Introduction aux systèmes NoSQL (Not Only SQL)

Bernard ESPINASSE
Professeur à Aix-Marseille Université (AMU)
Ecole Polytechnique Universitaire de Marseille

Avril 2013

1. De nouveaux besoins en gestion de données
2. Introduction aux systèmes NoSQL
3. Modèle « Clés-Valeurs »
4. Modèle « Colonnes »
5. Modèle « Documents »
6. Modèle « Graphes »

Plan

Principales sources du cours

Documents :
- C. Strauch, « Nosql databases », Lecture Notes, Stuttgart Media University, 2011.
- A. Foucret, « Livre blanc sur NoSQL », par Smile (http://www.smile.fr/Livres-blancs/Culture-du-web/NoSQL).
- S-K. Gajendran, « A Survey on NoSQL Databases ».
- S. Abiteboul, I. Manolescu, P. Rigaux, M-C Rousset, P. Senellart, « Web Data Management », Cambridge University Press 2011 (en ligne, la 3ème partie : http://webdam.inria.fr/Jorge/?action=chapters).
- J. Dean and S. Ghemawat, « MapReduce: Simplified Data Processing on Large Clusters », OSDI 2004

Présentations :
- P. Selmer, « NOSQL stores and Data analytics tools », Advances in Data Management, 2012.
- A.-C. Caron, « NoSQL », Université de Lille 1.
- M. Jaffré, P. Rauzy, « MapReduce », ENS, 2010.
- K. Tannir, « MapReduce : Algorithme de parallélisations des traitements », 2011, http://**blog.khaledtannir.net**/wp-content/.../KT-Presentation-MapReduce.pdf

1. Introduction

- **Nouveaux besoins en gestion de données**
- **Limites des SGBD Relationnels-transactionnels**
- **Le théorème de Brewer ou de CAP**
- **Le grand paysage des bases de données**

Nouveaux besoin en gestion de données (1)

Constat :

- essor des **très grandes plateformes** et **applications Web** (Google, Facebook, Twitter, LinkedIn, Amazon, …)
- **volume considérable de données** à gérer par ces applications nécessitant une **distribution** des données et leur traitement sur de nombreux serveurs
- ces données sont souvent associées à des **d'objets complexes** et **hétérogènes**

  *=> Limites des **SGBD traditionnels (relationnels et transactionnels)** basés sur SQL*

D'où nouvelles approches de stockage et de gestion des données :

- permettant une **meilleure scalabilité** dans des **contextes fortement distribués**
- permettant une **gestion d'objets complexes et hétérogènes** sans avoir à déclarer au préalable l'ensemble des champs représentant un objet
- regroupées derrière le terme **NoSQL** (proposé par **Carl Strozzi**), ne se substituant pas aux SGBD Relationnels mais les **complètant** en comblant leurs **faiblesses (Not Only SQL)**

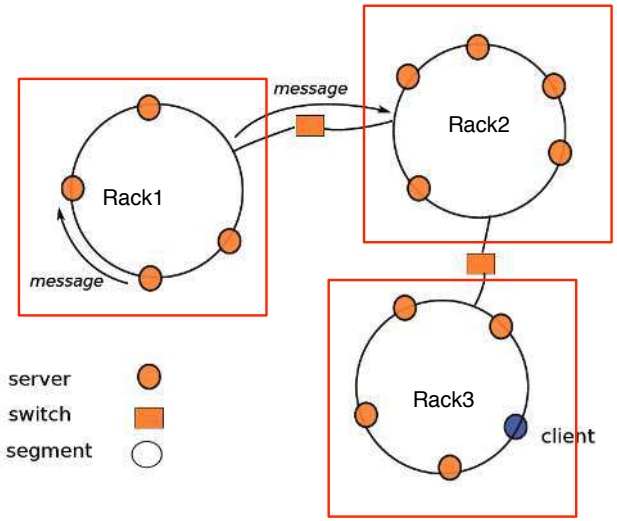
Exemple : Data Centers (1)

- Utilisent des **LANs** (Local Area Networks) avec **3 niveaux de communication** :
  - Les **serveurs** sont regroupés en « **Racks** » : *liaison réseau rapide, environ 1Go/sec*
  - un « **Data center** » consiste en un grand nombre de « **racks** », interconnectés par des **routeurs** (switches) : *liaison à 100 Mo/sec*
  - entre différents « **Data centers** » : *communication internet à 2-3 Mo/sec*
- Les **serveurs** communiquent par *envoi de messages*, ils ne partagent pas de disque ni de ressource de traitement = ***architecture « shared nothing »***

Exemple : Data Centers (2)

- **Ex. : Data center de Google (début 2010) :**
  - Un « **Data center** » Google contient entre **100 et 200** « **Racks** », chacun contenant **40 serveurs**
  - environ **5000 serveurs par** « **Data-center** » pour un total de **1 millions de serveurs** (estimation d'après la consommation électrique).
- **Ex. : Data center de Facebook (2010) :**
  - **2500 cpu** (serveurs)
  - **1 PetaByte d'espace disque** (= milleTerabytes = 1 million de Gigabytes)
  - Plus de **250 Gigabytes données compressées** (plus de 2 Terabytes non compressés)

Exemple : Data Centers (2)



Système distribué

- système logiciel permettant de coordonner plusieurs ordinateurs
- ordinateurs reliés par un réseau local (LAN)
- communiquant généralement par envoi de messages
- utilisant une architecture distribuée :
  - fonctionnant sur du matériel peu spécialisé
  - matériel facilement remplaçable en cas de panne

Application particulière : la **distribution de données** sur **plusieurs serveurs** organisés en « data centers » :

- gestion de volumes de données très importants
- assurer une continuité de service en cas d'indisponibilité de service sur un serveur

Gestion de données distribuée

On dispose d'un très grand ensemble de données sur lesquelles on doit leur appliquer des traitements. **2 stratégies** :

- **Par distribution des traitements (*scaling des traitements*) :**
  - **on distribue ces traitements** sur un nombre de machines important afin d'absorber des charges très importantes
  - on **envoie les données** aux endroits appropriés, et on enchaîne les exécutions distantes (scénario type **Workflow** implémentable avec des **web services**)
- **Par distribution des données (*scaling des données*) :**
  - on **distribue les données** sur un nombre important de serveurs afin de stocker de très grands volumes de données
  - on « **pousse** » **les programmes vers ces serveurs** (plus efficace de transférer un petit programme sur le réseau plutôt qu'un grand volume de données – Ex : algorithme **MapReduce**).

Limites des SGBD Relationnels-transactionnels (1)

SGBD Relationnels offrent :

- un système de **jointure entre les tables** permettant de construire des **requêtes complexes** impliquant plusieurs entités
- un système **d'intégrité référentielle** permettant de s'assurer que les **liens** entre les entités sont **valides**

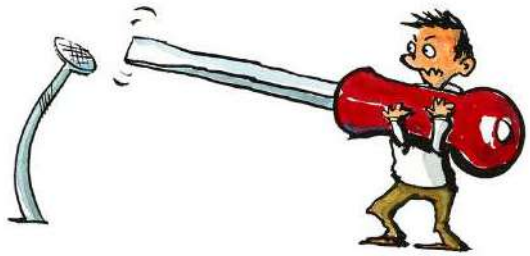
Contexte fortement distribué :

Ces mécanismes ont un coût considérable :

- avec la plupart des SGBD relationnels, les **données d'une BD liées entre elles** sont placées sur le **même nœud** du serveur
- si le **nombre de liens important**, il est de plus en plus **difficile de placer les données sur des nœuds différents**.

Limites des SGBD Relationnels-transactionnels ...

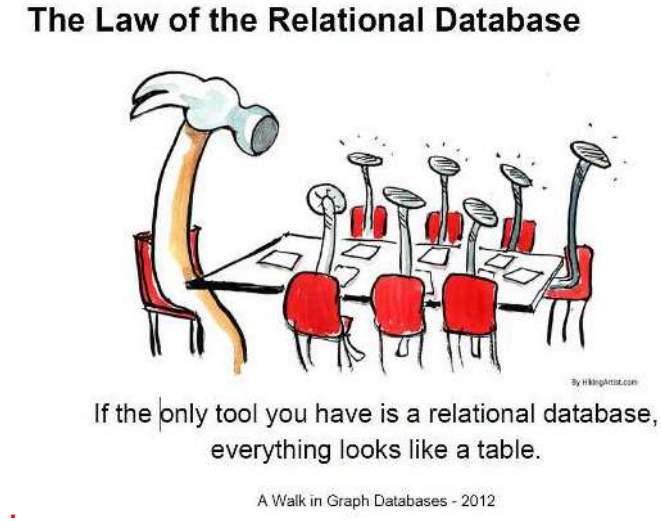
The Law of the Hammer



If the only tool you have is a hammer,
everything looks like a nail.

Abraham Maslow - The Psychology of Science - 1966

Limites des SGBD Relationnels-transactionnels …



Limites des SGBD Relationnels-transactionnels (2)

SGBD Relationnels sont généralement transactionnels :

=> gestion de transactions respectant les **contraintes ACID** (Atomicity, Consistency, Isolation, Durability)

Contexte fortement distribué :

Cette gestion a un coût considérable :

nécessaire de **distribuer les traitements** de données entre différents serveurs

- alors **difficile de maintenir les contraintes ACID** à l'échelle du système distribué entier
- tout en maintenant des **performances correctes**

=> la plupart des SGBD « NoSQL » relâchent les contraintes ACID, ou même ne proposent pas de gestion de transactions.

Modèle de données Relationnel vs. Agrégats (1)

Modèle relationnel :

- les données sont divisées en **ligne** (tuples – rows) avec des **colonnes** (columns) **prédéfinies** (attributs)
- il n'y a **pas de tuples emboîtés**
- il n'y a **pas de liste de valeurs**

Modèle agrégat :

- une **collection d'objets reliés**
- qui doivent être **traités ensemble.**

Modèle de données Relationnel vs. Agrégats (2)

Relational Instance

CUSTOMER

ID	NAME
1	Guido

PRODUCT

ID	NAME
1000	IPod Touch
1020	Monster Beat

BILLING_ADDRESS

ID	CUSTOMER_ID	ADDRESS_ID
1	1	55

ADDRESS

ID	STREET	CITY	POST_CODE
55	Chaumontweg	Spiegel	3095

ORDER

ID	CUSTOMER_ID	SHIPPING_ADDRESS_ID
90	1	55

ORDER_ITEM

ID	ORDER_ID	PRODUCT_ID	PRICE
1	90	1000	250.55
1	90	1020	199.55

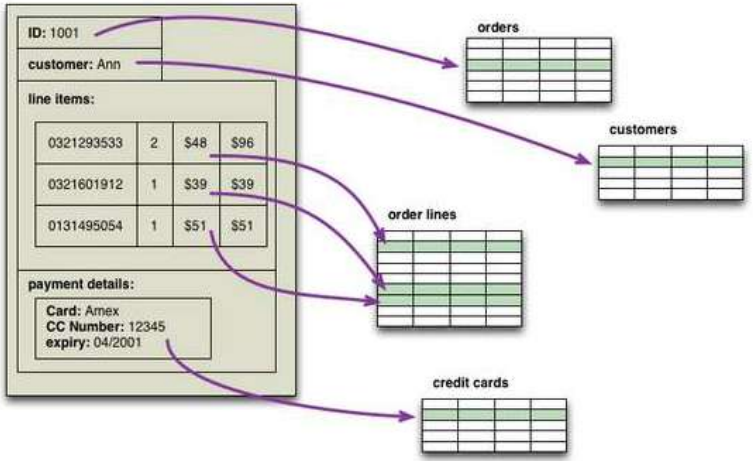
Aggregate Instance

```
{
„id":1,
„name":"Guido",
„billingAddress":[{„street":"Chaumontweg","city":"Spiegel","postCode":"3095"}]
}

{
„id":90,
„customerId":1,
„orderItems":[
 {
 „productId":1000,"price": 250.55, „produtName": „iPod Touch"
 },
 {
 „productId":1020,"price": 199.55, „produtName": „Monster Beat"
 }],
„sippingAddress":[{„street":"Chaumontweg","city":"Spiegel","postCode":"3095"}]
}
```

Modèle de données Relationnel vs. Agrégats (3)

(Martin Fowler, Aggregate Oriented Database, 19 January 2012)



Limitation des SGBDR : théorème de CAP (1)

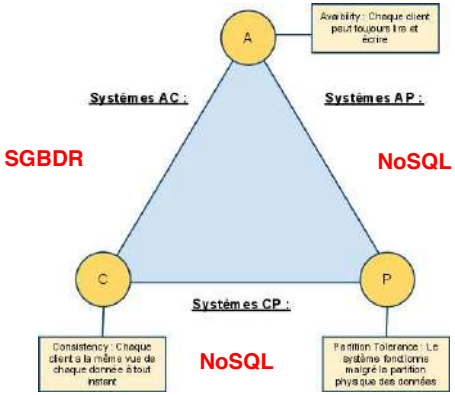
Théorème de CAP (Brewer, 2000) :

3 propriétés fondamentales pour les **systèmes distribués** :

- **C oherence** *ou Consistance* : tous les nœuds du système voient exactement les mêmes données au même moment
- **A vailability** *ou Disponibilité* : la perte de nœuds n'empêche pas les survivants de continuer à fonctionner correctement, les données restent accessibles
- **P artition tolerance** *ou Résistance au partitionnement* : le système étant partitionné, aucune panne moins importante qu'une coupure totale du réseau ne doit l'empêcher de répondre correctement (en cas de partitionnement en sous-réseaux, chacun doit pouvoir fonctionner de manière autonome)

Dans un système distribué, il est impossible d'obtenir ces 3 propriétés en même temps, il faut en choisir 2 parmi les 3

Limitation des SGBDR : théorème de CAP (2)



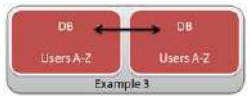
- Les **SGBDR** assurent les propriétés de *Consistance* et de *Disponibilité* (Availability) => *système **AC***
- Les **SGBD « NoSQL »** sont des *systèmes **CP** (Cohérent et Résistant au partitionnement) ou **AP** (Disponible et Résistant au partitionnement).*

Théorème de CAP (1)

Consistance (*Coherence*) : Un système S est dit **consistant** si on peut garantir qu'une fois qu'on y enregistre un état (disons "x = y"), il donnera le même état à chaque opération suivante, jusqu'à ce que cet état soit explicitement changé par quelque chose en dehors de S

- **Ex1:** Une instance de la BD est automatiquement pleinement consistante puisqu'il n'y a qu'un seul noeud qui maintient l'état.
- **Ex2** : Si 2 serveurs de BD sont impliqués, et si le système est conçu de telle sorte que toutes les clés de "a" à "m" sont conservées sur le serveur 1, et les clés "n" à "z" sont conservées sur le serveur 2, alors le système peut encore facilement garantir la cohérence

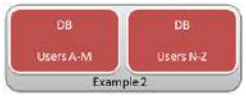
- **Ex3** : Supposons 2 BD avec des répliques. Si une des BD fait une opération d'insertion de ligne, cette opération doit être aussi faite (commited) dans la seconde BD avant que l'opération soit considérée comme complète.
  - Pour avoir une cohérence de 100% dans un tel environnement répliqué, les nœuds doivent communiquer
  - Plus le nb de répliques est grand plus les performances d'un tel système sont mauvaises.



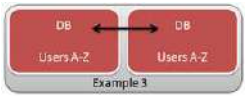
Théorème de CAP (2)

Disponibilité (*Availability*): Les BD dans **Ex1** ou **Ex2** ne sont pas fortement disponibles

- **dans Ex1** : si le nœud tombe en panne, on perd 100% des données
- **dans Ex2** : si le nœud tombe en panne, on perd 50% des données



- **dans Ex3** : une simple réplication de la BD sur un autre serveur assure une disponibilité de 100%.
  - augmenter le nb de nœuds avec des copies des données (répliques) augmente directement la disponibilité du système, en le protégeant contre les défaillances matérielles
  - les répliques aussi aider à équilibrer la charge d'opérations concurrences, notamment en lecture.



Théorème de CAP (3)

Résistance au partitionnement (*Partition-tolerance*):

- Supposons que soit assuré par réplication « consistance » et « disponibilité ». Dans le cas **Ex3**, supposons **2 serveurs de BD dans 2 Data-Centers différents**, et que ***l'on perde la connexion réseau*** entre les 2 Data-Centers, faisant que les 2 BD sont incapables de synchroniser leurs états.
- Si vous parvenez à gérer les opérations de lecture/écriture sur ces 2 BD, il peut être prouvé que les 2 serveurs ne seront plus consistants.
- une application bancaire gardant à tout moment «l'état de votre compte » est l'exemple parfait du problème des enregistrements inconsistants :
  - Si un client retire 1000 euros à Marseille, cela doit être immédiatement répercuté à Paris, afin que le système sache exactement combien il peut retirer à tout moment
  - Si le système ne parvient pas à le faire, cela pourrait mécontenter de nombreux clients
- Si les banques décident que la « consistance » est très importante, et désactivent les opérations d'écriture lors de la panne, alors la «disponibilité» du cluster sera perdu puisque tous les comptes bancaires dans les 2 villes seront désormais gelés jusqu'à ce que le réseau soit de nouveau opérationnel.

Le grand paysage des bases de données (1)

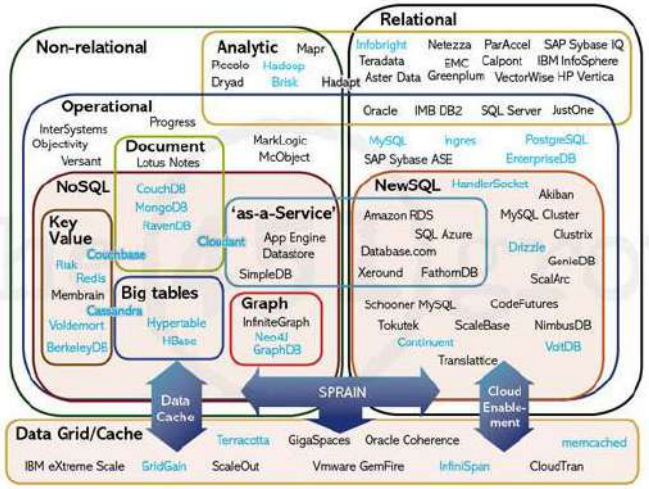
Report « NoSQL, NewSQL and Beyond: The answer to SPRAINed relational databases », 451 Group, April 15th, 2011 :

Face aux faiblesses des SGBDR en termes de **performance**, **d'évolutivité** et de **flexibilité** des besoins de traitement à **grande échelle** des données, alternatives suivantes avec des **architectures distribuées** :

- **NoSQL BD :** BD avec **schéma dynamique ou sans schéma**, BD magasins **de clés-valeur**, BD **de documents** et de **données graphiques, …**
- **NewSQL DB** : amélioration des **performances** grâce à de **nouveaux moteurs de stockage**, des technologies transparentes de **fragmentation**, de **nouveaux logiciels et matériels** : des ***BD radicalement nouvelles***
- **Data grid/cache products :** amélioration des **performances** des applications et de la BD par stockage des données en **mémoire** : **données persistante en cache**, **réplication et des données distribuées**, et **calcul exploitant le grid, …**

Le grand paysage des bases de données (2)

Report « NoSQL, NewSQL and Beyond: The answer to SPRAINed relational databases », 451 Group, April 15th, 2011 :



SPRAINed relational databases

- **SPRAIN** : acronyme qui désigne les 6 facteurs clés de l'adoption de technologies de gestion de données alternatives aux traditionnelles BDR
- ces BD alternatives doivent permettre de traiter de **très grands volumes de données**, supporter des **applications hautement distribuées** ou très **complexes**
- Ces **6 facteurs clés** sont :
  - **Scalability** (évolutivité) – hardware economics (économie de matériel)
  - **Performance** – BDR limitations
  - **Relaxed consistency** – CAP theorem (cohérence relachée)
  - **Agility** – polyglot persistence (agilité, persistance polyglotte)
  - **Intricacy** (intrication) – big data, total data
  - **Necessity** (nécessité) – open source

2. Introduction aux systèmes NoSQL

Introduction au NoSQL

Les BD NoSQL :

- Adoptent une **représentation de données non relationnelle**
- ne remplacent pas les BD relationnelles mais sont une **alternative**, un **complément** apportant des solutions plus intéressantes dans **certains contextes**
- apportent une **plus grande performance** dans le contexte des **applications Web** avec des **volumétries de données exponentielle**
- utilisent une **très forte distribution** de ces données et des traitements associés sur de **nombreux serveurs**
- font un **compromis sur le caractère « ACID » des SGBDR** pour plus de scalabilité horizontale et d'évolutivité.

L'adoption croissante des bases NoSQL par des grands acteurs du Web (Google, faceBook, …) => multiplication des offres de systèmes NoSQL

Caractéristiques générales de BD NoSQL

- **pas de schéma** pour les données ou **schéma dynamique**
- données de structures **complexes** ou **imbriquées**
- **données distribuées** : partitionnement horizontal des données sur plusieurs nœuds (serveurs) généralement par utilisation d'algorithmes « MapReduce »
- **réplication des données** sur plusieurs noeuds
- **privilégient la Disponibilité à la Cohérence (théorème de CAP)** : **AP** (Disponible + Résistant au partitionnement) plutôt que CP (Cohérent + Résistant au partitionnement)

  => n'ont en général pas de gestion de transactions

- mode d'utilisation : **peu d'écritures, beaucoup de lectures**

Partitionnement dans les syst. NoSQL : Sharding

« Sharding » :

- Mécanisme de **partitionnement horizontal** (par tuples) des données dans lequel les objets-données sont **stockées** sur des **nœuds serveurs différents** en **fonction d'une clé** (ex : fonction de hachage)

  *Ex : **partition = hash(o) mod n** avec o = objet-données à placer et n= nb de nœuds serveur disponibles*

- les données peuvent être aussi partitionnées afin que :
  - les **objets-données** accédés ou mis à jour **en même temps**, résident **sur le même nœud**
  - la **charge** soit **uniformément répartie** entre les **nœuds**
  - pour cela des **objet-données** peuvent être **répliquées**
- certains systèmes utilisent aussi un **partitionnement vertical** (par colonnes) dans lequel des parties d'un enregistrement sont stockées sur différents serveurs.

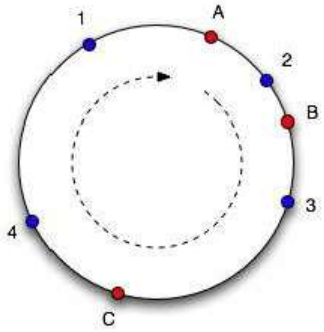
Partitionnement dans les syst. NoSQL : Consistent hashing (1)

« Consistent hashing » :

Mécanisme de partitionnement (horizontal) dans lequel les *objet-données* sont *stockés* sur des *nœuds-serveurs différents* en utilisant la **même fonction de hachage** à la fois pour le *hachage des objets* et le *hachage des nœuds :*

- nœuds et objets sont associés par une même fonction de hachage, et imaginés être placés sur un anneau (rack/cluster de serveurs)
  Ex : A, B, C sont des nœuds (serveurs) et 1, 2, 3, 4 sont des objets.
- un ***objet*** est associé au ***premier nœud*** serveur dans le ***sens horaire*** :
  Ex : les objets 4 et 1 sont associés au nœud A ; 2 à B ; 3 à C.

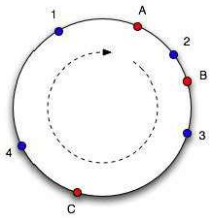
D'après Strauch



Partitionnement dans les syst. NoSQL : Consistent hashing (1)

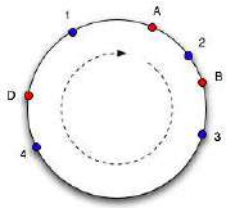
Suite …

- quand un ***nœud quitte*** l'anneau, les objets qui lui sont liés sont alors associés à leur ***nœud adjacent dans le sens horaire*** :
  Ex : le nœud C quitte l'anneau, 3 est alors associé avec 4 et 1 au nœud A



- quand un ***nœud entre*** dans l'anneau, il est placé (haché) sur l'anneau et ***des objets lui seront associés*** selon sa place dans l'anneau :
  Ex : le nœud D entre dans l'anneau, les objets 3 et 4 lui sont associés

D'après Strauch



Partitionnement dans les syst. NoSQL : MapReduce (1)

Map-Reduce :

- **Modèle de programmation parallèle** (framework de calcul distribué) pour le traitement de **grands ensembles de données**
- **développé par Google** pour le traitement de ***gros volumes de données*** en ***environnement distribué*** :
  - permet de *répartir la charge* sur un *grand nb de serveurs* (cluster)
  - *abstraction quasi-totale de l'infrastructure matérielle* **:** gère entièrement, de façon transparente le *cluster*, la *distribution de données*, la *répartition de la charge*, et la *tolérance aux pannes*
  - *ajouter* des machines *augmente la performance* (plug & play, scalable-friendly)
- la **librairie MapReduce** existe dans plusieurs langages (C++, C#, Erlang, Java, Python, Ruby…)

Partitionnement dans les syst. NoSQL : MapReduce (2)

Divers usages de Map-Reduce :

- **Utilisé par les grands acteurs du Web** notamment pour : *construire les index* (Google Search), *détection de spam* (Yahoo), *Data Mining* (Facebook) …
- **Mais aussi pour** :
  - De *l'analyse d'images astronomique*, de la *bioinformatique*, de la *simulation métrologique*, de *l'apprentissage automatique* (Machine Learning), des *statistiques*, …
  - *le calcul de la taille de plusieurs milliers de documents*
  - *trouver le nb d'occurrences d'un pattern* dans un très grand volume de données
  - *classifier de très grands volumes de données* provenant par exemple de paniers d'achats de clients (Data Mining)
  - …

Partitionnement dans les syst. NoSQL : MapReduce (3)

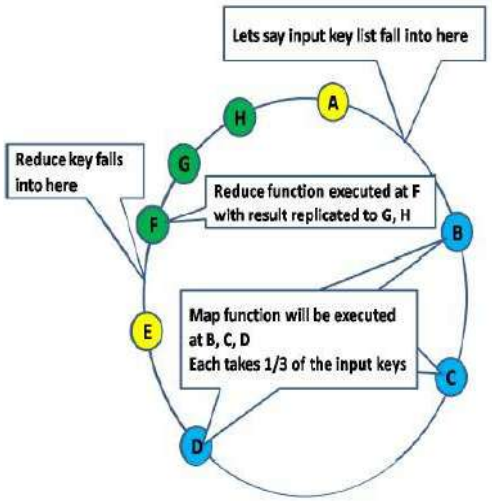
Usage de MapReduce en gestion de données :

- **Principales opérations** à faire sur des grands ensembles de données :
  1. **Itérer** sur un grand nombre d'enregistrements
  2. **Extraire** quelque chose ayant un intérêt de chacun de ces enregistrements
  3. **Regrouper** et **trier** les résultats intermédiaires
  4. **Agréger** tous ces résultats ensemble
  5. **Générer** le résultat final
- Dans le **modèle de programmation MapReduce**, le développeur implémente 2 fonctions : la fonction **Map** et la fonction **Reduce**
  - *opérations 1 et 2* : **fonction Map traite une paire clé/valeur** et génère un ensemble de paires de clés intermédiaires/valeurs
  - *opérations 3, 4 et 5* : **fonction Reduce fusionne toutes les valeurs intermédiaires** associées avec la même clé intermédiaire.

Partitionnement dans les syst. NoSQL : MapReduce (4)

- Appliqué à la BD, MapReduce traite un **ensemble de clés** en appliquant les fonctions **Map** et **Reduce** aux **nœuds** de stockage qui appliquent localement la fonction **Map** aux **clés** qui doivent être traitées et qu'ils possèdent
- les ***résultats intermédiaires*** peuvent être ***hachés comme des données ordinaires et traitées par les nœuds suivants dans le sens horaire***, qui appliquent la fonction Reduce aux résultats intermédiaires et produisent les résultats finaux
- du fait du ***hachage des résultats intermédiaires***, aucun **coordinateur** est utilisé pour diriger les nœuds de traitement vers ces **résultats**

D'après Strauch



Fondements des syst. NoSQL : MapReduce (5)

Fonctionnement de MapReduce :

- Traite des *données en entrée* pour en fournir des *résultats en sortie*
- Traitement constitué de *plusieurs tâches* dont chacune est *traitée indépendamment,* puis leurs *résultats sont combinés*
- On distingue 3 opérations majeures :
  - **Split** correspond à une opération de *découpage*
  - **Compute** correspond à une opération de *calcul*
  - **Join** correspond à l'opération de *regroupement* du résultat
- Dans le modèle de programmation MapReduce, le développeur implémente 2 fonctions :
  - la fonction **Map**
  - la fonction **Reduce**.

Fondements des syst. NoSQL : MapReduce (6)

Fonction Map : prend en entrée un ensemble de « Clé, Valeurs » et retourne une liste intermédiaire de « Clé1, Valeur1 » :

```
Map(key,value) -> list(key1,value1)
```

Fonction Reduce : prend en entrée une liste intermédiaire de « Clé1, Valeur1 » et fournit en sortie une ensemble de « Clé1, Valeur2 » :

```
Reduce(key1,list(value1)) -> value2
```

L'algorithme MapReduce s'exécute en **5 phases** :

1. La phase **Initialisation**
2. La phase **Map**
3. La phase **regroupement** (**Shuffle**)
4. La phase de **Tri**
5. La phase **Reduce**

Fondements des syst. NoSQL : MapReduce (7)

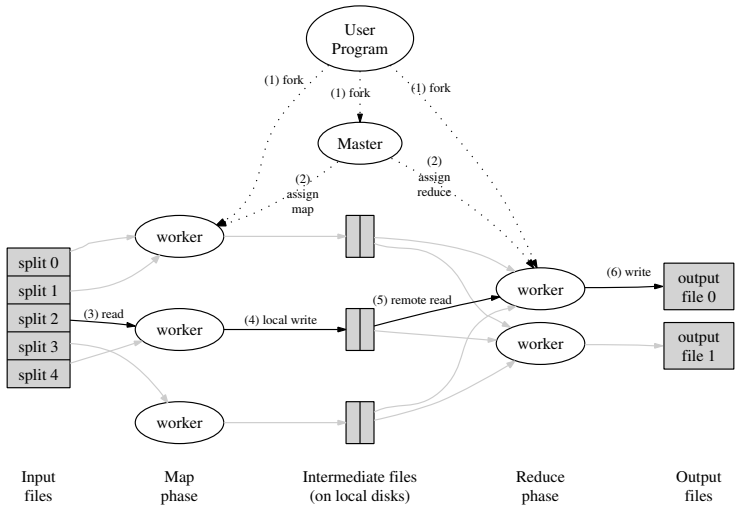
Fonctionnement de MapReduce :

- Lorsque l'application MapReduce est lancée, elle crée un composant « ***Master*** » responsable de la distribution des données, de la coordination de l'exécution de différentes unités de travail ou « ***Workers*** ».
- Le ***Master*** attribue aux ***Workers*** les tâches ***map*** et ***reduce***
- Un ***Worker*** possède **3 états**:
  - ***idle*** : il est *disponible* pour un nouveau traitement
  - ***in-progress*** : un *traitement est en cours* d'exécution
  - ***completed*** : il a *fini un traitement*, il en *informe* alors le *Master* de la taille, de la localisation de ses fichiers intermédiaires
- Le ***Master*** gère la synchronisation, la réorganisation, le tri et le regroupement des données :

  lorsqu'un ***Worker*** de type ***map*** a fini son traitement, le ***Master*** regroupe, trie et renvoie le résultat à un ***Worker*** de type ***reduce***

Fondements des syst. NoSQL : MapReduce (8)

Soit :



Fondements des syst. NoSQL : MapReduce (9)

Exemple de BD distribuée :

- BD de 1To de données
- 1800 serveurs
- les données sont découpées en 15000 morceaux d'environ 64Mo
- un seul serveur effectue la réduction (afin d'avoir les résultats dans un seul fichier)

Performances de MapReduce sur cette BD :

- démarrage relativement lent dû au temps de propagation du programme (±1mn).
- les Maps sont tous finis au bout d'environ 80s
- l'opération se termine en 150s.
- on atteint un pic de lecture de 30Go/s

Fondements des syst. NoSQL : Exemple MapReduce (1)

On souhaite **calculer le nombre de mots dans un document** :

la fonction ***Map, « mapFunc »*** est alors :

```
mapFunc(String key, String value):
// key: id du document;
// value: contenu du document
for each word w in value
EmitIntermediate (w, 1)
```

mapFunc a 2 arguments :

- la *clé (**key**) identifiant le document* dont on souhaite compter les mots
- le *contenu du document (**value**)*

l'ensemble des valeurs est parcouru par une boucle « for each » :

- pour chaque mot identifié, on appelle la méthode ***EmitIntermediate***
- elle place dans une zone intermédiaire le **mot** et la valeur **1** (correspondant à une occurrence).

Fondements des syst. NoSQL : Exemple MapReduce (2)

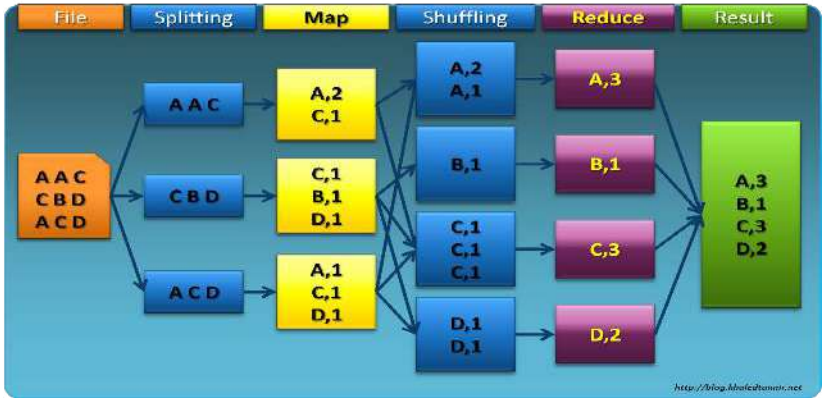
La fonction ***Reduce « reduceFunc »*** est :

```
reduceFunc(String key, Iterator values)
// key: un mot;  values: une liste de valeurs
result = 0
for each count v in values
result += v
EmitResult(result)
```

- **reduceFunc** a 2 arguments :
  - la ***clé*** (***key***) correspond à un *mot identifié par la fonction **mapFunc***
  - une ***liste de valeurs*** contenant le *nb d'occurrences de ce mot*
- la liste des valeurs correspond à l'ensemble des paires (mot, count) mises en zones intermédiaires par la fonction ***mapFunc***
- pour comptabiliser tous les mots du document, on initialise la variable ***result*** à 0, puis on itére sur l'ensemble des valeurs de la liste, puis chaque valeur est rajoutée à la variable ***result***
- à la fin de l'itération, la variable ***result*** contient le total du nb d'occurrence et le résultat est renvoyé par la fonction ***EmitResult***

Fondements des syst. NoSQL : Exemple MapReduce (3)

- Soit le **fichier du document en entrée contenant 3 lignes** (fractionnement du fichier en 3)
- chaque **ligne contient 3 lettres (AAC, CBD, ACD)**
- **traitement MapReduce effectué** pour compter les mots du document :



Fondements des syst. NoSQL : Exemple MapReduce (4)

Les étapes réalisées sont :

- **L'étape File** : on lit le fichier en entrée et on initialise les différents « *Workers MapReduce* »
- **L'étape Splitting** : on distribue les données à traiter sur les différents noeuds du cluster de traitement
- **L'étape Map** : on effectue le compte de chacune des lettres et ceci en local sur chaque noeud du cluster de traitement
- **L'étape Suffling** : on regroupe toutes les lettres ainsi que leur compte à partir de tous les noeuds de traitement
- **L'étape Reduce** : on effectue le cumule de toutes les valeurs de chaque lettre
- **L'étape Result** : on agrège tous les résultats des différentes étapes Reduce et on retourne le résultat final.

Fondements des syst. NoSQL : MVCC

Contrôle de Concurrence Multi-Version (MVCC) :

- Méthode de **contrôle de concurrence** couramment utilisée par les SGBD pour gérer des accès simultanés à la base de données avec mises à jour
- Dans une **BD NoSQL**, la gestion des mises à jour est faite :
  - non pas en *supprimant* une fraction contenant les données avant modification et en la *remplaçant* par une fraction contenant les données modifiées
  - mais en **marquant** les anciennes données comme **obsolètes** et en **ajoutant** une **nouvelle version** contenant les nouvelles données
  - il existe ainsi **plusieurs versions enregistrées**, une seule est la plus récente
- nécessite généralement un **balayage périodique** pour **supprimer** les objets de données obsolètes.

Fondements des syst. NoSQL : Vector-clock

Horloges vectorielles (Vector-clocks) :

- Les ensembles de données répartis sur nœuds peuvent être lus et modifiés sur chaque nœud et aucune cohérence stricte est assurée par des protocoles de transactions distribuées

  Problème** : comment faire des **modifications concurrentes
- Une **solution** : les **horloges vectorielles** :
  - un ***vecteur d'horloge*** est défini comme un n-uplet V [0], V [1], ..., V[n] des *valeurs d'horloge* à partir de chaque noeud.
  - **à tout instant le noeud i maintient un vecteur d'horloge** représentant son état et celui des autres nœuds répliques : (Vi [0] = valeur de l'horloge du premier noeud, Vi [1] = valeur de l'horloge du deuxième noeud, ... Vi [i] = sa propre valeur d'horloge, ... Vi [n] = valeur de l'horloge du dernier nœud)
  - les valeurs d'horloge peuvent être de réelles « timestamps » dérivées d'une horloge locale de nœud, du numéro de version/révision …

Typologie des BD NoSQL (1)

Stocker les informations de la **façon la mieux adaptée** à leur représentation => **différents types de BD NoSQL :**

- **type « Clé-valeur / Key-value » :** basique, chaque objet est identifié par une clé unique constituant la seule manière de le requêter
  - ***Voldemort, Redis, Riak, …***
- **type « Colonne / Column »** : permet de disposer d'un très grand nb de valeurs sur une même ligne, de stocker des relations « one-to-many », d'effectuer des requêtes par clé (adaptés au stockage de listes : messages, posts, commentaires, ...)
  - ***HBase, Cassandra, Hypertable, …***
- **type « Document »** : pour la gestion de collections de documents, composés chacun de champs et de valeurs associées, valeurs pouvant être requêtées (adaptées au stockage de profils utilisateur)
  - ***MongoDBn CouchDB, Couchbase, …***
- **type « Graphe »** : pour gérer des relations multiples entre les objets (adaptés au données issues de réseaux sociaux, …)
  - ***Neo4j, OrientDB, …***

Typologie des BD NoSQL (2)

	Key/Value Store	Column Store	Document Store	Graph Store
Design	Key/Value pairs; indexed by Key	Columns and Column Families. Directly accesses the column values	Multiple Key/Value pairs form a document. Values may be nested documents or lists as well as scalar values	Focus on the **connections** between data and fast navigation through these connections
Scalability / Performance	+++	+++	++	++
Aggregate-Oriented	Yes	Yes	Yes	No
Complexity	+	++	++	+++
Inspiration / Relation	Berkley DB, Memcached, Distributed Hashmaps	SAP Sybase IQ, Google BigTable	Lotus Notes	Graph Theory
NOSQL Products	Voldemort Redis Riak	HBase Cassandra Hypertable	MongoDB CouchDB Couchbase	Neo4j OrientDB DEX InfiniteGraph [Triple and Quad Stores]

3. BD NoSQL « Clé-Valeur »

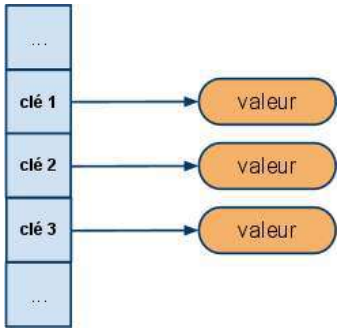
- **BD NOSQL modèle « Clé-Valeur »**
- **Forces & faiblesses des BD NoSQL « Clé-Valeurs »**

BD NOSQL modèle « Clé-Valeur » (1)

- Elles fonctionnent comme un **grand tableau associatif** et retourne une valeur dont elle ne connaît pas la structure
- leur modèle peut être assimilé à une **table de hachage (hashmap) distribuée**
- les données sont simplement **représentées** par un **couple clé/valeur**
- la valeur peut être une **simple chaîne de caractères**, ou un objet sérialisé…
- cette absence de structure ou de typage ont un impact important sur le requêtage : toute l'intelligence portée auparavant par les requêtes SQL devra être portée par **l'applicatif** qui interroge la BD.
- Implémentations les plus connues :
  - **Amazon Dynamo** (**Riak** en est l'implémentation Open Source)
  - **Redis** (projet sponsorisé par VMWare)
  - **Voldemort** (développé par LinkedIn en interne puis passage en open source).

BD NOSQL modèle « Clé-Valeur » (2)

- Chaque objet est identifié par une **clé unique** seule façon de le requêter
- la **structure de l'objet est libre**, souvent laissé à la charge du développeur de l'application (XML, JSON, ...), la base ne gérant généralement que des chaînes d'octets



BD NOSQL modèle « Clé-Valeur » (3)

- Leur exploitation est basée sur **4 opérations (CRUD):**
  - **C reate** : créer un nouvel objet avec sa clé → create(key, value)
  - **R ead** : lit un objet à partir de sa clé → read(key)
  - **U pdate** : met à jour la valeur d'un objet à partir de sa clé → update(key, value)
  - **D elete**: supprime un objet à partir de sa clé → delete(key)
- elles disposent généralement d'une simple **interface de requêtage HTTP REST** accessible depuis n'importe quel langage de développement
- ont des **performances très élevées** en lecture et en écriture et une **scalabilité horizontale considérable**
- le besoin en **scalabilité verticale est faible** du fait de la simplicité des opérations effectuées

Utilisations principales des BD NoSQL « Clés-Valeurs »

Utilisations principales des BD NoSQL type « Clés-Valeurs » :

- dépôt de données avec besoins de requêtage très simples
- système de stockage de cache ou d'information de sessions distribuées (quand l'intégrité relationnelle des données est non significative)
- les profils, préférences d'utilisateur
- les données de panier d'achat
- les données de capteur
- les logs de données
- …

Forces & faiblesses des BD NoSQL « Clé-Valeurs »

Forces :

- **modèle de données simple**
- **bonne mise à l'échelle horizontale** pour les lectures et écritures :
  - **évolutivité** (scalable)
  - **disponibilité**
  - **pas de maintenances requises** lors d'ajout/suppression de colonnes

Faiblesses :

- **modèle de données TROP simple :**
  - **pauvre** pour les **données complexes**
  - **interrogation seulement** sur **clé**
  - **déporte** une grande partie de la **complexité** de l'application sur la **couche application** elle-même

4. BD NoSQL « Colonne »

- **BD NOSQL modèle « Colonne » Xxxx**
- **Forces & faiblesses des BD NoSQL « Colonne »**

BD NOSQL modèle « Colonne » (1)

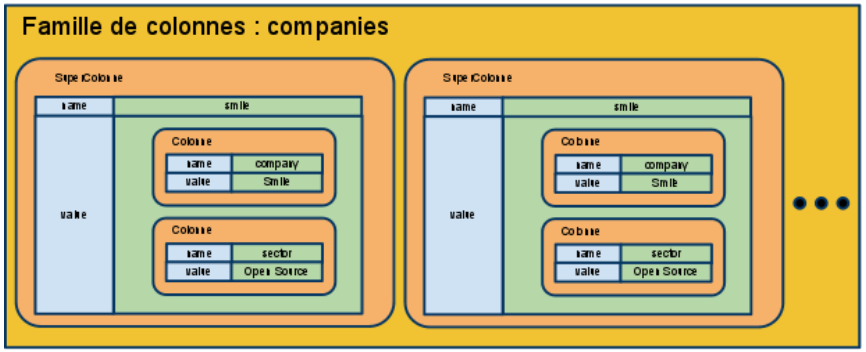
- Les données sont stockées par **colonne**, non par ligne
- on peut facilement **ajouter des colonnes** aux tables, par contre l'insertion d'une ligne est plus coûteuse
- quand les données d'une colonne se ressemblent, on peut facilement compresser la colonne
- modèle **proche d'une table dans un SGBDR** mais ici le nombre de colonnes :
  - est **dynamique**
  - peut **varier d'un enregistrement à un autre** ce qui évite de retrouver des colonnes ayant des valeurs NULL.
- Implémentations les plus connues :
  - **HBase** (Open Source de **BigTable** de Google utilisé pour l'indexation des pages web, Google Earth, Google analytics, ...)
  - **Cassandra** (fondation Apache qui respecte l'architecture distribuée de Dynamo d'Amazon, projet né de chez Facebook)
  - **SimpleDB** de Amazon.

BD NOSQL modèle « Colonne » (2)

Les **principaux concepts** associés sont les suivants :

- **Colonne :**
  - entité de base représentant un **champ de donnée**
  - chaque colonne est **définie par un couple clé / valeur**
  - une colonne contenant d'autres colonnes est nommée ***super-colonne***.
- **Famille de colonnes :**
  - permettent de **regrouper plusieurs colonnes** (ou super-colonnes)
  - les **colonnes sont regroupées par ligne**
  - **chaque ligne est identifiée par un identifiant unique** (assimilées aux tables dans le modèle relationnel) et sont **identifiées par un nom unique**
- **Super-colonnes :**
  - situées dans les familles de colonnes sont **souvent utilisées comme les lignes d'une table de jointure** dans le modèle relationnel.

BD NOSQL modèle « Colonne » (3)

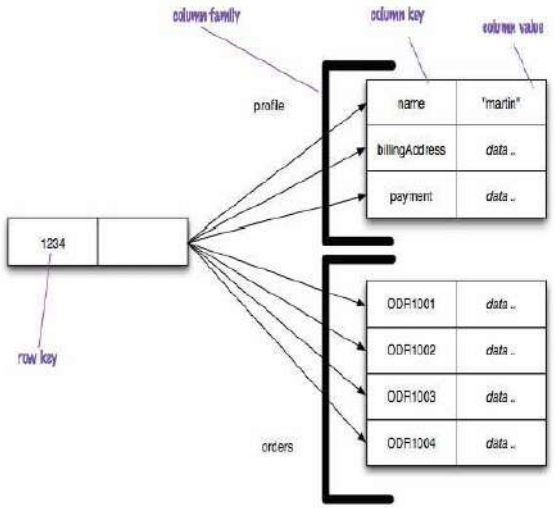


BD NOSQL modèle « Colonne » (4)

- Elles sont les plus **complexes** à appréhender des BD NoSQL, même si au final on a un schéma assez proche des bases documentaires
- elles sont très utilisées pour les **traitements d'analyse de données** et dans les **traitements massifs** (notamment via des opérations de type *MapReduce*).
- elles offrent **plus de flexibilité** que les BD relationnelles:
  - Il est possible **d'ajouter** une **colonne** ou
  - une **super colonne**
  - **à n'importe quelle ligne**
  - d'une **famille de colonnes**, **colonnes** ou **super-colonne** à tout instant.

BD NOSQL modèle « Colonne »

- *tuple* pour le client 1234
- *table* « Clients » partitionnée en 2 familles de colonnes : « Profile » et « Orders »
- chaque *famille de colonnes* a des *colonnes* (e.g. name et payment), des *super colonnes* avec un nom et un nombre arbitraire de colonnes associées
- chaque *famille de colonnes* peut être traitée comme une *table séparée* en terme de partitionnement (sharding) :
  - « Profile » pour le client 1234 peut être sur le nœud 1
  - « Orders » pour le client 1234 peut être sur le noeud 2



Forces & faiblesses des BD NoSQL « Colonne »

Forces :

- **Modèle de données** supportant des **données semi-structurées** (clairsemées)
- **naturellement indexé** (colonnes)
- **bonne mise à l'échelle à l'horizontale**
- **MapReduce** souvent utilisé en **scaling horizontal**,
- on peut voir les résultats de requêtes en temps réel

Faiblesses :

- A éviter pour des données interconnectés : si les relations entre les données sont aussi importantes que les données elles-mêmes (comme la distance ou calculs de la trajectoire)
- à éviter pour les lectures de données complexes
- exige de la maintenance - lors de l'ajout / suppression de colonnes et leur regroupements
- les requêtes doivent être pré-écrit, pas de requêtes ad-hoc définis "à la volée": NE PAS utiliser pour les requêtes non temps réel et inconnues.

Utilisations principales des BD NoSQL « Colonne »

- Les BD NoSQL type « Colonne » sont principalement utilisées pour :
  - Netflix l'utilise notamment pour le ***logging*** *et* ***l'analyse de sa clientèle***
  - Ebay l'utilise pour ***l'optimisation de la recherche***
  - Adobe l'utilise pour le ***traitement des données structurées*** et de ***Business Intelligence*** (BI)
  - Des sociétés de TV l'utilisent pour ***cerner leur audience*** et gérer le ***vote des spectateurs*** (nb élevé d'écritures rapides et analyse de base en temps réel (Cassandra)
  - peuvent être de ***bons magasins d'analyse des données semi-structurées***
  - utilisé pour la ***journalisation des événements*** et pour des ***compteurs***
  - …

5. BD NoSQL « Document »

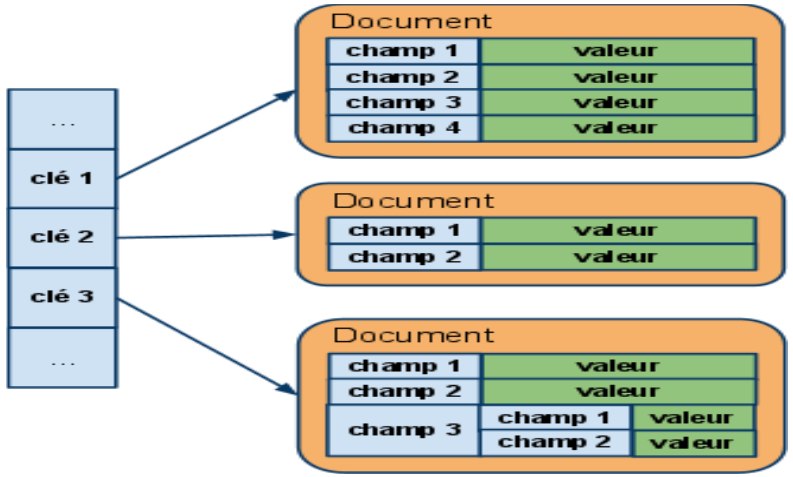
BD NOSQL modèle « Document » (1)

- Elles stockent une collection de "**documents**"
- elles sont basées sur le modèle « clé-valeur » mais la valeur est un **document** en **format semi-structuré hiérarchique** de type **JSON** ou **XML** (possible aussi de stocker n'importe quel objet, via une sérialisation)
- les **documents** n'ont **pas de schéma,** mais une **structure arborescente** : ils contiennent une liste de champs, un champ a une valeur qui peut être une liste de champs, ...
- elles ont généralement une **interface d'accès HTTP REST** permettant d'effectuer des requêtes (plus complexe que l'interface CRUD des BD clés/valeurs)
- Implémentations les plus connues :
  - **CouchDB** (fondation Apache)
  - **RavenDB** (pour plateformes « .NET/Windows » - LINQ)
  - **MongoDB, Terrastore, …**

BD NOSQL modèle « Document » (2)

- Un **document** est composé de **champs** et des **valeurs associées**
- ces **valeurs** :
  - peuvent être **requêtées**
  - sont soit d'un **type simple** (entier, chaine de caractère, date, ...)
  - soit elles-mêmes **composées** de plusieurs couples clé/valeur.
- bien que les documents soient structurés, ces BD sont dites "**schemaless**" : il n'est **pas nécessaire de définir au préalable les champs** utilisés dans un document
- les documents peuvent être très **hétérogènes** au sein de la BD
- permettent d'**effectuer des requêtes sur le contenu** des documents/objets : pas possible avec les BD clés/valeurs simples
- Elles sont principalement utilisées dans le **développement de CMS** (Content Management System - outils de gestion de contenus).

BD NOSQL modèle « Document » (3)



Forces & faiblesses des BD NoSQL « Document »

Forces :

- **modèle de données simple mais puissant** (expression de structures imbriquées)
- **bonne mise à l'échelle** (surtout si sharding pris en charge)
- **pas de maintenance de la BD requise** pour ajouter/supprimer des «colonnes»
- **forte expressivité de requêtage** (requêtes assez complexes sur des structures imbriquées)

Faiblesses :

- **inadaptée pour les données interconnectées**
- **modèle de requête limitée à des clés** (et indexes)
- peut alors être **lent pour les grandes requêtes** (avec MapReduce)

Utilisations principales des BD NoSQL « Document »

- **Les BD NoSQL type « Document » principalement utilisées pour :**
  - Enregistrement d'événements
  - Systèmes de gestion de contenu
  - Web analytique ou analytique temps-réel
  - Catalogue de produits
  - Systèmes d'exploitation
  - …

5. BD NoSQL « Graphe »

- **BD NOSQL modèle « Graphe »**
- **Forces & faiblesses des BD NoSQL « Graphe »**

BD NOSQL modèle « Graphe » (1)

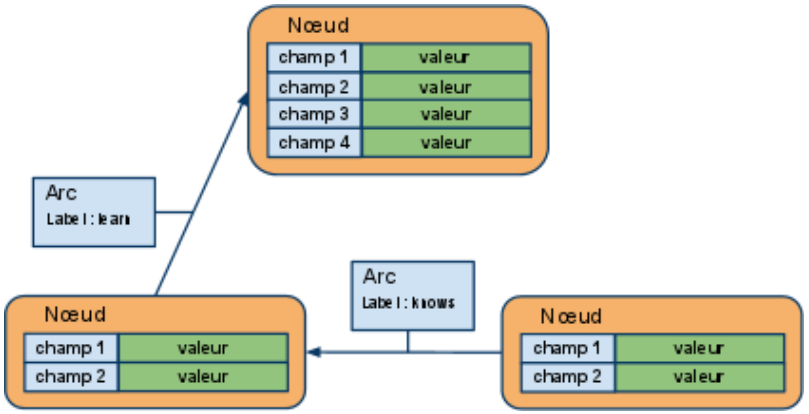
- Elles permettent la **modélisation**, le **stockage** et la **manipulation** de **données complexes liées par des relations** non-triviales ou variables
- modèle de représentation des données basé sur la **théorie des graphes**
- s'appuie sur les notions de **noeuds**, de **relations** et de **propriétés** qui leur sont rattachées.
- Implémentations les plus connues :
  - **Neo4J**
  - **OrientDB** (fondation Apache)
  - **…**

BD NOSQL modèle « Graphe » (2)

- **Elles utilisent :**
  - un **moteur de stockage** pour les **objets** (similaire à une base documentaire, chaque entité de cette base étant nommée nœud)
  - un **mécanisme de description d'arcs** (relations entre les objets), arcs orientés et avec propriétés (nom, date, ...)
- elles sont **bien plus efficaces que les BDR** pour traiter les problématiques liées aux réseaux (cartographie, relations entre personnes, ...)
- sont adaptées à la manipulation **d'objets complexes organisés en réseaux** : **cartographie**, **réseaux sociaux**, ..

BD NOSQL modèle « Graphe » (3)

- .



Forces & faiblesses des BD NoSQL « Graphe »

Forces :

- **modèle de données puissant**
- rapide pour les **données liées**, bien **plus rapide que SGBDR**
- **modèles d'interrogation bien établis et performants** : Tinkerpop pile (fournit un ensemble commun d'interfaces permettant aux différentes technologies informatiques graphiques de travailler ensemble, que le développeur utilise en cas de besoin), SPARQL et Cypher

Faiblesses :

- **Fragmentation** (sharding) :
  - Même si elles peuvent évoluer assez bien
  - Pour certains domaines, on peut aussi fractionner.

Utilisations principales des BD NoSQL « Graphe »

- Les BD NoSQL type « Document » sont principalement utilisées pour :
  - **Moteurs de recommandation**
  - **Business Intelligence (BI)**
  - **Semantic Web**
  - Social computing
  - Données géospatiales
  - Généalogie
  - Web of things
  - Catalogue des produits
  - Sciences de la Vie et calcul scientifique (bioinformatique, …)
  - Données liées, données hiérarchiques
  - Services de routage, d'expédition et de géolocalisation
  - Services financiers : chaîne de financement, dépendances, gestion des risques, détection des fraudes, …

BD NoSQL « Graphe » et Web Sémantique

Magasins de triplets RDF (Triple Stores) :

- the foundation of many Semantic Web systems
- encodés in format/langage **RDF**
- chaque ligne a une **structure «nœud-lien-noeud»** (sujet-prédicat-objet)
- possibilité de **joindre** des graphes ensemble automatiquement en faisant correspondre les identifiants des nœuds
- possibilité de **fusion automatique** de 2 graphes

  Ex: le graphe 1 a le noeud A relié à B et le graphe 2 le nœud B relié à C, l'union de ces graphes montre une relation de A à C.
- les **données RDF** interrogées via le protocole/langage de requête **SPARQL** permettant l'utilisation d'ontologies pour l'inférence *(Groupe W3C RDF Data Access de travail)*
- Ex : Virtuoso, Sesame, **Jena**

BD NoSQL « Graphe » : le futur ?

- Internet : net of computers
- Word WideWeb : web of documents
- (GGG) Giant Global Graph : graph of metadata

"I called this graph the Semantic Web, but maybe it should have been Giant Global Graph." - Tim Berners-Lee – 2007