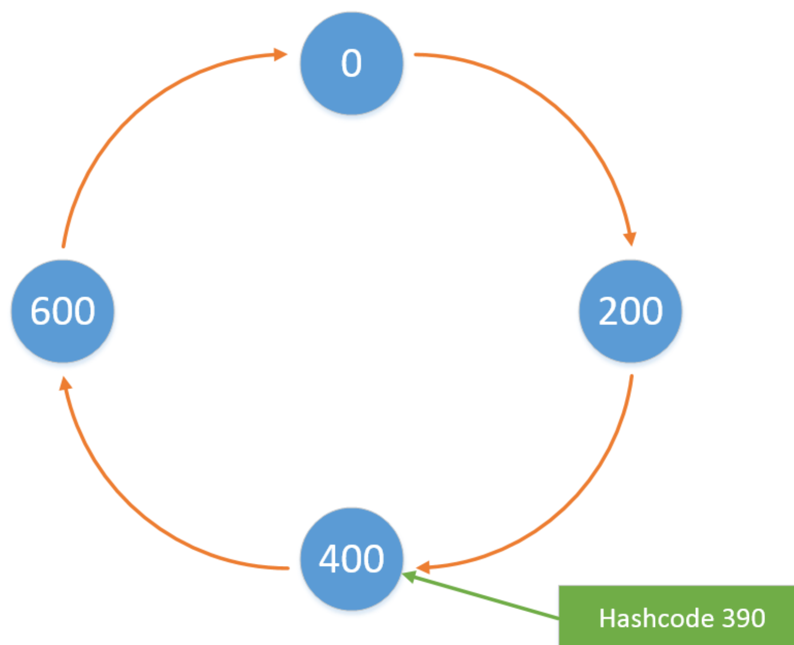
environnement distribué est souvent différente. Il arrive souvent que des nœuds soient ajoutés au cluster, que d'autres soient remplacé et d'autre supprimés. Que faire si un nœud venait s'ajouter à ce cluster de quatre nœuds ? Il faudrait redistribuer toutes les clés sur un modulo de cinq. Nous nous rendons très vite compte que ce processus n'est pas envisageable sur une taille d'architecture conséquente, telle que pourraient avoir besoin des sociétés comme Google. C'est justement là que réside le grand avantage du hachage consistant, qui permet l'ajout ou la suppression d'un nœud avec un dérangement minimal.

Cette solution a initialement été développée pour la mise en cache distribuée par David Ron Karger, professeur d'informatique au MIT. Elle a néanmoins été reprise par d'autres domaines dont, notamment, la répartition des données dans les systèmes distribués.

Le principe est le suivant : plusieurs nœuds pour un système distribué et une valeur de hachage définissant à chacun de ces nœuds la rangée de valeurs stockées attribuée. Après avoir obtenu la valeur de hachage à partir de sa clé, le système va chercher le nœud dont sa valeur est supérieure et y stocker la donnée. Voici une représentation d'un système distribuée en forme d'anneau et les nœuds positionnés sur cet anneau.

Figure 16

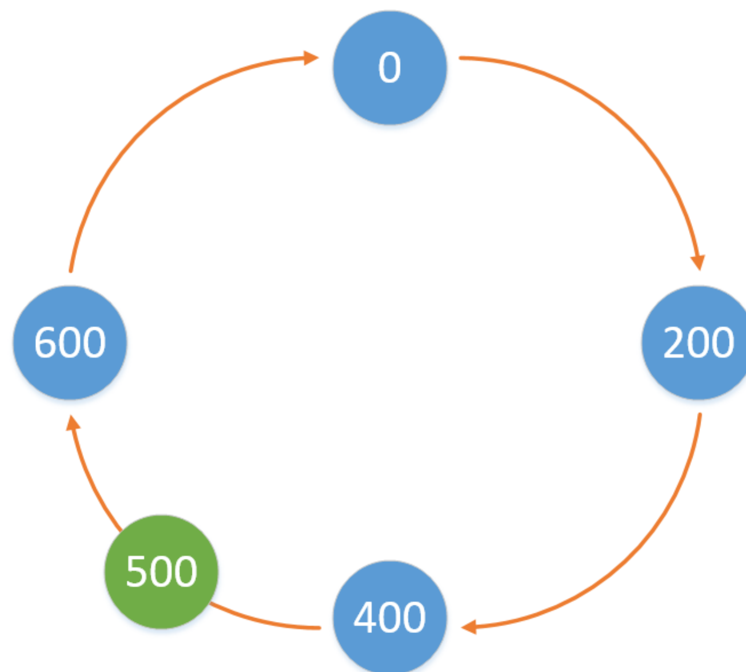
Répartition sur hachage consistant



Chaque nœud contient toutes les clés inférieures à sa valeur de hachage et supérieures à la valeur de hachage du nœud le précèdent.

Que va-t-il se passer si un nœud vient s'ajouter au système ? Il va d'abord venir se positionner sur le cluster. Pour ce faire, il va contacter son successeur et une fois entré en contact avec celui-ci, il va lui demander quel est son prédécesseur. Une fois déterminé qui était son successeur et son prédécesseur, il va leurs demander de l'insérer entre les deux, fractionnant la rangée de valeurs en deux. Une fois installé, il va récupérer une partie des clés de son successeur, comme le montre l'image ci-dessous.

Figure 17
Ajout d'un nouveau noeud



Ici le nouveau nœud récupèrera la rangée de 400 à 500. Si le nouveau nœud rajouté venait à être retiré, toutes ces clés seraient relocalisées sur le nœud succédant le nœud retiré et une information serait envoyée à son prédécesseur pour l'informer de son nouveau successeur. Le successeur lui, mettrait à jour la DHT. On peut effectivement voir que la redistribution des clés lors de l'ajout ou de la suppression d'un nœud n'affecte le système que de façon minimale.

5.1.2.3 Table de hachage distribuée

Comme déjà vu précédemment chaque nœud a le même niveau d'importance dans un système distribué sans maître et peut fournir un service complet. Il est donc essentiel que chaque nœud sache où se trouve l'information demandée par le client si celle-ci ne se trouve pas dans son propre serveur, afin de pouvoir rediriger la requête. De plus, la taille du système est modelable et des données peuvent changer de localisation (vu

précédemment) sur les nœuds. La solution à ces problèmes est donc le maintien d'une table de hachage sur chaque nœud. Une table de hachage distribuée est donc un grand annuaire consultable pour savoir où se trouve l'information recherchée. La table peut être entièrement maintenue sur chaque nœud, ou alors elle peut être partitionnée. Dans le premier cas, cette configuration permet un accès direct du client à la donnée. Dans le deuxième cas, il se peut que la donnée soit retrouvée après plusieurs sauts. En effet, un nœud qui contient toute la table de hachage peut directement rediriger le client au bon endroit, étant donné qu'il sait où se trouvent toutes les clés. Par contre, il peut arriver que le client demande une information sur un nœud ne contenant qu'une partie de la table de hachage et dont il ignore la localisation. Le nœud va donc demander à un autre nœud s'il ne saurait pas où se trouve l'information. Ce processus est appelé un accès à multiple « hops ». La table de hachage distribuée est échangée dans le système en utilisant le protocole de bavardage.

5.1.2.4 Réplication de données

5.2 MapReduce

Le traitement de données de façon distribuée soulève certaines questions, comment distribuer le travail entre les serveurs ? Comment synchroniser les différents résultats ? Comment gérer une panne d'une unité de traitement ? MapReduce répond à ces problèmes.

Il ne s'agit pas d'un élément de base de données, mais d'un modèle de programmation s'inspirant des langages fonctionnels et plus précisément du langage Lisp. Il permet de traiter une grande quantité de données de manière parallèle, en les distribuant sur divers nœuds d'un cluster. Ce mécanisme a été mis en avant par Google en 2004 et a connu un très grand succès auprès des sociétés utilisant des DataCenters telles que Facebook ou Amazon.

5.2.1 Principe de MapReduce

Le principe de MapReduce est simple: il s'agit de découper une tâche manipulant un gros volume de données en plusieurs tâches traitant chacune un sous-ensemble de ces données. MapReduce est vu en deux étapes. La première étape, appelé « map » consiste à dispatcher une tâche, ainsi que les données quelle traite en plusieurs sous-tâches traitant chacune d'elles un sous-ensemble de données. La deuxième étape se nomme « Reduce ». Dans cette partie il s'agit de récupérer les résultats des différents serveurs (à savoir les sous-tâches) et de les consolider. Nous reviendrons plus en détail sur le sujet plus bas. Bien que le principe soit relativement simple, il l'est surtout grâce à l'apport du modèle MapReduce simplifiant au maximum les complexités de

l'informatique distribuée. Il permet ainsi aux développeurs de se focaliser sur le traitement proprement dit. Le fait que MapReduce soit développé en java permet de s'abstraire de l'architecture matérielle pour qu'un framework MapReduce puisse tourner sur un ensemble de machines hétérogènes.

5.2.2 Programmation fonctionnelle

Avant d'approfondir sur le sujet il est important de faire un détour sur les orientations de MapReduce à la programmation fonctionnelle. Les langages fonctionnels sont très différents de la majorité des autres langages de programmation dits impératifs. Ces types de langages se basent sur le concept de machine d'état, où les différents traitements de données produisent des changements d'état. Un état est donc l'ensemble des valeurs présentes dans un système à un moment donné. Un langage manipule donc cet état à travers différentes instructions comme par exemple des affectations de valeur à des variables, le branchement conditionnel, le bouclage, etc.

Cependant ce type de langage pouvant affecter l'état d'un système peut produire des effets de bord sur d'autres sous-ensembles du système. Cela veut dire qu'une fonction peut modifier un état autre que sa valeur de retour, comme par exemple une variable statique ou globale. La situation devient problématique si plusieurs serveurs peuvent changer l'état général du système. Le langage fonctionnel répond à cette problématique en rejetant le changement d'état et la mutation de données, et souligne l'application des fonctions. La programmation de type fonctionnelle permet donc de gérer le traitement de données de manière sûre, car elle ne maintient qu'un état transitoire. En effet les résultats obtenus sont stockés de manière locale, et aucun état n'est maintenu hors du temps d'exécution de la fonction. Cela permet de diminuer les effets de bord et de gagner en souplesse lors des traitements.

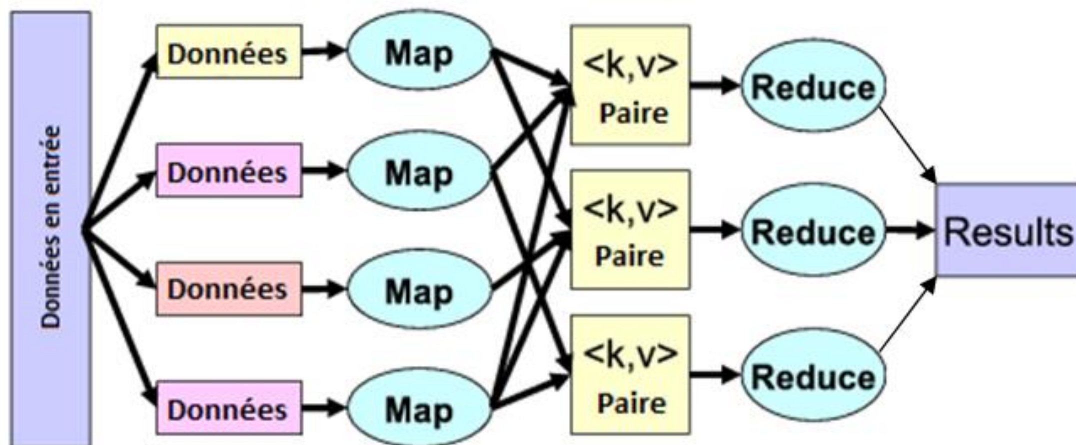
5.2.3 Fonctionnement de MapReduce

Comme dit précédemment, le modèle MapReduce consiste en deux étapes, représentées toutes deux par des fonctions. Dans la fonction map on prend en entrée un ensemble de « clé-valeurs », le nœud fait une analyse du problème et il va le séparer en sous-tâches, pour pouvoir les redistribuer sur les autres nœuds du cluster. Dans le cas nécessaire, les nœuds recevant les sous-tâches refont le même processus de manière récursive. Les sous-tâches des différents nœuds sont traitées chacune d'entre elles dans leur fonction map respective et vont retourner un résultat intermédiaire. La deuxième étape nommé « Reduce », consiste à faire remonter tous les résultats à leur nœud parents respectif. Les résultats se remonte donc du nœud le plus bas jusqu'à la racine. Avec la fonction Reduce, chaque nœud va calculer un résultat partiel en

associant toutes les valeurs correspondant à la même clé en un unique couple (clé – valeur). Une fois obtenu ce couple (clé-valeur), le résultat est remonté au nœud parent, qui va refaire le même processus de manière récursive jusqu'au nœud racine. Quand un nœud termine le traitement d'une tâche, un nouveau bloc de données est attribué à une tâche Map. Voici un schéma illustrant tout ça.

Figure 18

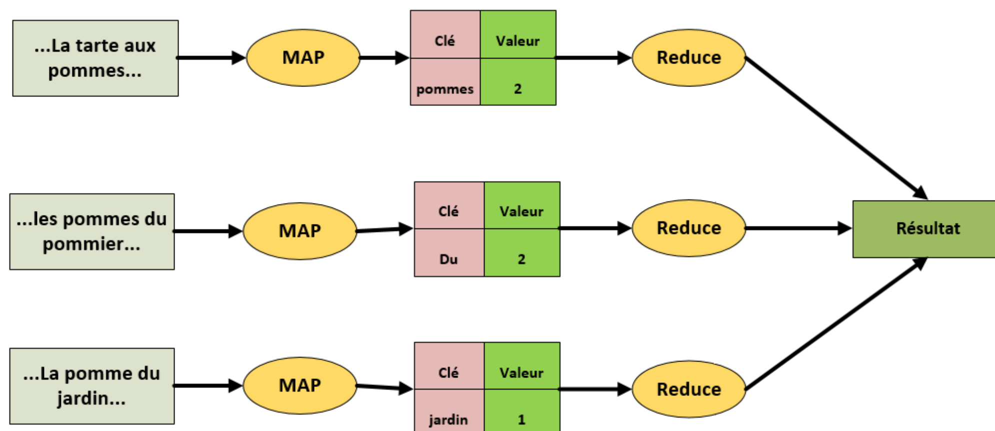
Schéma MapReduce



Voici un exemple d'utilisation. Le problème ici sera de compter le nombre d'occurrence de chaque mot. Comme données en entrée nous allons avoir des fichiers texte. La fonction Map aura pour but de décomposer le texte en couple de mots clé-valeur. La fonction Reduce va prendre les couples avec la même clé et compter la totalité des occurrences pour ne faire qu'une seul paire clé-valeur par mot. → [(m,[1,1,1,…])m2,[1,1,1,..],…]. L'illustration ne montre qu'une partie du processus et est incomplète.

Figure 19

Exemple comptage de mot



5.2.4 Les avantages

Voici une liste de quelques avantages qu'apporte MapReduce :

- Simplicité d'utilisation : l'utilisateur peut se focaliser sur son propre code et négliger l'aspect du traitement distribué, traité de manière totalement transparente.
- Polyvalence : il est adaptable à de nombreux type de traitement de données, comme par exemple les fouilles de données, calculs de taille de plusieurs milliers de documents.
- Sa faculté de décomposition d'un processus en plusieurs tâches distribuables, et cela sur une multitude de nœuds.

5.2.5 Les inconvénients

Voici une liste de quelques désavantages de MapReduce :

- Il n'y a qu'une seul entrée pour les données. Par conséquent, les algorithmes qui nécessitent plusieurs éléments en entrée sont mal supportés, MapReduce étant prévu pour lire un seul élément en entrée et de produire un élément en sortie.
- La tolérance aux pannes ainsi que la scalabilité horizontale, font que les opérations du MapReduce ne sont pas toujours optimisées pour les entrées/sorties. De plus le flux de données en deux étapes le rend très rigide, car pour passer à l'étape suivante, l'étape courante doit être terminée. Tous ces problèmes réduisent donc la performance.
- Ne supporte pas les langages de haut niveau tel que le SQL.

5.2.6 Hadoop

Le modèle de programmation MapReduce est implémenté dans la plupart des moteurs NoSQL, comme par exemple CouchDB, MongoDB et Riak pour n'en citer que quelques un. Il est difficile de parler de MapReduce sans parler de Hadoop qui est aujourd'hui le framework le plus populaire.

Hadoop est donc un framework open source développé en java, faisant partie des projets de la fondation de logiciel Apache depuis 2009. Il est destiné à faciliter le développement d'applications distribuées et scalables, permettant la gestion de milliers de nœuds ainsi que des pétaoctets de données.

Doug Cutting l'un des fondateurs de Hadoop, travaillant à l'époque sur le développement de Apache Lucene, cherchait une solution quant à la distribution du traitement de Lucene afin de bâtir le moteur d'indexation web Nutch. Il décidait donc de s'inspirer de la publication de Google sur leur système de fichier distribué GFS (Google File System). Premièrement, renommer NDFS, il sera rebaptisé HDFS pour HadoopDistributedFileSystem. Hadoop est aujourd'hui l'un des outils les plus pertinents pour répondre aux problèmes du Big Data.

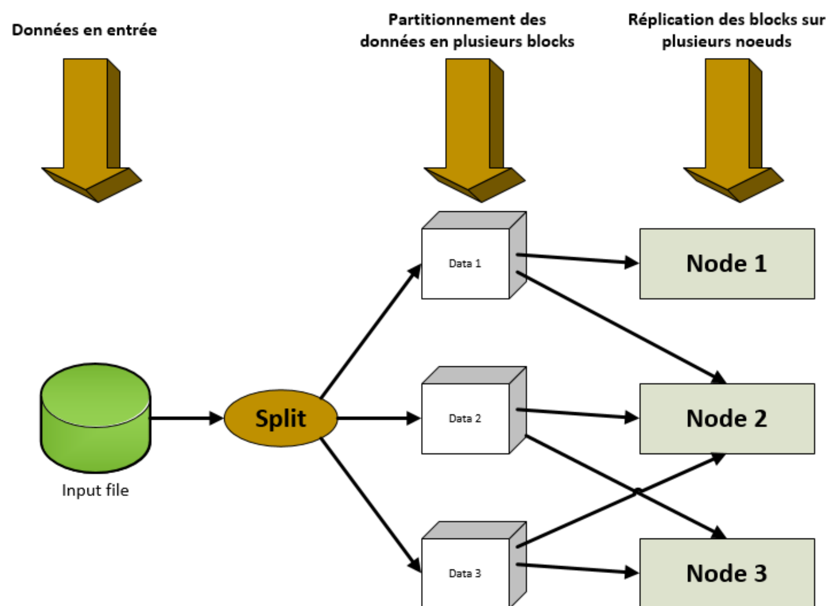
Hadoop s'inspire de deux produits, le premier est le « Google FileSystem », un système de fichiers distribués. Le deuxième est bien évidemment le modèle de Programmation MapReduce, qui, en 2004, avait été mis en avant par la firme de MountainView dans l'une de leurs publications.

5.2.6.1 Hadoop Distributed FileSystem (HDFS)

Dans Hadoop, les différents types de données, qu'elles soient structurées ou non, sont stockées à l'aide du HDFS. Le HDFS va prendre les données en entrée et va ensuite les partitionner en plusieurs blocs de données. Afin de garantir une disponibilité des données en cas de panne d'un nœud, le système fera un réplica des données. Par défaut les données sont répliquées sur trois nœuds différents, deux sur le même support et un sur un support différent. Les différents nœuds de données peuvent communiquer entre eux pour rééquilibrer les données.

Figure 20

Hadoop Distributed FileSystem



5.2.6.1.1 Typologie d'un cluster Hadoop

Hadoop repose sur un schéma dit « maître-esclave » et peut être décomposé en cinq éléments.

Le nom du nœud (Name Node) : Le « Name Node » est la pièce centrale dans le HDFS, il maintient une arborescence de tous les fichiers du système et gère l'espace de nommage. Il centralise la localisation des blocs de données répartis sur le système. Sans le « Name Node » , les données peuvent être considérées comme perdues car il s'occupe de reconstituer un fichier à partir des différents blocs répartis dans les différents « Data Node ». Il n'y a qu'un « Name Node » par cluster HDFS.

Le gestionnaire de tâches (Job Tracker) : Il s'occupe de la coordination des tâches sur les différents clusters. Il attribue les fonctions de MapReduce aux différents « Task Trackers ». Le « Job Tracker » est un « Daemon » cohabitant avec le « Name Node » et ne possède donc qu'une instance par cluster.

Le moniteur de tâches (Task tracker) : Il permet l'exécution des ordres de mapReduce, ainsi que la lecture des blocs de données en accédant aux différents « Data Nodes ». Par ailleurs, le « Task Tracker » notifie de façon périodique au « Job Tracker » le niveau de progression des tâches qu'il exécute, ou alors d'éventuelles erreurs pour que celui-ci puisse reprogrammer et assigner une nouvelle tâche. Un « Task Tracker »

est un « Deamon » cohabitant avec un « Data Node », il y a un donc un « Task Tracker » par « Data Node ».

Le nœud secondaire (Secondary node) : N'étant initialement pas présent dans l'architecture Hadoop, celui-ci a été ajouté par la suite afin de répondre au problème du point individuel de défaillance (SPOF- Single point of failure). Le « Secondary Node » va donc périodiquement faire une copie des données du « Name Node » afin de pouvoir prendre la relève en cas de panne de ce dernier.

Le nœud de données (Data Node) : Il permet le stockage des blocs de données. Il communique périodiquement au « Name Node » une liste des blocs qu'il gère. Un HDFS contient plusieurs nœuds de données ainsi que des réplications d'entre eux. Ce sont les nœuds esclaves.

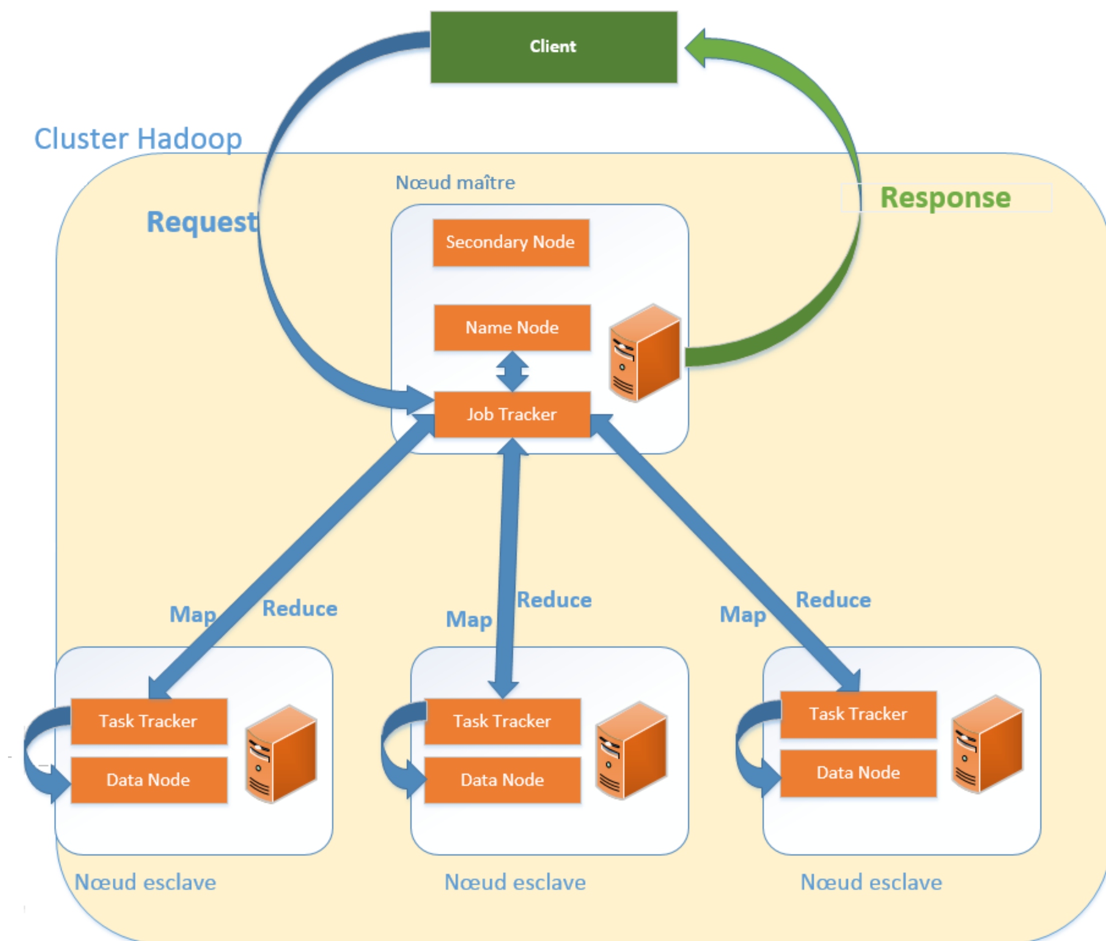
Figure 21
Architecture d'un cluster Hadoop

Un cluster Hadoop peut-être constitué de machines hétérogènes, que ce soit au niveau du hardware comme au niveau software (système d'exploitation). Cependant il est bien plus simple d'administrer un cluster de type homogène.

5.2.6.2 Implémentation de MapReduce dans Hadoop

Figure 22

Fonctionnement de MapReduce dans Hadoop



Voici comment fonctionne le processus MapReduce dans le framework Hadoop. Pour commencer, le « Job Tracker » va recevoir une requête cliente « Map », il va alors demander au « Name Node » quelles sont les informations nécessaires ainsi que leur localisation dans le cluster pour répondre à la requête du client. Une fois la réponse reçue, le « Job Tracker » enverra la fonction « map » aux différents « Task Trackers » . Les « Task Trackers » vont alors chercher les données correspondant au traitement sur leur « Data Nodes ». Une fois que les fonctions « map » ont terminés leur traitement, les résultats de ceux-ci sont stockés. Il est bon de noter que les données sont compilées au niveau de chaque « Data Nodes », et non pas de manière centralisée. C'est ce qui fait la caractéristique principale de Hadoop.

Une fois que la partie « Map » est terminée, c'est la partie Reduce qui commence à faire remonter les différents résultats et les consolider en un seul résultat final (voir 5.2.3 fonctionnement de MapReduce) pour répondre à la requête du client.

5.3 La gestion de la cohérence des données

Nous avons précédemment vu (dans le chapitre 3.3 cohérence contre disponibilité) que le théorème de CAP démontre le fait que dans un environnement distribué, il était impossible de maintenir les trois contraintes (qui sont la cohérence, la disponibilité et la résistance au morcellement) en même temps. Nous avons noté que les bases de types NoSQL privilégiaient la disponibilité au détriment de la cohérence des données. Une telle affirmation pourrait cependant induire les personnes en erreur, pouvant leur faire croire qu'il n y a aucun traitement de la cohérence dans les moteurs NoSQL, ce qui n'est pas tout à fait vrai. En effet, pour répondre à la forte cohérence des SGDBR, les moteurs NoSQL appliquent la cohérence finale (eventual consistency).

La cohérence finale est donc un modèle de programmation qui affirme que pour la mise à jour d'une donnée, au bout d'un certain temps, celle-ci aura été mise à jour sur tous ses réplicas, on dit alors que le système a été convergé. Lors d'une mise à jour d'une information dans une base NoSQL, la donnée n'est pas totalement verrouillée et peut être accessible en lecture. Tout ceci n'est pas sans conséquence et de nombreux cas d'incohérence se créent. C'est donc pour cela que des outils de réconciliation ont été mis en place.

5.3.1 Anti – Entropie

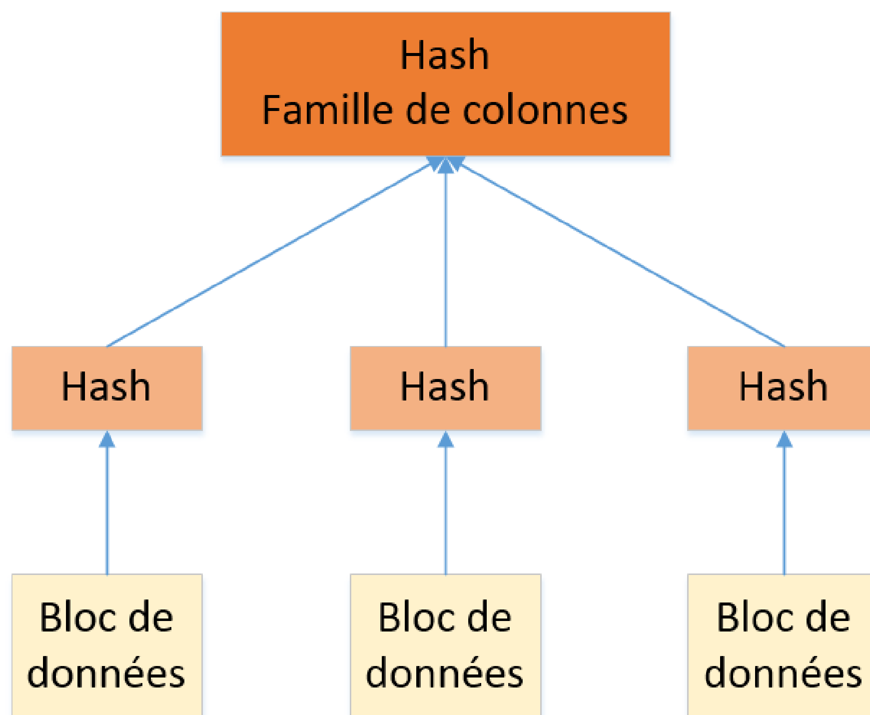
Le processus qui consiste à concilier les différences entre plusieurs réplicas de données distribuées s'appelle anti-entropie. Entropie signifiant un état de désordre, anti-entropie pourrait se traduire comme un « anti-désordre ». L'anti-entropie se base sur deux principes ;

Le premier est le HashTree qui est un mécanisme de comparaison de volume de données. Il sera utilisé dans ce contexte pour comparer deux bases de données qui sont supposées être de parfaits réplicas. En comparaison avec d'autres algorithmes de hachage, l'avantage du HashTree est que son processus est beaucoup moins long et couteux. Au lieu de comparer la valeur de hachage ligne par ligne, le HashTree va calculer le hachage d'une table tout entière et comparer sur cette unique valeur. Si les deux valeurs sont égales, cela signifie que les tables sont identiques. Dans le cas contraire, cela veut dire qu'elles ne sont pas identiques et que l'algorithme va descendre à un niveau plus détaillé en créant des hachages sur des critères plus fins et ainsi répéter le même processus jusqu'à trouver l'élément différenciateur. Nous pouvons tout de suite

nous rendre compte du gain de temps et de traitement de donnée que le HashTree apporte dans un contexte de « Big Data ». Voici une représentation du HashTree schématisé pour un système Cassandra :

Figure 23

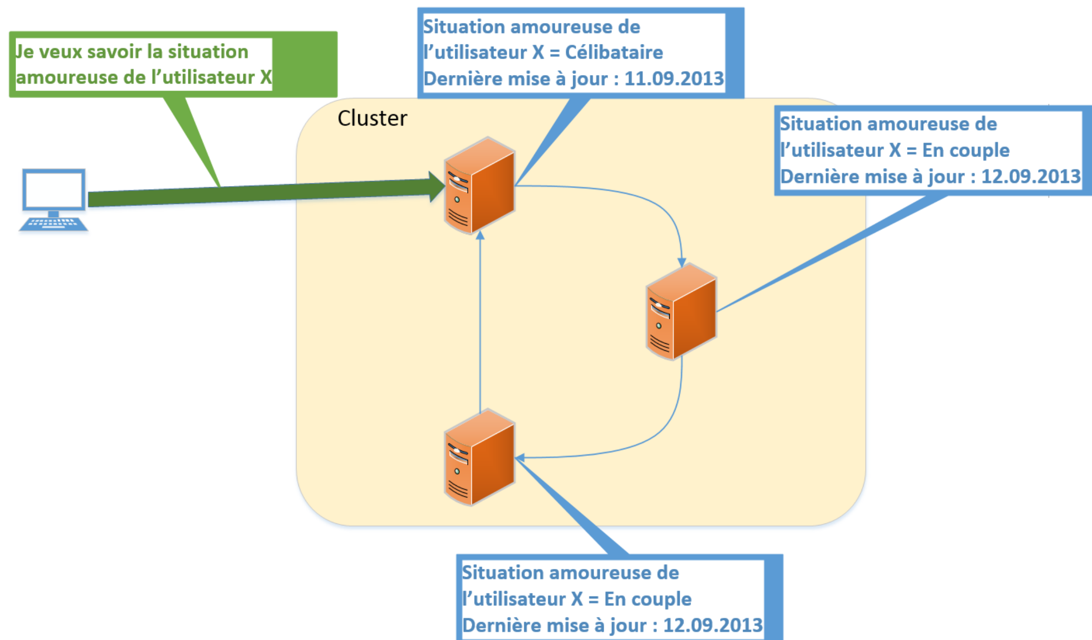
Représentation du HashTree sur Cassandra



Le deuxième principe est celui de « la gestion des conflits ». Dans un système distribué où l'on peut mettre à jour la même donnée se trouvant sur des serveurs différents au même moment, une gestion des conflits est essentielle. En effet, comment pourrait-on savoir si l'information provenant du serveur que l'on a interrogé est la dernière mise à jour ?

Figure 24

Représentation de conflit



Il existe diverses solutions implémentées par les moteurs NoSQL pour la gestion de conflit. Les techniques les plus utilisées sont les « timestamps » et les « vectorclocks ». L'illustration ci-dessus représente la gestion du conflit dans un système tel que Cassandra où chaque valeur possède un « timestamps ». Un timestamps est donc la date et l'heure à laquelle l'information a été mise à jour. Cassandra définit plusieurs niveaux de consistance, et de façon générale, lors de la lecture (pour ne parler que de cet exemple), un des niveaux consiste à lire la donnée sur plusieurs nœuds afin de comparer les « timestamps » pour choisir l'information la plus à jour.

6. Les avantages / désavantages du NoSQL

SQL et NoSql ont chacun son lot d'avantages et de désavantages ; aucune des deux solutions n'est donc meilleure que l'autre. En fonction des types de besoins que peut avoir une entreprise par exemple, une solution SQL pourrait être plus avantageuse qu'une solution NoSql, et vice-versa. Notons que NoSQL signifie « Not Only Sql ».

6.1 Les avantages

6.1.1 Plus évolutif

NoSQL est plus évolutif. C'est en effet l'élasticité de ses bases de données NoSQL qui le rend si bien adapté au traitement de gros volumes de données. Au contraire, les bases de données relationnelles ont souvent tendance à utiliser la scalabilté verticale, qui

consiste à augmenter les capacités du serveur (plus d'espace, ajout de RAM, augmentation de la puissance du processeur) quand celui-ci atteint ses limites. Mise à part le coût élevé que peut engendrer ce genre de processus, il n'est tout simplement pas viable dans un contexte de BigData. Il est donc de loin préférable d'adopter une solution NoSQL utilisant la scalabilité horizontale, dont le principe est d'ajouter des serveurs en parallèle afin de mieux répartir la charge. La scalabilité horizontale offre d'autres avantages non négligeables, comme une grande tolérance aux pannes ou les coûts réduits relatifs à l'achat du matériel (plus besoin d'acheter de serveurs extrêmement puissants).

6.1.2 Plus flexible

N'étant pas enfermée dans un seul et unique modèle de données, une base de données NoSQL est beaucoup moins restreinte qu'une base SQL. Les applications NoSQL peuvent donc stocker des données sous n'importe quel format ou structure, et changer de format en production. En fin de compte, cela équivaut à un gain de temps considérable et à une meilleure fiabilité. Il va sans dire qu'une base de données relationnelle doit être gérée attentivement. Un changement, aussi mineur soit-il, peut entraîner un ralentissement ou un arrêt du service.

6.1.3 Plus économique

Les serveurs destinés aux bases de données NoSQL sont généralement bon marché et de faible qualité, contrairement à ceux qui sont utilisés par les bases relationnelles. De plus, la très grande majorité des solutions NoSQL sont open–source, ce qui reflète d'une part une économie importante sur le prix des licences, mais aussi une vitesse de développement du produit beaucoup plus grande.

6.1.4 Plus simple

Les bases de données NoSQL ne sont pas forcément moins complexes que les bases relationnelles, mais elles sont beaucoup plus simples à déployer. La façon dont elles ont été conçues (réparation automatique, données distribuées, simplicité des modèles de données) permet une gestion beaucoup plus légère. Toutefois, la présence de DBA (Database administrator) dans un environnement NoSQL est quand même nécessaire (pour le moment), ne serait-ce que pour la gestion de la performance et pour la disponibilité d'une banque de données critique.

6.1.5 Le Cloud Computing

NoSQL et le cloud s'associent de façon naturelle. En effet, le cloud computing répond extrêmement bien aux besoins en matière de scalabilité horizontale que requièrent les bases de données NoSQL. De plus, la facilité de déploiement et de gestion de ces bases

en fait un partenaire de choix pour le cloud computing, permettant aux administrateurs de se concentrer davantage sur l'aspect logiciel sans devoir se soucier du matériel qu'ils utilisent. Le service d'Amazon « DynamoDb » en est l'exemple parfait.

6.2 Les désavantages

6.2.1 Fonctionnalités réduites

Les bases de données NoSQL sont principalement conçues pour stocker des données et offrent très peu de fonctionnalités autres que celle-là. Lorsque les transactions entrent dans l'équation, les bases de données relationnelles restent le meilleur choix. Voilà pourquoi les bases NoSQL ne pourront jamais remplacer entièrement SQL (ce n'est pas le but d'ailleurs).

6.2.2 Normalisation et Open Source

Etrangement, le critère d'open source pour les bases de données NoSQL est à la fois sa plus grande force et sa plus grande faiblesse. La raison est qu'il n'existe pas encore de normalisation fiable pour NoSQL

6.2.3 Performances et évolutivité au détriment de la cohérence

En raison de la façon dont les données sont gérées et stockées dans ces bases, la cohérence des données pourrait bien être une préoccupation. Comme déjà vu précédemment, les bases de données NoSQL font l'impasse sur les propriétés dites ACID afin de mieux répondre aux besoins de performances et d'évolutivité. La cohérence des données est donc un facteur moins important. Selon le besoin, ces caractéristiques peuvent être un sérieux atout tout comme un paramètre irrecevable. S'il faut faire face à de très grosses montées en charge de très gros volumes de données, NoSQL est le système adéquat. Si on traite des données sensibles (transactions bancaires ou autres), la cohérence des données est indispensable.

6.2.4 Manque général de maturité

Bien que les bases de données NoSQL soient présentes depuis longtemps, leurs technologies sont encore immatures par rapport à celles des bases relationnelles. Cela se traduit également par un manque d'administrateurs et de développeurs ayant les compétences dans ce système. Les bases de données ont beau être « administrator-friendly » (simples à administrer), cela n'a pas de sens si les administrateurs en question ne savent pas comment les utiliser. A l'heure actuelle, les bases de données relationnelles sont beaucoup mieux implémentées dans les entreprises. Elles disposent d'un nombre plus grand de fonctionnalités et de professionnels qui comprennent comment les gérer.

6.3 Conclusion

Etant donné que, ces dernières années, les bases de données NoSQL ont gagné en popularité avec l'émergence du BigData et qu'elles ont pu venir à bout de problèmes que les bases de données relationnelles étaient incapables de résoudre, elles restent néanmoins limitées dans d'autres domaines importants. Un administrateur doit par conséquent examiner soigneusement le type de bases de données le mieux adapté à ses besoins, car un mauvais choix pourrait avoir de très lourdes conséquences.

7. Benchmark

La deuxième partie de ma recherche consiste en un rapport d'un travail pratique, le but est de reprendre un script de base de données SQL et de le réécrire sur Cassandra, qui est un moteur NoSQL de type colonne.

7.1 Présentation du sujet

Lors de mes études à la HEG j'ai dû développer, dans le cadre d'un cours de génie-logiciel, un mini-programme permettant la gestion d'un vidéo club : Easy Location. Ce projet était divisé en deux étapes. La première partie consistait en la mise en place de la base de données, ce qui comprend la réalisation des modèles MCD,MLD, et MPD, ainsi que celle des différents scripts (script de création de base de données, création d'utilisateurs, création des tables, etc.). La deuxième partie était le développement du software en langage .Net.

Dans le cadre de la présente recherche, mon travail consiste donc à reprendre la partie BDD de ce projet et de la traduire sur Cassandra. L'intérêt d'une base de données NoSQL repose sur le fait qu'elle soit implémentée sur plusieurs machines afin de répartir la charge de travail. Malheureusement, l'expérimentation n'aura pas lieu en raison d'un manque de moyens matériels et d'un volume insuffisant de données à traiter.

7.2 Le modèle de données Cassandra

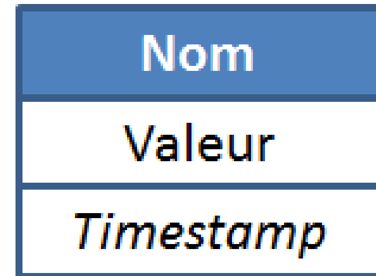
Afin de mieux comprendre la mise en place de la base de données, il est important de nous attarder quelques instants sur le modèle de données utilisé par Cassandra, c'est-à-dire sur les différents éléments qui composent cet outil.

7.2.1 La colonne

La colonne est la plus petite valeur possible dans un système Cassandra. C'est un triplet composé du nom de la colonne, de la valeur qu'elle possède, et de son timestamp. Ce dernier indique l'heure et la date à laquelle l'information a été enregistrée. Il est utilisé par le système pour savoir quelle est l'information la plus actuelle. Une colonne peut contenir plusieurs valeurs, comme dans une collection de chaînes de caractères. Elle peut aussi ne pas en avoir du tout.

Figure 25

Description d'une colonne

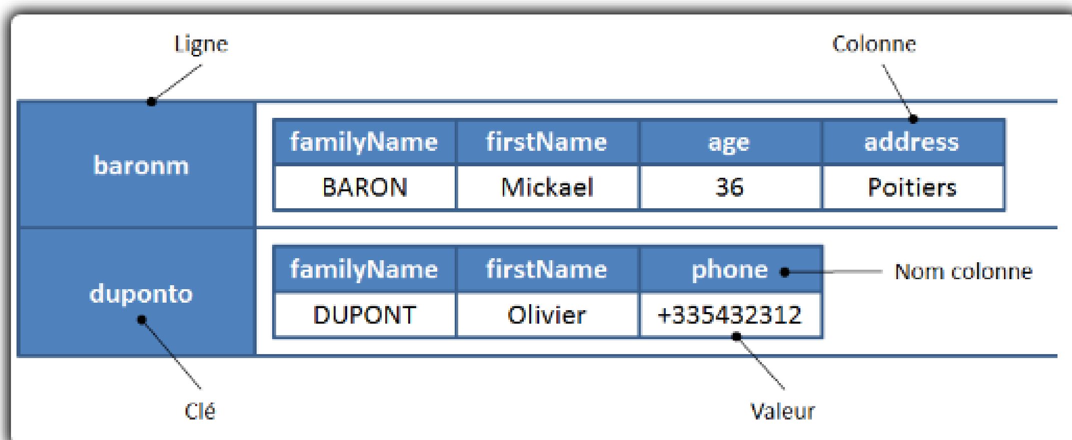


7.2.2 La ligne

Une ligne est un enregistrement. Elle est composée d'un ensemble de colonnes. Chaque ligne est identifiée par une clé, qu'on appellera « rowKey ». Il est possible d'utiliser des colonnes comme clé primaire d'une ligne. Une ligne peut contenir jusqu'à deux milliards de colonnes.

Figure 26

Description d'une ligne



Dans l'illustration ci-dessus, on remarque que la deuxième ligne ne contient pas autant de colonnes que la première et que les noms de colonnes sont différents. Cela s'explique par le fait que le schéma d'une base de données orientée colonne est dynamique, que ce soit par rapport au nombre de lignes ou par rapport au nombre de colonnes (voir le chapitre 3.4.2 Les bases de données orientées colonne).

7.2.3 Famille de colonnes

Une famille de colonnes est un regroupement logique de lignes (d'enregistrements). Dans le monde relationnel, elle est comparable à une table.

Figure 27

Description d'une famille de colonnes

Persons

baronm	familyName	firstName	age	address
	BARON	Mickael	36	Poitiers

duponto	familyName	firstName	phone
	DUPONT	Olivier	+335432312

Lorsqu'on crée une famille de colonnes, il est possible d'y définir des informations concernant les métadonnées des colonnes, c'est-à-dire d'y définir les différents attributs d'une table. Or, c'est au moment de l'enregistrement d'une ligne que l'on décide des colonnes qui seront exploitées. On distingue deux types de familles de colonnes : la famille de colonnes statique, où les colonnes sont définies lors de la création ou la modification de celle-ci, et la famille de colonnes dynamique, où les colonnes sont définies lors de la création ou la modification d'une ligne.

7.2.4 Le keyspace

Le keyspace est un regroupement des différentes familles de colonnes. On peut le considérer comme un schéma de base de données.

7.2.5 Les différents types d'attributs

Tout comme SQL, Cassandra propose plusieurs types d'attributs. Voici un tableau présentant les types en question. La première partie du tableau contient les types sous leur format natif, tandis que la seconde partie montre leur type correspondant en format CQL.

Figure 28

Différents types proposés par Cassandra

Type Interne	Nom CQL
BytesType	blob
AsciiType	ascii
UTF8Type	text, varchar
IntegerType	varint
Int32Type	int
LongType	bigint
UUIDType	uuid
TimeUUIDType	timeuuid
DateType	timestamp
BooleanType	boolean
FloatType	float
DoubleType	double
DecimalType	decimal
CounterColumnType	counter

7.3 Mise en place de la base de données EasyLocation sur Cassandra

Dans ce chapitre, je vais aborder les différentes étapes de la création de la base de données Easy Location sur le système Cassandra. Il convient de noter que ces étapes ne sont pas toutes reproductibles entre un environnement Oracle et Cassandra. C'est la raison pour laquelle je vais uniquement me consacrer aux parties qui peuvent l'être.

Bien qu'historiquement Cassandra-CLI ait été le premier outil à pouvoir manipuler une base Cassandra, je vais utiliser CQL Shell, qui est non seulement plus commun, mais aussi très similaire à SQL dans sa syntaxe. CQL Shell est apparu suite à l'arrivée du langage CQL.

7.3.1 Création de la base de données

La création du keyspace est l'une des premières étapes par lesquelles il faut passer pour mettre sur pied une base de données sur Cassandra. Pour rappel, le keyspace est le schéma de la base de données. C'est dans ce schéma-là que je vais créer toutes les familles de colonnes nécessaires à l'application.

*CREATE KEYSPACE **Easy_Location** WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 3};*

Grâce à cette commande, j'ai créé le keyspace portant le nom de « Easy_Location ». Le paramètre « replication » est obligatoire et sert à définir les stratégies de réplication de la base entre les différents nœuds du cluster. Or, il n'est pas nécessaire dans le contexte du projet, étant donné que la base ne se trouve que sur une seule machine. C'est

pourquoi je ne vais pas me pencher non plus sur les différentes stratégies de réplication. Dans le cadre de cette recherche, je me contenterai donc d'appliquer la stratégie par défaut.

7.3.2 Gestion des utilisateurs

Dans le système Cassandra, la gestion des droits d'utilisateurs correspond à l'authentification et à l'autorisation internes. L'authentification interne consiste en la gestion des noms d'utilisateurs et de leur mot de passe par Cassandra. Quant à l'autorisation interne, elle gère les différents droits accordés à un utilisateur sur l'ensemble de la base de données.

Afin que le système d'authentification et d'autorisation soit actif, il faut changer quelques paramètres dans un fichier de configuration. En effet, sur Cassandra, le système d'authentification et d'autorisation est par défaut désactivé. Pour l'activer, il faut modifier les deux lignes suivantes dans le fichier « cassandra.yaml » :

```
# authentication backend, implementing IAuthenticator; used to identify users
#authenticator: org.apache.cassandra.auth.AllowAllAuthenticator
authenticator: org.apache.cassandra.auth.PasswordAuthenticator
# authorization backend, implementing IAuthorizer; used to limit access/provide
permissions
#authorizer: org.apache.cassandra.auth.AllowAllAuthorizer
authorizer: org.apache.cassandra.auth.CassandraAuthorizer
```

Ensuite, pour éviter de devoir s'identifier à chaque fois avec le client « CQLSH », je vais créer un fichier, que je nommerai « .cqlshrc » et dans lequel je vais saisir les informations suivantes :

```
[authentication]
username = cassandra
password = cassandra
```

Il convient de noter que « cassandra » est le « superuser » par défaut. Une fois le fichier « .cqlshrc » créé, il faut le sauvegarder dans le répertoire racine de Cassandra et redémarrer le serveur. Il est bon de savoir que le système d'authentification géré par Cassandra n'est pas l'unique possibilité. Des systèmes externes, comme un annuaire LDAP, peuvent aussi être utilisés. Ici, l'authentification et l'autorisation sont gérées par le système Cassandra lui-même ; c'est la raison pour laquelle on parle d'authentification et d'autorisation internes.

Passons maintenant à la création des différents utilisateurs d'Easy Location. Il existe deux types de profils utilisateur prédéfinis dans Cassandra : le « superuser », qui

possède tous les droits, et le « user » à qui il faudra attribuer les droits. Dans un premier temps, il faut créer les deux utilisateurs de l'application, qui sont :

- **User_user :** simple utilisateur soumis à certaines restrictions sur les fonctionnalités du logiciel.
- **User_admin :** utilisateur ayant le rôle d'administrateur. Il a donc accès à toutes les fonctionnalités du logiciel.

Il faut également créer un profil utilisateur pour le développement :

- **User_adminBdd :** profil auquel le développeur en charge de l'application peut se connecter. Il possède absolument tous les droits possibles au sein de la base de données.

Afin de rester cohérent avec le modèle de données qui va être présenté, nous ne verrons ci-après que quelques droits que nous allons octroyer aux utilisateurs. La totalité du script sera fournie en annexe. Je procéderai dans la manière suivante pour chaque utilisateur ; création de l'utilisateur dans un premier temps et attribution des droits dans un deuxième temps.

Création User_user :

*CREATE USER **User_user** WITH PASSWORD '**user'** NOSUPERUSER;*

Attribution des droits:

*GRANT SELECT ON TABLE **eas_client** TO **User_user**;*
*GRANT SELECT ON TABLE_**eas_clientLocations** TO **User_user**;*
…
*GRANT MODIFY ON TABLE **eas_client** TO **User_user**;*
…

Création User_admin;

*CREATE USER **User_admin** WITH PASSWORD '**admin**' NOSUPERUSER;*

Attribution des droits:

*GRANT SELECT ON KEYSPACE **easy_location** TO **User_admin**;*
*GRANT MODIFY ON KEYSPACE **easy_location** TO **User_admin**;*

Création User_adminBdd;

*CREATE USER **User_adminBdd** WITH PASSWORD '**adminBdd'** SUPERUSER;*

On peut remarquer deux choses qui se différencient des commandes SQL : la permission MODIFY, qui englobe les permissions « INSERT », « DELETE » et « UPDATE », et les commandes « NOSUPERUSER/SUPERUSER ».

Le « SUPERUSER » détient le plein pouvoir de la base de données. C'est l'équivalent de l'administrateur d'un système. Par conséquent, si on attribue le rôle « SUPERUSER » à l'utilisateur « User_adminBdd », il n'est plus nécessaire de lui octroyer d'autres permissions.

7.3.3 La modélisation par la dénormalisation

Concernant la modélisation de la base de données, je me suis posé la question suivante : comment modéliser la base de données dans un contexte NoSQL ? En effet, il n'est pas possible de traiter la partie modélisation NoSQL de la même façon qu'on le fait dans les bases de données relationnelles. Dans ces dernières, la modélisation s'effectue au moyen d'un ensemble de tables qui peuvent se joindre entres elles grâce aux clés primaires et aux clés étrangères afin de répondre aux différentes requêtes et, par la même occasion, de supprimer la redondance de données. C'est ce qu'on appelle la modélisation par la normalisation. Dans une base de données NoSQL orientée colonne comme Cassandra, qui rejette la notion de jointure, cette approche n'est tout simplement pas possible. Il faut donc concevoir un modèle de données en fonction des requêtes qui vont solliciter la base de données. Dans ce type de modèle, les données sont la plupart du temps dupliquées sur plusieurs familles de colonnes (tables en SQL) afin d'optimiser et de simplifier le traitement des requêtes. C'est ce qu'on appelle la dénormalisation. La modélisation de la base de données EasyLocation nous permettra d'y voir un peu plus clair sur ces deux façons de modéliser : la normalisation et la dénormalisation.

7.3.4 Modèle de données « Easy_location »

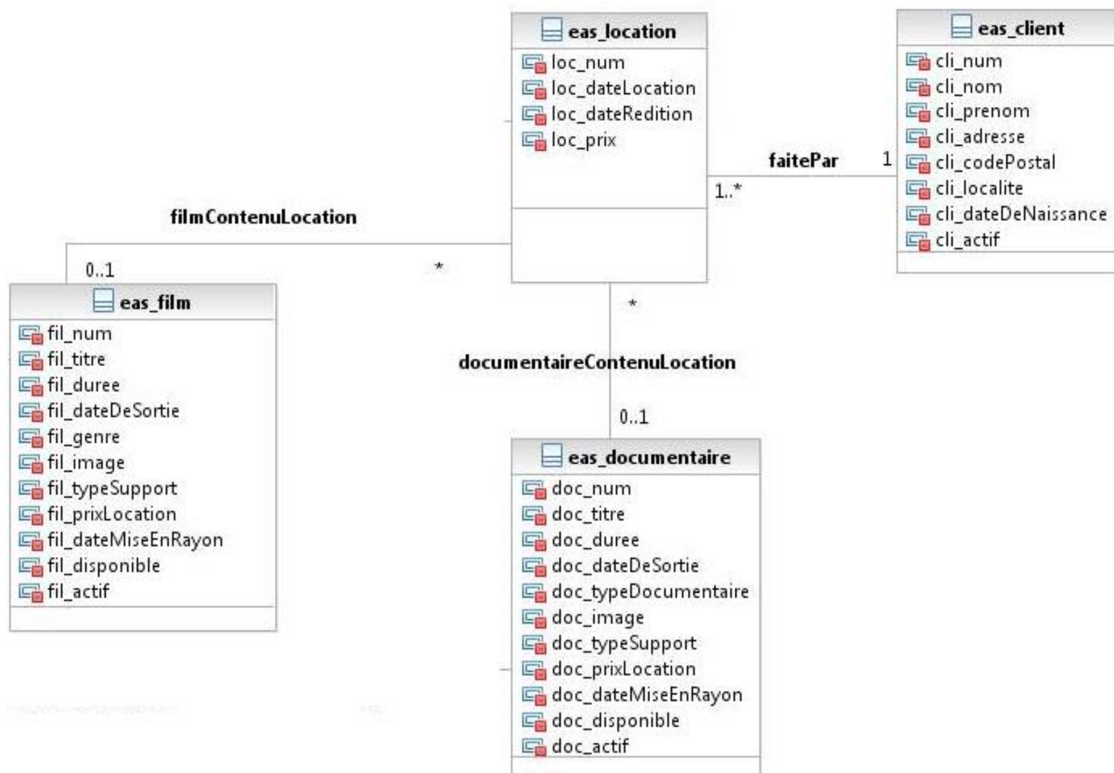
Afin de ne pas surcharger le chapitre, nous allons aborder ici qu'une petite partie du schéma de données pour démontrer en quoi consiste la dénormalisation. La totalité de la modélisation est disponible en annexe.

Dans ce contexte non exhaustif, l'utilisateur peut, grâce au logiciel « Easy Location », consulter le profil d'un membre du vidéo club (nom, prénom, adresse, locations effectuées, locations en cours, etc.). Il peut aussi lister toutes les vidéos du magasin. Les vidéos, quant à elles, se divisent en deux catégories : les films et les documentaires.

Voici le modèle de données qui permet de répondre à ces différentes requêtes dans une base de données relationnelle.

Figure 29

Modèle conceptuel de données



Si l'utilisateur souhaite voir les différents clients du vidéo club, une simple requête sur la table « eas_client » permet d'y répondre. S'il souhaite consulter l'historique de location d'un client particulier, deux requêtes avec une jointure de trois tables sont nécessaires. Une première requête, qui joint les tables « eas_client », « eas_location » et « eas_film », permet de consulter l'historique des films. La seconde, qui joint les tables « eas_client », « eas_location » et « eas_documentaire », permet de consulter l'historique des documentaires. Pour le dernier cas d'utilisation, qui concerne la consultation de toutes les vidéos disponibles en magasin, il suffit de requêter les tables « eas_film » et « eas_documentaire » tour à tour.

Dans le modèle ci-dessus, j'ai choisi deux catégories de vidéos différentes car il est intéressant par la suite de voir l'utilité d'un modèle dit « schema-less ». En effet, la rigidité du modèle relationnel impose que la structure de la table soit définie à l'avance et que celle-ci soit respectée lors de l'insertion de données. Etant donné que les films et les documentaires possèdent des caractéristiques différentes, il faut créer deux tables bien distinctes. Nous allons voir que ce procédé n'est pas nécessaire dans un contexte NoSQL. Voici la même modélisation, en utilisant la méthode de dénormalisation.

Figure 30

Modélisation par la dénormalisation

Avant d'entrer dans les détails, il faut savoir qu'il n'existe aucune recette miracle de modélisation. Celle-ci s'effectue en fonction des besoins spécifiques du client. Comme je l'ai mentionné précédemment, Cassandra (et une grande majorité des bases NoSQL) rejette la notion de jointure de table. Le principe consiste donc à ce que chaque famille de colonnes (tables en SQL) puisse retourner l'information nécessaire pour chaque cas de figure cité plus haut. Si l'utilisateur veut consulter le profil client, une requête sera donc faite sur la famille de colonnes « eas_client ». Dans le cas d'une consultation de l'historique de location d'un client, une requête sera faite sur la famille de colonnes « eas_clientLocations ». Dans le dernier cas de figure, qui consiste à consulter la base de données des vidéos disponibles, une requête sur la famille de colonnes « eas_video » sera nécessaire. Deux éléments importants se dégagent de cette analyse. Premièrement, j'ai dupliqué les colonnes dans la famille de colonnes « eas_clientLocations ». En effet, comme je l'ai déjà expliqué, il arrive qu'il soit nécessaire de dupliquer l'information sur plusieurs de ces familles de colonnes dans les cas où une requête ne peut être faite que sur une seule famille de colonnes à la fois. Deuxièmement, comparé au premier modèle conceptuel de données (Figure-29), il n'y a plus qu'une seule famille de colonnes (ou table, tout dépend dans quel contexte de base de données on se situe) : « eas_video », qui regroupe les deux tables « eas_film » et « eas_documentaire » avec une colonne « video_type » permettant de différencier les vidéos. Pour la famille de colonnes «eas_video », dont les colonnes peuvent varier (par

exemple, si la vidéo est un documentaire, elle aura une caractéristique « eas_typeDocumentaire », alors que si c'est un film, elle aura une caractéristique « eas_genre »), je vais utiliser le concept de famille de colonnes dynamique.

Comme je l'ai mentionné plus haut, Cassandra travaille sur un type de schéma dit « schema-less », ce qui signifie qu'il n'est pas nécessaire de définir la structure d'une famille de colonnes avant de pouvoir y ajouter des données. Nous avons vu également qu'il existe deux types de familles de colonnes.

Les familles de colonnes statiques, pour lesquelles il faut définir au préalable la structure des métadonnées, sont utilisées lorsqu'on on connaît à l'avance le type de contenu des données et qu'on est sûr qu'il ne changera pas. Etant donné que les types informations sur un client ainsi que son historique de location ne changeront pas, les familles de colonnes « eas_client » et « eas_clientLocations » seront de type statique.

La seconde catégorie de familles est la famille de type dynamique. Dans une famille de colonnes dynamique, il n'est pas nécessaire de définir la structure au préalable. Ainsi, les colonnes sont définies de façon dynamique lors de l'insertion des données. Chaque ligne d'enregistrement a donc sa propre structure de colonnes (voir le passage des familles de colonnes du chapitre 7.2 Modèle de données de Cassandra). Généralement, on utilise les familles dynamiques quand on ne sait pas si les types d'informations seront toujours les mêmes. C'est pourquoi « eas_video » sera de type dynamique.

7.3.5 L'indexation

Bien entendu, l'indexation n'est pas de la plus grande utilité dans un contexte comme celui-ci. Malheureusement, je ne dispose pas d'une base de données conséquente afin de démontrer l'amélioration en performance de l'indexation. Cependant, il est toujours bon de savoir comment l'indexation fonctionne ; c'est la raison pour laquelle je l'ai aussi appliquée au projet.

Cassandra offre deux types d'index : les index primaires et les index secondaires. Les premiers sont les index de la « rowKey » d'une ligne d'enregistrement et sont principalement utilisés par le système pour le partitionnement des données sur les différents nœuds d'un cluster. Ceux auxquels je vais plus m'intéresser pour ce cas de figure sont les index secondaires, c'est-à-dire ceux que je vais appliquer aux colonnes. Dans quels cas de figure est-il judicieux de créer des index secondaires ? L'élaboration d'index secondaires est très efficace lorsque beaucoup d'enregistrements contiennent la même valeur pour une colonne donnée. Par exemple, supposons que nous avons une famille de colonnes utilisateur avec des millions d'utilisateurs habitant en Suisse. Une

indexation pourrait porter, par exemple, sur leur canton d'origine. L'indexation servant à améliorer la vitesse de réponse d'une requête sur un critère donné, il est évident que nous allons indexer des colonnes qui seront sollicitées lors d'une recherche dans l'application.

Sur le modèle de « Easy_Location », je vais donc indexer deux colonnes qui se situent sur des familles de colonnes différentes (les colonnes indexées sont surlignées en rouge dans le modèle de données de la figure-30 du chapitre « 7.3.4 Modèle de données Easy_Location ». La première indexation s'effectuera sur la colonne « client_localité » de la famille de colonnes « eas_client ». Elle contribuera à améliorer la performance lorsqu'une recherche des clients par ville sera effectuée. Quant à la deuxième indexation, elle s'effectuera sur la colonne « video_type » de la famille de colonnes « eas_clientLocations ». Son but sera d'accroître la performance lors d'une recherche dans l'historique d'un client par type de vidéo. La façon d'indexer les colonnes sera traitée au chapitre 7.3.7 La création de familles de colonnes.

7.3.6 Utilisation de Cassandra-cli pour une meilleure approche

Bien que j'aie décidé d'utiliser le langage CQL dans le cadre de ce travail, j'ai préféré revenir au « Cassandra-cli » pour la création de famille de colonnes et pour l'insertion. Je trouve que « Cassandra-cli » est bien plus en accord avec la théorie des différents types de familles de colonnes. La première chose qui surprend dans le langage CQL est qu'il n'est pas possible de créer une famille de colonnes sans schéma ; on doit impérativement définir une colonne. Ensuite, lorsqu'on veut ajouter une colonne durant l'insertion de données de façon dynamique, il faut d'abord créer la colonne en question à l'aide de la commande « Alter table » afin de modifier la structure de la famille de colonnes au niveau métadonnées. En d'autres termes, il n'est pas possible d'insérer une donnée dans une colonne qui n'existe pas : il faut dans un premier temps la créer et ensuite y insérer la donnée. On peut alors se demander où est l'aspect de création dynamique du schéma lors de l'insertion, car Oracle le fait tout aussi bien. Bien que les données soient en fin de compte stockées de la même manière dans la base, le CQL peut s'avérer très déstabilisant en ce qui concerne les familles de colonnes dynamiques. Dans la communauté Cassandra, c'est un sujet qui fait débat à tel point qu'un article « Does CQL support dynamic columns / wide rows ? » a été publié par Datastax afin de répondre à la question.

7.3.7 La création de familles de colonnes

A présent, il est temps de passer à la création des familles de colonnes. Je vais donc créer deux familles statiques : « eas_client » et « eas_clientLocations ».

Création de la famille de colonnes « eas_client »

```
CREATE COLUMN FAMILY eas_client
WITH comparator = UTF8Type
AND key_validation_class = UTF8Type
AND column_metadata = [
{column_name:client_num, validation_class: Int32Type}
{column_name:client_nom, validation_class: UTF8Type}
{column_name:client_prénom, validation_class: UTF8Type}
{column_name:client_adresse, validation_class: UTF8Type}
{column_name:client_codePostal, validation_class: Int32Type }
{column_name:client_localite, validation_class: UTF8Type, index_type: KEYS}
{column_name:client_dateDeNaissance, validation_class: DateType }
{column_name:client_actif, validation_class: BooleanType}
]);
```

Création de la famille de colonnes « eas_clientLocations»

```
CREATE COLUMN FAMILY eas_clientLocations
WITH comparator = UTF8Type
AND key_validation_class = UTF8Type
AND column_metadata = [
{column_name:client_num, validation_class: Int32Type}
{column_name:client_nom, validation_class: UTF8Type}
{column_name:location_dateLocation, validation_class: DateType }
{column_name:location_dateRédition, validation_class: DateType }
{column_name:location_prix, validation_class: Int32Type }
{column_name:video_num, validation_class: UTF8Type}
{column_name:video_type, validation_class: UTF8Type, index_type: KEYS }
{column_name:video_titre, validation_class: UTF8Type }
{column_name:video_typeSupport, validation_class: UTF8Type }
]);
```

On constate qu'en général la création d'une famille de colonnes statique est très similaire à celle d'une table SQL. Il y existe cependant quelques éléments supplémentaires dont il est important de parler. Le premier de ces éléments est le « comparator », qui est le type de données utilisé pour les noms de colonnes. Il est employé par le système pour effectuer différents types de tris. C'est pourquoi il est important de ne pas se tromper lors de la définition de celui-ci, au risque d'avoir des tris inappropriés sur le disque. Le comparator n'est pas obligatoirement demandé lors de la définition de la famille de colonnes, mais il est fortement conseillé de le définir. Le deuxième élément est le « validator », qui est le type de données de la valeur des colonnes. Un type « validator » doit être attribué à chaque colonne. C'est le même principe qu'en SQL. Enfin, il convient de noter l'indexation des colonnes (elles sont marquées en rouge). On peut voir qu'il est extrêmement simple d'indexer une colonne. L'indexation peut s'effectuer soit au moment

de la création de la famille de colonnes, soit après, en mettant à jour la table à l'aide de la commande « Update ». Cependant, si l'indexation sur une colonne ne se fait pas lors de la création de la famille de colonnes, les données qui y seront insérées avant l'indexation de la colonne ne seront pas indexées.

Regardons maintenant la création de notre famille de colonnes dynamique « eas_video ».

Création de la famille de colonnes « eas_video»

*CREATE COLUMN FAMILY **eas_video***
WITH key_validation_class=UTF8Type
AND Comparator= UTF8Type
AND default_validation_class = UTF8Type;

On constate ici que, pour créer une famille de colonnes dynamique, il n'est effectivement pas obligatoire de définir des colonnes. En revanche, il est dans ce cas nécessaire de définir un « default_validation_class » par défaut, qui sert à valider toutes les valeurs de la colonne en question. Les colonnes seront créées dynamiquement au moment de l'insertion de données, mais ce point sera abordé plus en détail au chapitre suivant.

7.3.8 Insertion de données

En ce qui concerne l'insertion de données, il faut reconnaître que l'automatisation fournie par le langage CQL par rapport à la création de la « rowkey » (qui est l'équivalent de la clé primaire dans les bases de données relationnelles) est bien plus confortable que le client « Cassandra-cli ». En effet, lorsqu' on crée une famille de colonnes en langage CQL, on définit la rowkey sur un attribut.

Voici un exemple :

*CREATE TABLE **client** (*
***Client_num** int,*
***Client_nom** varchar,*
***Client_prenom** varchar,*
…
*PRIMARY KEY (**client_num**)) ;*

On remarque que le langage CQL permet de créer la « rowkey » automatiquement lors des insertions de données. En revanche, avec le client «Cassandra –cli », la « rowkey » doit être insérée manuellement en plus des différentes colonnes que contient la ligne. Je vais donc procéder à des insertions de client avec les deux solutions afin de voir les différences qui existent entre elles. Voici une insertion de données en CQL :

INSERT INTO eas_client (client_num,client_nom,client_prenom,client_adresse,client_codePostal,client_localite,) VALUES (1,'guijarrot','alain',1205,'rue de carouge 95'','Genève');

Bien que j'aie utilisé le client CQLSH pour l'insertion, je vais consulter les données avec le client « Cassandra-cli » pour deux raison. Premièrement, « Cassandra-cli » met bien en évidence la rowKey de chaque ligne, ce que ne fait pas CQLSH. Deuxièmement, les données qu'il présente sont plus lisibles. Voici ce que l'on obtient lors d'une interrogation avec le client Cassandra-cli.

Figure 31

Interrogation table client

```
[default@easy_location] LIST eaS_client;
Using default limit of 100
Using default cell limit of 100
-------------------
RowKey: 1
=> (name=, value=, timestamp=1380730425528000)
=> (name=client_adresse, value=727565206465206361726f756765203935, timestamp=138
0730425528000)
=> (name=client_codepostal, value=31323035, timestamp=1380730425528000)
=> (name=client_localite, value=47656e657665, timestamp=1380730425528000)
=> (name=client_nom, value=6775696a6172726f74, timestamp=1380730425528000)
=> (name=client_prenom, value=616c61696e, timestamp=1380730425528000)

1 Row Returned.
Elapsed time: 16 msec(s).
[default@easy_location] _
```

On constate que la valeur de la « rowkey » correspond à la valeur d'insertion de la colonne « client_num ». La valeur de chacune des colonnes n'est pas visible car le comparator utilisé entre les différents clients n'est pas le même. Toutefois, ce n'est pas vraiment ce que je tente de démontrer dans le présent travail.

Voici maintenant une insertion effectuée avec le client « Cassandra-cli »

*SET eas_client[2][**'client_num'**]= 2;*
*SET eas_client[2][**'client_nom'**]='Piazza';*
*SET eas_client[2][**'client_prenom'**]= 'Adriano' ;*
*SET eas_client[2][**'client_adresse'**]= 'Av de la praille 32' ;*
*SET eas_client[2][**'client_codePostal'**]= '1227' ;*
*SET eas_client[2][**'client_localite'**]= 'Carouge' ;*

Voici le résultat que l'on obtient lors de l'interrogation :

Figure 32

Interrogation table client

```
[default@easy_location] list eas_client;
Using default limit of 100
Using default cell limit of 100
-------------------
RowKey: 2
=> (name=client_adresse, value=av de la praille 32, timestamp=1380734631865000)
=> (name=client_codePostal, value=1227, timestamp=1380734791424000)
=> (name=client_localite, value=Carouge, timestamp=1380734821157000)
=> (name=client_nom, value=pantaleo, timestamp=1380734536070000)
=> (name=client_num, value=2, timestamp=1380734510031000)
=> (name=client_prenom, value=giuseppe, timestamp=1380734565934000)

1 Row Returned.
Elapsed time: 29 msec(s).
[default@easy_location]
[default@easy_location] _
```

Avec le client « Cassandra-cli », lorsqu' on utilise les commandes d'insertion, on doit définir sur quelle ligne (commande surlignée en vert) et sur quelle colonne on va effectuer la modification. Si la ligne en question n'existe pas, la commande « Set » va agir comme une insertion. Si la ligne existe déjà, la commande « Set » va agir comme une mise à jour. Ainsi, la colonne « client_num » n'est plus qu'un simple attribut ici. J'ai tout de même décidé de maintenir cette colonne pour garder une certaine cohérence avec le modèle de données. Avec cette façon d'insérer manuellement la « rowKey », on s'expose à un plus grand risque d'erreur humaine.

Bien que j'aie réussi à démontrer certaines différences que je trouvais intéressantes en travaillant sur les deux clients pour l'insertion des données, il existe à ce jour des incompatibilités entre ces deux langages. C'est le cas, par exemple, de l'impossibilité de définir le comparator manuellement dans le langage CQL lors de la création d'une famille de colonnes. Cette incompatibilité est source de problèmes d'insertion depuis le client « Cassandra-cli » ainsi que de visualisation de données (voir la Figure-32).

7.4 Conclusion

Il est temps de tirer quelques conclusions sur ce travail pratique. Sur le plan technique, trois éléments principaux sont à retenir.

Le premier d'entre eux est le modèle de données de Cassandra lui-même. Habitué aux bases de données relationnelles, j'ai trouvé intéressant de voir les différences qu'apporte Cassandra dans la structure du modèle.

Ensuite, il était intéressant de voir les deux clients (Cassandra-cli et CQLSH) permettent d'attaquer la base de données Cassandra. J'ai trouvé des points positifs chez chacun

d'entre eux. Cela dit, comme Cassandra-cli n'évoluera plus, il est évident que les développeurs vont de plus en plus se tourner vers le langage CQL (ce qui est déjà le cas). On peut remarquer que le langage CQL est très similaire au très célèbre langage SQL des bases relationnelles, sans doute pour permettre un passage plus facile des différents experts SQL, mais aussi dans un but de standardisation. Cependant, comme je l'ai déjà dit, je pense qu'il est plus judicieux, dans un contexte d'apprentissage, d'utiliser le client Cassandra-cli. En effet, la façon dont il manipule la base de données est plus en phase avec les diverses théories enseignées.

Le dernier point technique est la modélisation par la dénormalisation. Etant donné que Cassandra est un système de base de données qui rejette la notion de jointure, l'approche utilisée pour les bases de données relationnelles n'était tout simplement pas possible. Habitué aux systèmes relationnels et à la normalisation des données dont le but est d'éviter la redondance au maximum, cette approche a été au départ déstabilisante. Elle a été cependant très instructive, et je sais dorénavant qu'il n'y pas qu'une seule façon de modéliser le schéma d'une base de données.

L'une des questions à laquelle il fallait répondre dans le cadre de ce travail pratique était de savoir si une base de données NoSQL, telle que Cassandra, était adaptée à ce genre d'application. La réponse (on peut se le dire sans trop de surprises) à cette question est non, et ceci pour différentes raisons. Tout d'abord, Cassandra (ainsi que les autres bases de données NoSQL) tire son principal avantage de son architecture distribuée, permettant de répartir la charge de travail entre plusieurs machines. Ici, nous n'allons tirer aucun profit de cet atout, car l'application est destinée à une machine « standalone ». Par conséquence il n'y aura pas de requêtes multiples à gérer, ni des accès concurrentiels au niveau de la base de données. La deuxième raison est l'aspect volumétrique des données. Soyons réalistes : même si le vidéo club connaissait un franc succès au point d'atteindre un millier de clients, un simple système de gestion de base de données centralisé suffirait largement à la gestion de l'application.

Non seulement le système Cassandra n'est pas nécessaire dans un tel cas, mais il aurait un effet négatif. En effet, son utilisation requiert la duplication des données dans la base (voir chapitre sur la dénormalisation), ce qui augmente les risques d'incohérences entre les différentes familles de colonnes. Il faudra également faire plus d'opérations en cas de modification d'une donnée. Par exemple, imaginons que nous avons une cliente de longue date qui vient annoncer son changement de nom de famille suite à son mariage. Dans un système relationnel, la modification ne s'appliquera que sur la table « eas_client », alors que, dans le système Cassandra, il faudra appliquer le changement

non seulement sur la famille de colonnes « eas_client » mais aussi sur la famille de colonnes «eas_clientLocations », car les données du client sont répétées à chaque nouvelle location (voir modèle de données « Easy_location »). Toutes ces duplications de données entraînent aussi une consommation supplémentaire de l'espace en disque.

Tout ceci confirme ce que nous disions initialement sur le fait que les bases de données NoSQL ne sont pas forcément meilleures que les bases relationnelles, et vice-versa. La phase d'analyse d'un administrateur de base de données sur le choix de celles-ci est donc très importante et doit être faite très soigneusement. Un mauvais choix pourrait avoir un effet très néfaste pour la suite des évènements.

Bibliographie

BRUCHEZ, Rudi, 2013. *Les bases de données NoSQL – Comprendre et mettre en oeuvre*. Paris.

Webographie

[1] Wikipédia – NoSQL [consulté le 30.07.2013] Disponible à l'adresse :

http://fr.wikipedia:.org/wiki/NoSQL

[2] Inzecloud – Qu'est ce que le NoSQL ? [consulté le 30.07.2013] Disponible à l'adresse :

http://www.inzecloud.fr/quest-ce-que-le-nosql/

[3] Encyclopeadia Universalis – Systèmes informatiques [consulté le 27.07.2013] Disponible à l'adresse :

http://www.universalis.fr/encyclopedie/systemes-informatiques-systemes-de-gestion-de-bases-de-donnees/2-bref-historique-et-typologie/

[4] Wikipédia – Structured Query Language [consulté le 27.07.2013] Disponible à l'adresse :

http://fr.wikipedia.org/wiki/SQL

[5] Wikipédia – Théorème CAP [consulté le 13.08.2013] Disponible à l'adresse :

http://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_CAP

[6] MarginWeb – Quand et pourquoi utiliser une base de données NoSQL ? [consulté le 01.08.2013] Disponible à l'adresse :

http://www.marginweb.com/blog/20121110/quand-et-pourquoi-utiliser-une-base-de-donnees-nosql

[7] LinuxFr – Base de données : Petit état des lieux du NoSQL [consulté le 01.08.2013] Disponible à l'adresse :

http://linuxfr.org/news/petit-etat-des-lieux-du-nosql

[8] Macobelix – Rapport Architecture logicielle [consulté le 06.08.2013] Disponible à l'adresse :

http://macobelix.polytech.unice.fr/Cours/Sar03/Exposes/11R-BigData_NoSQL.pdf

[9] Xebia – NoSQL Europe : Bases de données orientées colonnes et Cassandra [consulté le 06.08.2013] Disponible à l'adresse :

http://blog.xebia.fr/2010/05/04/nosql-europe-bases-de-donnees-orientees-colonnes-et-cassandra/

[10] Mathieu ROGER – Synthèse d'étude et projets d'intergiciels [consulté le 06.08.2013] Disponible à l'adresse :

http://blog.octera.info/wp-content/uploads/2010/11/Base-NOSQL.pdf

[11] Amazon web services – Amazon DynamoDB [consulté le 13.08.2013] Disponible à l'adresse :

http://aws.amazon.com/fr/dynamodb

[12] Developpez.com– Introduction à la base de données NoSQL Cassandra [consulté le 13.08.2013] Disponible à l'adresse :

http://soat.developpez.com/articles/cassandra/

[13] Developpez.com– Introduction au NoSQL Apache CASSANDRA [consulté le 13.08.2013] Disponible à l'adresse :

http://mbaron.developpez.com/nosql/cassandra/installation-outils-administration/

[14] DATASTAX– Internal authentication [consulté le 17.09.2013] Disponible à l'adresse :

http://www.datastax.com/documentation/cassandra/1.2/webhelp/index.html?pagename=docs&version=1.2&file=index#cassandra/security/secure_about_native_authenticate_c.html

[15] Wikipédia – MongoDB [consulté le 14.08.2013] Disponible à l'adresse :

http://fr.wikipedia.org/wiki/MongoDB

[16] NoSQL – LIST OF NOSQL DATABASES [consulté le 14.08.2013] Disponible à l'adresse :

http://nosql-database.org/

[17] Journal du net – MongoDB : une gestion intelligente de la montée en charge [consulté le 14.08.2013] Disponible à l'adresse :

http://www.journaldunet.com/developpeur/outils/comparatif-des-bases-nosql/mongodb.shtml

[18] InfoQ – Les bases Orientées Graphes, NoSQL et Neo4J [consulté le 15.08.2013] Disponible à l'adresse :

http://www.infoq.com/fr/articles/graph-nosql-neo4j

[19] GREEN DATA CENTER – Advantages of NoSQL [consulté le 15.08.2013] Disponible à l'adresse :

http://greendatacenterconference.com/blog/the-five-key-advantages-and-disadvantages-of-nosql/

[20] Tech Nirvana – The five key advantages(and disadvantages) of NoSQL [consulté le 15.08.2013] Disponible à l'adresse :

http://technirvanaa.wordpress.com/tag/nosql-disadvantages/

[21] LePPF – Le NULL en SQL [consulté le 21.08.2013] Disponible à l'adresse :

http://www.leppf.fr/spip.php?article33

[22] Adam MARCUS – The NoSQL Ecosystem [consulté le 23.08.2013] Disponible à l'adresse :

http://www.aosabook.org/en/nosql.html

[23] talks !– Les patterns des grands du Web – Sharding [consulté le 23.08.2013] Disponible à l'adresse :

http://blog.octo.com/sharding/

[24] Anne Benoit – Algorithmique des réseaux et des télécoms [consulté le 27.08.2013] Disponible à l'adresse :
http://graal.ens-lyon.fr/~abenoit/reso05/cours/2-p2p.pdf

[25] Wikipédia – Consistent hashing [consulté le 27.08.2013] Disponible à l'adresse :

http://en.wikipedia.org/wiki/Consistent_hashing

[26] Wikipédia – MapReduce [consulté le 29.08.2013] Disponible à l'adresse :

http://fr.wikipedia.org/wiki/MapReduce

[27] Stéphane Genaud – MapReduce : un cadre de programmation parallèle pour l'analyse de grandes données [consulté le 30.08.2013] Disponible à l'adresse :

http://icps.u-strasbg.fr/~genaud/courses/sd/mapreduce.pdf

[28] Wikipédia – Hadoop [consulté le 30.08.2013] Disponible à l'adresse :

http://fr.wikipedia.org/wiki/Hadoop

[29] Journal du Net – Hadoop en 5 questions [consulté le 04.09.2013] Disponible à l'adresse :

http://www.journaldunet.com/solutions/systemes-reseaux/definition-d-hadoop.shtml

[30] bulletins-electroniques.com– Hadoop une technologie en plein essor [consulté le 04.09.2013] Disponible à l'adresse :

http://www.bulletins-electroniques.com/actualites/72829.htm

[31] Hadoop Wiki– Hadoop [consulté le 06.09.2013] Disponible à l'adresse :

http://wiki.apache.org/hadoop/

[32] Wikipédia– Eventual consistency [consulté le 12.09.2013] Disponible à l'adresse :

http://en.wikipedia.org/wiki/Eventual_consistency

[33] SYBASE– Dénormalisation de tables et de colonnes [consulté le 13.09.2013] Disponible à l'adresse :

http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.infocenter.dc31014.1520/doc/html/rad1232020345198.html

[34] DATASTAX– Does CQL support dynamic columns / wide rows ? [consulté le 26.09.2013] Disponible à l'adresse :

http://www.datastax.com/dev/blog/does-cql-support-dynamic-columns-wide-rows

[35] DATASTAX– About column families [consulté le 30.09.2013] Disponible à l'adresse :

http://www.datastax.com/docs/0.8/ddl/column_family

[36] DATASTAX– About indexes in Cassandra [consulté le 2.10.2013] Disponible à l'adresse :

http://www.datastax.com/docs/1.0/ddl/indexes

Annexe 1 : Script CQL de création des utilisateurs et D'attribution des permissions

Création des utilisateurs :

*CREATE USER **User_user** WITH PASSWORD '**user'** NOSUPERUSER;*
*CREATE USER **User_admin** WITH PASSWORD 'admin' NOSUPERUSER;*
*CREATE USER **User_adminBdd** WITH PASSWORD **'adminBdd'** SUPERUSER;*

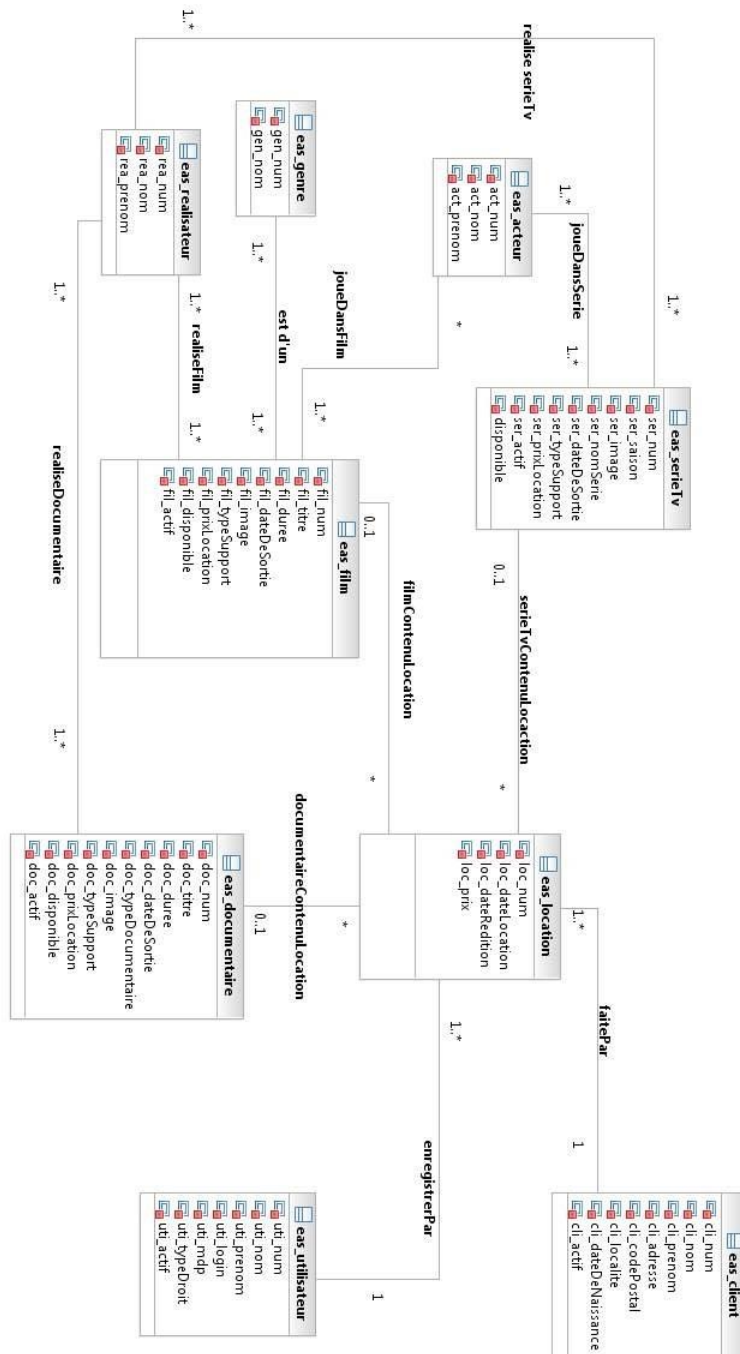
Attribution des permissions:

*GRANT SELECT ON TABLE **eas_client** TO **User_user**;*
*GRANT SELECT ON TABLE_**eas_clientLocations** TO **User_user**;*
*GRANT SELECT ON TABLE **eas_video** TO **User_user**;*
*GRANT MODIFY ON TABLE **eas_client** TO **User_user**;*
*GRANT MODIFY ON TABLE **eas_clientLocations** TO **User_user;***
*GRANT MODIFY ON TABLE **eas_video** TO **User_user**;*
*GRANT SELECT ON KEYSPACE **easy_location** TO **User_admin;***
*GRANT MODIFY ON KEYSPACE **easy_location** TO **User_admin;***

Annexe 2 : Modèle de données relationnelle Easy_Location

Figure 33

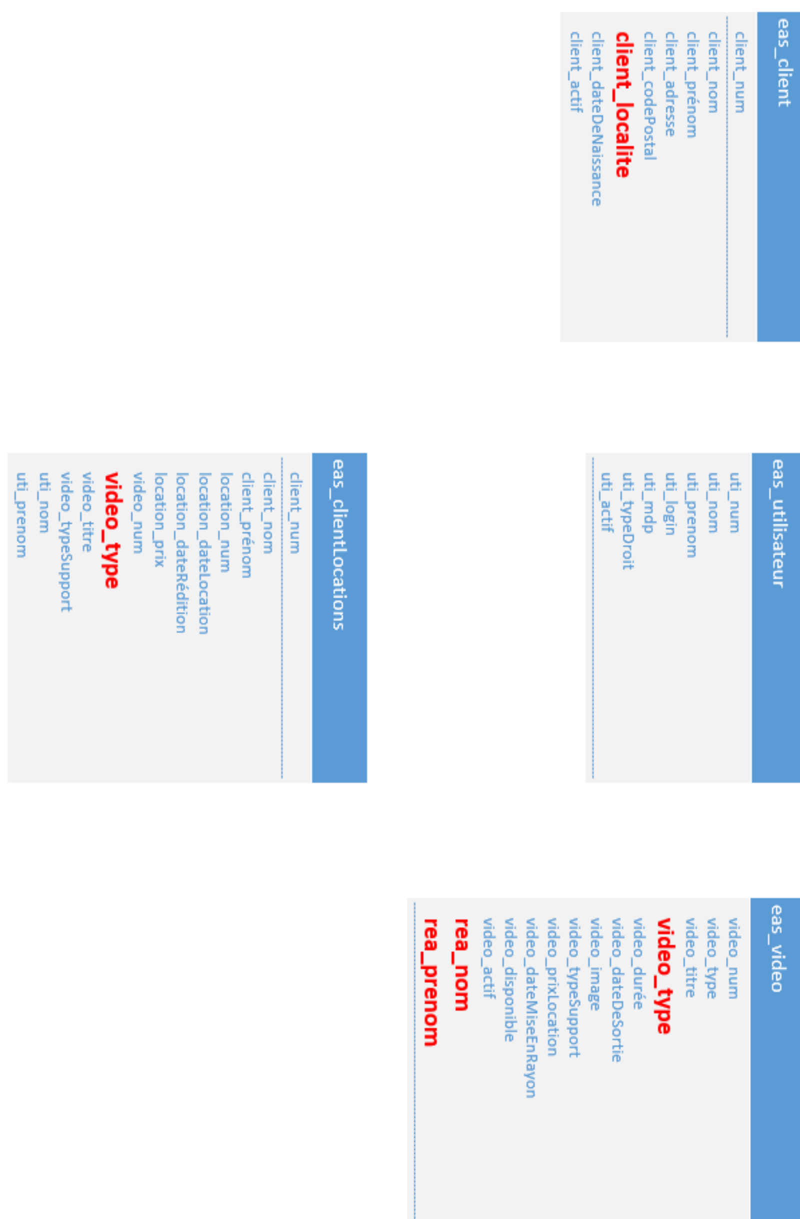
Modèle de données SQL Easy Location



Annexe 3 : Modèle de données Easy_Location dénormalisé

Figure 34

Modèle de données dénormalisé



Note : les colonnes en couleur rouge, sont celles indexées.

Annexe 4 : Création des familles de colonnes avec le client « Cassandra-cl »

Création de la famille de colonnes « eas_client »

*CREATE COLUMN FAMILY **eas_client***
WITH comparator = UTF8Type
AND key_validation_class = UTF8Type
AND column_metadata = [
*{column_name:**client_num**, validation_class: Int32Type}*
*{column_name:**client_nom**, validation_class: UTF8Type}*
*{column_name:**client_prénom**, validation_class: UTF8Type}*
*{column_name:**client_adresse**, validation_class: UTF8Type}*
*{column_name:**client_codePostal**, validation_class: Int32Type }*
*{column_name:**client_localite**, validation_class: UTF8Type, **index_type: KEYS**}*
*{column_name:**client_dateDeNaissance**, validation_class: DateType }*
*{column_name:**client_actif**, validation_class: BooleanType}*
]);

Création de la famille de colonnes « eas_clientLocations»

*CREATE COLUMN FAMILY **eas_clientLocations***
WITH comparator = UTF8Type
AND key_validation_class = UTF8Type
AND column_metadata = [
*{column_name:**client_num**, validation_class: Int32Type}*
*{column_name:**client_nom**, validation_class: UTF8Type}*
*{column_name:**location_dateLocation**, validation_class: DateType }*
*{column_name:**location_dateRédition**, validation_class: DateType }*
*{column_name:**location_prix**, validation_class: Int32Type }*
*{column_name:**video_num**, validation_class: UTF8Type}*
*{column_name:**video_type**, validation_class: UTF8Type, **index_type: KEYS** }*
*{column_name:**video_titre**, validation_class: UTF8Type }*
*{column_name:**video_typeSupport**, validation_class: UTF8Type }*
*{column_name:**uti_nom**, validation_class: UTF8Type }*
*{column_name:**uti_prenom**, validation_class: UTF8Type }*
]);

Création de la famille de colonnes « eas_video»

*CREATE COLUMN FAMILY **eas_video***
WITH key_validation_class=UTF8Type
AND Comparator= UTF8Type
AND default_validation_class = UTF8Type;

Création de la famille de colonnes « eas_video»

*CREATE COLUMN FAMILY **eas_utilisateur***
WITH comparator = UTF8Type
AND key_validation_class = UTF8Type

```
AND column_metadata = [
{column_name:uti_num, validation_class: Int32Type}
{column_name: uti_nom, validation_class: UTF8Type}
{column_name: uti_prénom, validation_class: UTF8Type}
{column_name:uti_login, validation_class: UTF8Type}
{column_name:uti_mdp, validation_class: UTF8Type }
{column_name:uti_typeDroit, validation_class: UTF8Type,}
{column_name:uti_actif, validation_class: Boolean }

]);
```