Chapitre 8

Emergence et principes des BdD *NoSQL* : souplesse, volumétrie et performance

Ce chapitre commence par résumer l'évolution des BdD avec la mise au point du modèle relationnel, ses fondements mais aussi ses limitations, qui ont conduit à l'apparition d'autres types de BdD, dites *NoSQL*. Les concepts introduits ou mis en valeur par le *NoSQL*, pour répondre aux nouveaux besoins des grandes industries du web, sont présentés dans leur contexte d'émergence. Les différents types de BdD *NoSQL* sont également présentés, classées selon le type de schéma de donnée qu'elles utilisent (clé/valeur, colonne, index inversé, clé/document, graphe). Enfin, le positionnement des annuaires LDAP est discuté, et le format de données JSON, très prisé des solutions *NoSQL*, est décrit en fin de chapitre.

8.1 Motivations et émergence des BdD *NoSQL*

Les technologies de BdD NoSQL sont apparues dans les entreprises du web (Google bien sur, puis Yahoo !, Amazon, Facebook. . .), qui étaient confrontées à des besoins de gestion et d'analyse de données pour lesquels il n'existait pas de solution satisfaisante. Ce sont donc de nouveaux besoins industriels qui ont menés à l'émergence rapide de l'approche NoSQL.

8.1.1 Les BdD avant l'approche *NoSQL*

Les aspects historiques

Edgar Frank Codd élabore des stratégies d'organisation des données au sein d'IBM au milieu des années 1960. Il publie un premier document interne en 1969, puis un article dans une revue ACM en 1970 qui pose les bases du modèle relationnel (voir le haut de la figure 8.1). Ce modèle repose sur des relations entre les valeurs des données, et non pas sur des pointeurs entre entités comme dans les modèles précédents. Il inclut aussi la manipulation des données à travers une algèbre relationnelle et un langage de haut niveau associé, et il dissocie la représentation et l'interrogation des données d'une part, et leur stockage sur disque d'autre part. Au début cette séparation semblait ne pas pouvoir être performante : comment être performant en interrogeant des données sans savoir comment elle sont stockées ? Mais finalement, les moteurs des bases de données finiront par être plus performant que les développeurs humains pour stocker et accéder à des données sur disques.

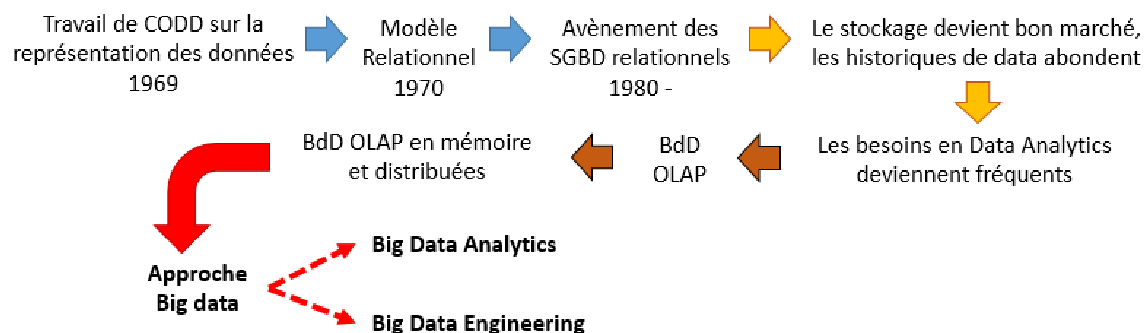

FIGURE 8.1 – Evolution des technologies de BdD

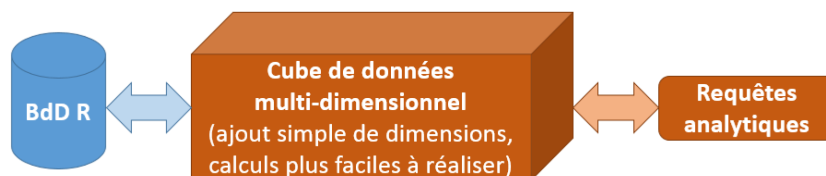

FIGURE 8.2 – Cube de données pour requêtes analytiques, au dessus d'une BdD relationnelle

Les bases de données relationnelles deviennent ensuite un standard dans les années 1980, avec des implantations performantes sur de gros systèmes distribués, mais aussi avec des implantations légères sur de simples PC.

Les contraintes et forces du modèle relationnel

Quelle que soit la taille de la BdD et du système sous-jacent, une BdD relationelle doit en théorie respecter 13 règles édictées par Codd. En conséquence, une BdD relationelle impose de nombreuses contraintes à ses concepteurs.

Les attributs ont des types précis que l'on pourrait qualifier de *scalaires* (entier, date, string de taille fixée...), les types de taille variable sont exclus ou déconseillés (listes, tableaux sans taille maximale...). Toutes les lignes d'une relation ont les mêmes colones, c'est-à-dire la même suite d'attributs. Un attribut sans valeur dans une ligne doit être présent avec une valeur *NULL*. La BdD ne peut être interrogée que par un langage relationnel qui respecte pleinement l'algèbre relationelle. Les modifications sur la BdD peuvent être regroupées et appliquées en une seule séquence atomique, garantissant l'intégrité de la base (lors de son interrogation elle intègre alors toutes ou aucune modifications)...

Au final, la conception d'une BdD relationnelle repose sur celle d'un *schéma* de base après une méthodologie de modélisation, où on minimise la redondance de représentation des données. L'interrogation de la BdD se fait ensuite par *jointure* de plusieurs relations, et des mécanismes internes au SGBD (comme l'indexation) optimisent l'exécution de ces opérations. Toutefois de nombreuses jointures sur de grosses relations peuvent demander beaucoup de mémoire et de temps de traitement. Un schéma de base est en général plus favorable à certaines interrogations, alors que pour d'autres il demandera plus de jointures.

L'impact du *Data Analytics*

Les requêtes transactionnelles, comme la recherche de clients habitant une ville donnée et possédant une voiture, sont bien supportées par le modèle relationnel. En revanche, l'analyse des données visant à calculer des statistiques complexes sur une grosse partie de la base sont plus

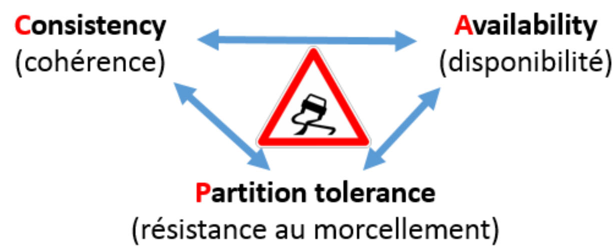

FIGURE 8.3 – Les 3 objectifs impossible à concilier ("problème CAP")

difficilement supportées. Ces requêtes peuvent demander de nombreuses jointures et surtout des calculs importants sur des données volumineuses. Mener l'interrogation par un langage purement relationnel n'est alors pas très adapté. Plus formellement, cette différence de besoins a mené à identifier deux types de requêtes et de moteurs de base : les requêtes OLTP et OLAP (deux types de requêtes définis par Codd).

Il arrive que les données soient peu fréquemment mises à jour, où qu'au contraire la BdD relationnelles reçoive un flux quasi continu de mises à jour. Mais dans tous les cas, des requêtes de type OLTP (*OnLine Transaction Processing*) cherchant à extraire des informations essentiellement selon des conditions sur les valeurs de certains attributs, correspondent bien aux capacités des BdD relationnelles, initialement conçues pour cela.

En revanche, les requêtes d'analyse de données (*Data Analytics*) ont émergé plus tard, quand la capacité de stockage a permis de sauvegarder à bon marché des historiques très volumineux de nombreuses données d'entreprises. Dès lors, beaucoup d'entreprises ont souhaité réaliser des analyses statistiques de leurs données, parfois à travers quasiment toutes leurs bases, pour *calculer* de nouveaux indicateurs et dégager des tendances, et non pas seulement pour extraire certaines valeurs factuelles de la BdD. Mais les BdD traditionnelles se sont avérées inadaptées pour traiter ces requêtes OLAP (*OnLine Analytical Processing*). Des réorganisations des schémas des BdD ont alors été étudiées pour faciliter le traitement de ces requêtes. Des schémas en étoile sont apparus, avec une relation centrale à laquelle se rattachent beaucoup d'autres, puis des solutions sous forme de *cubes de données multi-dimensionnels* ont vu le jour (voir figure 8.2). On peut facilement ajouter des dimensions à un cube de données (une pour chaque grandeur répertoriée et analysée) et déclencher des calculs d'analyses statistiques à travers de grandes parties du cube. Pour plus d'efficacité, on a vu apparaître des bases de données OLAP avec un stockage entièrement en mémoire, quitte à les distribuer pour utiliser la mémoire de plusieurs PC (*in memory OLAP*). Des développements de *Complex Event Processing* (CEP) ont alors émergé, qui supportent des flux de mises à jour du cube de données, et qui recalculent presque en permanence des indicateurs statistiques sur les données.

Toute cette évolution, initiée avec les BdD OLAP, constitue les prémisses de l'analyse statistiques de données d'entreprises (incluse dans le *Data Science*), et du design de nouvelles solutions de stockage et d'accès aux données (incluse dans le *Data Engineering*). Les démarches de *Big Data* et les technologies *NoSQL* ont suivi peu après. La figure 8.1 résume cette évolution.

8.1.2 Le problème *CAP*

Les moteurs de BdD relationnelles veillent à maintenir une intégrité forte de leur base. Pour cela, ils s'appuient notamment sur des mécanismes de mise-à-jour atomiques, qui garantissent qu'une séquence de modifications sera exécutée entièrement ou pas du tout (en cas de problème). La BdD sera alors dans l'état final désiré ou restera dans son état initial, et on pourra garantir qu'une lecture de donnée retournera toujours la version cohérente la plus récente de la donnée. Il

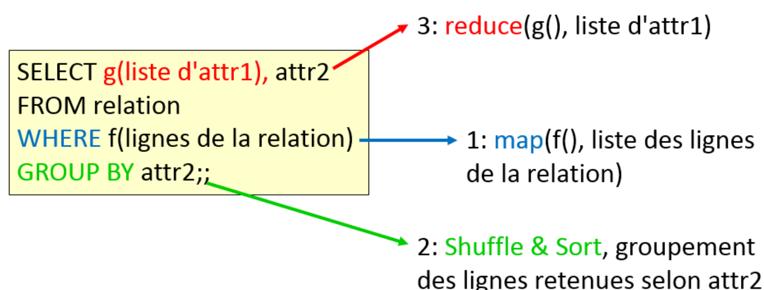

FIGURE 8.4 – Résolution d'une requête SQL type en Map-Reduce

s'agit de la propriété de *consistency*, qui doit être vérifié en permanence.

Mais un autre objectif des moteurs de BdD est que la base soit disponible à tout moment : les mises-à-jour ne doivent pas bloquer la BdD. Des actions légitimes sur la base ne doivent pas empêcher celle-ci de répondre : en l'absence de panne la base doit toujours répondre à chaque requête. Cette deuxième propriété est celle de *(high) availability* (ou *haute disponibilité*) [1].

Malheureusement, quand on distribue de plus en plus une BdD afin de tenir la montée en charge (supporter de plus en plus de requêtes concurrentes sans ralentir), une troisième propriété doit être vérifiée : celle de *partition tolerance* (ou de *résistance au morcellement*). La BdD doit continuer à fonctionner correctement en cas de pertes de messages entre ces composants, ou de dysfonctionnement de certaines parties.

En 2000 puis 2002, Eric Brewer affirme que seuls 2 de ces 3 propriétés peuvent être satisfaites sur un système distribué (puis deux chercheurs du MIT publieront un article visant à démontrer cette affirmation). La figure 8.3 illustre ces trois objectifs de **C***onsistency*, de *(high)* **A***vailability*, et de **P***artition tolerance*, qui sont impossibles à concilier à large échelle et qui constituent le *problème CAP*. Face à ce problème, les BdD NoSQL prennent le parti de privilégier la volumétrie et la vitesse de traitement, quitte à sacrifier des propriétés de cohérence, car maintenir une cohérence forte sur un système à *large échelle* coûte trop cher en temps. C'est donc le *C* de *CAP* qui est oublié en *NoSQL* à large échelle.

8.1.3 Contexte d'émergence et principes fondateurs du *NoSQL*

Les besoins de nouvelles technologies de BdD ont émergé dans les industries de pointe du web, suite à des limitations embarrassantes qu'elles avaient rencontré avec la technologie des BdD relationnelles. Nous pouvons identifier et retracer les principales étapes de cette émergence du *NoSQL*.

Le paradigme *Map-Reduce* pour reproduire des requêtes SQL classique

C'est Google qui a tout d'abord été confronté au stockage et à la gestion de très gros volumes de données, et qui a inventé un système de fichiers distribués nommé *Google File System* (GFS). Google communique sur GFS fin 2003 dans un symposium informatique (de l'ACM). Puis, en 2004 Google présente son principe et sa réalisation de *Map-Reduce*, un mécanisme permettant de distribuer des traitements à appliquer sur les données distribuées à l'aide de GFS. Le terme *Map-Reduce* s'inspire directement des fonctions *map* et *reduce* des langages fonctionnels. La fonction *map* prend deux arguments une fonction f et un ensemble de données E, applique la fonction f à tous les éléments de l'ensemble E, et retourne une liste de résultats. La fonction *reduce* prend

1. En fait, le terme de *haute disponibilité* possède d'autres définitions, dans d'autres contextes, qui le relie au contraire à la résistance aux pannes

aussi deux arguments : une fonction g et la liste d'éléments retournés par la fonction *map*. La fonction g s'applique à cette liste et la "réduit". Par exemple, on peut réduire une simple liste de valeurs numériques à sa somme, ou à son élément maximum ou minimum... Google réalise cette double opération *Map-Reduce* à très large échelle à travers son *système de fichiers distribué* GFS, et pose ainsi les deux premières briques du *data engineering*.

Mais avant d'aller plus loin, tâchons de comprendre pourquoi un mécanisme de *Map-Reduce* est une brique très utile pour interroger des données (alors qu'elle n'est quasiment pas utilisé en calcul scientifique). Dans une BdD relationnelle, une structure de requête classique est de la forme SELECT-FROM-WHERE-GROUPBY. Comme illustré sur la figure 8.4, l'application de la condition du *Where* (1) revient à appliquer sur chaque ligne de la relation la fonction de test booléenne *f(...)*. Puis l'exécution du *Group by* va tout d'abord (2) trier les lignes retenues selon l'attribut *attr2* en regroupant celles présentant la même valeur pour cet attribut, et va ensuite (3) appliquer la fonction *g(...)* sur la liste des attributs *attr1* d'un ensemble de lignes de même attribut *attr2*. Ceci correspond à appliquer une fonction *map(f, lignes de la relation)*, puis à trier et regrouper les lignes retenues selon l'attribut *attr2*, et enfin à appliquer la fonction *reduce(g, liste d'attr1)* aux attributs *attr1* de chaque groupe de lignes.

Une opération *Map-Reduce*, avec une fonction *reduce* appliquée par groupe de résultats de la fonction *map*, permet donc de refaire un "Select...From...Where...Group by...". Sa mise en œuvre sera sans doute plus complexe que l'emploi de SQL, mais sera aussi plus générique : elle fonctionnera même si les éléments stockés dans la base ne sont pas de simples attributs de types scalaires comme dans une BdD relationnelle. Par exemple, si les éléments stockés sont des données structurées complexes, comme des fichiers de données, un SGBD relationnel ne pourrait pas être employé, alors qu'un mécanisme de *Map-Reduce* au dessus d'un système de fichiers distribués sera utilisable.

Hadoop : un premier *Map-Reduce* distribué open-source

La présentation par Google en 2003-2004 de son système de fichiers distribué et de son mécanisme de *Map-Reduce* distribué avait fait une forte impression, mais ces outils restaient propriété et exclusivité de Google. Inévitablement, une implantation libre vit le jour peu de temps après. Hadoop fut développé initialement par Doug Cutting pour mettre au point un moteur de recherche libre. Il repose sur le système de fichier distribué HDFS (*Hadoop Distributed File System*), et sur un mécanisme de *Map-Reduce* distribué exploitant HDFS. L'environnement de développement choisi fur l'écosystème extrêmement portable de Java. Hadoop a été assez rapidement adopté, et des sociétés comme Yahoo ! et Facebook annonçait déjà en 2012 avoir des dizaines de milliers de machines exploitées par Hadoop.

Le développeur d'applications basées sur Hadoop écrit essentiellement les fonction *map* et *reduce*, sous forme de classes Java. Des versions par défaut sont disponibles qui réalisent l'identité (pas de filtrage des données d'entrée, pas de réduction des données filtrées). Le développeur peut aussi écrire quelques classes permettant d'optimiser le code *Map-Reduce* pour les données traitées, comme des *Combiner* ou des *Partitioner* (voir chapitre 6). A l'exécution, les nœuds de données sont transformés en nœuds de traitement, et accueillent des *processus map* et des *processus reduce*. Mais la partie la plus complexe d'Hadoop, qui assure le routage des données de sortie des *processus map* vers les entrées des *processus reduce*, n'a pas à être re-développée. En cas d'utilisation d'une architecture distribuée comme un cluster de PC, ou un *cloud*, cette opération de routage/re-brassage des résultats intermédiaires peut être complexe à implanter. De plus, elle n'a rien à voir avec la logique métier de l'application et le développeur applicatif ne devrait pas s'en préoccuper. C'est donc une opération de spécialiste du calcul distribué qui est fournie clé en main par Hadoop au développeur applicatif.

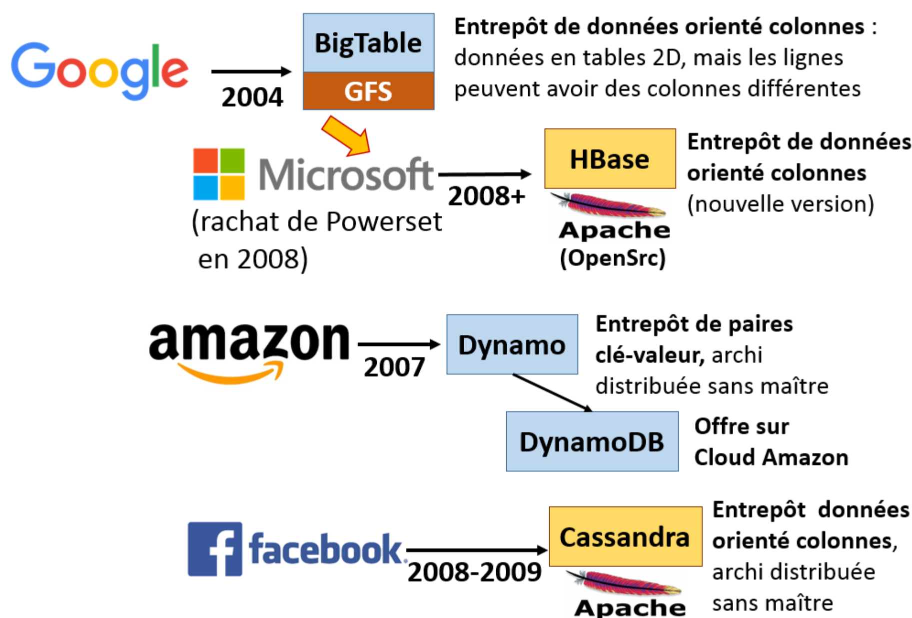

FIGURE 8.5 – Emergence des technologies *NoSQL* en environnement industriel

Les premières BdD NoSQL

A partir d'un système de fichier distribué à large échelle puis d'un mécanisme de *Map-Reduce* au dessus, les premières BdD *NoSQL* ont pu voir le jour.

Google avait tout d'abord développé BigTable en 2004 : une BdD distribuée bâtie sur son système de fichiers distribués GFS et sur son propre mécanisme de *Map-Reduce* pour traiter les requêtes d'interrogation des données (voir figure 8.5). Il s'agissait d'un entrepôt de données *orienté colonnes*. Les données sont encore organisées en lignes et colonnes, mais toutes les lignes n'ont pas forcément les mêmes colonnes (modèle beaucoup plus souple que celui des BdD relationnelles). Par la suite, M. Stack et J. Kellerman en implanteront une autre version pour un moteur de recherche de la société Powerset, qui sera rachetée par Microsoft en 2008, et qui deviendra finalement le logiciel libre HBase, projet important de la sphère *NoSQL*.

En 2007 Amazon développe un entrepôt de paires *clé-valeur* appelé Dynamo, totalement distribué avec une architecture sans maître. Il est resté un produit propre d'Amazon totalement intriqué à l'architecture logicielle d'Amazon. Toutefois, Amazon propose une *offre cloud* d'entrepôt de *clé-valeur* appelé DynamoDB. Evidemment, le stockage des données se fait dans un *cloud*, c'est-à-dire chez Amazon et non pas chez l'utilisateur, ce qui peut rebuter certains clients.

Un peu plus tard (2008-2009) Facebook a développé aussi un entrepôt de données totalement distribué mais orienté colonnes, qui s'inspira des technologies de Google et d'Amazon, et appelé Cassandra. L'objectif initial était de proposer aux utilisateurs une recherche rapide dans les boites aux lettres de leurs comptes. Aujourd'hui Cassandra est très utilisée, par exemple par Twitter.

Différences avec une solution relationnelle standard

Les premières solutions *NoSQL* sont donc apparues en quelques années dans des sociétés du *web*, pour répondre à leurs besoins concrets. La figure 8.6 représente un premier empilement logiciel grossier mais générique, menant à diverses solutions *NoSQL* au dessus d'un large ensemble d'unités de stockage distribuées. Pour l'utilisateur, les différences principales avec une solution relationnelle et SQL standard sont :

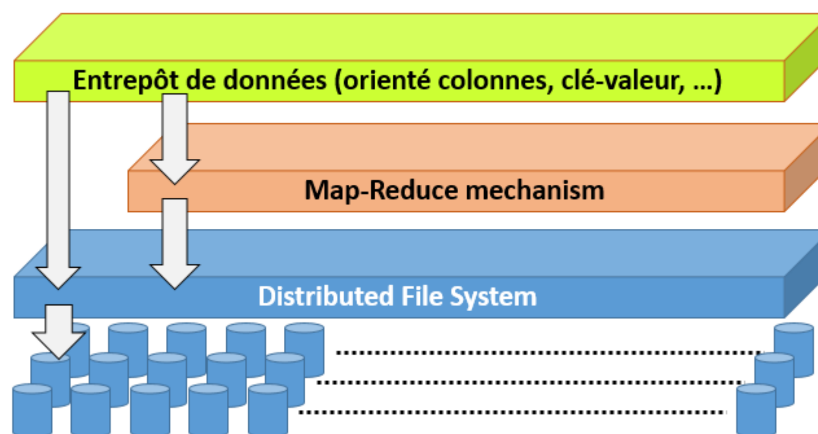

FIGURE 8.6 – Principe (grossier) d'une architecture *NoSQL*

– *Une grande souplesse dans la nature des données stockées*. Par exemple, les lignes d'un entrepôt orienté colonnes n'ont pas forcément les mêmes colones, les valeurs d'un entrepôt clé-valeur peuvent être des fichiers textes au format JSON (voir plus loin). . .

– *L'exploitation d'une très grosse volumétrie dans des temps restant raisonnables*, grâce notamment à un relâchement des contraintes d'intégrité. Par exemple, la mise à jour d'une valeur (d'un document) se fera de manière atomique (en exclusion mutuelle avec des lectures de la valeur), mais une suite de mises à jour sur plusieurs documents ne pourra pas se faire de manière atomique. Il sera donc possible de lire un ensemble de valeurs en cours de mise à jour et légèrement incohérentes. En contrepartie, les mises à jour et les lectures resteront rapides même sur de très gros volumes de données.

– *L'augmentation des performances par la distribution massive* du stockage et des traitement. Par exemple, en utilisant suffisamment de PC pour disposer d'assez de mémoire on peut charger toute la base en mémoire, et accélérer le traitement des requêtes supportées en *NoSQL* (suite des mécanismes des BdD OLAP).

On s'éloigne donc de la rigueur des BdD relationnelles pour plus de souplesse, plus de volume, et plus de vitesse. D'autres différences avec les BdD relationnelles existent aussi sur les stratégies de contrôle des accès aux données (sécurisation des données), et de résistance aux pannes.

8.2 Classification et cibles des technologies *NoSQL*

Une façon simple de classifier les différents moteurs de BdD *NoSQL* est de se focaliser sur le type de structures de données qu'ils manipulent. On peut distinguer 5 grandes classes bien identifiées, plus une 6me qui regrouperait toutes les BdD *NoSQL* spécifiquement conçues pour un type original d'application.

8.2.1 BdD stockant des paires *clé-valeur*

Expérimentalement, il est apparu que la plupart des applications demandaient à lire des données à partir d'identifiants de celles-ci, ce qui créa le besoin pour des BdD *NoSQL* stockant des paires *clé-valeur*. C'est la solution la plus simple, à la fois pour le développeur qui stocke des

associations clé-valeur de manière explicite, et pour le moteur de la BdD qui ne propose que des mécanismes d'accès simplifié aux paires clé-valeur.

Le composant essentiel des moteurs de ces bases est leur *fonction de hachage*, qui distribue et retrouve les paires dans un système distribué. Habituellement ces moteurs sont très efficaces pour retrouver des valeurs à partir de clés. En revanche, peu de fonctionnalités sont disponibles pour écrire des requêtes évoluées. C'est au développeur d'écrire ses routines de travail sur les données extraites, souvent dans un paradigme *Map-Reduce*.

Notons qu'initialement les *valeurs* étaient des blocs binaires, dont la structure interne restait inaccessible au moteur de la BdD. Il n'était donc pas possible de modifier un champ seulement de la valeur, ou de rechercher et filtrer des valeurs à partir d'un de leur champ. Mais progressivement, ces BdD se sont mises à supporter des valeurs au format JSON (voir section 8.4) et à se rapprocher des BdD *NoSQL* orientées documents (voir section suivante).

Redis et *Riak* sont des moteurs de BdD orientés paires de clé-valeurs.

8.2.2 BdD *orientées documents*

Dans les BdD orientées documents on associe des clés à des documents avec une structure hiérarchique, comme des documents au format JSON (voir section 8.4). Les valeurs ne sont donc plus opaques, et les moteurs des BdD peuvent manipuler leurs champs : ajouter, supprimer, modifier, indexer et tester un champ précis.

La motivation initiale pour stocker des documents hiérarchiques vient des pages *web* que doivent souvent retourner les applications *web*. Ces pages *web* contiennent du texte structuré à l'aide de balises (balises XML par exemple), qui peut très bien se représenter au format JSON. Un document JSON associé à une clé peut ainsi contenir et fournir immédiatement une information structurée que l'on obtient à la suite de plusieurs jointures dans une BdD relationnelle. On parle alors de *réaliser la jointure lors de l'écriture*, quand on crée le document structuré et qu'on le stocke dans la base *NoSQL*. L'accès à l'information est ensuite plus rapide, mais le mécanisme est moins souple : si on désire rechercher, filtrer et assembler une autre information que celle stockée dans le fichier associé à la clé, alors l'effort à fournir sera plus important et demandera plus de temps.

Une autre motivation pour accéder aux contenus des documents, est de pouvoir réaliser des index de documents textuels. Toutefois, la réalisation d'index et d'index inversés performants est un problème complexe dont le traitement a engendré des outils *NoSQL* spécifiques (voir section 8.2.4).

Notons que les BdD orientées paires clé-valeur tendent à évoluer pour accepter des documents JSON comme valeurs. Elles tendent ainsi à se rapprocher des BdD orientées documents, et la différence entre ces deux types de BdD *NoSQL* s'atténue.

MongoDB est un exemple très populaire de BdD *NoSQL* orientée documents.

8.2.3 BdD *orientées colonnes*

Les BdD *NoSQL* orientées colonnes peuvent paraître au premier abord plus proches des BdD relationnelles que les précédentes. Comme les BdD relationnelles elles se présentent à l'utilisateur sous forme d'ensembles de tables 2D, où chaque ligne est identifiée par une clé associée à un ensemble de valeurs stockées dans les colonnes de la ligne. Mais elles sont en fait très différentes :

– Alors que les tables des BdD relationnelles sont statiques et que toutes les lignes possèdent les mêmes colonnes, les tables des BdD *NoSQL* orientées colonnes sont beaucoup plus souples.

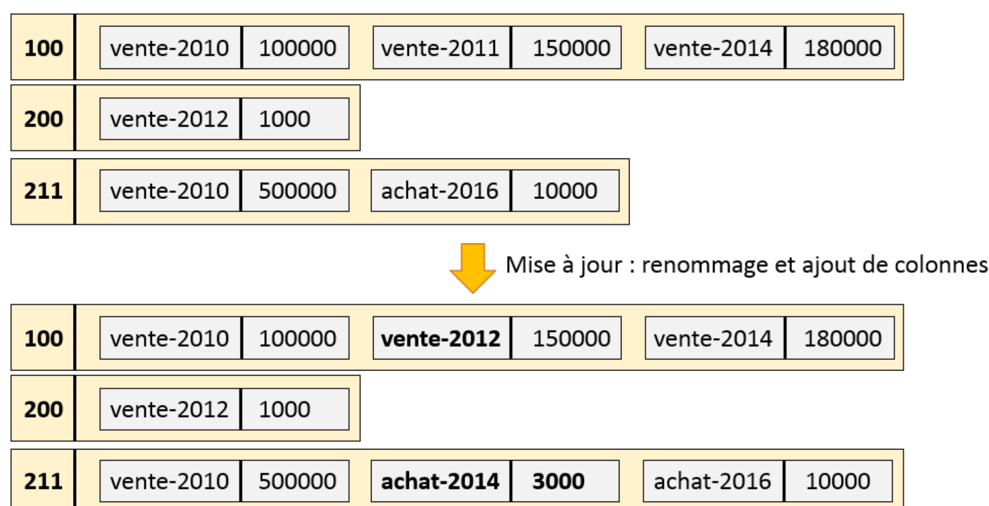

FIGURE 8.7 – Principe des tables des BdD *NoSQL* orientées colonnes

Les colonnes peuvent varier en nombre et en nom d'une ligne à l'autre, et ces nombres et noms peuvent changer dans le temps (voir figure 8.7). Si une colonne n'a pas de sens pour une ligne, alors on ne sera pas obligé de la créer pour cette ligne, et si elle cesse d'avoir un sens on pourra la supprimer. Si un nouveau concept émerge dans ce qui est décrit par une ligne, alors on pourra lui rajouter une colonne. Il ne devrait donc pas y avoir de place perdue lorsque des champs n'existent pas dans certaines lignes (il n'y a pas d'espace mémoire consommé par des valeurs *NULL*, contrairement aux cas des BdD relationnelles).

– Les requêtes possibles sur ces BdD *NoSQL* sont en revanche simples et minimalistes, comparées aux BdD relationnelles. On peut typiquement effectuer des requêtes par clé, avec une condition sur un intervalle de colonnes.

  Ex : chercher les colonnes de la ligne de clé 100, chercher les colonnes de nom compris entre "achat-2010" et "achat-2012" pour une ligne de clé 1000, chercher la colonne "vente" pour les lignes 100 à 200...

En fait, ces BdD *NoSQL* ont notamment été conçues pour stocker des relations de type *one-to-many*, évolutives et aboutissant à de très gros volumes. On en trouve particulièrement sur le *web* où une donnée peut-être reliée à un million d'autres.

Même si les colonnes ne sont pas toutes les mêmes pour chaque ligne, cette approche en lignes et colonnes sans case vide permet un stockage puis des accès très efficaces. Il est parfois nécessaire pour une BdD de réorganiser son stockage sur disque suite à des modifications du nombre ou des noms de ses colonnes, mais globalement le stockage reste très efficaces. En contrepartie, les requêtes supportées efficacement sont limitées.

HBase et *Google Big Table* sont des exemples très connus de BdD *NoSQL* orientées colonnes.

8.2.4 BdD réalisant des *index inversés*

La construction d'un index d'un ensemble de documents, et plus encore d'un index inversé (liste des mots associés à la listes de leurs emplacements dans les documents sources) est nécessaire pour que les moteurs de recherche puissent répondre rapidement aux requêtes qui leurs sont faites. Mais habituellement, l'index inversé est beaucoup plus gros que l'ensemble de données

initial ! Le stockage de l'index justifie/impose donc de le compresser. Cependant, il faut éviter de perdre du temps à le décompresser (par parties) chaque fois que l'on veut s'en servir. Il faut utiliser un format de compression permettant à la fois de réduire significativement le volume de l'index, et de le décompresser à la volée très rapidement, voire de l'exploiter en mode compressé.

L'index est donc une grosse structure à calculer, à compresser, à mettre à jour quand de nouveaux documents viennent compléter la BdD, et à décompresser très rapidement à la demande. Des algorithmes spécifiques et très optimisés sont donc nécessaires, et sont apportés par une BdD *NoSQL* réalisant des index inversés, contrairement à une simple BdD orientée documents.

Elasticsearch est un bon exemple d'outils du monde *NoSQL* qui réalise efficacement des index inversés.

8.2.5 BdD orientées *graphes*

Les BdD *NoSQL* orientées graphes sont destinées à stocker et à analyser des données organisées en graphes, comme les réseaux sociaux. Ces BdD proposent divers algorithmes optimisés de parcours et d'analyse de graphes, et stockent leurs données de manière à permettre une exécution très rapide de ces algorithmes (parfois sous forme de graphes de structures de données référencées entre elles par adresses). Ce qui rend leur utilisation simple, confortable et très efficace pour saisir et analyser des graphes.

Une comparaison aux BdD relationnelles s'impose. En théorie le modèle relationnel de Codd permet de représenter et de stocker tout type de données, et bien sur il permettrait de représenter les nœuds d'un graphe et les arcs entre ses nœuds. Mais il les stockerait dans des tables 2D contenant des clés et des valeurs (les attributs), et non pas des adresses d'autres nœuds. Tout parcours de graphe se ferait alors par exécution de nombreuses opérations de *jointures* coûteuses, au lieu de simplement suivre des enchaînements d'adresses de données. En résumé, une BdD relationnelle serait lente pour analyser un graphe.

Une comparaison aux BdD *NoSQL* orientées clé-valeur est également intéressante. Les BdD clé-valeur n'utilisent qu'un minimum d'espace pour stocker chaque donnée et ne supportent que des requêtes élémentaires. Ces *entrepôt de données* peuvent ainsi stocker des ensembles de données de très grande volumétrie, et distribuer le traitement des requêtes qui sont naturellement parallélisables. Ces caractéristiques intéressent particulièrement les *majors* du *web* comme Google. Mais en contrepartie, l'insertion d'un graphe (comme un réseau social) dans un entrepôt de paires clé-valeur est fastidieux et à la charge de l'application, tout comme les opérations d'analyse du graphe qui ne sont absolument pas fournies ni facilités par l'entrepôt. Donc, à moins d'avoir vraiment des volumes de données titanesques, il est nettement préférable d'utiliser une BdD *NoSQL* orientée graphes pour stocker et analyser des graphes.

Neo4J est un exemple de BdD *NoSQL* orientée graphes.

8.3 Positionnement des annuaires LDAP

La première standardisation du protocole LDAP date de 2002 (fiche RFC-3377), et il découle d'un protocole encore plus ancien appelé X500. Les annuaires LDAP sont donc antérieurs au mouvement *NoSQL*, et constituent une alternative au BdD relationnelles pour représenter, stocker et exploiter des données. De plus ils sont très utilisés, il est donc intéressant d'étudier leur positionnement vis-à-vis de l'approche *NoSQL*. Sont-ils proches ou différents des solutions *NoSQL* ?

Les annuaires LDAP sont conçus pour stocker des informations *hiérarchiques* dans un arbre, avec des concepts de données parentes et filles pour chaque nœud de l'arbre. Ils sont également optimisés en lecture, donc conçus pour des données beaucoup plus souvent lues que mises à jour. En

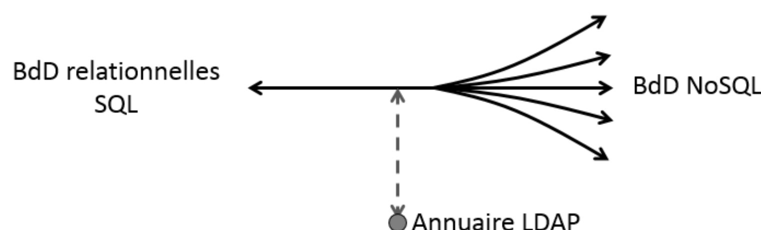

FIGURE 8.8 – Positionnement des annuaires LDAP : un modèle et un outil à part

fait ces *annuaires* sont typiquement utilisés pour stocker des descriptions de systèmes d'information, naturellement hiérarchiques, et accedées par de nombreuses applications mais plus souvent en lecture qu'en écriture. On peut ainsi décrire facilement des groupes d'utilisateurs, puis chaque utilisateur avec son login et son certificat, ainsi que des branches de réseaux, et par exemple, pour chaque branche ses machines, et pour chaque machine ses installations de logiciels et de licences...

A première vue on pourrait penser que l'on est proche d'une BdD *NoSQL* orientée documents, et stockant des données structurées au format JSON (voir section 8.4). Mais 3 différences essentielles existent :

– Les premier mécanismes de contrôle d'accès aux données (*Access Control List*) des BdD *NoSQL* étaient beaucoup moins riches que ceux du protocole LDAP. Un annuaire LDAP permet facilement de restreindre les accès d'un utilisateur (ou d'un groupe) à certaines parties seulement de son arborescence. Les BdD *NoSQL* avaient initialement tendance à faire une hypothèse de *trusted user*, c'est-à-dire à supposer que le contrôle des accès étaient fait en amont, et que seuls des utilisateurs ayant droit de consulter les données pouvaient se trouver sur le réseau d'accès à la BdD. Heureusement cette faiblesse de contrôle d'accès des BdD *NoSQL* tend à s'atténuer.

– Les annuaires LDAP reposent sur des principes et des mécanismes anciens, et leurs implantations sont peu performantes : des opérations simples sont réalisées de manière complexes, à l'opposé des principes du *NoSQL*.

– Enfin, les annuaires LDAP imposent un schéma de BdD *hiérarchique en arbre*. Certes ce schéma permet de définir assez librement les informations stockées dans les nœuds de l'arbre, mais certains le considèrent quand même contraignant vis-à-vis de solutions *NoSQL* qui tentent de ne pas imposer de schéma ou de type de données.

Les annuaires LDAP ne sont donc pas des BdD relationnelles, mais ne sont pas non plus des solutions *NoSQL*. Comme illustré sur la figure 8.8 ils occupent une position à part, et sont maintenant d'une technologie un peu ancienne.

8.4 Format de fichiers de données *JSON*

Remarque : Cette présentation du langage JSON (*JavaScript Object Notation*) est directement inspirée du site web officiel de ce langage : `http://www.json.org`.

JSON est un format très simple d'échange de données en texte, que quasiment tous les langages de programmation peuvent implanter d'une façon ou d'une autre. En fait, il existe des *valeurs*, et

FIGURE 8.9 – Grammaire des *collections* ou *objets* de JSON

FIGURE 8.10 – Grammaire des *tableaux* de JSON

FIGURE 8.11 – Grammaire des *valeurs* de JSON

seulement deux types de structures : les *collections* et les *tableaux*.

– Une *collection (ou objet) de couples nom-valeur* commence par une "{", contient des paires nom-valeur séparés par des "," et se termine par une "}". Une paire nom-valeur est composée d'une string (le nom) et d'une valeur séparées par " :". Voir figure 8.9.

– Un *tableau de valeurs* commence par un "[", contient une suite de valeurs séparées par des "," et se termine par une "]". Voir figure 8.10.

– Une valeur peut être terminale : une string, un nombre, un booléen ou "null", ou bien être une collection (ou objet) de paires nom-valeur ou un tableau de valeurs. Voir figure 8.11.

A titre d'exemple, voici une donnée structurée stockée dans un fichier texte au format JSON et contenant 1 collection de 3 paires nom-valeurs (dont 2 avec le même nom "livre"). Ensuite, chaque valeur est elle-même une collection de paires nom-valeur dont la valeur est un tableau de petites collections. . .

```
{
 "livre" : {
    "Vue de l'esprit" : [ {"auteurs" : ["Hofstadter", "Dennet"] },
                          {"éditions" : "InterEditions"},
```

```
                        {"année" : 1987}
                      ],
    "Eagle" : [ {"auteurs" : ["Kidder"] },
                {"éditions" : "Flammarion"},
                {"année" : 1982}
              ]
    },
 "film" : {
    "2001 odyssée de l'espace" : [ {"réalisateurs" : ["Kubrick"] },
                                   {"année" : 1968}
                                 ]
    },
 "livre" : {
    "Cosmos" : [ {"auteurs" : ["Sagan"] },
                 {"éditions" : "Mazarine"},
                 {"année" : 1981}
               ]
    }
}
```

Une telle donnée structurée au format JSON peut être insérée très naturellement dans une BdD *NoSQL* orientée documents. Mais bien entendu, il existe d'autres façons de structurer ces informations dans un format JSON.

8.5 Exercices

8.5.1 Origine des technologies *NoSQL*

1. Dans quel milieu sont nés les premiers environnements *NoSQL* ?
2. Quels besoins spécifiques ont fait émerger une technologie de BdD autre que celle des (traditionnelles) bases relationnelles ?

8.5.2 Données stockées dans une base *NoSQL*

1. Pourquoi dit on que le *Join* est à faire lors de l'écriture d'une collection de données dans une base *NoSQL* ?
2. Quelle sont les conséquences du manque de *schéma de base* dans les bases *NoSQL* ?

Bibliographie

[1] G. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.

[2] B. Azarmi. *Scalable Big Data Architecture*. Apress, 2016.

[3] R. Bruchez. *Les bases de données NoSQL et le Big Data*. Eyrolles, 2ème edition, 2016.

[4] R. Bruchez and M. Lutz. *Data science : fondamentaux et études de cas*. Eyrolles, 2015.

[5] B. Chapman, G. Jost, R. Van Der Pas, and D. Kuck. *Using OpenMP*. The MIT Press, 2008.

[6] K. Chodorow. *MongoDB, the Definitive Guide*. O'Reilly, 2ème edition, 2013.

[7] K. Dowd and Ch. Severance. *High Performance Computing*. O'Reilly, 2nd edition, 2008.

[8] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 1999.

[9] J.L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31 :532–533, 1988.

[10] H. Karau, A. Konwinski, P.Wendell, and M.Zaharia. *Learning Spark*. O'Reilly, 1st edition, 2015.

[11] H. Karau and R. Warren. *High Performance Spark*. O'Reilly, 1st edition, 2017.

[12] M. Kirk. *Thoughtful Machine Learning with Python*. O'Reilly, 2017.

[13] P. Lemberger, M. Batty, M. Morel, and J-L. Raffaelli. *Big Data et Machine Learning*. Dunod, 2015.

[14] D. Miner and A. Shook. *MapReduce Design Patterns*. O'Reilly, 2013.

[15] T. White. *Hadoop. The definitive Guide*. O'Reilly, 3rd edition, 2013.

[16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets : A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.