MongoDB

Table des matières

I - MongoDB – présentation

1. Grandes lignes de mongoDB

MongoDB est une **base de données "No-SQL"** de la famille "**orientée document**".

Une base de données "mongo" n'est pas structurée comme un ensemble de tables relationnelles reliées entre elles par les jointures (clef-étrangères référençant des clefs primaires).

Une base "mongoDB" est simplement structurée en un ensemble de collections de documents/enregistrements au format "BSON / JSON" .

Cette structure au premier abord un peu simpliste permet d'obtenir de bonnes performances dans un contexte de répartition en mode "cloud computing" et/ou dans le cas où les données sont très nombreuses et assez peu reliées entre elles (ex : enregistrement d'infos "spatio-temporelles" à ultérieurement analyser d'un point de vue statistique : big data ou …) .

Il est tout de même possible d'exploiter (via des traitements "ad-hoc") certains liens logiques (ref → id) entre documents/enregistrements "JSON/BSON" de façon obtenir un comportement proche de celui d'une base de données relationnelles .

BSON correspond à une version binarisée du format JSON et est utilisé en interne comme format d'enregistrement des données sur le disque.

1.1. Points forts et points faibles de mongoDB

Points forts de mongoDB	Point faibles de mongoDB
• Supporte de gros volumes • Mode distribué (réplication, sharding , ….) • Simple et efficace • données au format JSON • ...	• Moins de possibilités dans les requêtes qu'avec le traditionnel SQL • Moins structuré , moins de contraintes d'intégrité • ...

1.2. Comparaison rapide "SQL - mongo"

SQL	*MongoDB*
Tables (avec colonnes figées)	**Collections** d'enregistrements/documents JSON (avec éventuelle variété possible sur parties secondaires)
INSERT INTO Client **VALUES** (1 , "…" , "…")	db.clients.**insert**({id:1 , nom:"..." , prenom: "…"})
SELECT * FROM Client	db.clients.**find**()
SELECT * FROM Client **WHERE** nom='toto'	db.clients.find(**{** nom : "toto" **}**)
SELECT * FROM Client WHERE nom LIKE '%D%'	db.clients.find({ nom : /D/ })
DELETE FROM Client WHERE nom = 'toto'	db.clients.**remove**({ nom : "toto" })
UPDATE Client **SET** prenom='alex' WHERE nom='Therieur'	db.clients.**update**({nom: "Therieur"}, {**$set: {** prenom: "alex" **}**})

II - Installation et utilisation de mongoDB

1. Téléchargement et pré-installation de mongoDB

mongodb-win32-x86_64-2008plus-ssl-3.2.1-signed.msi (*environ 100 Mo* à télécharger depuis "***https://www.mongodb.org/downloads****#production*") permet une installation sous windows .

L'installation s'effectue par défaut dans **C:\Program Files\MongoDB\Server\3.2** .

2. Installation et configuration élémentaire MongoDB

Pour configurer une base de données sur l'ordinateur local , on peut :

1) préparer un répertoire (par exemple : C:\Prog\MongoDB)

2) préparer (par exemple) les sous répertoires suivants :
- **data/db** *(pour les fichiers internes des données de la base)*
- **data/log** *(pour les fichiers de log)*
- **dataset** *(pour import/export de fichiers ".json")*

3) écrire un petit fichier de configuration "**mongodb.cfg**" :

```
systemLog:
   destination: file
   path: C:\Prog\MongoDB\data\log\mongod.log
storage:
   dbPath: C:\Prog\MongoDB\data\db
```

4) lancer éventuellement l'installation de "mongoDB" en tant que service windows (en tant qu'administrateur) :

install_MongoDB_service_as_admin.bat

```
set MONGO_HOME=C:\Program Files\MongoDB\Server\3.2
"%MONGO_HOME%\bin\mongod.exe" --config "C:\Prog\MongoDB\mongodb.cfg" --install
pause
```

5) démarrer ou arrêter si nécessaire le service "mongoDB" (en tant qu'administrateur) :

net_start_MongoDB_service_as_admin.bat

```
net start MongoDB
pause
```

net_stop_MongoDB_service_as_admin.bat

```
net stop MongoDB
pause
```

ou bien effectuer un démarrage direct (sans service windows préalablement installé) :

```
"%MONGO_HOME%\bin\mongod.exe" --dbpath "C:\Prog\MongoDB\data\db"
```

3. Eventuel hébergement cloud via MongoLab

L'entreprise "**MongoLab**" se propose d'héberger des bases de données au format "mongoDB" sur un "Cloud" (ex : Amazon , Google ,Azure) .

Cette entreprise propose une version gratuite "***free / sandBox / limitée à 500Mo***" qui est suffisante pour effectuer quelques essais rapides en mode "développement" .

Mode opératoire :

- Créer (gratuitement) un compte pour l'administration de la base . Mémoriser les paramètres.
- Valider l'adresse e-mail pour activer le compte "mongoLab" . Se connecter.
- Choisir une souscription de service (ex : " free/sandbox" ou …)
- Créer une nouvelle base de données (ex : "**my_mongo_db**")
- Créer un utilisateur (ex : "*powerUser*" , "secret*Pwd*")
- Lire les paramètres (URL de la base , …)
- Administrer et gérer la base à distance (via mongo , mongoimport , ...)

4. Import de données au format ".json"

L'option --drop demande à d'abord supprimer tous les éléments de la collection avant d'importer le contenu du fichier .json .

L'option --jsonArray permet d'importer un tableau d'objet "json" si le fichier à importé est structuré de cette façon (par défaut le fichier .json ne comporte qu'un document/objet à importer).

import_clients_dataset_in_test_db.bat

```
set MONGO_HOME=C:\Program Files\MongoDB\Server\3.2
cd /d "%~dp0"
"%MONGO_HOME%\bin\mongoimport" --db test    --collection users    --drop
                                          --file dataset/clients.json   --jsonArray
pause
```

NB: La base locale par défaut est test , sans username ni password

Si l'on veut initialiser une base distante (hébergée par exemple par MongoLab) , il faudra alors préciser l'URL de celle ci ainsi que les informations d'authentification (username, password) :

```
"%MONGO_HOME%\bin\mongoimport" -h ds041494.mongolab.com:41494 --db my_mongo_db
--collection comptes  --drop -u powerUser -p secretPwd --file ../dataset/comptes.json    --jsonArray
```

Le répertoire relatif ***dataset*** comporte par exemple les fichiers suivants :

clients.json

```
[
{
"_id" : 1,
"comptes" : [ 1 , 2 ],
"nom": "Therieur",
"prenom": "alain",
"adresse" : { "idAdr" :1 ,"rue" : "123 - rue elle" , "codePostal" : "75000", "ville" : "Par ici"  },
"telephone" : "0102030405" ,
"email" : "alain.therieur@ici-ou-la-bas.fr"
}
]
```

comptes.json

```
[
{
"_id" : 1 ,
"label" : "compte 1 (courant) - mongoDB",
"solde" : 601.0
},
{
"_id" : 2 ,
"label" : "compte 2 (LDD) - mongoDB",
"solde" : 3201.0
},
{
"_id" : 3 ,
"label" : "compte 3 (PEL) - mongoDB",
"solde" : 6501.0
}
]
```

5. "Shell" pour dialogue interactif avec mongoDB

lancer_mongo_shell.bat

```
set MONGO_HOME=C:\Program Files\MongoDB\Server\3.2
cd /d "%~dp0"
"%MONGO_HOME%\bin\mongo.exe"
pause
```

Et dans le cas d'une base distante :

```
mongo  ds041494.mongolab.com:41494/my_mongo_db  -u powerUser -p secretPwd
```

Exemples de commandes (mongo shell) :

db.*collectionName*.find();
db.clients.find();

…

6. URI/URL de connexion à une base mongoDB

(ex : depuis code nodeJs) :

```
mongodb://<dbuser>:<dbpassword>@ds041494.mongolab.com:41494/my_mongo_db
```

exemple :

```
mongodb://powerUser:secretPwd@ds041494.mongolab.com:41494/my_mongo_db
```

III - Instructions et requêtes - mongoDB

1. Structure d'une base mongoDB

Collections d'enregistrements/documents JSON .

Par défaut , l'identifiant (unique) d'un document JSON enregistré dans une base mongoDB est **_id** .

2. Principales instructions - mongoDB

2.1. Insertions

```
db.clients.insert({ _id:1 , nom:"..." , prenom: "…"})
```

NB : dans le cas d'un insert ou le _id n'est pas précisé , un **_id** est **généré par défaut** au format **ObjectId("a2334bc...ac445788")** .

```
db.clients.insert({ nom:"veut" , prenom: "quiNen"})
```

2.2. Recherches

db.clients.**find**()

db.clients.find({ nom : "toto" })

db.clients.find({ nom : /D/ }) *clients avec nom comportant D*

db.clients.find({ _id : { **$in : [1, 5, 8]** } }) *clients dont le _id est dans [1, 5 , 8]*

2.3. Suppressions

db.clients.**remove**({ nom : "toto" })

2.4. Mises à jour

db.clients.**update**({nom: "Therieur"},
{**$set:** { prenom: "alex" }})

3. Aggregation (pipeline)

Une requête de type ".**aggregate**()" permet d'enchaîner une suite de traitements afin de produire un résultat "json" très spécifique (ex : somme / moyenne après un équivalent de "group-by" SQL).

.aggregate() est à paramétrer par un tableau de traitements qui seront automatiquement déclenchés les uns après les autres via une logique de "pipeline" : la sortie d'un traitement N est l'entrée du traitement N+1 .

Exemple concret:

Soit la collection "***operations***" au format suivant :

```
[… ,
{
"_id" : 3,
"compte" : 1,
"label" : "salaire - mongoDB",
"montant" : 2001,
"dateOp" : "2015-03-18"
}
]
```

Alors la commande suivante

```
db.operations.aggregate([
        {
        $match : { montant : { $lt : 2500 } }
        },
        {
        $group : { _id : "$compte", total : { $sum : 1 }}
         },
        {
        $sort : { total : -1}
        }
         ]);
```

Va appliquer dans l'ordre :

1) un filtrage sur les opérations de montant < 2500

2) un regroupement des opérations rattachées à un même compte et pour chaque regroupement le nombre d'opérations

3) un tri par ordre décroissant(-1)

et retourner par exemple les valeurs suivantes :

```
{ "_id" : 1, "total" : 2 }
{ "_id" : 2, "total" : 1 }
```

NB : avec la variante **$group : { *_id* : "$*compte*", *total* : { $sum : "$montant" }}**
on aurait obtenu la somme des montants des opérations associées à chaque compte .

4. Jointures (depuis mongoDB 3.2)

Depuis la version 3.2 de MongoDB, il est possible d'utiliser $lookup() à l'intérieur de .aggregate() de façon à obtenir un à peu près équivalent d'une jointure entre deux collections.
Le paramétrage "from" correspond à la collection à joindre / intégrer , "foreignField" est souvent le _id de cette collection et "as" correspond au nom de la propriété à produire (il est possible de choisir la même valeur que celle de "localField" si l'on souhaite écraser la valeur de liaison/référencement.

Par défaut $lookup() produit un tableau .
Dans le cas d'une ralation "one-to-one" ou "many-to-one" , on pourra combiner $lookup() avec $unwind (dans .aggregate()) pour obtenir un résultat unique (sans []) .

Exemple :

```
db.operations.aggregate([
        {
        $lookup : {
                "localField": "compte",
                "from": "comptes",
                "foreignField": "_id",
                "as": "compteForThisOp"
                }
        }
  ]);
```

retourne :
{ "_id" : 1, "compte" : 1, "label" : "achat xy - mongoDB", "montant" : -51, "dateOp" : "2015-01-20", "compteForThisOp" : [{ "_id" : 1, "label" : "compte 1 (courant) - mongoDB", "solde" : 551 }] }
{ "_id" : 2, "compte" : 2, "label" : "achat zz - mongoDB", "montant" : -91, "dateOp" : "2015-02-08", "compteForThisOp" : [{ "_id" : 2, "label" : "compte 2 (LDD) - mongoDB", "solde" : 3251 }] }

et

```
db.operations.aggregate([
        {
        $lookup : {
                "localField": "compte",
                "from": "comptes",
                "foreignField": "_id",
                "as": "compte"
                }
        },
        {
        "$unwind": "$compte"
        }
  ]);
```

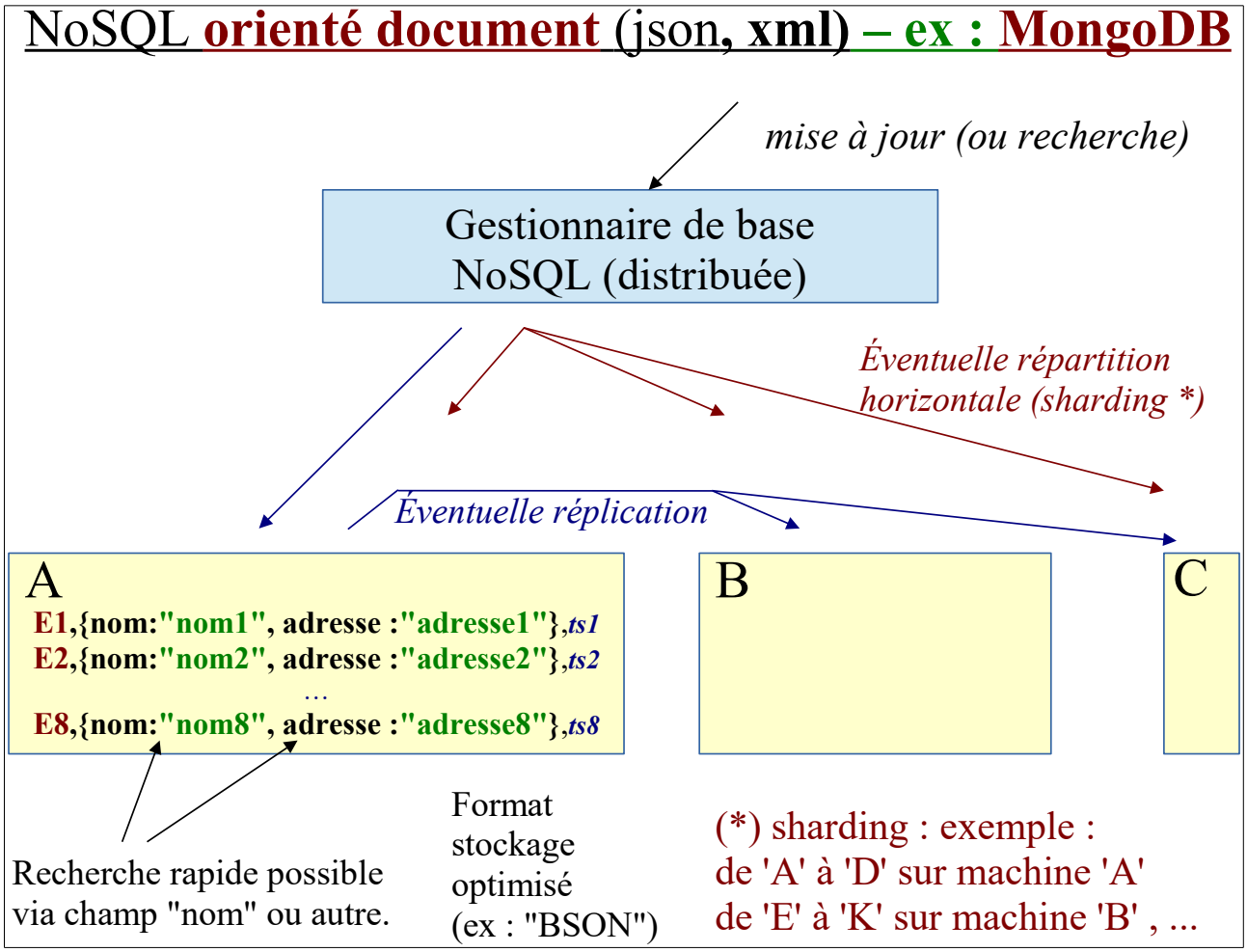
retourne :

{ "_id" : 1, "compte" : { "_id" : 1, "label" : "compte 1 (courant) - mongoDB", "solde" : 551 }, "label" : "achat xy - mongoDB", "montant" : -51, "dateOp" : "2015-01-20" }
{ "_id" : 2, "compte" : { "_id" : 2, "label" : "compte 2 (LDD) - mongoDB", "solde" : 3251 }, "label" : "achat zz - mongoDB", "montant" : -91, "dateOp" : "2015-02-08" }

IV - Admin mongoDB (réplication , sharding, ...)

1. Vue d'ensemble - administration de mongoDB

2. Replication et Sharding



V - MongoDB – aspects divers et avancés

1. Indexation

....

2. Autres aspects divers et avancés

…

ANNEXES

VI - Annexe – Généralités sur le cloud computing

1. Le "Cloud Computing"

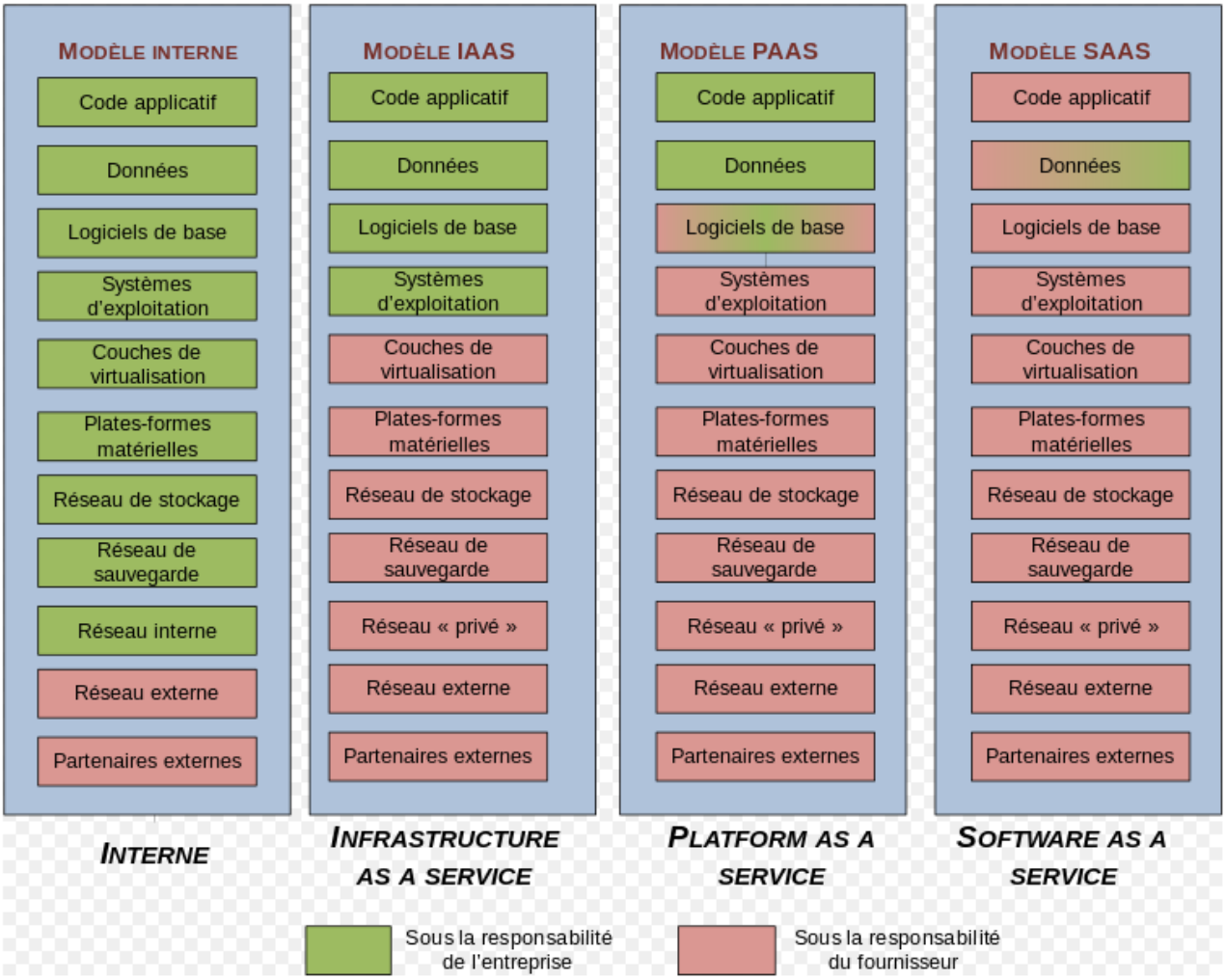
1.1. Architecture / terminologie "Cloud"

Un nuage (anglais *cloud*) est un une métaphore désignant un ensemble de matériel, de raccordements réseau et de logiciels qui fournit des services sophistiqués que les individus et les collectivités peuvent exploiter à volonté depuis n'importe où dans le monde.

Le *cloud computing* est un basculement de tendance : au lieu d'obtenir de la puissance de calcul par acquisition de matériel et de logiciel, le consommateur se sert de puissance mise à disposition par un fournisseur via Internet.

Les caractéristiques essentielles d'un nuage sont la disponibilité mondiale en self-service, l'élasticité, l'ouverture, la mutualisation et le paiement à l'usage :

- *ressources en self-service*, et adaptation automatique à la demande. La capacité de stockage et la puissance de calcul sont adaptées automatiquement au besoin d'un consommateur. Ce qui contraste avec la technique classique des hébergeurs où le consommateur doit faire une demande écrite à son fournisseur en vue d'obtenir une augmentation de la capacité - demande dont la prise en compte nécessite évidemment un certain temps. En *cloud computing* la demande est automatique et la réponse est immédiate.
- *ouverture*. Les services de *cloud computing* sont mis à disposition sur l'Internet, et utilisent des techniques standardisées qui permettent de s'en servir aussi bien avec un ordinateur qu'un téléphone ou une tablette.
- *mutualisation*.La mutualisation permet de combiner des ressources hétérogènes (matériel, logiciel, trafic réseau) en vue de servir plusieurs consommateurs à qui les ressources sont automatiquement attribuées. La mutualisation améliore la scalabilité et l'élasticité et permet d'adapter automatiquement les ressources aux variations de la demande.
- *paiement à l'usage*: la quantité de service consommée dans le *cloud* est mesurée, à des fins de contrôle, d'adaptation des moyens techniques et de facturation



Niveaux *(aas=as a service)*	**Caractéristiques**
SAAS (Software aas)	On loue un logiciel entièrement géré par le fournisseur (ServApp + Base_données , ….). On a simplement besoin de postes clients avec des navigateurs internet . Seule la sauvegarde des données (copies locales) est facultativement à gérer.
PAAS (Platform aas)	On loue une plate-forme logicielle d'un certain type (ex : Serv App Java/Jee Tomcat" + bases MySql ou …..) ou bien "Node-js" + "MongoDB") entièrement gérée par un fournisseur d'hébergement puis l'on déploie de code d'une application à faire fonctionner
IAAS (Infrastructure aas)	On loue des machines virtuelles auprès d'un fournisseur d'hébergement (ex : Gandi , OVH) avec par exemple un O.S. "linux" , puis on installe la base de données et le serveur d'applications que l'on préfère (ex : Mysql et Tomcat7) pour ensuite installer et faire fonctionner des applications.
Tout en interne	On achète les ordinateurs "serveurs" , les "systèmes

	d'exploitation" , les logiciels "SGBDR" , "compta" , "…" . On administre tout nous même.

Plus on s'appuie sur une couche basse , plus on a de liberté pour choisir l'O.S. , les logiciels que l'on préfère mais on doit gérer soit même plein de choses.

Plus on s'appuie sur une couche haute , moins on à de choses à gérer mais plus on devient dépendant de l'offre (large ou pas) et de la qualité de service (bonne ou pas) du fournisseur.

La maîtrise d'œuvre sera souvent intéressée par une offre Iaas ou Paas . Cela lui permettra de fournir sa "valeur ajoutée" sous forme de logiciel à déployer et faire fonctionner de façon à ensuite satisfaire les besoins d'une maîtrise d'ouvrage.

Spécificité importante d'une application Saas :
- bien gérer les sauvegardes des données
- bien gérer la sécurité et la disponibilité
- bien gérer la séparation des données "client1" , "client2" , … , "clientN" si une application Saas est partagée/louée par plusieurs clients

1.2. Technologies pour le "cloud computing"

Windows Azure (en tant que plate-forme),
→ Plateforme "Cloud" de *Microsoft*

Amazon Web Services (AWS)
→ Offre cloud de la société Amazon (historiquement une des premières offres)

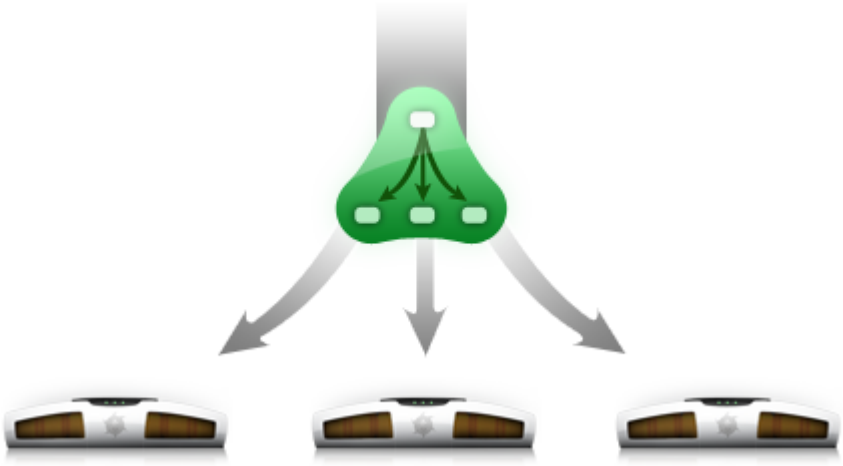
Google App Engine
→ API ...

1.3. Exemples d'hébergements "cloud"

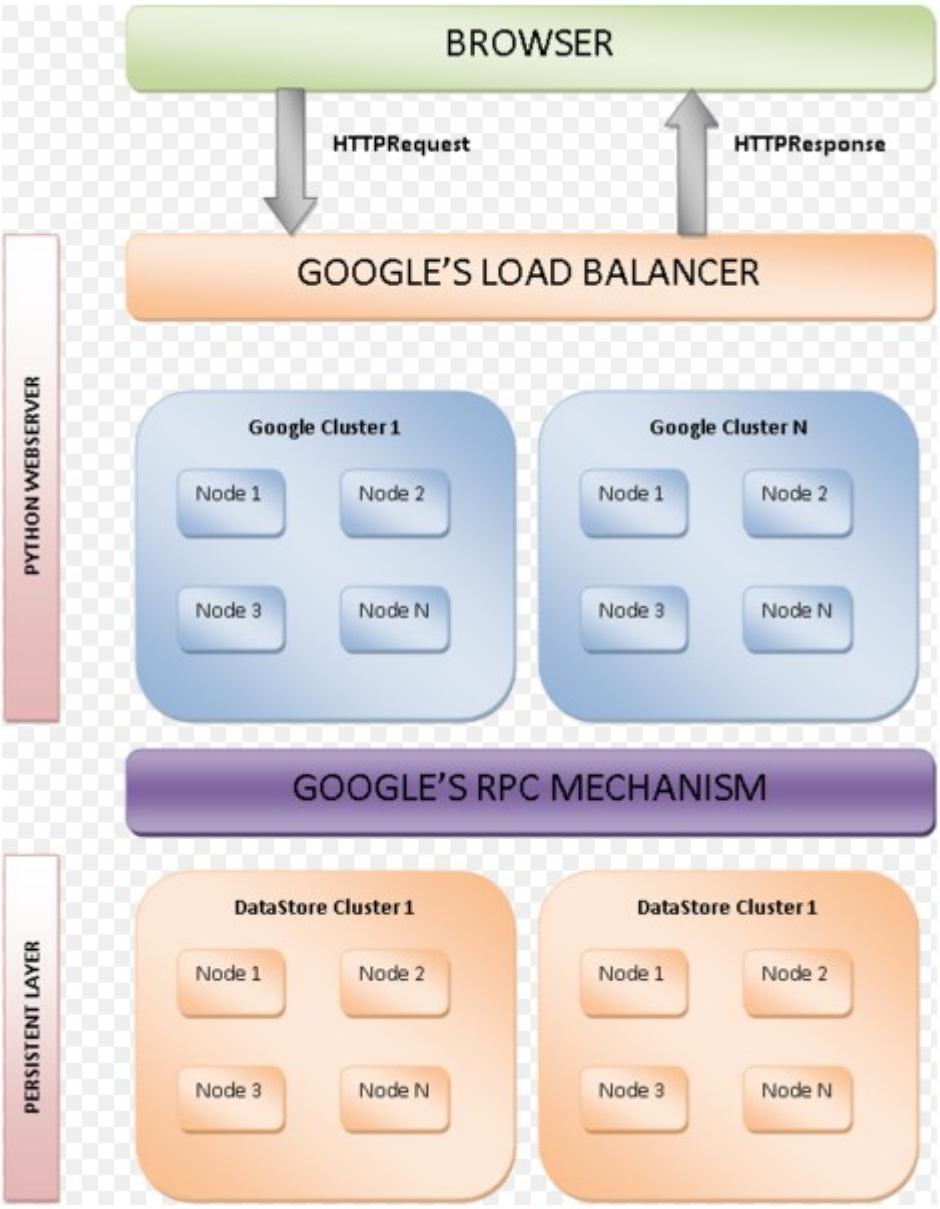
Gandi.net	Essentiellement Iaas , un peu Paas (php , node-js, ...)
Windows Azure (hébergement)	Microsoft
"**CloudLayer** / **SoftLayer**"	IBM

"Gandi.net"

Load balancing via "accélérateur web" :



"Google App Engine" :



VII - Annexe – Généralités "No SQL"

1. Bases SQL et NoSQL

1.1. Rappel (ACID)

A.C.I.D. ==> Atomicity , Consistency , Isolation , Durability

- L' **Atomicité** désigne le comportement "**tout ou rien**" (Le tout vu en tant qu'élément unique et atomique doit soit réussir , soit échouer). Il n'y a pas de demi-mesure.
- La **Consistance** d'une transaction désigne le fait que **les différentes opérations doivent laisser le système dans un état stable et cohérent**.
- Le concept d' **Isolation** signifie ici que **2 transactions concurrentes n'interfèrent pas entre elles** (Points critiques: résultats intermédiaires et opérations annulées).
- La **Durabilité** indique que **les résultats d'une transaction doivent absolument être mémorisés de façon durable** (sur un support physique) de façon à survivre suite à une éventuelle défaillance(un fichier de Log peut également être très utile).

1.2. Bases de données SQL (relationnelles) / SGBDR

- Forte consistance (intégrité garantie)
- Transactionnelles (ACID)
- Très matures / actuellement traditionnelles (adoptions faciles/habituelles)

mais pas très scalables en écriture du fait des contraintes "ACID" : détérioration des performances lorsque beaucoup de charge (utilisateurs simultanés) et de volume (big data).

1.3. Bases NoSQL (Not only SQL)

NoSQL désigne une catégorie de SGBD qui n'est plus fondée sur l'architecture classique des bases relationnelles. L'unité logique n'y est plus la table et les données ne sont en général pas manipulées avec SQL.

Origine/Historique :

À l'origine, servant à manipuler des bases de données géantes pour des sites web de très grande audience tels que Google, Amazon.com, Facebook ou eBay1, le NoSQL s'est aussi étendu par le bas après 2010. Il renonce aux fonctionnalités classiques des SGBD relationnels au profit de la simplicité. Les performances restent bonnes avec la montée en charge (scalabilité) en multipliant simplement le nombre de serveurs, solution raisonnable avec la baisse des coûts, en particulier si les revenus croissent en même temps que l'activité.

Les systèmes géants sont les premiers concernés : énorme quantité de données, structuration relationnelle faible (ou de moindre importance que la capacité d'accès très rapide, quitte à multiplier

les serveurs).

Un modèle typique en NoSQL est le système clé-valeur, avec une base de données pouvant se résumer topologiquement à un simple tableau associatif unidimensionnel avec des millions — voire des milliards — d'entrées. Parmi les applications typiques, on retrouve des analyses temps-réel, statistiques, du stockage de logs (journaux), etc.

En 2009 à San Francisco des développeurs de logiciels NoSQL qui, selon le magazine Computerworld, « racontent comment ils ont renversé la tyrannie des coûteux et lents SGBD relationnels par des moyens plus simples et plus rapides de manipuler des données ». Selon Jon Travis, un des présentateurs de la conférence, « les SGBD relationnels offrent trop, alors que les produits NoSQL offrent exactement ce dont vous avez besoin ».

Beaucoup de produits NoSQL peuvent manipuler de très grandes quantités de données (centaines de téraoctets) et offrent une évolutivité à la charge . Ces produits NoSQL ne sont pas toujours décrits comme des SGBD, mais quelquefois comme des logiciels de *stockage* de données.

Théorie :

Un système de gestion de base de données (SGBD) permet de réaliser des transactions atomiques, cohérentes, isolées, et durables (ACID).

Les capacités ACID garantissent que si plusieurs utilisateurs font de manière simultanée des modifications des données, toutes les modifications vont être prises en compte, dans un ordre précis et maîtrisé de manière à avoir un résultat cohérent (intégrité des données) avec l'historique des modifications faites par chacun. La mise en œuvre stricte des capacités ACID entraîne des coûts logiciels importants et un niveau de performance moindre à infrastructure matérielle équivalente.

Les SGBD d'annuaires ont servi de modèle en permettant de lever certaines de ces contraintes en fonction de l'usage, en particulier dans les cas où la grande majorité des accès aux bases de données consistent en lectures sans modification (dans ce cas seule la propriété de persistance importe).

Pour faire face à des volumes importants de données, accédés de différents endroits du monde, il faut pouvoir répliquer ces données sur différentes machines physiques, c'est ce que l'on appelle un environnement distribué. Le théorème CAP démontre qu'il n'est pas possible d'assurer des transactions totalement ACID dans un environnement distribué.

Les bases de données NoSQL répondent aussi au théorème du CAP d'Eric Brewer qui est plus adapté aux systèmes distribués. Ce théorème énonce que tout système distribué peut répondre aux contraintes suivantes:

- **Cohérence** : tous les noeuds du système voient exactement les mêmes données au même moment
- **Haute disponibilité (Availability)** : en cas de panne, les données restent accessibles
- **Tolérance au Partitionnement** : le système peut être partitionné

Mais le théorème du CAP précise aussi que seulement deux de ces trois contraintes peuvent être respectées en même temps.
D'autres caractéristiques communes aux différentes bases de données NoSQL peuvent être citées tel que le partitionnement horizontal sur plusieurs noeuds, la réplication des données, les schémas dynamiques ou l'absence de schéma.

Le protocole PAXOS est très efficace pour la lecture dans un environnement distribué, beaucoup moins pour l'écriture / modification et il ne supporte pas les transactions ACID.

Les solutions du marché implémentent ce protocole en ajoutant leurs techniques propres pour limiter les conséquences de l'impossibilité d'ACID lors des écritures et mises à jour de données.

Problèmes de l'atomicité :

- une opération atomique ne permet pas des accès entrelacés
- une partie de la base de données est verrouillée lors de l'écriture

==> *En se passant de l'atomicité , on a plus besoin de verrouillage* .

Problèmes de la consistance :

- toutes les transactions doivent être totalement commités ou annulées
- tous les membres d'un cluster doivent avoir les mêmes données (synchronisations immédiates sur tous les autres nœuds)
- respect des schémas

==> ***En se passant de la consistance immédiate, on a plus besoin d'avoir exactement et immédiatement les mêmes données sur tous les serveurs, on synchronise lorsque l'on peut (avec un léger différé).***

Problèmes de la durabilité:

- avant de répondre au client , on doit être sûr que les données ont été écrites sur le disque.

==> ***En se passant de la durabilité , on n'a plus besoin d'attendre la confirmation de l'écriture.***

NB : chaque technologie NoSQL gère à sa façon les aspects précédents (atomicité , consistance , durabilité) . On ne se passe évidemment pas de tout ça .
C'est essentiellement l'aspect "synchronisation non immédiate" qui est mis en avant .

1.4. Types de bases de données NoSQL :

Les solutions NoSQL existantes peuvent être regroupées en 4 grandes familles.

- **Clé / valeur** : Ce modèle peut être assimilé à une hashmap distribuée. Les données sont, donc, simplement représentées par un couple clé/valeur. La valeur peut être une simple chaîne de caractères, un objet sérialisé… Cette absence de structure ou de typage ont un impact important sur le requêtage. En effet, toute l'intelligence portée auparavant par les requêtes SQL devra être portée par l'applicatif qui interroge la BD. Néanmoins, la communication avec la BD se résumera aux opérations PUT, GET et DELETE. Les solutions les plus connues sont Redis, Riak et Voldemort créé par LinkedIn.
- **Orienté colonne** : Ce modèle ressemble à première vue à une table dans un SGBDR à la différence qu'avec une BD NoSQL orientée colonne, le nombre de colonnes est dynamique. En effet, dans une table relationnelle, le nombre de colonnes est fixé dés la création du schéma de la table et ce nombre reste le même pour tous les enregistrements dans cette table. Par contre, avec ce modèle, le nombre de colonnes peut varier d'un enregistrement à un autre ce qui évite de retrouver des colonnes ayant des valeurs NULL. Comme solutions, on retrouve principalement HBase (implémentation Open Source du modèle BigTable publié par Google) ainsi que Cassandra (projet Apache qui respecte l'architecture distribuée de Dynamo d'Amazon et le modèle BigTable de Google).
- **Orienté document** : Ce modèle se base sur le paradigme clé valeur. La valeur, dans ce cas, est un document de type JSON ou XML. L'avantage est de pouvoir récupérer, via une seule clé, un ensemble d'informations structurées de manière hiérarchique. La même opération dans le monde relationnel impliquerait plusieurs jointures. Pour ce modèle, les implémentations les plus populaires sont CouchDB d'Apache, RavenDB (destiné aux plateformes .NET/Windows avec la possibilité d'interrogation via LINQ) et MongoDB.
- **Orienté graphe** : Ce modèle de représentation des données se base sur la théorie des graphes. Il s'appuie sur la notion de noeuds, de relations et de propriétés/attributs qui leur sont rattachées. Ce modèle facilite la représentation du monde réel, ce qui le rend adapté au traitement des données des réseaux sociaux. La principale solution est Neo4J.

Les deux mouvements "orienté colonne" et "orienté document" découlent bien du système clé valeur et c'est la nature ou la structure de la valeur qui diffère.

Le marché :

Dans le marché des SGBD *NoSQL* se trouvent Cassandra, MongoDB, Voldemort, CouchDB et SimpleDB. Selon Oracle, le « battage » autour de ces produits vient du fait qu'ils sont impliqués dans de grands sites web tels que Facebook, LinkedIn ou Amazon.com. C'est un marché jeune, encore sans réel leader (en 2011). Le marché évolue rapidement et les comparatifs de produits y sont rapidement dépassés. Lors d'un sondage réalisé en 2010 auprès de professionnels de l'informatique, 44 % des sondés répondaient encore qu'ils n'ont jamais entendu parler de NoSQL.

1.5. SQL vs NoSQL

Les SGBD relationnels sont largement répandus dans les entreprises. Dimensionnés pour une quantité d'informations et un nombre d'utilisateurs typiques d'une entreprise, ils ont pour fonction principale le traitement de transactions.

Ils montrent cependant leurs limites lorsqu'ils sont utilisés dans un périmètre plus large, tel qu'un site web populaire, en répartition de charge(load balancing), fréquenté par des millions de visiteurs dans le monde entier : les SGBD relationnels exigeraient alors des logiciels et des ordinateurs coûteux ainsi que des compétences en optimisation peu répandues.

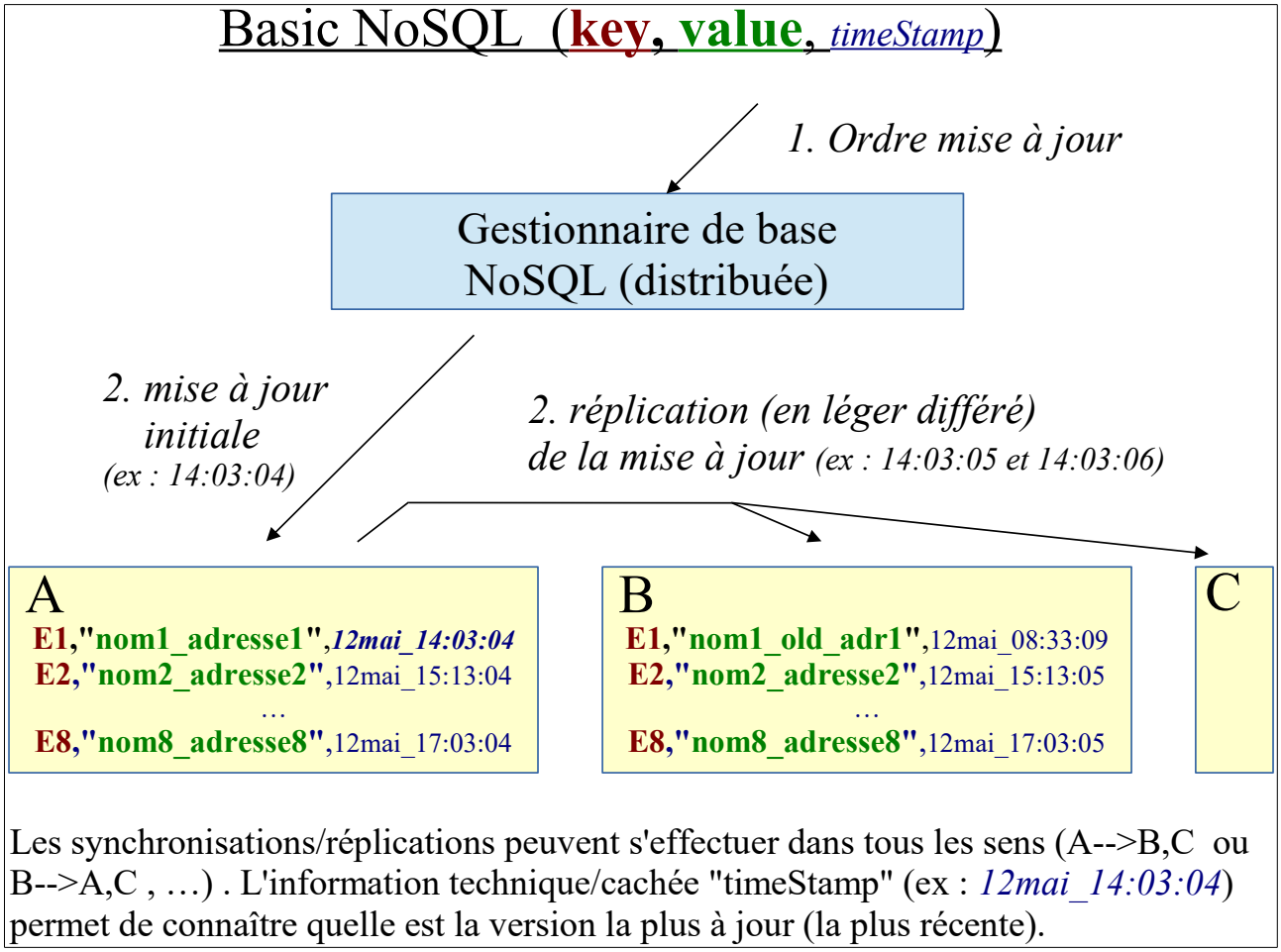
Ce segment de marché est de ce fait occupé par les logiciels *NoSQL*, conçus spécifiquement pour un usage de type Internet. Ces produits abandonnent la représentation matricielle de l'information et le langage de commande SQL en échange d'une simplicité, d'une performance et surtout d'une scalabilité accrues. La complexité de mise en œuvre du traitement des transactions a été réduite dans le but d'obtenir des services plus simples et plus spécialisés.

Les deux domaines, bases relationnelles et NoSQL, répondant à des besoins et des contraintes différentes, coexistent souvent dans les architectures logicielles de certains domaines métiers.

Choix selon questions suivantes:

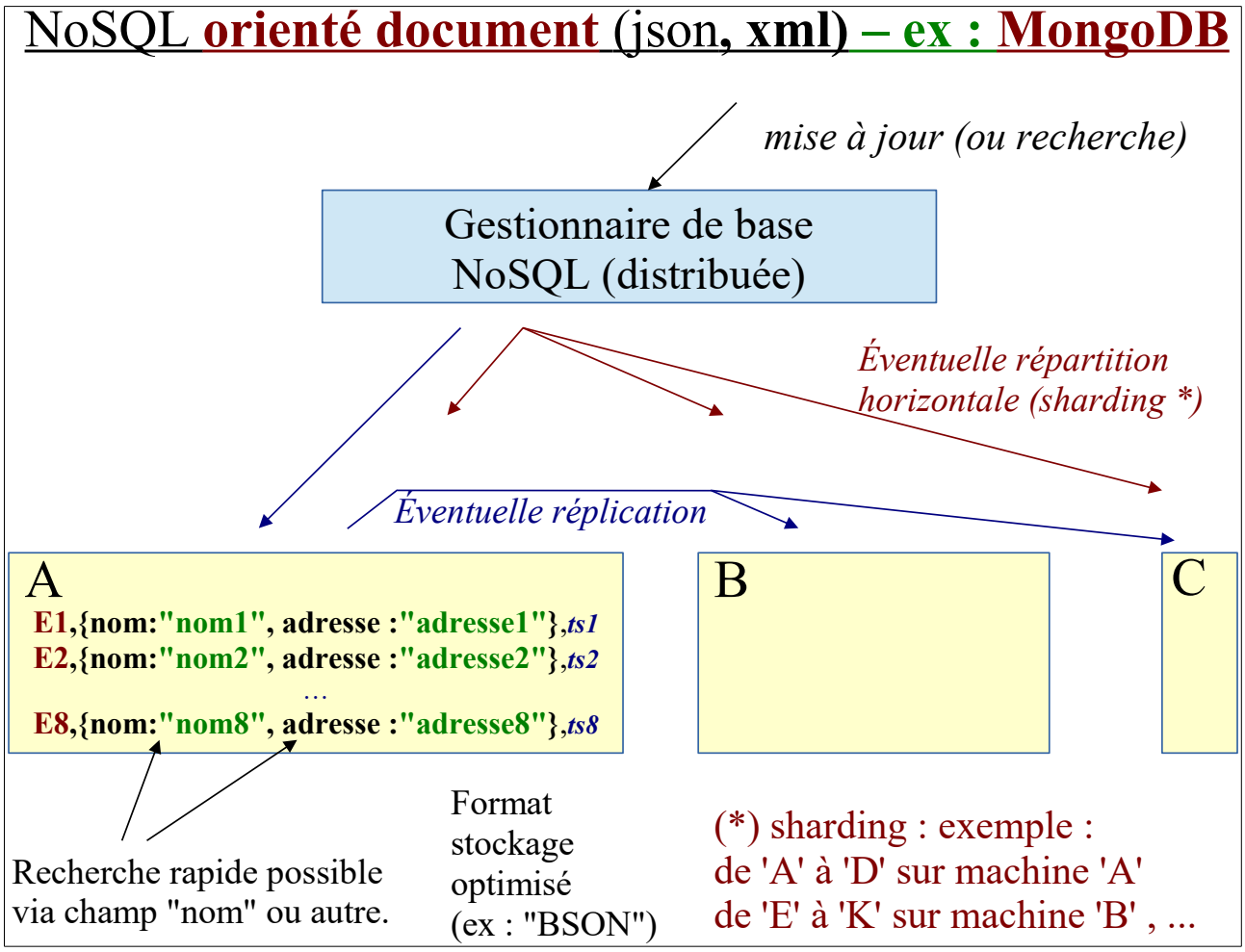
- Comment veut-on structurer nos données ?
- Combien de données à manipuler ?
- Ratio "lecture/écriture" ?

1.6. Basic NoSQL (clef,valeurs)

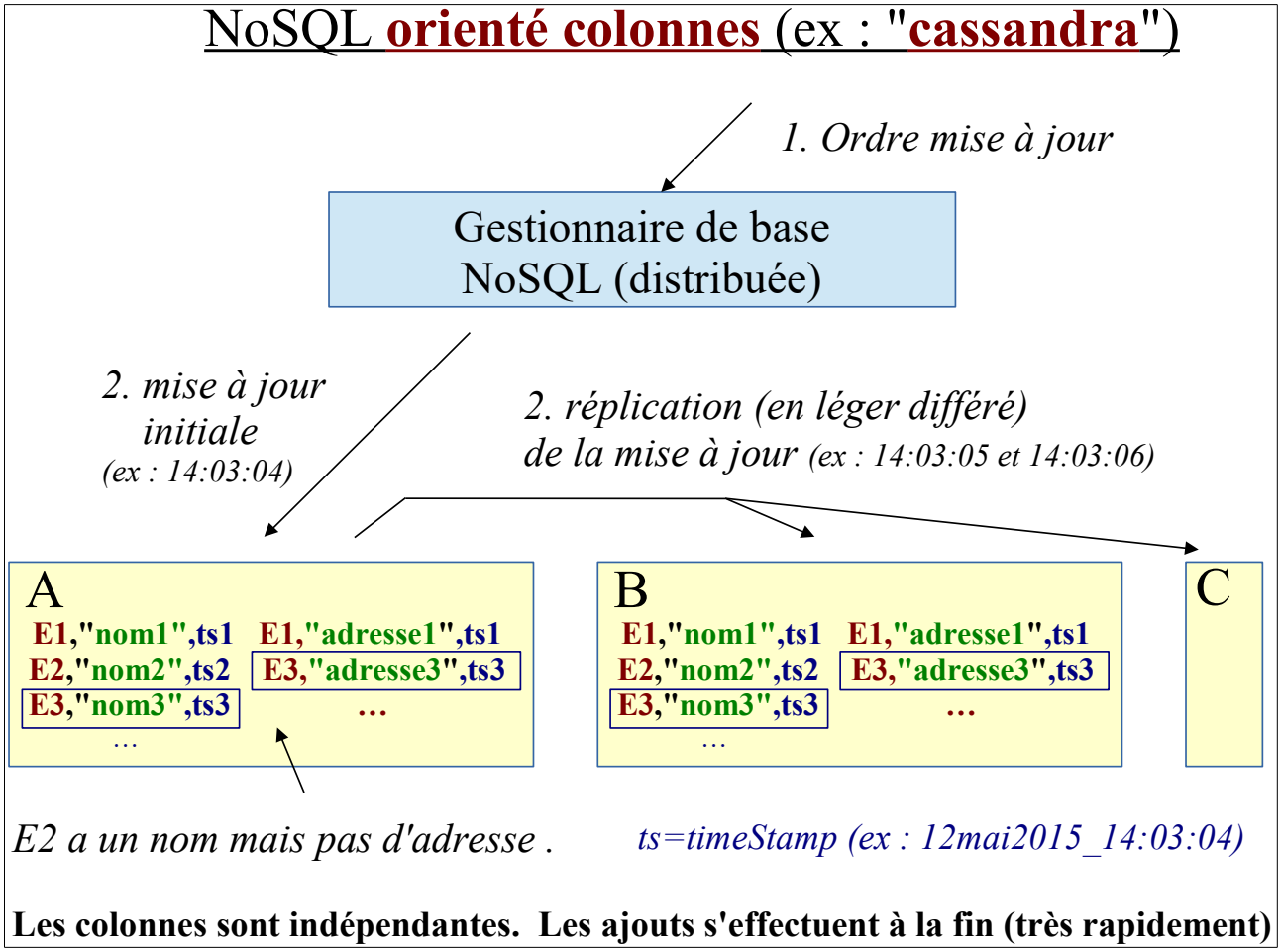


Les synchronisations/réplications peuvent s'effectuer dans tous les sens (A-->B,C ou B-->A,C , …) . L'information technique/cachée "timeStamp" (ex : *12mai_14:03:04*) permet de connaître quelle est la version la plus à jour (la plus récente).

1.7. Orienté document



1.8. Orienté colonnes



VIII - Annexe – Api pour mongoDB (node , java, ...)

1. Api java pour mongoDB

...

2. Api javascript / node pour mongoDB

Dans l'éco-système nodeJs, on trouve essentiellement 2 Api (complémentaires) pour s'interfacer avec mongoDB :

- mongo (api de base)
- mongoose (mapping objet-mongo)

2.1. Api mongoDB élémentaire

…

2.2. Api mongoose

IX - Annexe – Bibliographie, Liens WEB + TP

1. Bibliographie et liens vers sites "internet"

2. TP