Préparation à l'examen de *NoSQL*

Wéry Benoît

6 décembre 2017

Chapitre 1

Vrai ou Faux

1. *Dans une entreprise, les données vivent souvent plus longtemps que les logiciels.* Vrai

   Une entreprise *utilise* des logiciels et *stocke* des données en veillant à garder le plus d'indépendance possible entre ces deux éléments. Ainsi, un changement d'implémentation de la structure des données ne devrait pas affecter le bon fonctionnement du logiciel.

   Là où les données doivent persister dans le temps, un logiciel peut être amené à être remplacé (par obsolescance, optimisation,...)

2. *Dans une entreprise, les logiciels vivent souvent plus longtemps que les données.* Faux

3. *Le passage du relationnel au NoSQL se fait généralement au profit d'une diminution des garanties relatives à la consistance des données.* Vrai

   Le NoSQL offre bien des performances, tant au niveau de la gesion et de l'accès aux données (distribution cluster, dispatching des requêtes, ...) que de la robustesse aux crash par exemple MAIS tout cela se fait en acceptant une perte de consistance des données.

   En pratique, la consistance "instantanée" est difficilement atteignable (voir impossible dans certains cas) mais les données seront tôt ou tard mises jour, ce qui induit une consistance "à terme".

   Rem : pour plus de détails sur les types de consistances,... voir à partir de question 54

4. *Le passage du relationnel au NoSQL se fait généralement au profit d'une diminution de la quantité de données stockables.* Faux

   ... stockables VS stockées...

   Le NoSQL a émmergé d'un besoin de stocker de nouveaux formats de données (ex : stocker directement des objets plutôt que de devoir retrouver leur informations indépendantes pour le reconstruire), malgré la *puissance* et la *stabilité* des base de données relationnelles, celles-ci ne sont plus suffisantes dans certaines applications.

Le modèle NoSQL, est adapté à une distribution des données sur différentes machines/serveurs (archi *cluster*). De ce fait, l'espace de stockage est donc ajustable facilement, la quantité de données stockables peut être augmentée en ajoutant de nouvelles machines. (Contrairement au modèle relationnel pour lequel la DB se trouve sur une machine unique et dont l'espace de stockage est donc difficilement agrandissable...)

5. *Le passage du relationnel au NoSQL se fait généralement au profit de l'abandon de la possibilité de lire des données de manière concurrente.* Faux

   Le modèle relationnel permet d'accéder de manière concurrente aux données (en R/W) via l'utilisation de **transactions**, les données sont stockées de façon consistante et persistante.

   Le modèle NoSQL, quant à lui, permet un accès concurrent aux données MAIS elles ne sont pas garanties consistantes (ou du moins, la consistance est plus difficile à garantir et généralement elle s'obtient "à terme"). De plus, les données d'une BDD pouvant être stockées sur des serveurs différents (contrairement au relationnel -> 1 machine/BDD), une architecture par réplication de données permet d'améliorer l'accès concurrent aux données, en dispatchant les requêtes entre les machines.

6. *Le passage du relationnel au NoSQL se fait généralement au profit d'une diminution des garanties relatives à la persistance des données.* ???

   "la gestion de la persistance des données réfère au mécanisme responsable de la sauvegarde et de la restauration des données. Ces mécanismes font en sorte qu'un programme puisse se terminer sans que ses données et son état d'exécution ne soient perdus". A VERIFIER

7. *Tout comme pour le relationnel, l'organisation des données en NoSQL suit un modèle mathématique rigoureux.* Faux

   Là où le relationnel se base sur des modèles mathématiques standards pour définir la structure de la BDD, le NoSQL ne suit pas de schéma (ex : possibilité d'ajout de champs sans contrôle).

   De cette façon, on obtient une structure **flexible** avec une BDD qui n'est pas figée dans le temps, ce qui est utile dans le cas de *prototypages*.

8. *Le NoSQL est particulièrement adapté à des traitements de données de type OLTP* Faux

   Online Transactional Processing (OLTP)... modèle qui utilise des données dans un but purement transactionnel -> gestion

   Une transaction doit faire appel à des données consistantes, ce qui n'est pas garantit dans le cas du NoSQL ...

9. *Le NoSQL est particulièrement adapté à des traitements de données de type OLAP.* Vrai

   Online Analytical Processing (OLAP)... modèle qui utilise les données dans un but d'analyse et de prédictions -> statistiques

10. *Toutes les bases de données de type NoSQL satisfont les propriétés ACID.* Faux

    Proriétés ACID...
    · **A**tomicity : une transaction fait tout ou rien

- **C**onsistency : la BDD change d'un état valide vers un autre état valide, les données sont constamment à jour
- **I**solation : une transaction doit être exécutée sans avoir connaissance de l'existance des autres
- **D**urability : une transcation validée et confirmée est sotckée, durable dans le temps

De telles conditions correspondent au modèle relationnel.

Un autre ensemble de conditions, le modèle BASE, permet de gérer les *pertes de consistance* en maintenant la fiabilité et correspond donc à l'approche NoSQL :

- **B**asically **A**vailable : le système renverra toujours une réponse, même si la donnée n'est pas à jour ou qu'il s'agit d'un message d'erreur.
- **S**oft state : l'état change constamment au cours du temps (même lorsqu'il n'y a pas d'inputs) pour essayer de se stabiliser entre les serveurs de la BDD
- **E**ventual consistency : le système finira tôt ou tard par être consistant

11. *Un système distribué peut toujours garantir la consistence des données sur tous ses nœuds.* Faux

    Les données peuvent être différentes d'un noeud à l'autre en fonction des mises à jour qui ont été faites ou non.

12. *Le mouvement de l'Open Data consiste à fournir librement des données récoltées pour permettre à la communauté de les analyser.* Vrai

    Il s'agit de données collectées par de grands groupes (Nasa, Nsa, Union Européenne,...) et qui sont rendues publiques. On peut considérer cela comme une situation "win/win" puisque les utilisateurs ont librement accès à une quantité gigantesque d'informations et qu'ils peuvent eux-mêmes en faire des analyses, utiles dans ce cas-ci pour les groupes qui fournissent les données.

13. *Le mouvement de l'Open Data à créer des logiciels open source permettant d'analyser des données massives (big data).* ???

    ??? sens phrase ???

    Le **Big Data** consiste en l'**augmentation du volume de données** manipulées (ex : données scientifiques, réseaux sociaux, opérateurs tél., indicateurs écono et socio,...).

    Il y un réel *défi* afin de gérer et traiter cette énorme quantité de données et pour y parvenir, il faut se tourner vers d'autres modèles de BDD que le relationnel car il n'est plus adéquat.

14. *Google File System (GFS) est un moteur de base de données NoSQL.* Vrai

    GFS est un **système de fichiers distribué** optimisé pour fonctionner sur un *cluster*. Il est utilisé pour gérer des fichiers de taille importante (découpés en blocs et stockés sur différentes machines) qui sont rarement supprimés ou réécrits. La plupart du temps, les accès portent sur la lecture de larges zones ou des ajouts en fin de fichiers. Pour chaque fichier, il en existe une/des copie(s) pour subdvenir aux éventuelles pannes de serveurs.

15. *Apache Hadoop est une implémentation propriétaire de MapReduce, commercialisée par Oracle.* Faux

    Il s'agit d'une implémentation **libre** de MapReduce (en Java)

Rappel : MapReduce... il s'agit d'un *modèle de programmation* dont le but est de traiter et générer de grandes quantités de données. Il se base sur des algorithmes parllèles et distribués sur un cluster. Une implémentation de MapReduce se base sur deux fonctions :

(a) MAP effectue un **traitement sur une liste** (tri, filtre,...)

(b) REDUCE regroupe les données en un résultat.

16. *L'intégration des données dans une seule base pour les partager entre plusieurs applications permet d'obtenir les meilleures garanties en terme de préservation de l'intégrité des données.*

17. *Limiter l'accès aux bases de données d'une entreprise à une seule application qui offre une API aux autres permet de rendre un changement de leur structure plus facile.*

18. *Une base de données NoSQL (clé-valeur, document et colonne) stocke des agrégats que l'on peut comparer aux tables du modèle relationnel* Vrai

Là où le modèle relationnel utilise des tables avec des rangées pour stocker les données, le modèle NoQSL utilise des "agrégats" qui sont des **unités d'information** qui peuvent être traîtées, stockées et échangées de façon atomique.

Ces unités de données sont **plus complexes** et **plus structurées** que leurs homologues du modèles relationnel, qui sont de simples tuples. Les possibilités de traîtement qui en résultent sont donc plus complètes.

19. *Une collection de paires clé-valeur peut être assimilée à une table relationnelle à deux colonnes dont la colonne représentant les clés est la clé primaire de la table.* Vrai

Les paires "clé-valeur" sont identifiables au moyen de la clé. L'avantage remarquable de ce système est que **la clé n'a pas de forme stricte**!!

La clé est un *string* [...]

De plus, la clé n'a pas conscience du format de la valeur qu'elle refère car celle-ci est de type **blob (= Binary Large OBject)**. Autrement dit, la valeur peut être quelconque (ex : *sting, JSON, XML, objet, ...*), c'est à l'application de gérer le format de chaque valeur. De ce fait, le modèle n'a ni schéma, ni structure pour le stockage et il est donc impossible de récupérer une partie de la valeur (-> tout ou rien).

Les avantages de ce modèle sont : efficacité de recherche et la simplicité

20. *Supprimer la valeur associée à une clé est l'une des trois opérations de base que l'on peut réaliser sur une base de données clé-valeur.* Vrai/ Faux ???

Les trois opérations de base sont :

(a) **Récupérer** une valeur à partir de sa clé [Read(key)]

(b) **Définir** la valeur d'une clé [Create/Update(key, value)]

(c) **Supprimer** une clé [Delete(key)]

On ne sait pas supprimer la valeur d'une clé sans supprimer la clé elle-même.

21. *Dans une base de données clé-valeur, il est généralement prévu de rechercher toutes les clés dont les valeurs satisfont une certaine propriété.* Faux

    La clé étant de type *string*, elle est utilisé pour définir des formats/patterns utilisés pour des recherches efficaces.

22. *Il est possible d'imposer des contraintes sur les domaines des valeurs des paires clé-valeur d'une base de données clé-valeur.*

    COMPLETER [...]

23. *Distribuer les données sur un cluster de machines fait partie des éléments mis en place dans le monde NoSQL.* Vrai

    Le modèle NoSQL se prête bien à la distribution des données (agrégats) sur différentes machines physiques (= clusters, architecture en noeuds).

Intérêts du cluster...

- gestion de grandes quantités de données, *scale out* : la capicité de stockage peut être facilement augmentée en ajoutant de nouvelles machines et ce "à chaud"
- meilleur trafic R/W : la répartition de la charge permet de fluidifier le trafic des requêtes
- résister à des ralentissements, pannes réseaux : les machines peuvent être inscrites chez différents Fournisseurs d'Accès Internet (FAI) et le réseau est de ce fait moins sensible à une éventuelle panne chez un FAI ou une saturation

24. *Il est possible de faire du sharding de données pour une base de données se trouvant sur une machine unique.* Vrai ? (via machine virtuelle) -> mais complètement inutile

    Le sharding est une technique de distribution des données qui consiste à répartir la charge entre différents serveurs de façon horizontale (= [...]).

En sharding, les données **ne sont pas répliquées** et chaque serveur est accessible en **R/W**. Pour optimiser la distribution, on veillera à rassembler sur un même noeud les données qui sont *accédées ensemble* (probabilité d'accès commun et localisation géographique). C'est le client qui a connaissance de la répartition des données et qui est donc responsable d'interroger la bonne machine.

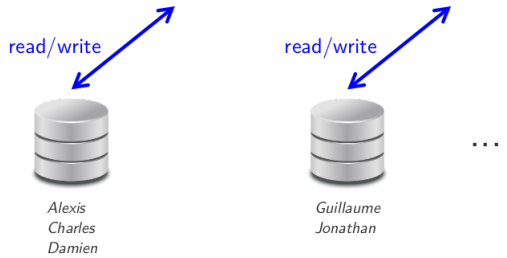

FIGURE 1.1 – Distribution des données : modèle sharding [1]

Avantages :
✔ fraction de la DB pour accélérer son traîtement et la rendre plus facile à gérer
✔ avantages cluster (résistance aux pannes, *scale out*, ...)

Inconvénients :
✘ définir bonne répartition des données -> risque de surcharge d'un serveur
✘ pas de résilience (ni en lecture, ni en écriture) -> si un serveur tombe, une partie de la DB n'est plus du tout accessible

25. *Le sharding permet de récupérer les données en cas de corruption grâce à un stockage redondant de ces dernières sur plusieurs serveurs pouvant être physiquement à des endroits différents.* Faux

En sharding les données ne sont pas répliquées contrairement aux modèles de réplications *master/slave* ou *peer-to-peer*.

26. *Réplication de données et sharding sont incompatibles.* Faux

Il est possible de combiner les techniques de *sharding* et *replication* ce qui offre les avantages de résilience (R, W ou les deux) et de répartition de la DB.

On peut combiner le sharding avec :
- le *master/slave* : plusieurs maîtres (responsable chacun d'une partie de la DB) ou rôle mixtes (esclave pour certaines données et maîtres pour d'autres)
- le *peer-to-peer* : fraction de la DB (sharding) et réplication sur N noeuds

27. *La réplication master-slave offre la propriété de résilience à la lecture.* Vrai

Dans le modèle de réplication master/slave, on distingue deux types de noeuds

(a) le **master** : responsable des données et de leur mise à jour, il est accessible en R/W
(b) les **slaves** : répliques complètes (!!) du maître, accessibles seulement en lecture

Les requêtes des clients sont redirigées suivant que ce soit pour de la lecture ou de l'écriture. Comme le master est le seul accessible en lecture, il doit communiquer ses modifications aux esclaves pour les mettre à jour.

Ce modèle possède la propriété de *read resilience*, càd que si le maître crash, le système sera toujours accessible en lecture via les esclaves. De plus, il est possible d'élir (manuellement ou automatiquement) un esclave comme nouveau maître si celui-ci tombe.

Avantages :
✔ robustesse en cas de crash et *read resilience*
✔ fluidification du trafic pour les requêtes de lecture
✔ avantages cluster (résistance aux pannes, *scale out*, ...)

Inconvénients :
✘ pas adapté lorsqu'il y a beaucoup d'écritures -> surcharge maître
✘ nécessite plus d'espace de stockage puisque chaque noeud est une copie complète du maître

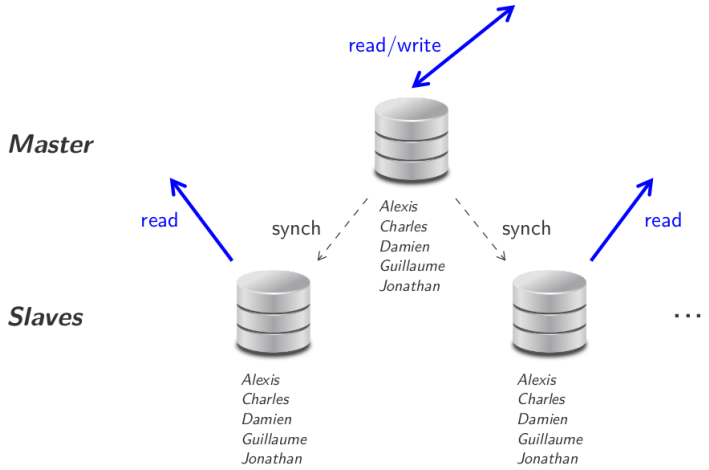

FIGURE 1.2 – Distribution des données : modèle de réplication master/slave [1]

✘ les valeurs lues par plusieurs utilisateurs peuvent être différentes par inconsistence -> la cohérence des données n'est pas garantie, puiqu'il faut que le maître ait synchronisé ses modifications avec les esclaves

28. *En utilisant une réplication master-slave, les données deviennent complètement inaccessibles une fois que le master tombe.* Faux

Par la propriété de *read resilience*, les données seront toujours accessibles en lecture si le maître crash (avec risque de perte d'information si l'esclave n'était pas à jour par rapport au maître). Elles ne seront disponibles en écriture **que si** un esclave est désigné pour remplacer le maître (-> pas *write resilience*)

29. *La consistence des données est plus compliquées à garantir avec une réplication master-slave qu'avec une réplication peer-to-peer.* Vrai

Dans le modèle de réplication *peer-to-peer*, les noeuds sont tous égaux, ils sont **accessibles en lecture et en écriture**. A chaque écriture sur un noeud, tous les autres doivent être mis à jour, ce qui peut créer des conflits d'écriture concurrente si une même donnée est modifiée à deux endroits différents en même temps.

Avantages :

✔ robustesse en cas de crash et *complete resilience*

✔ fluidification du trafic pour les requêtes de lecture ET d'écriture

Inconvénients :

✘ pas adapté lorsqu'il y a beaucoup d'écritures -> lourdeur synchronisations

✘ nécessite plus d'espace de stockage puisque chaque noeud contient toute la DB

✘ risques de conflits d'écritures concurrentes

✘ peu évoluable -> l'ajout d'un nouveau noeud impose de le connecter à TOUS les autres noeuds existants

30. *La consistence des données est plus compliquées à garantir avec une réplication peer-to-peer*

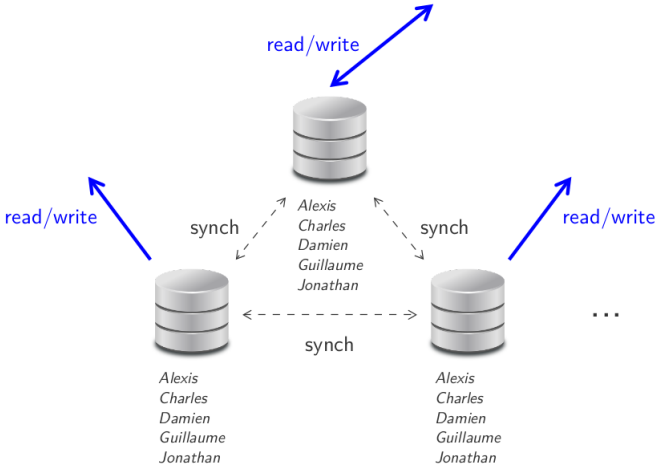

FIGURE 1.3 – Distribution des données : modèle de réplication peer-to-peer [1]

qu'avec une réplication master-slave. Faux

En master/slave, les modifications peuvent **seulement** être effectuées via le maître. Il y a donc des risques d'inconsistence des données le temps que les esclaves soient mis à jour MAIS les modifications seront partout les mêmes.

En peer-to-peer, des modifications peuvent être faites **sur n'importe quel noeud** qui communique ensuite à tous les autres ses modifications. Des écritures concurrentes sur différents noeuds peuvent donc engendrer des problèmes d'inconsistence de données qui sont plus difficile à vérifier. (Rem : une solution serait de synchroniser tous les noeuds sur une même date/heure et d'utiliser un serveur dédié pour stocker les informations de mise à jour des données. Ainsi, on peut facilement déterminer la plus récente)

31. *En utilisant une réplication master-slave, une lecture sur le master assurera toujours d'obtenir les données les plus récentes* Vrai

Puisque le master est **le seul noeud accessible en écriture**, ses données sont les plus à jour (contrairement aux esclaves qui doivent être synchronisés).

32. *Les buckets de Riak permettent de segmenter les données en plusieurs collections d'agrégats.* Vrai

Riak est un **moteur NoSQL décentralisé/distribué** (= Système de gestion de base de données SGBD) orienté clé-valeur. Il est très bon pour monter en charge, càd augmenter le volume de stockage, et a une haute tolérance aux pannes (peut enlever facilement un noeud en panne sans perte d'intégrité des données).

Ce SGBD stocke les clés dans des ***buckets***, qui agissent comme des **espaces de noms** pour les clés. Autrement dit deux clés peuvent porter le même nom si elles se trouvent dans des buckets différents.

Dans le modèle clé-valeur, lorsqu'on fait une requête sur une clé, toute la valeur est retournée et il n'est pas possible de cibler certaines parties (gauche fig[**?**]). Le système de buckets remédie d'une certaine façon à ce problème en permettant la **segmentation des données** au sein d'un même bucket (droite fig[**?**]). Ainsi, un ensemble logique d'informations est identifé par un coupe

Bucket –> *Key* unique et il est possible de rechercher toutes les informations qui correspondent à une même clé.

Dans l'exemple, de la figure [**?**], les données données du profil de l'utilisateur et les informations de sa session ont été séparées car il est plus intéressant de pouvoir y accéder séparemment. Chaque regroupement de donnée est alors identifé par une clé générique (elle portera le même nom dans tous les buckets, qui eux sont uniques), par exemple *"sessionID_ userProfile"*. Cette clé, porteuse d'information grâce au formatage particulier de son nom, permettra ensuite d'accéder directement à un ensemble d'informations relatives à l'utilisateur, au lieu de récupérer l'ensemble de ses données.

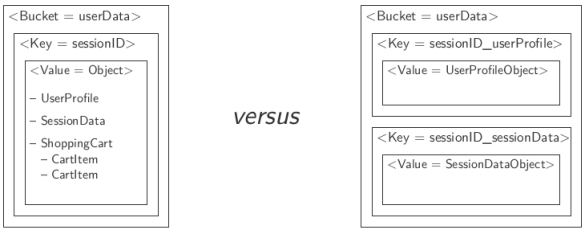

FIGURE 1.4 – Riak - système de *buckets* [1]

Avantages :

· pratique lorsqu'on sait que les données d'une même valeur ne seront jamais utilisées en même temps -> évite de récupérer l'ensemble du contenu de la valeur

33. *On ne peut pas stocker des arbres binaires comme valeurs avec Redis.* Faux

Redis est un moteur de base de données en mémoire, utilisé pour permettre une manipulation la plus rapide possible des données. Il permet de stocker **uniquement 5 types de valeurs** ;

(a) CHAINE DE CARACTÈRES, NUMÉRIQUES OU BINAIRES
(b) LISTE (DE CHAINES) : ajout/retrait à gauche ou à droite
(c) ENSEMBLE DE CHAINES NON TRIÉ ET SANS DOUBLONS
(d) HACHAGE (DICTIONNAIRE) NON HIÉRARCHIQUE
(e) ENSEMBLE TRIÉ AVEC ASSOCIATION D'UNE NOTE PAR ÉLÉMENT

Il n'est pas possible de créer un arbre binaire à partir d'une des structures ci-dessus.

34. *Redis garantit la persistance de données.* Faux

De base, Redis stocke les données en mémoire (cache ou virtuelle) uniquement. Une fois le serveur quitté, **toutes les données sont perdues**.

Cependant, il est de possible de configurer une sauvegarde régulière sur le disque et de la recharger.

35. *Les bases de données orientée colonnes optimisent le stockage disque pour des tables qui contiennent de nombreuses lignes.* Vrai

Dans le modèle orienté-colonnes, différentes techniques de compressions existent pour réduire la taille de stockage des fichiers. Ces méthodes se basent sur le type de données de chaque

colonne et sur le nombre de lignes qui ont une valeur est identique. Ci-dessous, trois exemples de mécanismes de compressions :

(a) RUN-LENGTH ENCODING : utilisé lorsque beaucoup de données similaires sont regroupées (triées).

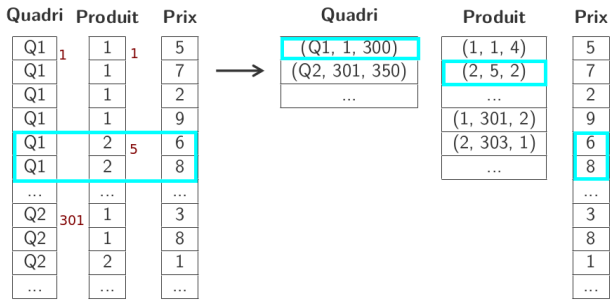

FIGURE 1.5 – Technique de compression "*Run-Length Encoding*" [1]

Les données identiques sont regroupées et ce sur différents niveaux. Pour éviter d'écrire sur le disque X fois la même donnée, on utilise alors un **tuple *(Valeur, Adresse, Nombre)*** où l'adresse correspond à la ligne du changement de valeur. Dans l'exemple (Fig [**?**]), la valeur Q1 commence à partir de la ligne 1 et est répliquée 300 fois. En passant au niveau inférieur, on peut voit que la valeur 2 (associée à Q1) commence à la ligne 5 et est répliquées 2 fois.

A l'étage le plus haut, il y aura donc autant de tuple qu'il y a de valeur différentes et à l'étage le plus bas, il y aura autant de valeur qu'il y a de lignes initialement.

En repartant de la colonne "Prix", on peut alors retrouver le "Produit" et le "Quadri" qui sont associés à une valeur grâce au tuples.

(b) BIT-VECTOR ENCODING : utilisé lorsqu'il n'y a beacoup de lignes MAIS peu de valeurs différentes (elles ne doivent pas être triées).

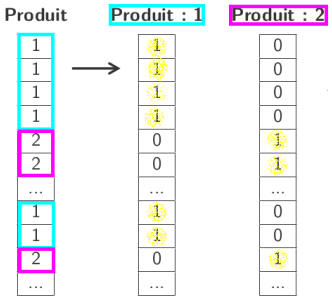

FIGURE 1.6 – Technique de compression "*Bit-Vector Encoding*" [1]

Chaque **valeur est représentée par un vecteur** dont la longeur est égale au nombre de ligne de la colonne de départ. Dans ce vecteur, on remplace la valeur associée par un 1 et les autres par 0. On ne stocke donc plus que quelques (= nombre de valeurs différentes) vecteurs de n bits pour représenter toute l'information au lieu d'un seul gros vecteur, dont la taille dépend des valeurs à encoder (sur x bits chacune).

(c) DICTIONARY ENCODING : utilisé lorsque des motifs de valeurs se répètent.

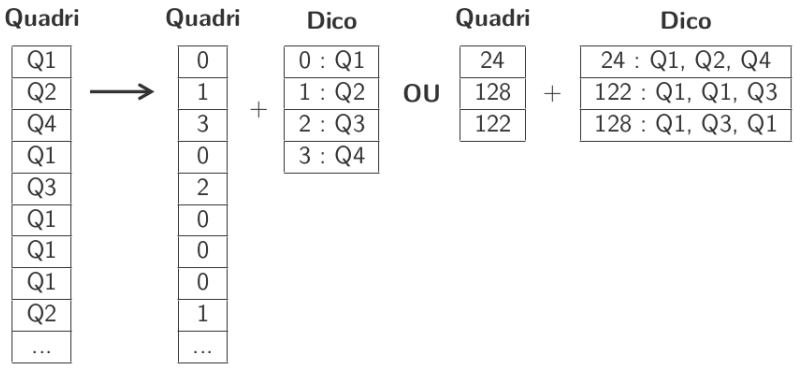

FIGURE 1.7 – Technique de compression "*Dictionary Encoding*" [1]

Deux approches sont possibles :
- on peut substituer chaque valeur par une autre qui est encodée sur moins de bits et associé la table de conversion à la nouvelle colonne (de même taille) ainsi obtenue
- on peut remplacer un motifs de valeurs et y associer une table ... [? ? ?]

36. *Une base de données orientée colonnes est très adaptée lorsqu'on a plus d'opérations d'écriture que de lecture.* Faux

Dans le modèle orienté-colonnes, les colonnes/familles de colonnes se trouvent dans des fichiers distincts. Lorsqu'on fait une lecture sur certaines colonnes, il suffit d'ouvrir les fichiers qui y correspondent. Dès lors que l'on fait une écriture pour ajouter une nouvelle entrée, il faut nécessairement ouvrir tous les fichiers...

37. *Une base de données orientée colonnes est très adaptée lorsqu'on a plus d'opérations de lecture que d'écriture.* Vrai

38. *Une base de données orientée colonnes est un map à deux niveaux.* Vrai

Ce map à deux niveaux se compose de la façon suivante :
- au premier niveau -> une PAIRE CLÉ-VALEUR identifie une ligne
- au second niveau -> une MAP DE COLONNES forme les familles

Les colonnes qui sont "souvent accédées ensemble" sont regroupées en familles et stockées dans un même fichier. Lors de la conception d'une DB en colonnes, il est important de penser aux regroupements des colonnes car bien que l'ajout d'une colonne soit puisse se faire facilement "à chaud", des changements sur les familles sont difficilement envisageables par la suite.

Contrairemnet au modèle relationnel où toutes les colonnes d'une lignes doivent absolument être remplies (induit beaucoup de valeurs *NULL*), en orienté-colonne les lignes peuvent avoir des clés colonnes différentes.

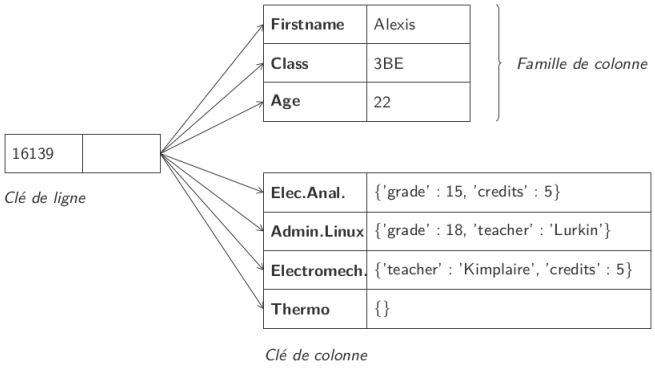

FIGURE 1.8 – Une base orientée-colonne est un *map à deux niveaux* [1]

39. *Dans une base de données orientée colonnes, les familles de colonnes sont de préférence définies une fois pour toute lors de la création de la table.* Vrai

40. *L'avantage de l'utilisation de colonnes plutôt que de lignes est d'offrir une vitesse d'écriture plus grande de nouveaux enregistrements.* Faux

L'avantage du modèle colonnes est d'offrir une **vitesse de lecture plus grande**.

En effet, en relationnel classique (*stockage lignes*), les lignes sont physiquement stockées les unes après les autres sur le disque. Dès lors, lorsqu'on cherche à obtenir les valeurs de certaines colonnes seulement, il faut malgré tout parcourir l'entièreté de chaque ligne. Ceci devient très contraignant, augmente considérablement le temps de lecture, lorsque le nombre de colonnes est important.

Ce problème ne se pose plus en orienté-colonne, car toutes les données d'une même colonne se trouvent dans un même fichier et sont donc physiquement contigües sur le support de stockage. La lecture des valeurs d'une (ou plusieurs) colonne(s) se fait dès lors beaucoup plus rapidement.

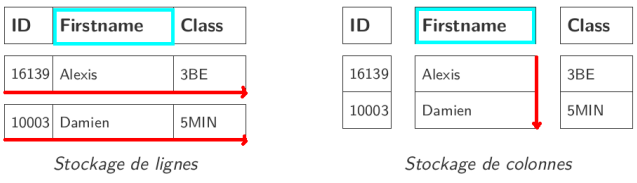

FIGURE 1.9 – Modèles de stockage sur le disque [1]

41. *L'avantage de l'utilisation de colonnes plutôt que de lignes est d'offrir un meilleur taux de compression des données stockées.* Vrai

En travaillant en colonnes, on peut notamment optimiser les algorithmes en fonction du types des valeurs de la colonne. (Suite réponse Q35)

42. *L'avantage de l'utilisation de colonnes plutôt que de lignes est d'offrir de meilleures performances lors de la lecture de tous les enregistrements d'une table.* Faux

Le modèle orienté-colonnes offre de meilleures performances à la lecture uniquement lorsqu'on travaille sur certaines colonnes de la table, car on utilise les fichiers relatifs à ces colonnes (et non toute la DB).

Dès lors que l'on fait une requête sur l'ensemble des enregistrements, la modèle devient un inconvénient car il nécessite d'ouvrir et de traiter TOUS les fichiers contenant les colonnes, ce qui alourdit davantage la lecture par rapport au modèle relationnel.

43. *Une base HBase peut servir d'input/output de MapReduce (Hadoop)* Vrai

HBase est un SGBD orienté-colonne qui s'exécute par dessus le système de fichiers HDFS (= [...]). Il permet a une base de servir d'input/output de MapReduce.

Les données sont organisées dans des "*tables*" identifiables par un nom. Chaque ligne possède une clé unique pour l'identifier et les colonnes sont regroupées en "*familles*" (identifiables par chaines caract.). Toutes les lignes possèdent les même familles de colonnes mais celles-ci peuvent être remplies ou non.

En HBase, les données des cellules sont *versionnées*. Les différentes versions sont identifiées par un *Timestamp*.

Ainsi, pour obtenir la donnée d'une cellule, il faut préciser : *Table -> Clé -> Famille Col. -> Colonne -> Timestamp*

Les **familles de colonnes sont stockées dans des HFile séparés**, ce qui facilite la structure de la DB et l'indexation des données. L'avantage de HBase est alors de proposer la distribution et la réplication des données sur un *cluster*.

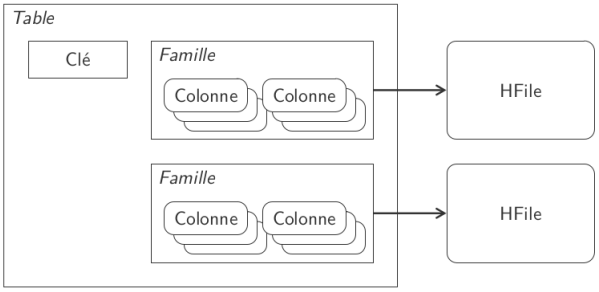

FIGURE 1.10 – Modèle de données en HBase [1]

Pour l'écriture des données, celles-ci sont d'abord placées dans un buffer, le *memstore*, qui écrit ensuite les changements dans un HFile [...]

44. *Une base HBase peut servir de fichiers avec GFS (Google File System)* ???

45. *Une base de données orientée graphe stocke deux collections d'agrégats appelés nœuds et arêtes.* Vrai

Le modèle orienté-graphe stocke deux types d'éléments :
- des NOEUDS : représentent des entités quelconques avec des propriétés
- des ARÊTES : représentent des relations UNIDIRECTIONNELLES entre deux noeuds et possèdent une propriété. (*Rem : pour faire un lien bidirictionnel, il faut donc nécessairement utiliser deux arêtes*

Puisqu'il n'y a pas de restriction sur les types des noeuds/relations, le graphe représente une collection hétérogène d'information dont l'avantage principal est de permettre une **traversée rapide** des relations, sans que celles-ci ne doivent être reconstruites.

Cette architecture est particulière utile dans les cas suivants :
- les collections *très riche de liens* entre les entités
- routage/dispatching et localisation
- moteur de recommandations automatiques

46. *La suppression d'un nœud dans une base de données orientée graphe implique la suppression de toutes les relations partant et arrivant sur ce nœud.* Vrai

Il ne peut **pas exister de lien mort** dans le graphe, autrement dit la suppression d'un noeud implique nécessairement de supprimer toutes les arêtes adjacentes.

47. *Il est impossible de stocker une liste de personnes dans une base de données orientée graphe* Faux

On distingue différents *types de relations* dans le graphe dont une qui permet de représenter une **liste chaînée**. La relation est un "pointeur" vers l'élement suivant de la liste.

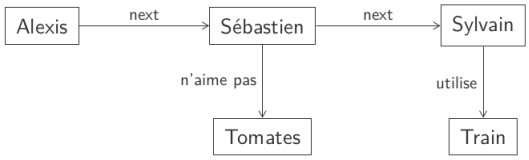

FIGURE 1.11 – Représentation d'une liste chaînée en modèle orienté-graphe [1]

48. *SPARQL est un langage de requêtes générique permettant d'interroger n'importe quelle base de données NoSQL.* Faux

SPARQL (***S**PARQL **P**rotocol and **R**DF **Q**uery **L**anguage)* est un langage de requêtes adapté spécifiquement à la structure des graphes RDF.

Tout ceci fait partie de ce qui constitue plus communément le **Web Sémantique** qui est une extension standardisée du Web, pour l'utilisation de formats de données. Il s'agit donc du *Web des données* qui permet de partager des informations structurées entre des applications. (*« le Web sémantique fournit un modèle qui permet aux données d'être partagées et réutilisées entre plusieurs applications, entreprises et groupes d'utilisateurs »* W3C)

Dans ce contexte, le RDF (***R**esource **D**estcription **F**ormat*) est un **modèle de graphe** destiné à représenter de façon formelle les ressources web. Il est le langage de base du Web sémantique et permet de stocker les données sous forme de triplets "(Sujet) — prédicat —> (Objet)" pour marquer la relation entre deux entités.

Le SPARQL, quant à lui, est le langage de requêtes qui interroge les graphes RDF au moyen de quatre types d'instructions :

(a) SELECT : extraire un sous-graphe d'un graphe RD sous forme de table

(b) CONSTRUCT : engendrer un nouveau graphe complétant un autre

(c) ASK : poser une question (True/False)

(d) DESCRIBE : extraire un graphe RDF

49. *Gremlin est un langage de requêtes générique permettant de décrire des traversées de graphe.* Vrai

Il est utilisé avec deux types de bases de données :
- OLTP : OrientDB,...
- OLAP : Apache Giraph, Hadoop,...

Dans ce langage, une requête décrit la traversée à faire pour récupérer des informations, on effectue des opérations sur les noeuds.

50. *Neo4j supporte les transactions ACID.* Vrai

Neo4j est un **système de gestion de graphes** qui supporte les transactions de type ACID et qui interroge les données au moyen du CQL (***C**ypher **Q**uery **L**anguage*), un langage à travers HTTP (requêtes de types : CREATE, MATCH et SET).

Il permet d'associer aux noeuds et aux arêtes des *propriétés* sous forme de paires "clé-valeur" et d'ajouter des *labels* aux noeuds, ce qui permet de les regrouper en catégories.

Le résultat d'une requête est un *chemin*, càd une séquence de noeuds avec des relations.

51. *Les bases de données orientée graphe sont très adaptée pour le sharding.* Faux

Dans le modèle orienté-graphe, toutes les données d'un même graphe doivent nécessairement se trouver sur une mêmemachine. Il n'est donc pas adapté à la distribution de type *sharding*.

52. *OrientDB offre la possibilité d'utiliser le sharding de données.* Vrai

Dans OrientDB (SGBD multi-domaines : graphe, clé-valeur, colonnes et document), les données sont représentées sous forme de **classes** qui possèdent : une *popriété* par "colonne" (similaire aux tables du relationnel), des *contraintes* et des *enregistrements*.

Ces classes peuvent être réparties sur différentes machines, offrant donc la possibilité d'utiliser la répartition des données en mode *sharding*.

53. *Une approche pessimiste de la consistance des données consiste à se limiter à un serveur unique pour le stockage des données.* Faux

La consistance de mise à jour des données n'est pas évidente à garantir lorsque plusieurs utilisateurs essaie de modifier une donnée en même temps -> conflit d'écritures (*write-write)*

En cas d'échec de mise à jour, on risque de : perdre une mise à jour ou perdre de la consistance au niveau des données

Pour éviter au plus possible la perte de cohérence des données, deux approches sont envisageables :

(a) PESSIMISTE : approche plus lourde qui consiste à **éviter tout conflits d'écritures**, par exemple en utilisant un système de locks (lorsqu'un *user* souhaite faire un W, il fait une demande pour obtenir un verrou sur cette donnée afin qu'elle ne puisse pas être modifiée par qqun d'autre entre temps). **Ce principe ne doit pas forcément se faire sur un serveur unique, il peut tout aussi bien fonctionner dans un modèle distribué**.

(b) OPTIMISTE : approche plus "souple" qui **tolère les conflits d'écritures**. Elle peut conserver une des mises à jour, suivant un règlement d'ordre de traitement des requêtes, ou alors conserver toutes les mises à jour en notifiant les *users* du conflit. Il est alors de leur responsabilité de le régler ou du moins d'en être conscient.

Remarque : se limiter à un serveur unique ne garantit la consistance des données en cas de conflits d'écriture, il faut également mettre en place des mécanismes de protection

54. *Garantir la consistance de lecture empêchera tout conflit de type write-write.* Faux

La garantie de consistance de lecture **empêchera tout conflit de type *read-write***, de telle sorte que si plusieurs *users* lisent une même donnée, ils obtiennent la même information ET que cette information ait une consistance logique.

Exemple de conséquences dues à un conflit de lecture et écriture simultannés :

- Si une information est composée de plusieurs documents/entités et que ceux-ci sont ré-écrits au même moment que la lecture est faite, certaines parties risque d'être mises à jour et d'autre non. On se retrouve alors avec des documents dans des versions différentes et donc une information qui perd son sens logique de départ.

55. *Garantir la consistance de mise à jour empêchera tout conflit de type read-write.* Faux

La garantie de consistance de mise à jour **empêchera tout conflit de type *write-write***. Un tel conflit a lieu lorsque plusieurs *users* modifient une même donnée en même temps et ses conséquences sont : perte d'une mise à jour et inconsistance des données (un *user* à modifié une donnée en partant d'une version qui n'est plus la bonne).

56. *Garantir la consistence de réplication est impossible avec un système peer-to-peer.* Vrai

Si la "consistance de réplication" est garantie, alors plusieurs *users* qui lisent une même donnée obtiendront la même information, bien que la donnée se trouve répliquée sur des noeuds différents, par exemple, ou dans des mémoires différentes (cache, RAM, disque...).

Cette inconsistance provient des problèmes de diffusion des changements d'information entre les différentes parties du système et est typique du modèle NoSQL car il est destiné à être distribué (*cluster*). Cependant, on peut parler de consistance "à la longue" car en pratique, l'information finira *toujours* par être mise à jour sur tous les noeuds.

Cette inconsistance est influencée par le nombre de noeuds mais égalemnet par le délai du réseau, lié au temps de communication entre les entités (qui augmente avec la distance et qui n'est pas forcément maîtrisable).

Garantir la consistance de réplication est *"impossible"* dans un système peer-to-peer aussi petit soit-il. En effet, une approche (bien que lourde) pour limiter les problèmes de consistance serait de vérifier que la donnée ait la même valeur sur tous les noeuds MAIS le temps de faire cette vérification, la donnée pourrait avoir changé sur un noeud puisque chacun est accessible en écriture -> pas de garantie ...

57. *Si mes données sont répliquées sur quatre nœuds, avec W = 2, il suffit de lire deux nœuds pour lire l'information la plus à jour.* Faux

Read Quorum : R + W > N

- R nombre de noeuds à contacter pour une lecture
- W nombre de noeuds impliqués dans l'écriture
- N facteur de réplication

~~Si N=4 et W=2, il faut lire au moins 3 noeuds pour obtenir l'information la plus à jour (R > 4-2). En effet, en ne lisant que deux des quatre noeuds, il se pourrait qu'ils contiennent tout les deux une même ancienne valeur (puisque W=2) alors qu'en lisant 3 des noeuds, ...~~ LE PORBLEME RESTE LE MEME

En effet, en lisant 3 noeuds, il y en aura forcément un sur les trois qui sera différent MAIS si les deux autres ont la même valeur et qu'elle correspond à une ancienne valeur alors la consistance n'est pas garantie.

Le problème vient du fait que le *write quorum* (W > N/2) n'est pas respecté dans l'énoncé. Il faudrait que $W \geq 3$ pour garantir de lire l'information la plus à jour.

58. *Si mes données sont répliquées sur quatre nœuds, il suffit d'impliquer W = 2 nœuds dans l'écriture pour assurer une consistence des données.* Faux

Le *write quorum* nous dit que W > N/2 pour garantir de lire l'information la plus à jour, il faudrait donc que $W \geq 3$ si N = 4.

59. *L'utilisation d'un timestamp comme version stamp est moins lourd à déployer que d'utiliser un GUID (Globally Unique Identifier).* Faux ? ?

Le timestamp est plus lourd car il nécessite la synchronisation de tous les noeuds sur une même horloge (via un serveur dédié, fuseau horaire,...) mais contrairement au GUID, il permet de comparer l'instant de changement de plusieurs versions d'une donnée.

60. *L'utilisation d'un GUID (Globally Unique Identifier) comme version stamp est moins lourd à déployer que d'utiliser un timestamp.* Vrai

61. *L'utilisation d'un GUID (Globally Unique Identifier) comme version stamp permet de retrouver la version la plus récente d'une donnée.* Faux

Le GUID ne permet pas de trouver la version la plus récente d'une donnée par comparaison,

il permet simplement de [...] *identifier la version de changement ? ? ?*

62. *Le write optimiste est très cher à implémenter dans un modèle clé-valeur.*

63. *Dans une base de données orientée colonnes, les données transitent par plusieurs espaces de stockage avant leur destination finale permanent.*

64. *Il est possible d'utiliser de la réplication master-slave avec une base de données orientée graphe pour rendre les lectures plus performantes.*

65. *Les bases de données orientée documents permettent d'effectuer des transactions atomiques au niveau d'un document.*

66. *Les bases de données orientée documents permettent d'effectuer des transactions atomiques au niveau d'une collection.*

67. *On peut changer le modèle d'une base de données NoSQL entre clés-valeurs, colonnes et documents, tout en garantissant exactement le même ensemble de propriétés.*

68. *Le passage vers le NoSQL permet de se passer des ORMs.*

69. *Le NoSQL est très adapté pour stocker des données très uniformes.*

70. *Le passage du relationnel au NoSQL rend les calculs à effectuer sur les données plus lents suite à un éventuel cout de transfert des données au sein du cluster.*

71. *L'opération Map s'applique à une collection de documents et renvoie un collection modifiée.*

72. *L'opération Reduce produit un résultat unique.*

73. *ElasticSearch est une base de données NoSQL.*

74. *ElasticSearch possède des caractéristiques similaires aux bases de données NoSQL.*

75. *Une migration de données en relationnel ou en NoSQL implique toujours un stockage physique de données redondantes.*

Bibliographie

[1] Sébastien Combéfis. Bases de données avancées - NoSQL. Ecole Centrale des Arts et Métiers, Bruxelles, 2017.