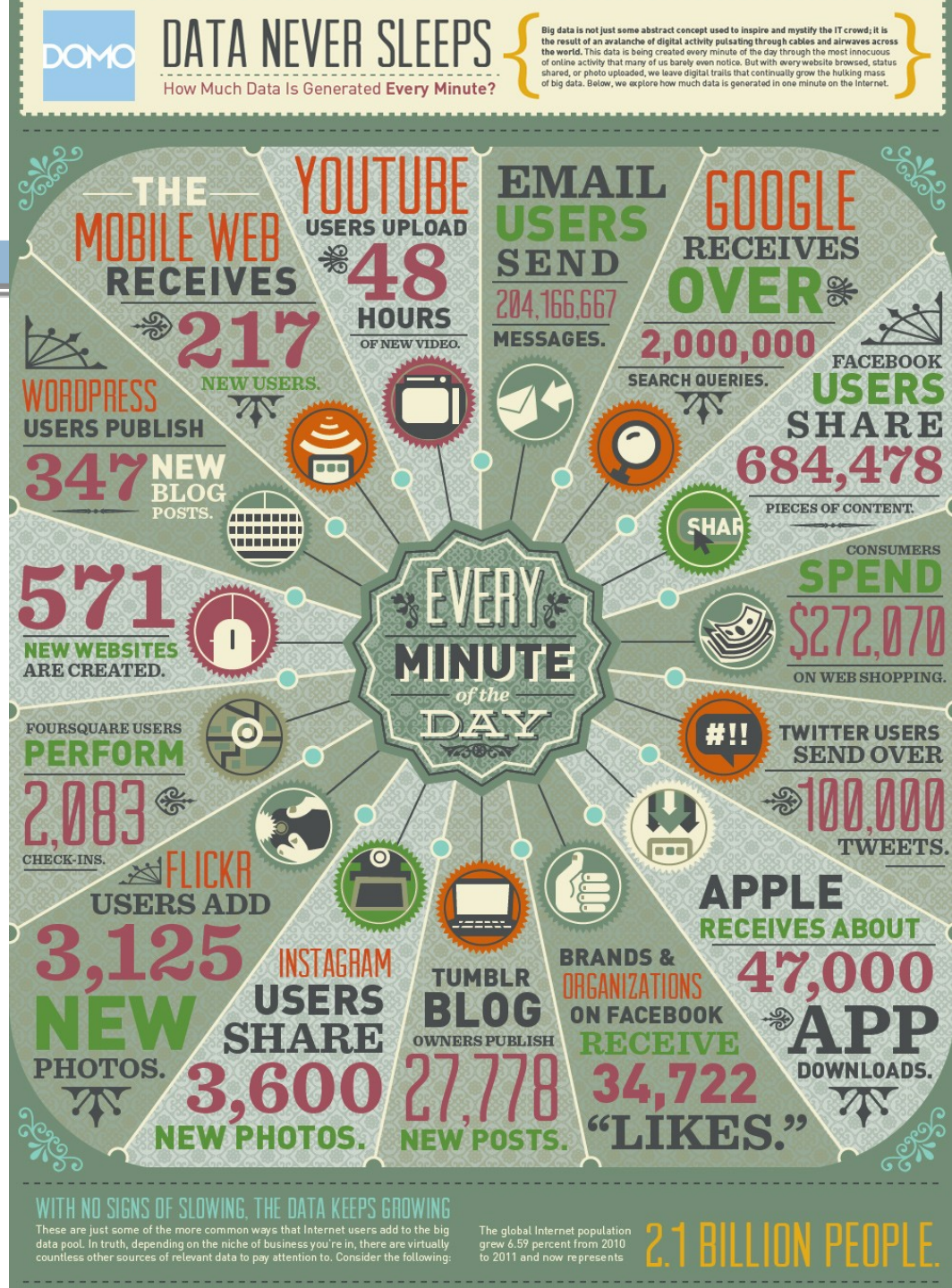


NoSQL

Plan de la présentation

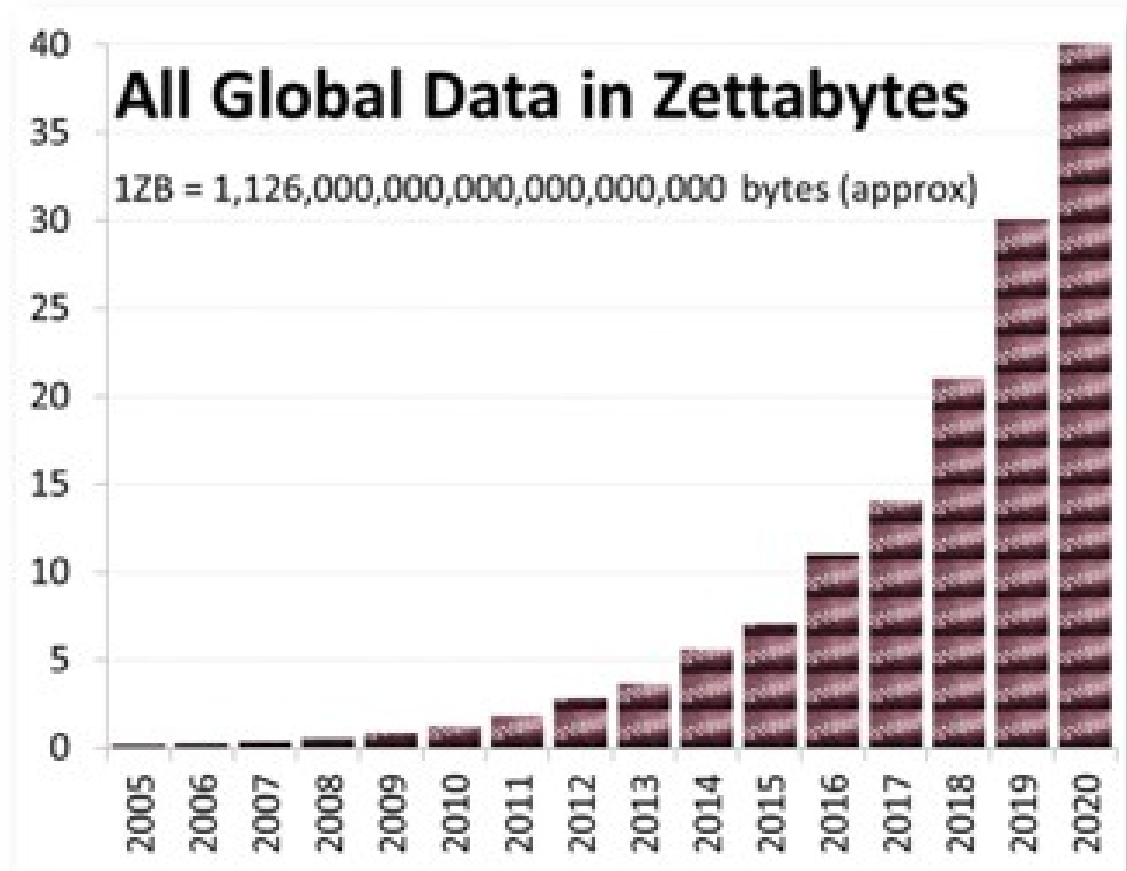
- Introduction
- NoSQL : principes
- Les types de BDs NoSQL
- Quelques exemples

1 minute =



Et ?

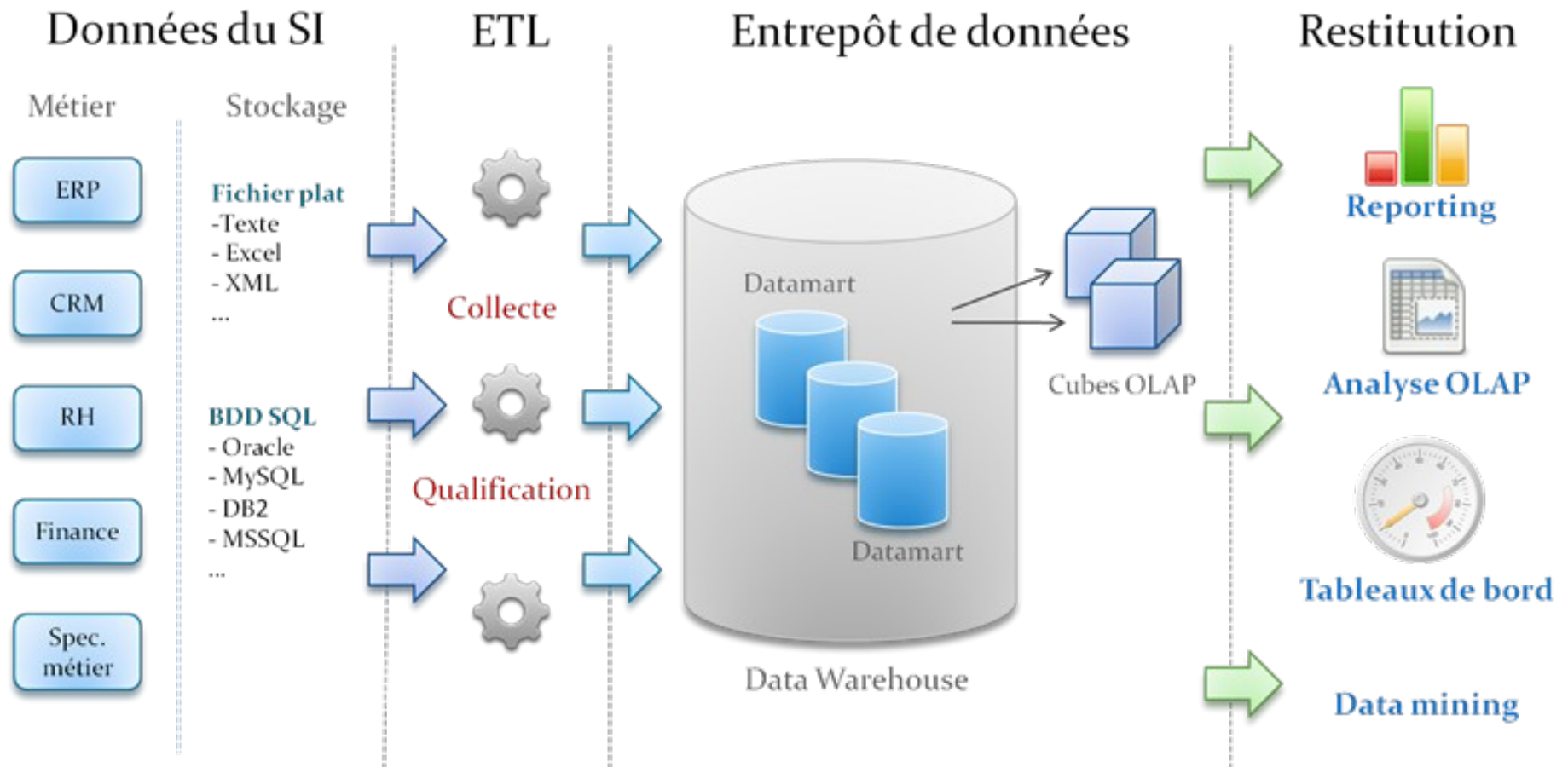
- Données produites : double (+) tous les 2 ans



Contexte

- Chaque jour, la quantité de données produites explose
 - + de users
 - + d'usages (mobiles, réseaux sociaux, ...)
 - + de machines qui en génèrent (IOT)
- Opportunités d'analyses + larges et + fines
=> valorisation de l'information
- Mais SGBDr et outils d'aide à la décision « classiques » débordés par la volumétrie des données
- Algos d'analyse classiques (BI) ne passent pas à l'échelle

Et pépé ?



Soufflé par les 3V du Big Data

- Volume : Go/To => Po/Eo
- Variety : types très variés, texte...
- Velocity : données créées de + en + vite, traitements « temps réel »

Nouvelle problématique

- Stockage : plus sur 1 serveur ! Mais 100, 1000...
BD NoSQL
- Frameworks de requêtage
Hadoop, Spark, ...
- Solutions de visualisation et de data mining
adaptées
- Format de fait : JSON
- Données dans le cloud => analyse dans le cloud



Les données [dans le Cloud] : NoSQL

Source :

Xebia, Michaël Figuière

[http://www.slideshare.net/mfiguiere/
presentations](http://www.slideshare.net/mfiguiere/presentations)

A propos de NoSQL

Not *Only*

No SQL

Relational

Contrairement aux idées reçues

- NoSQL n'est pas un remplaçant des SGBDR

↳ *The right tool for the right job*

- NoSQL reste un domaine d'innovation

↳ *Mais déjà déployé en production !*

- NoSQL est un écosystème riche et complexe

↳ *« Le diable est dans le détail »*

Au commencement

- Des cas d'usage différents mais des besoins similaires :
 - Performance
 - Disponibilité ($> 99.99\%$)
 - Résilience
 - Scalabilité horizontale

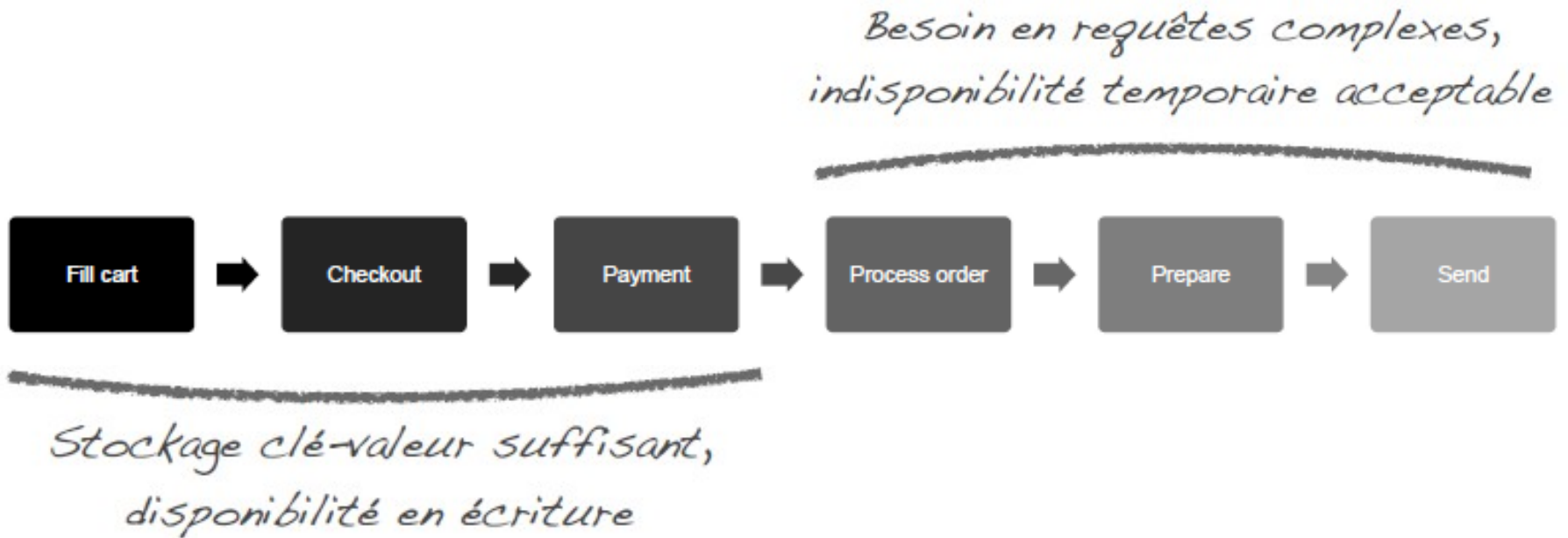
amazon.com.

- Création de Dynamo
- Dernier incident majeur en 2004
- < 40 min d'indisponibilité par an

Google™

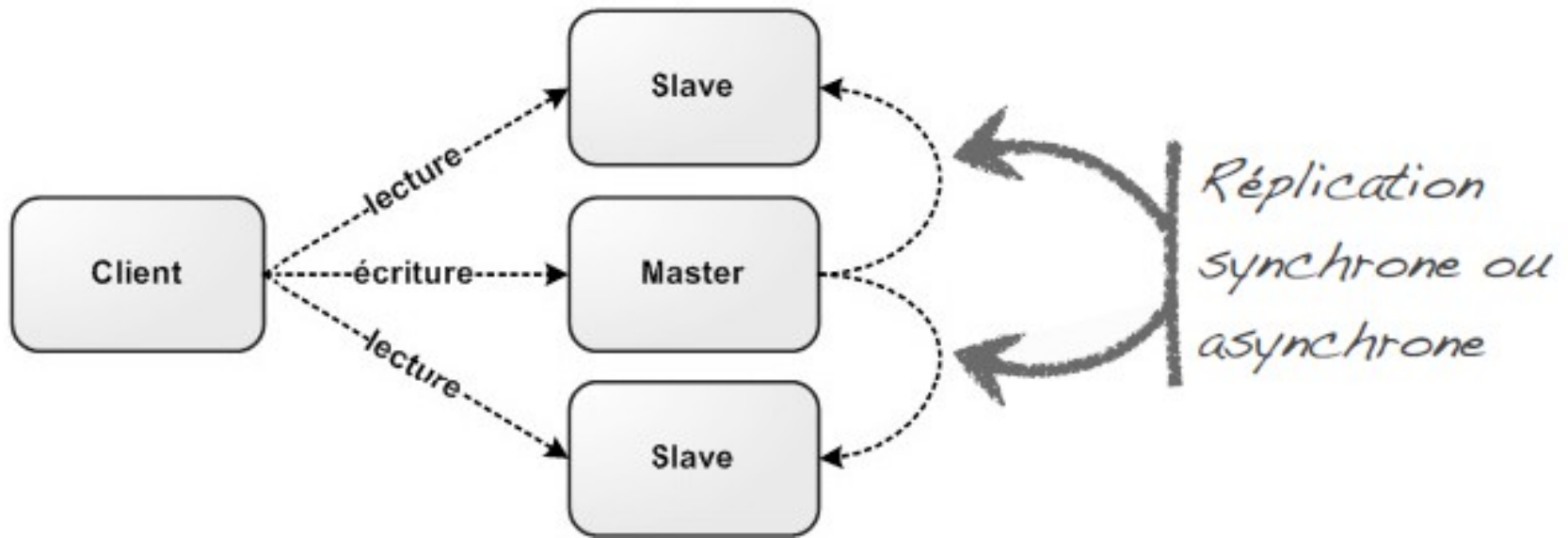
- Création de BigTable + MapReduce
- Toutes les pages Web du monde
- Fonctionnement online et offline

Amazon : naissance de Dynamo

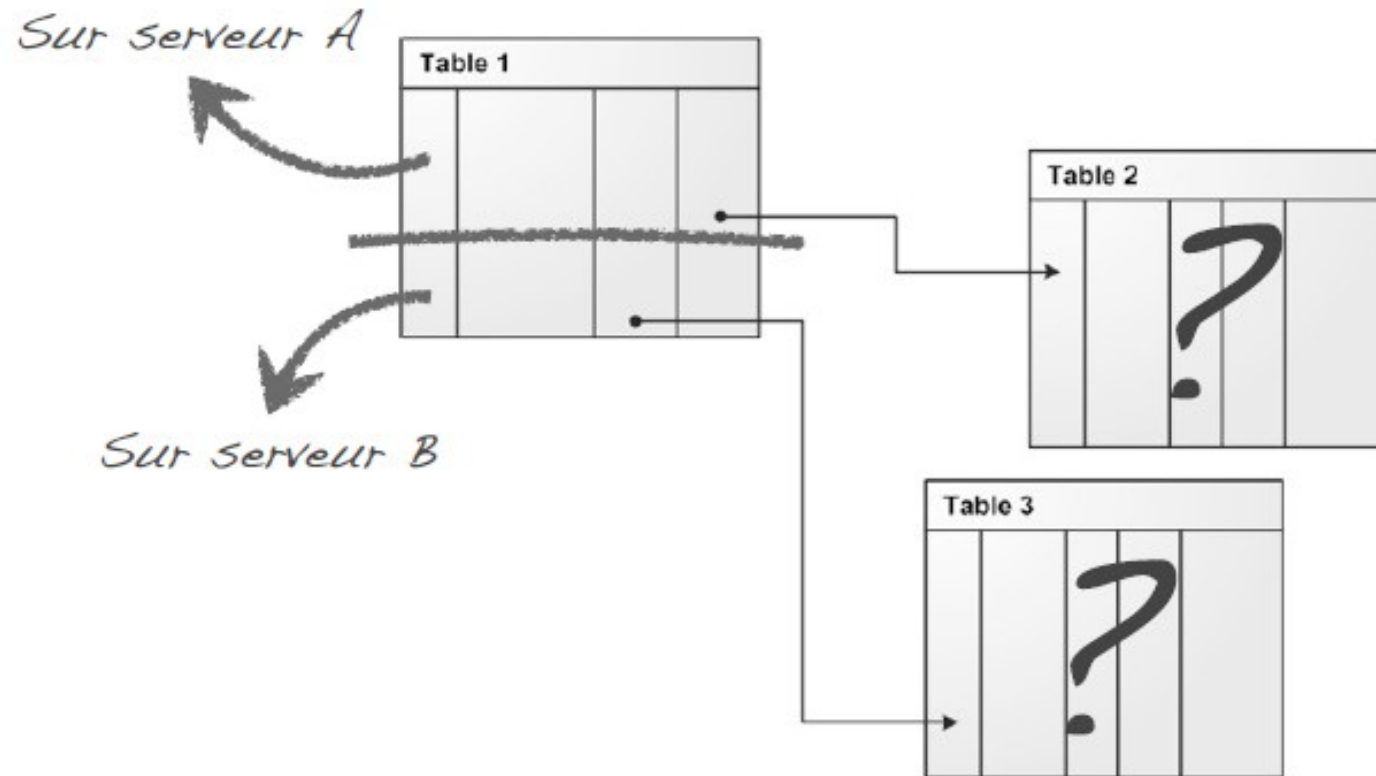


Assurer la Scalabilité : SGBDr

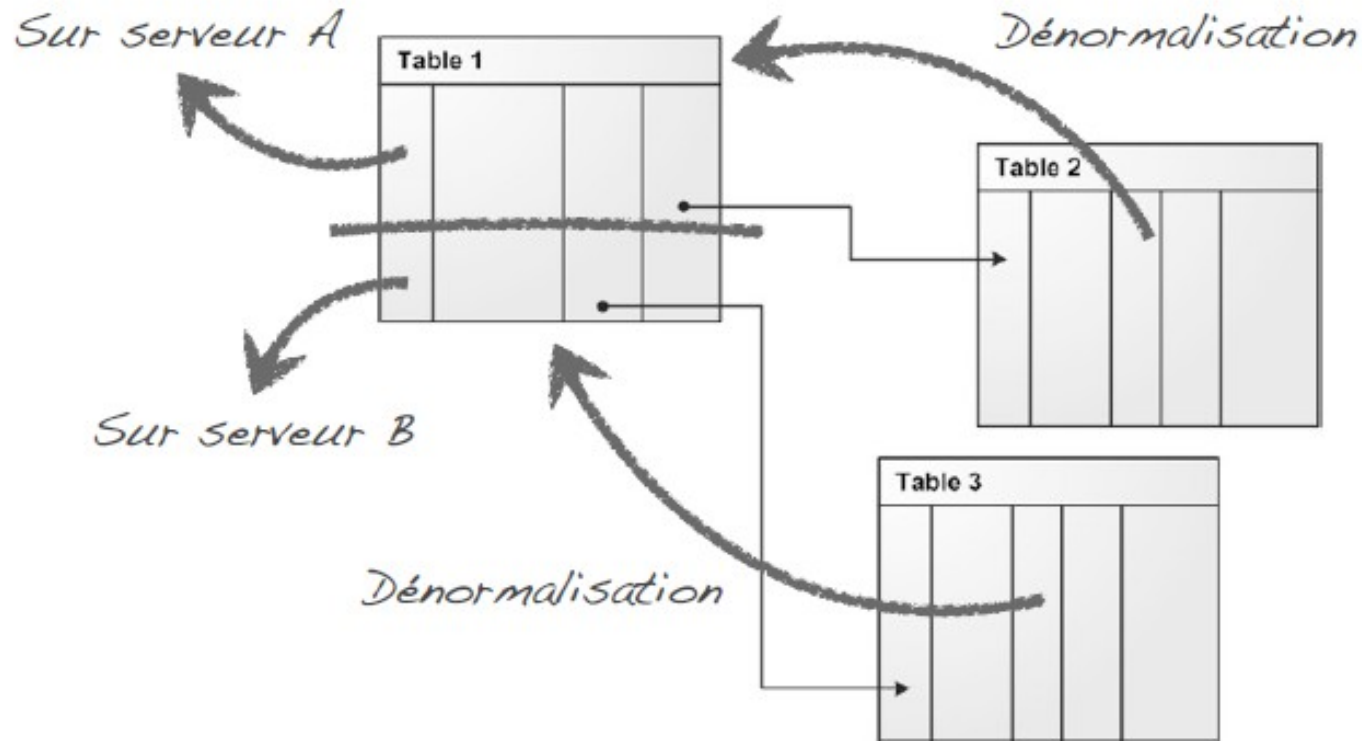
*Mise en oeuvre
typique avec MySQL*



Sharding avec SGBDr



Sharding avec SGBDr



- On perd alors beaucoup d'intérêt du Relationnel

Sharding avec SGBDr : problèmes

- Pour garder de bonnes performances, les relations many-to-many et many-to-one nécessitent d'être dénormalisées
- Gestion du resharding
- Code applicatif complexifié

table hachage à BDD clé-valeur

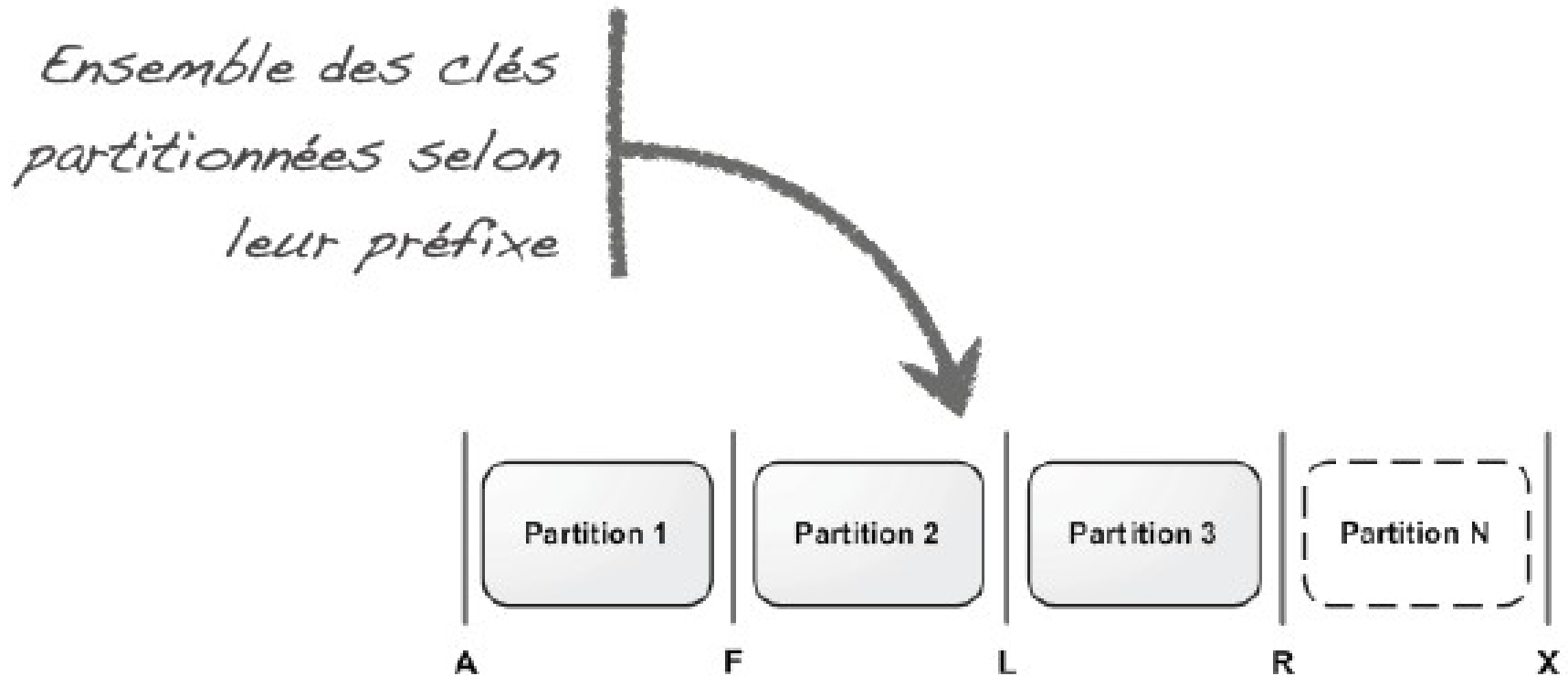


table hachage à BDD clé-valeur

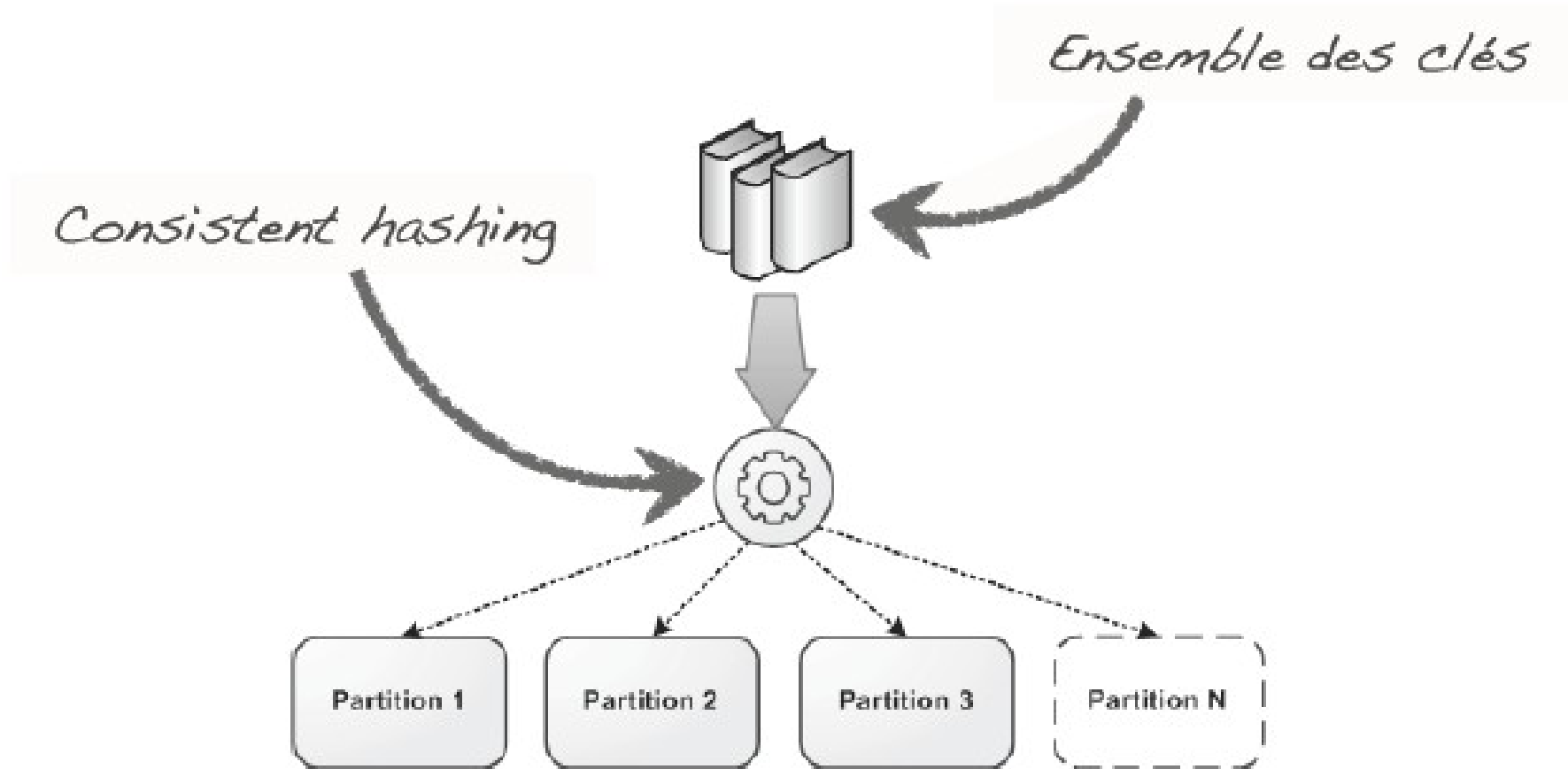
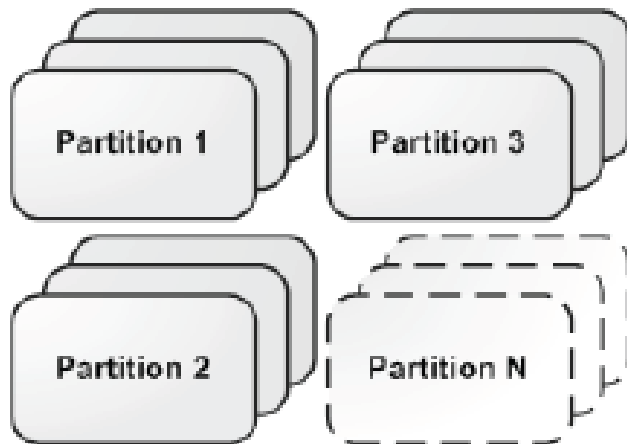
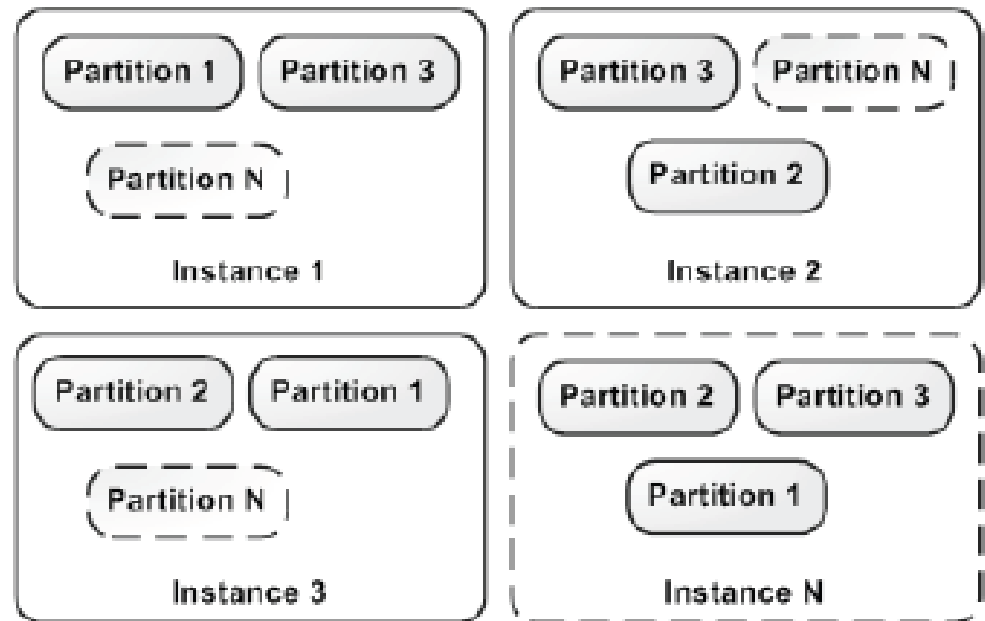


table hachage à BDD clé-valeur

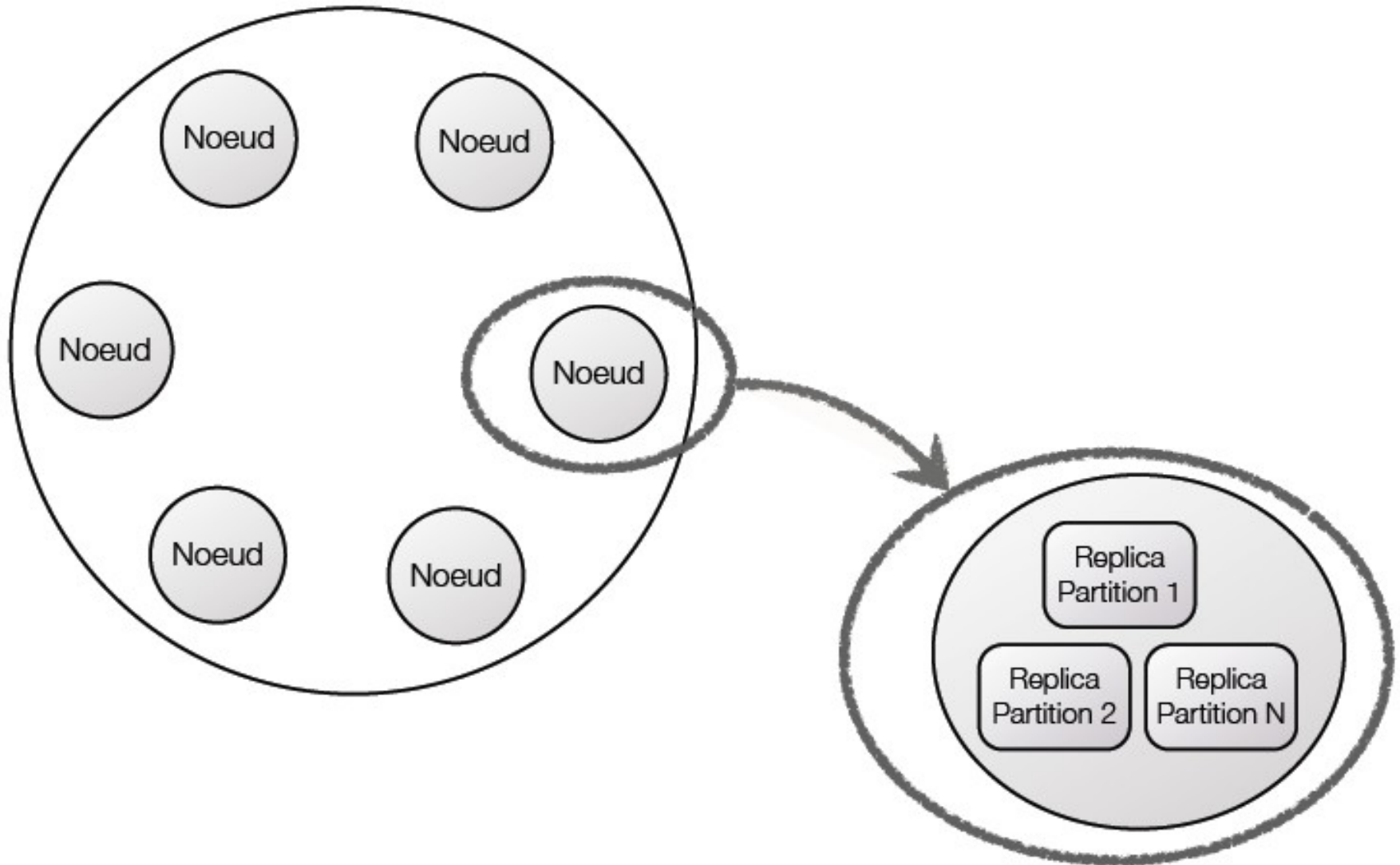


Une partition par instance

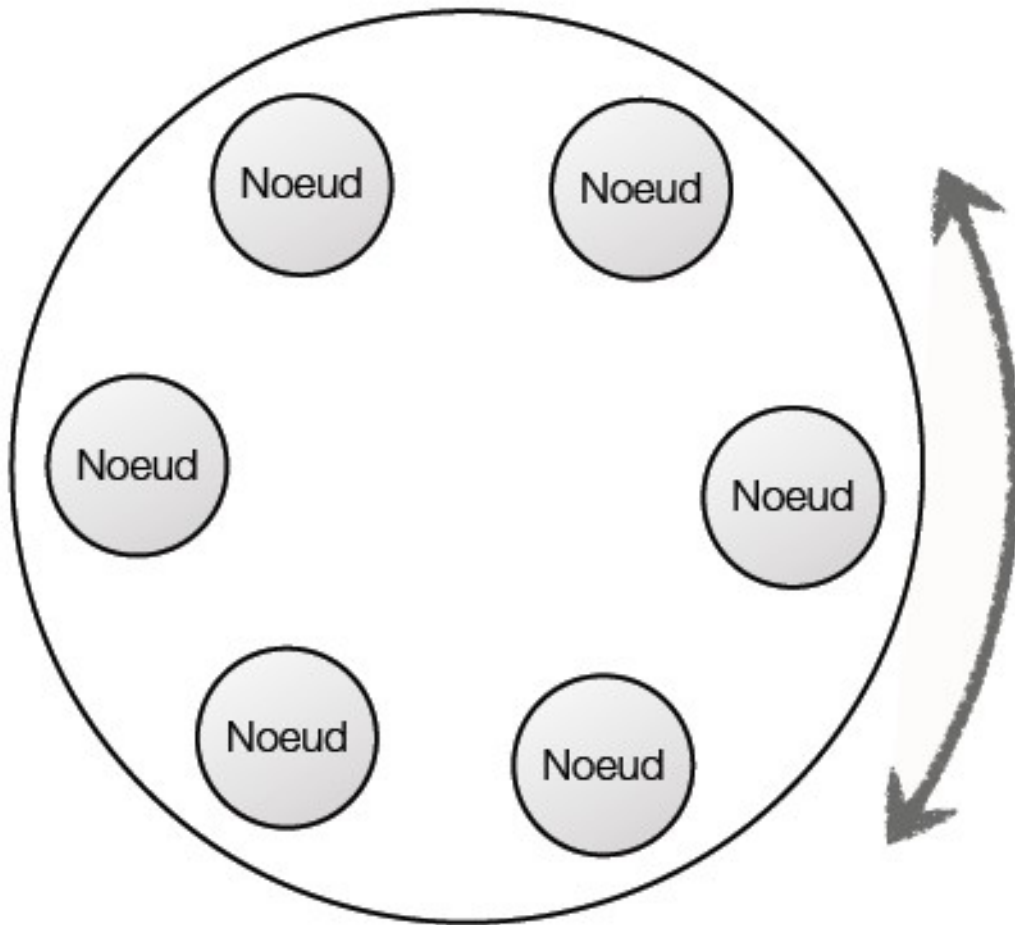


Multiples partitions par instance

Noeuds en anneaux



Noeuds en anneaux



*Communication de
proche en proche
pour diffuser les
changements de
topologie*

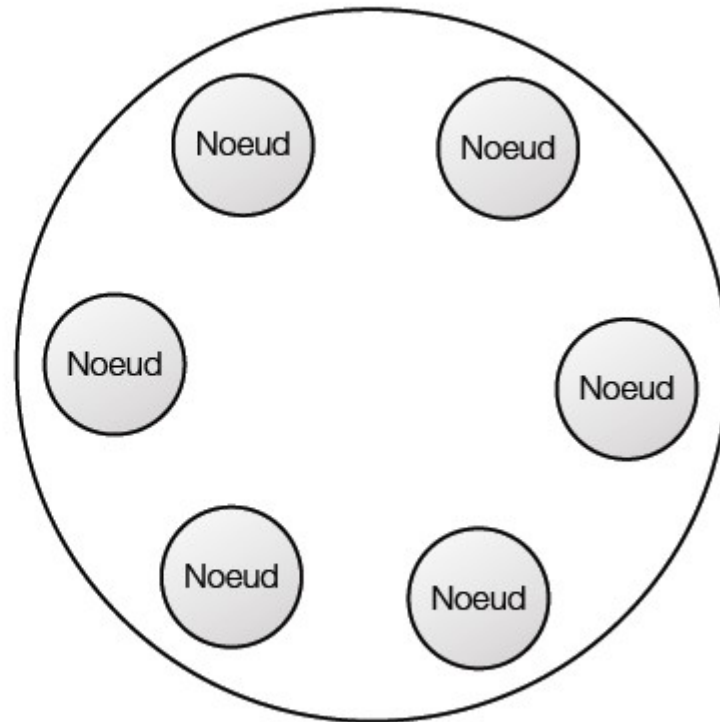
Interactions Client/Serveur

Client

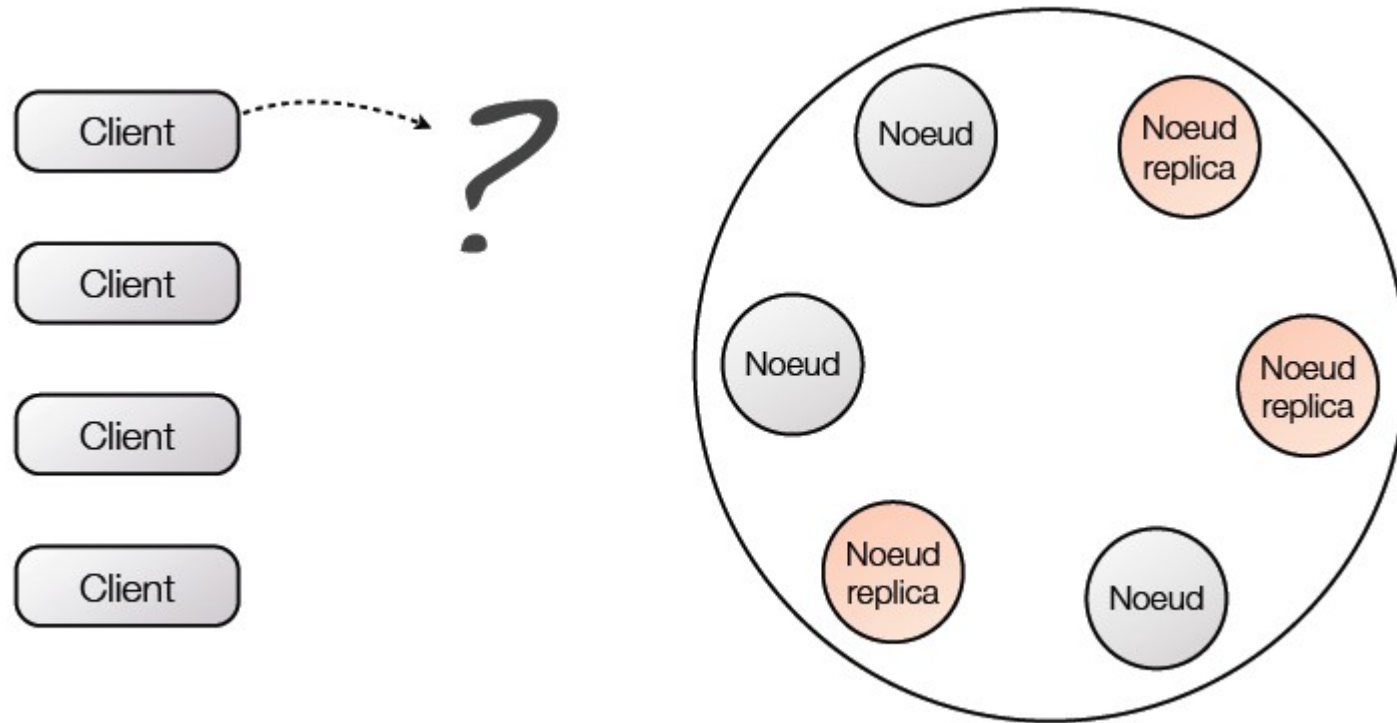
Client

Client

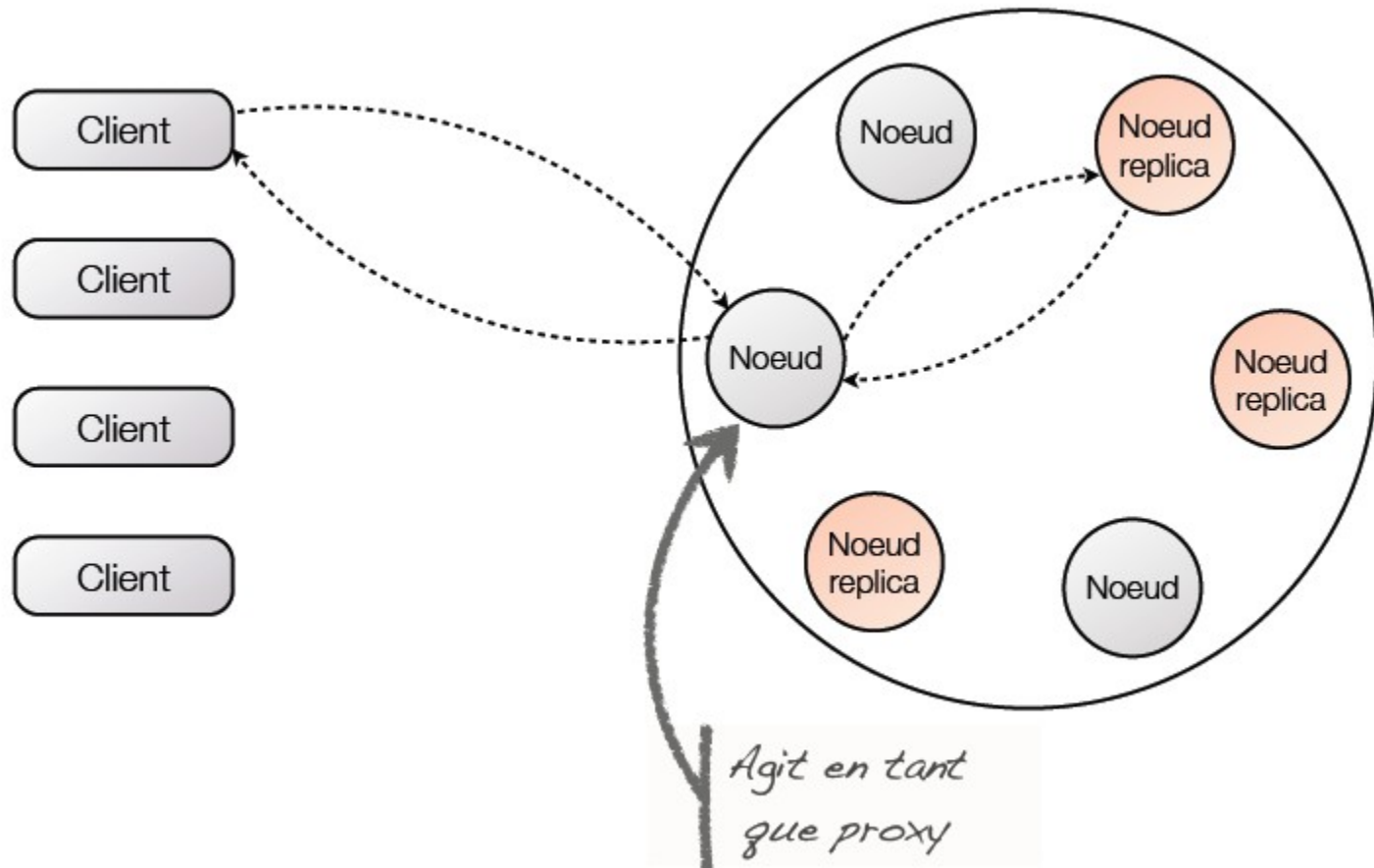
Client



Interactions Client/Serveur



Interactions Client/Serveur



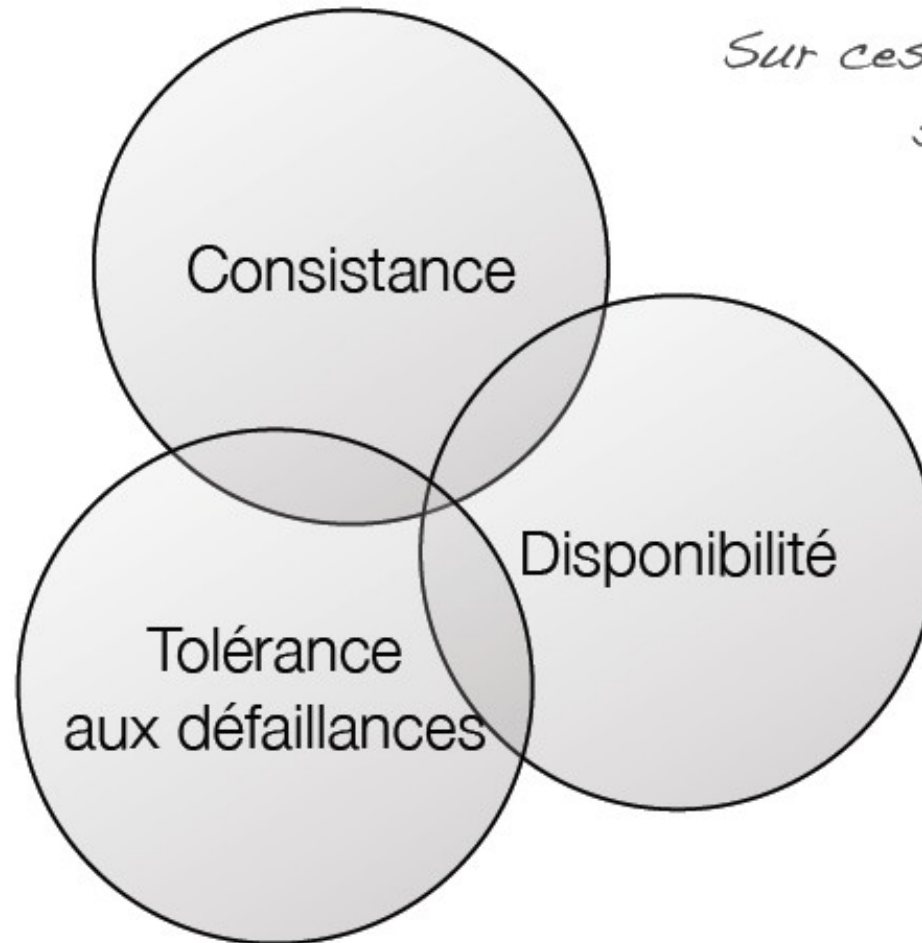
Que devient ACID ?

- **Atomique** : la suite d'opérations est indivisible, en cas d'échec en cours d'une des opérations, la suite d'opérations doit être complètement annulée (*rollback*).
- **Cohérente** : le contenu de la base de données à la fin de la transaction doit être cohérent sans pour autant que chaque opération durant la transaction donne un contenu cohérent.
- **Isolée** : lorsque deux transactions A et B sont exécutées en même temps, les modifications effectuées par A ne sont ni visibles par B, ni modifiables par B tant que la transaction A n'est pas terminée et validée (*commit*).
- **Durable** : Une fois validé, l'état de la base de données doit être permanent, et aucun incident technique ne doit pouvoir engendrer une annulation des opérations effectuées durant la transaction.

Que devient ACID ?

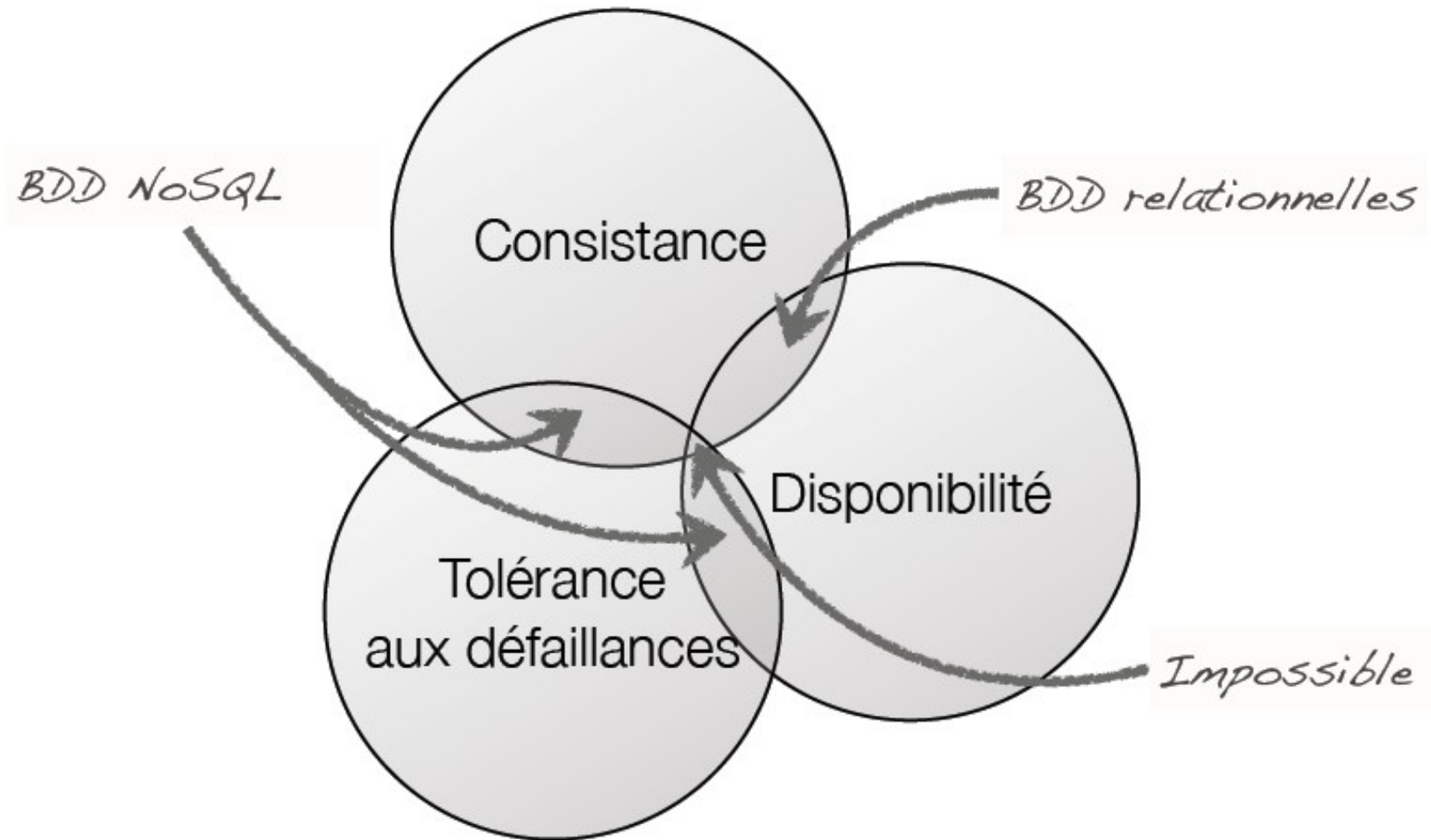
- Tout accès réseau/machine est faillible
- Des concessions doivent être faites sur le modèle de données
- Des concessions doivent être faites sur la consistance

Le théorème CAP

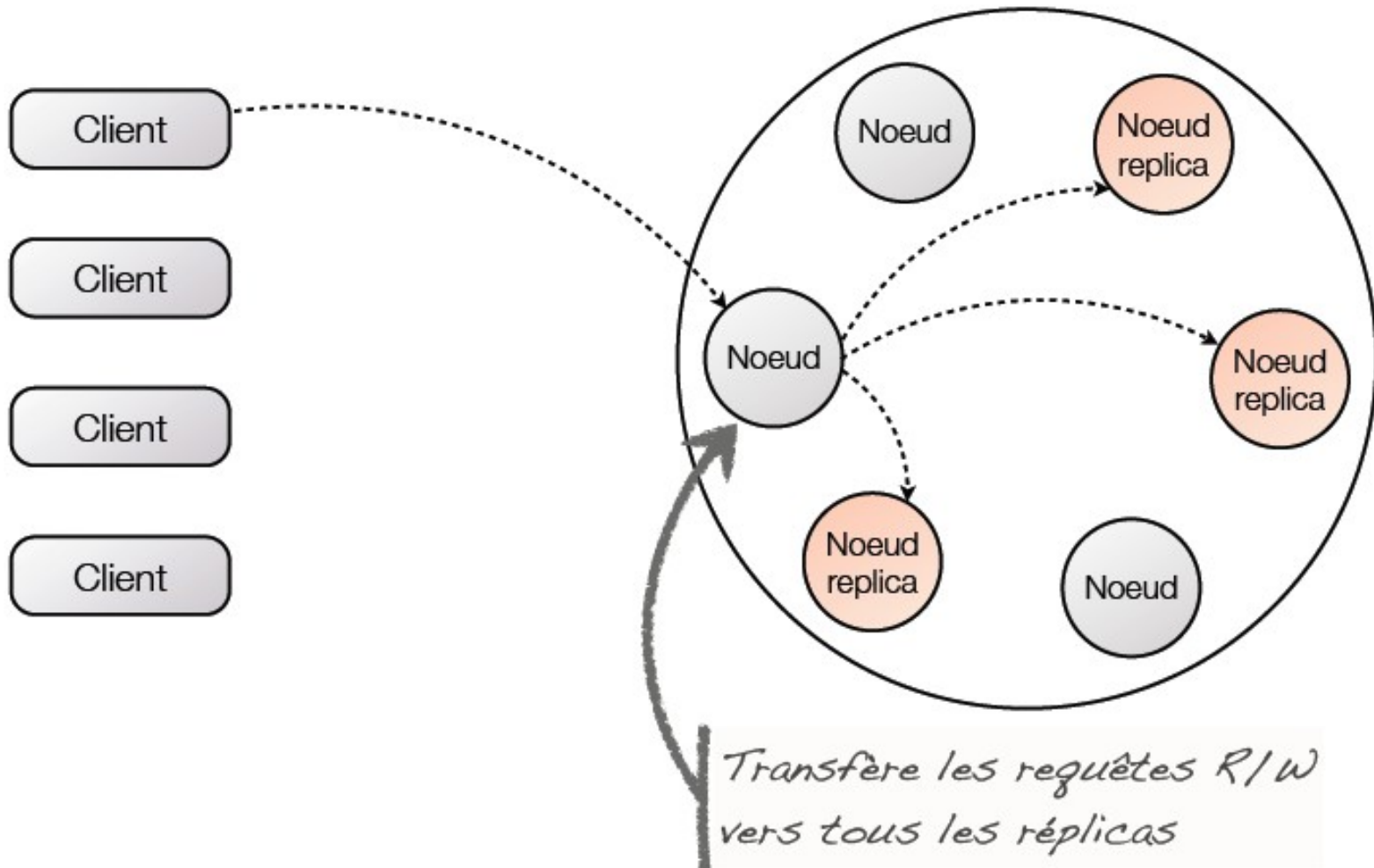


*Sur ces 3 propriétés,
seules 2 sont
réalisables
à la fois*

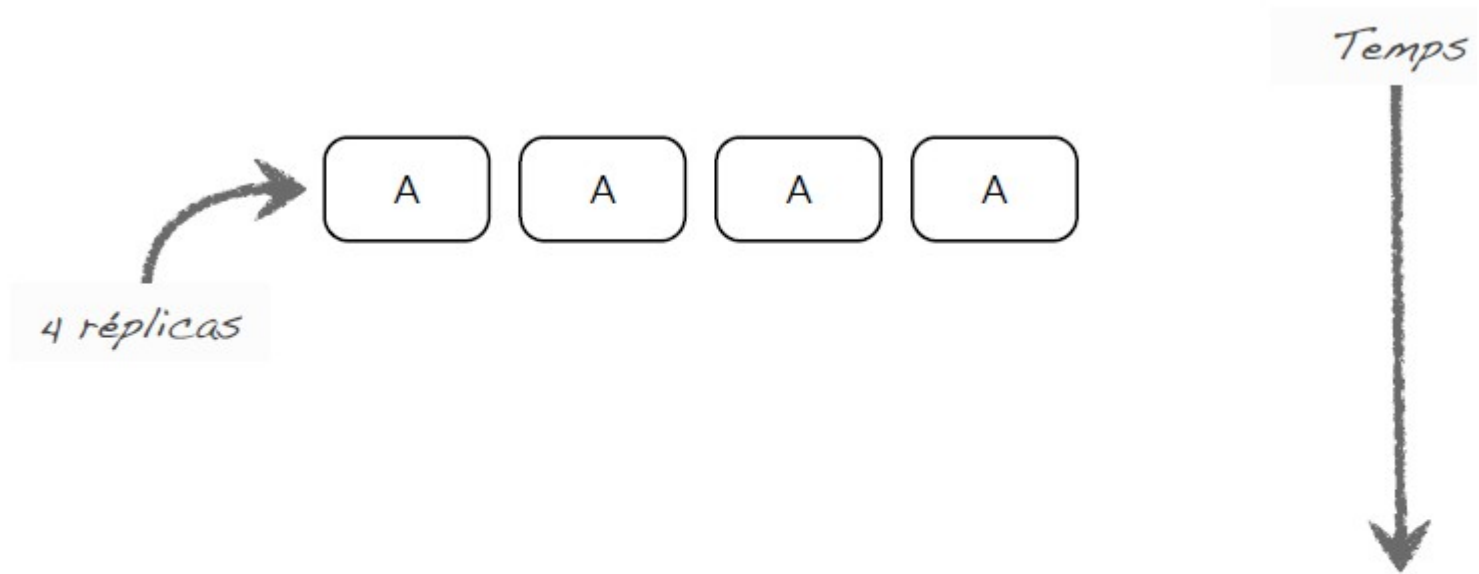
Le théorème CAP



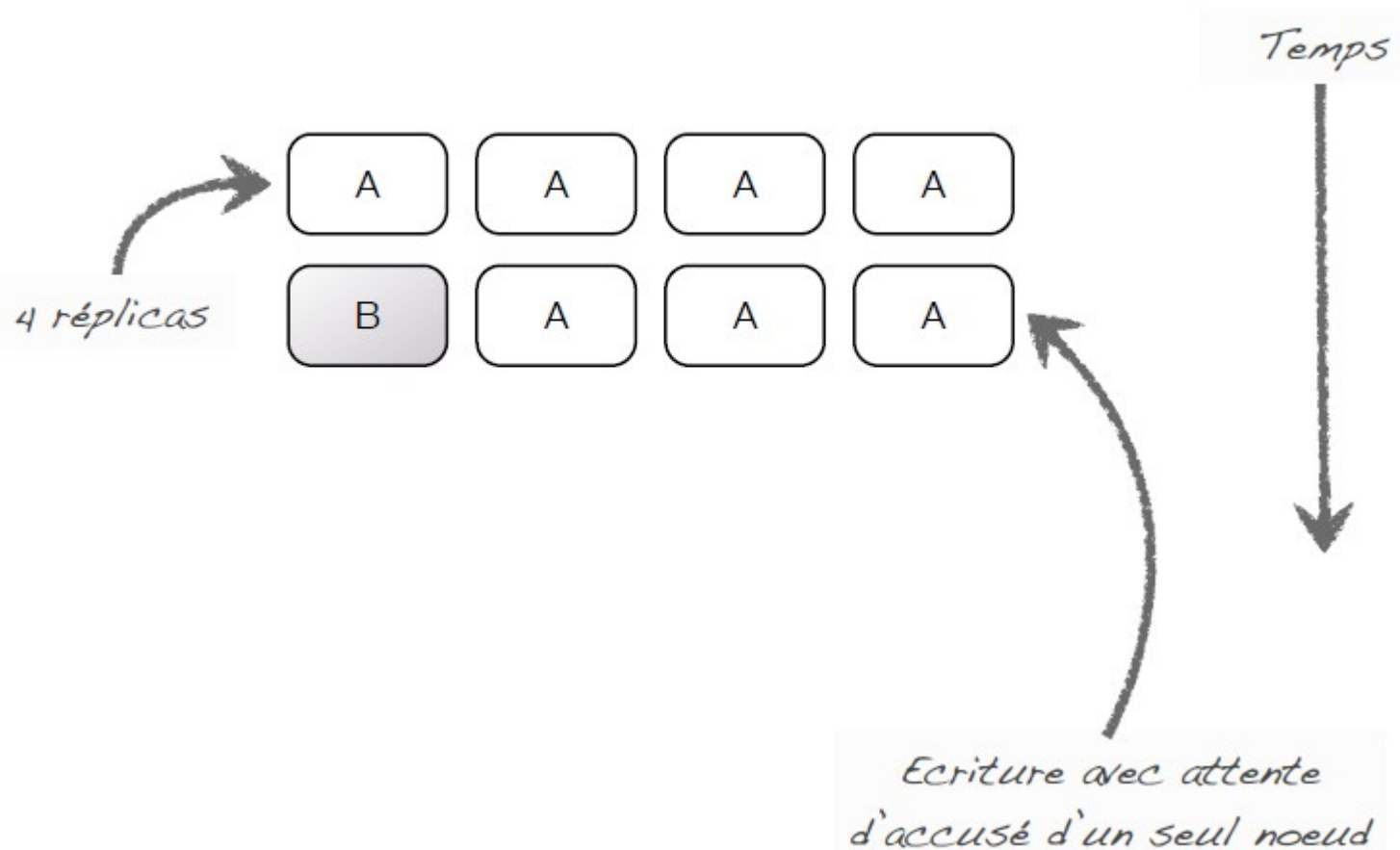
Consistance éventuelle



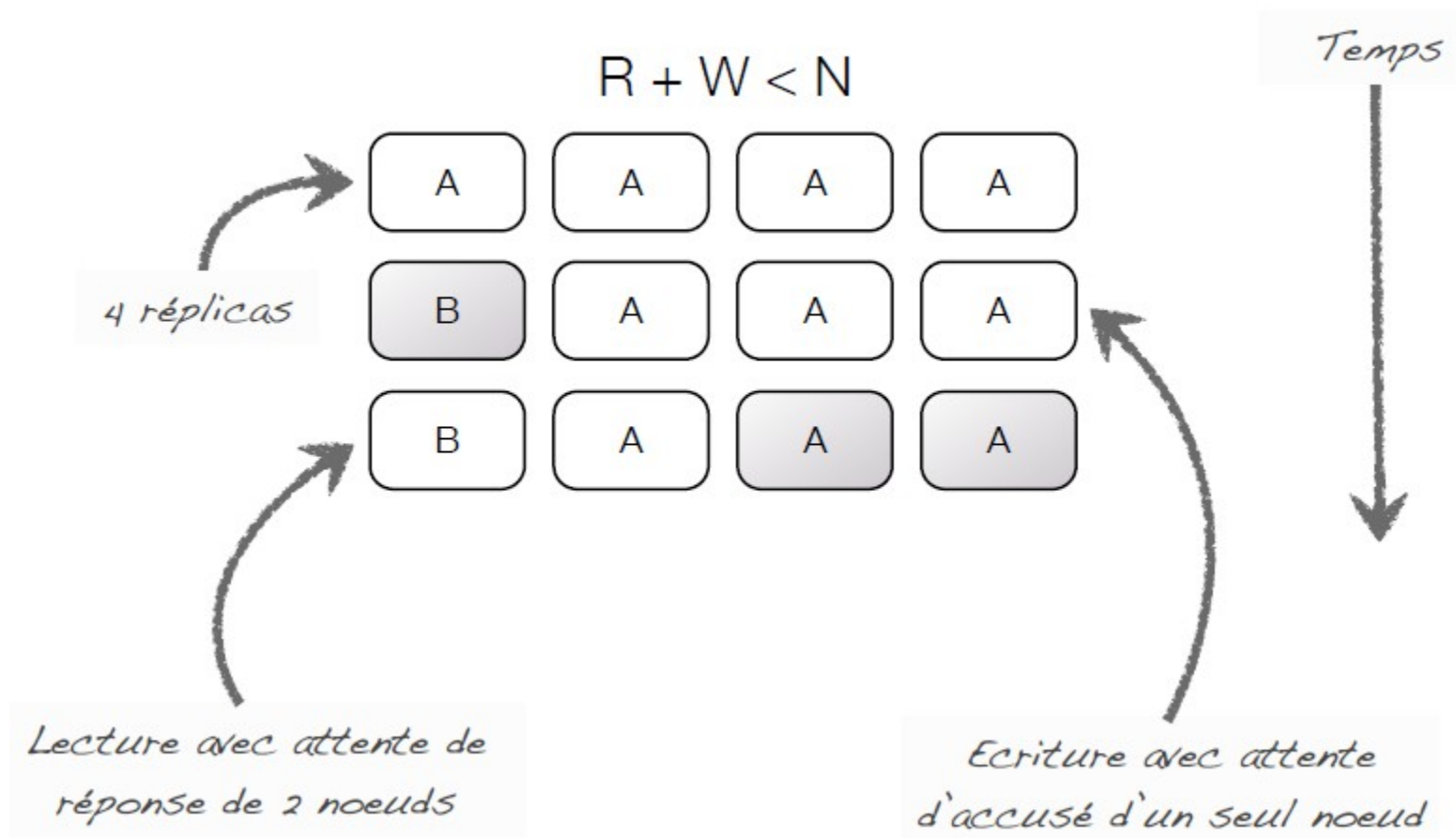
Consistance selon le nb de réponses attendues



Consistance selon le nb de réponses attendues

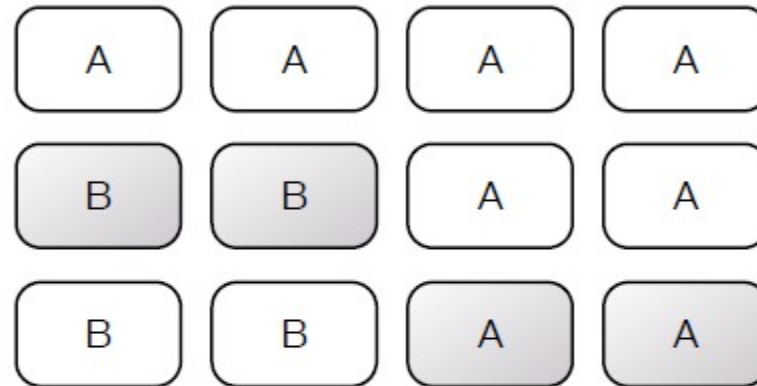


Consistance selon le nb de réponses attendues



Consistance selon le nb de réponses attendues

$$R + W = N$$

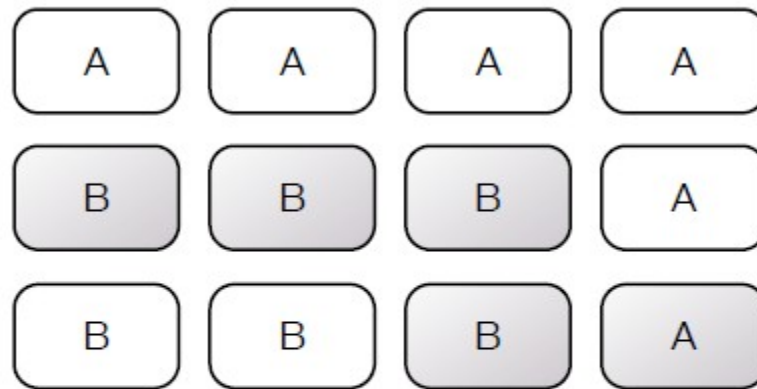


*Lecture avec attente de
réponse de 2 noeuds*

*Ecriture avec attente
d'accusé de 2 noeuds*

Consistance selon le nb de réponses attendues

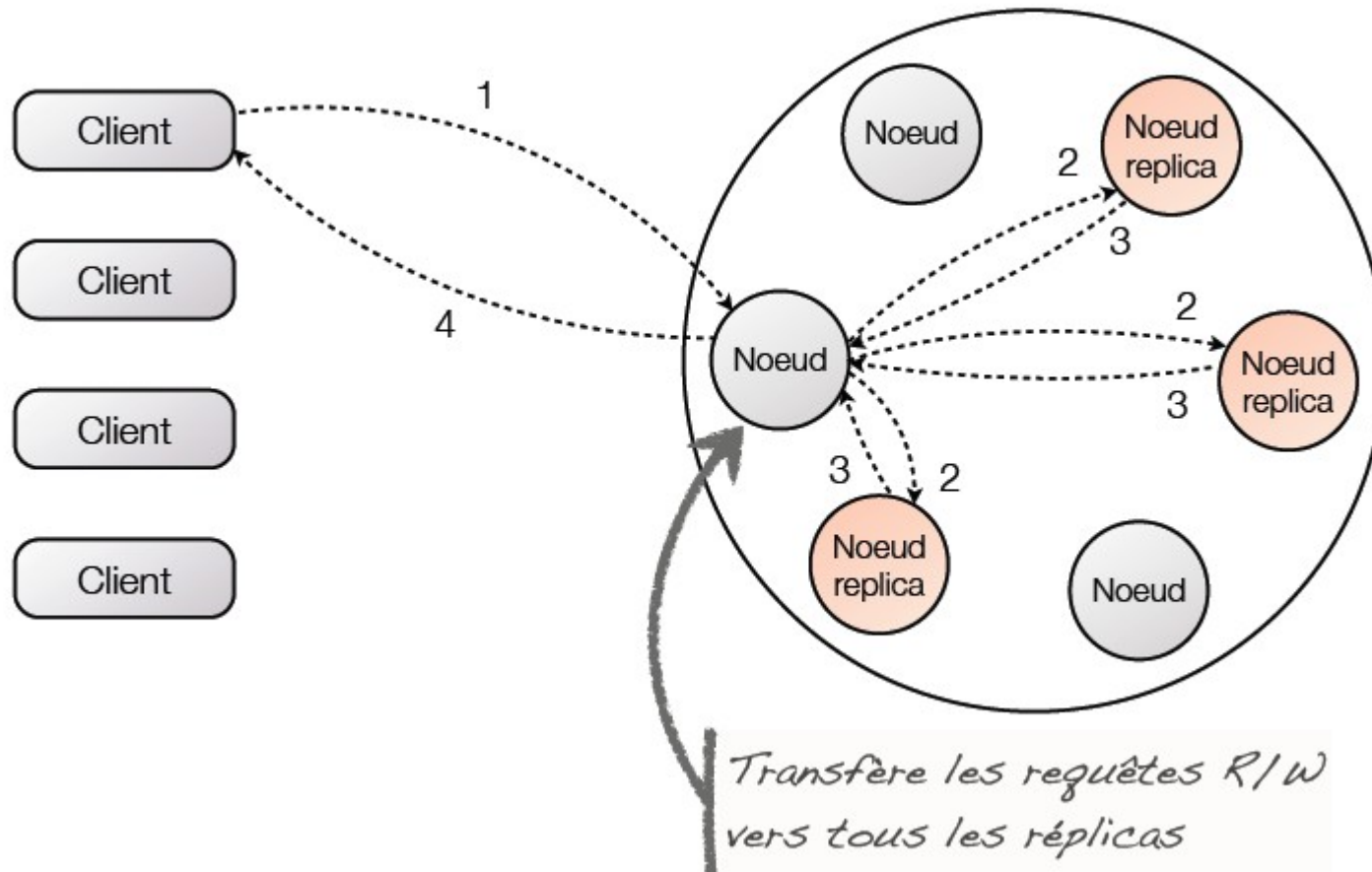
$$R + W > N$$



*Lecture avec attente de
réponse de 3 noeuds*

*Ecriture avec attente
d'accusé de 2 noeuds*

Consistance apparente au Client



Atomicité et Isolation

- Les données ne sont plus co-localisées
↳ *Localisation non prédictible dans le temps*
- Les transactions distribuées nuiraient à la disponibilité et aux performances
- Atomicité et Isolation par opération sur une clé

Update

- Gestion concurrence multi-version
 - Idem que dans SGBD pour accès à une ressource en lecture ou écriture
- Mise-à-jour:
 - différent des SGBDR où on supprime la donnée et on écrit la nouvelle
 - en NoSQL étiquette sur l'ancienne version pour dire que obsolète, ajout de la nouvelle version
 - anciennes versions nettoyées périodiquement
- Problème de cohérence des données dupliquées
 - Horloges vectorielles (un noeud garde un vecteur de versions/timestamp pour lui et les autres noeuds répliques)

Durabilité

- Ecriture sur un ou plusieurs disques

↳ *La réplication permet de renforcer la durabilité*

- Ecriture multiples en mémoire

↳ *La réplication apporte la durabilité*

- En mémoire avec écriture asynchrone sur disque

↳ *Pas de durabilité*

BD NoSQL : caractéristiques

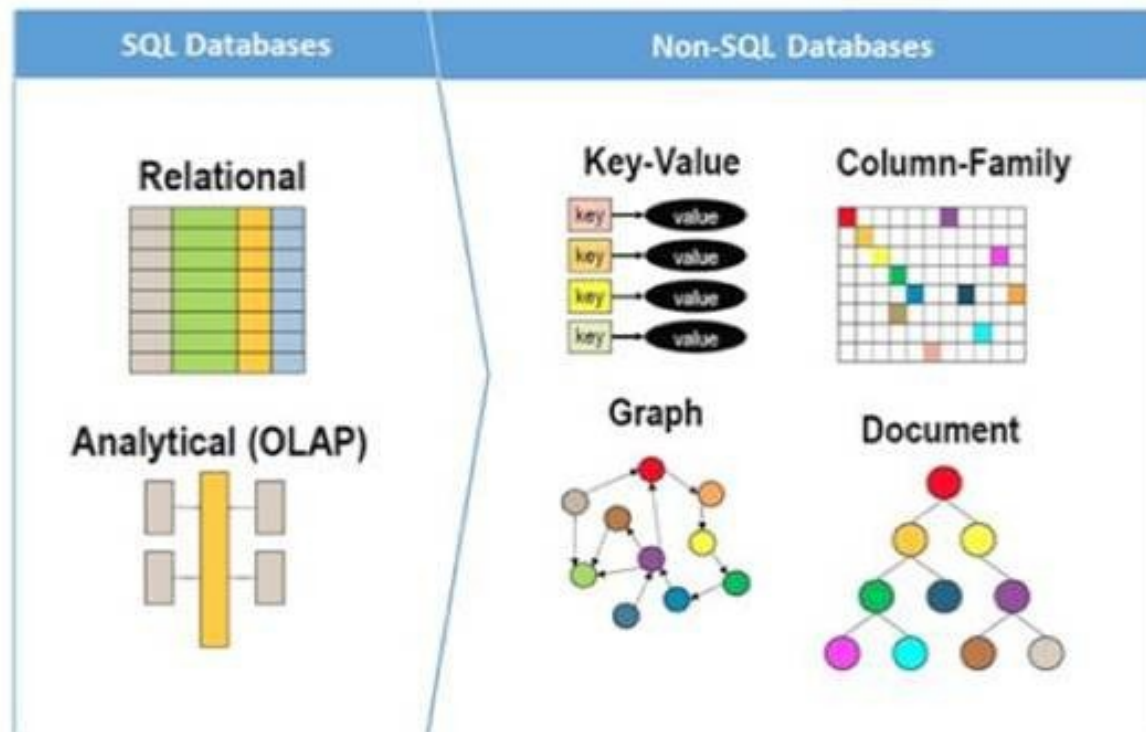
- Pas de relations
 - Pas de schéma physiques ou dynamiques
- Données complexes
 - Imbrication, tableaux
- Distribution de données (milliers de serveurs)
 - Parallélisation des traitements (Map/Reduce)
- Replication des données
 - Disponibilité vs Cohérence (pas de transactions)
 - Peu d'écritures, beaucoup de lectures

Quelques produits NoSQL

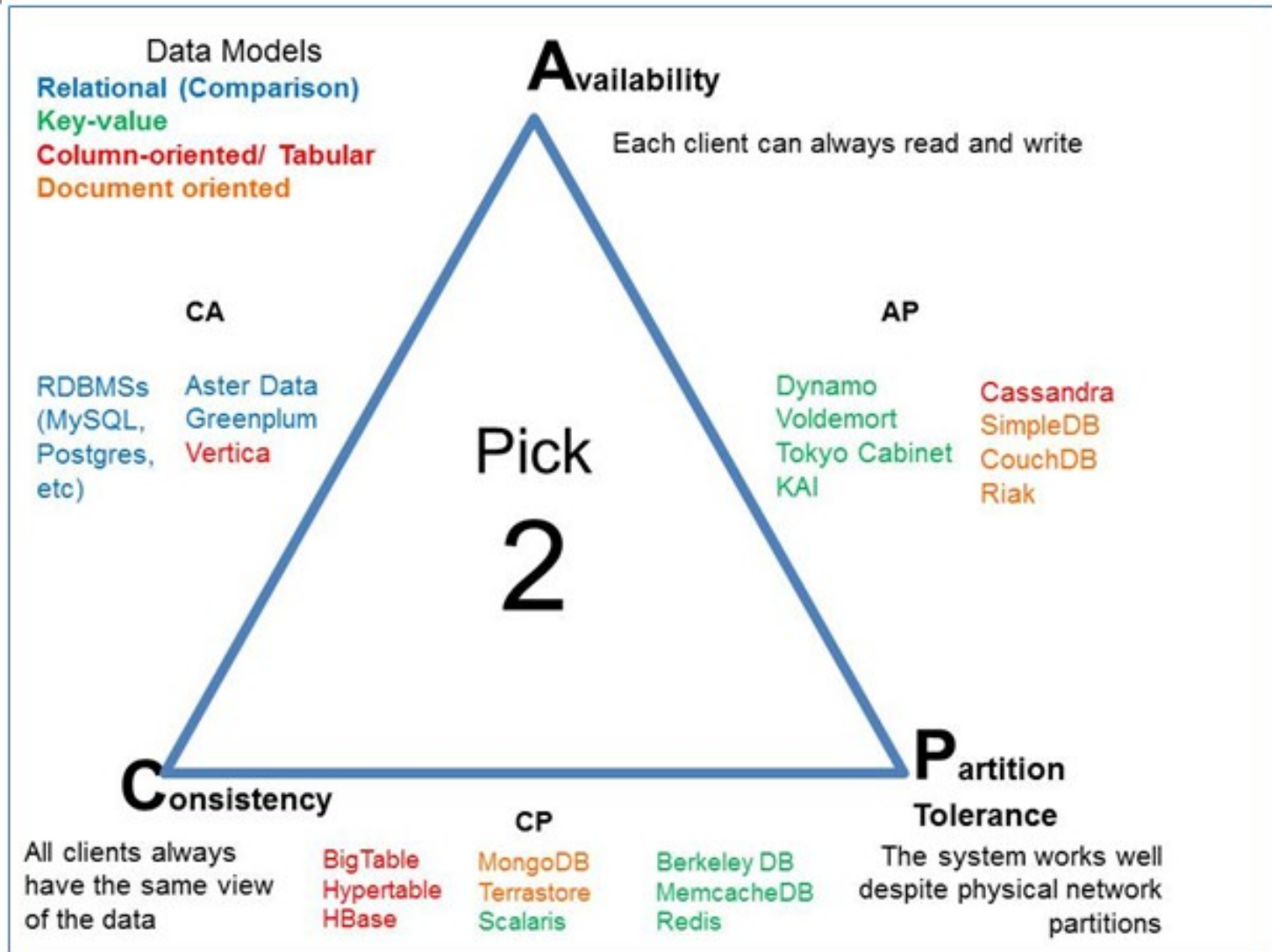
- BigTable (Google)
- Cassandra (Facebook, Twitter, Digg)
- CouchDB
- DynamoDB
- HBase
- MongoDB (SourceForge.net)
- Neo4j
- Project Voldemort (LinkedIn)
- Redis
- Riak
- SimpleDB (Amazon.com)

Une grande famille...

noSQL: “Not Only SQL”



Positionnements CAP



□ **Clé-valeur (Key-Value Store)**

- Données identifiées par clé unique (utilisées pour l'interrogation)
- *DynamoDB, Voldemort, Redis, Riak, MemcacheDB*

□ **Colonnes (Column data)**

- Relation 1-n “one-to-many” (messages, posts)
- *HBase, Cassandra, Hypertable*

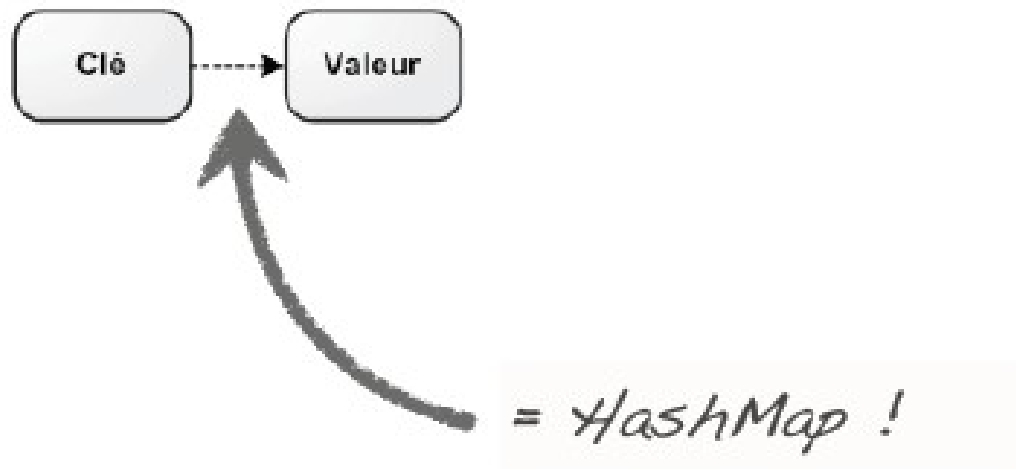
□ **Documents**

- Données complexes, attributs/valeurs
- *MongoDB, CouchDB, Terrastore*

□ **Graphes**

- Réseaux sociaux...
- *Neo4j, OrientDB, FlockDB*

BDD orientée Clé-Valeur



BDD orientée Clé-Valeur

- Similaires à un “*HashMap*” distribué
- Couple Clé+Valeur
 - Pas de schéma pour la valeur (chaîne, objet, entier, binaires...) qui peut donc être différente pour chaque tuple
- Conséquences :
 - Pas de structure ni de types
 - Pas d'expressivité d'interrogation (pré/post traitement pour manipuler concrètement les données)
- DynamoDB (Amazon), Redis (VMWare), Voldemort (LinkedIn)

BDD orientée Clé-Valeur

- Opérations CRUD (HTTP)
 - Create(key,value)
 - Read(key)
 - Update(key,value)
 - Delete(key)
- Passage à l'échelle au niveau horizontal
 - partitionnement/distribution des tuples
- Difficile au niveau vertical
 - segmentation des données

BDD orientée Clé-Valeur

□ Avantages:

- Modèle très simple
- Très bonne scalabilité en lecture et en écriture
- Modification facile du contenu associé à une clé et par extension du « schéma » (ajout d'une colonne par ex.)
- Augmente la disponibilité

□ Inconvénients:

- Interrogation simple car accès par clé => la complexité reportée au pré/post traitement

Exemple avec Riak

```
// Connexion à l'instance Riak
RiakClient riak = new RiakClient("http://localhost:8098/riak");

// Ecriture
RiakObject o = new RiakObject("bucket", "key", "value");
riak.store(o);

// Lecture
FetchResponse r = riak.fetch("bucket", "key");
if (r.hasObject()) {
    o = r.getObject();
    System.out.println(
        "bucket: " + o.getBucket()
        + ", key: " + o.getKey()
        + ", value: " + o.getValue());
}
```

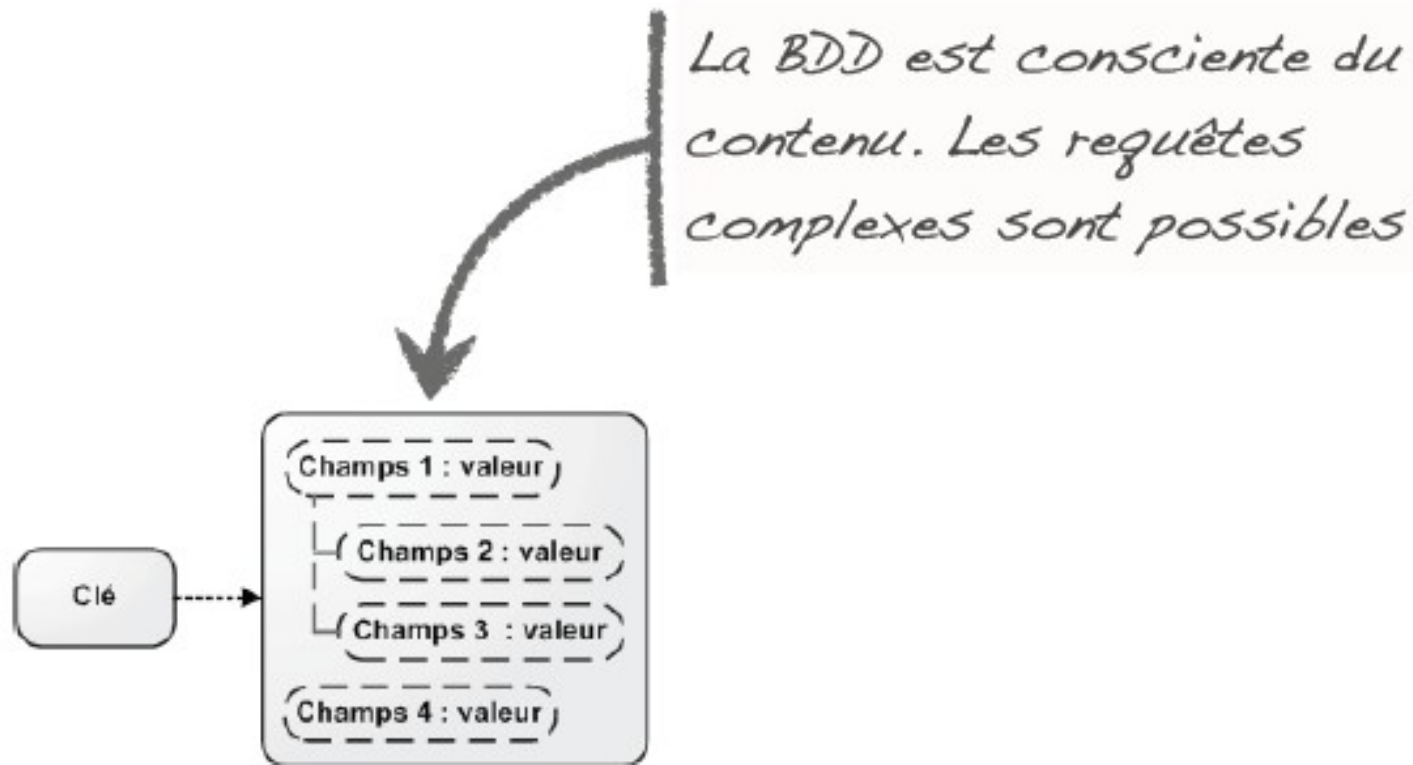
Exemple avec Riak

```
// Définition d'un coefficient de réplication de 3 pour le bucket
RiakBucketInfo bucketInfo = new RiakBucketInfo();
bucketInfo.setNVal(3);
riak.setBucketSchema("bucket", bucketInfo);

// Ecriture avec W=2
riak.store(o, RequestMeta.writeParams(2));

// Lecture avec R=1
client.fetch("bucket", "key", RequestMeta.readParams(1));
```

BDD orientées Documents



BDD orientées Documents

- Basé sur le modèle clé-valeur
 - Ajout de données semi-structurées (JSon ou XML)
- Requêtes : Interface HTTP
 - Plus complexe que CRUD
- MongoDB, CouchDB (Apache), RavenDB, Terrastore

BDD orientées Documents

- Comme clé-valeur mais chaque valeur est un document
- Un document est composé de champs et de valeurs associées
 - Types simples existent (Int, String, Date)
 - Schéma non nécessaire (peut varier d'un document à l'autre)
- Imbrication de données (schéma arborescent): chaque donnée peut être aussi composée de couples clé-valeur
- Chaque donnée du document peut être interrogée

BDD orientées Documents

□ **Avantages :**

- Modèle simple mais bonne puissance d'expression (structure imbriquée)
- Interrogation de tout attribut (+indexation)
- Passe facilement à l'échelle (surtout si sharding supporté)

□ **Inconvénients :**

- Inter-connexion de données complexe
- Interrogation sur la clé (+index)

Exemple MongoDB

- Une collection MongoDB en revanche pourrait se présenter de la manière suivante :

```
{ "_id": ObjectId("4efa8d2b7d284dad101e4bc9"),  
  "Nom": "DUMONT",  
  "Prénom": "Jean",  
  "Âge": 43 },  
{ "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),  
  "Nom": "PELLERIN",  
  "Prénom": "Franck",  
  "Âge": 29,  
  "Adresse": {  
    "Rue" : "1 chemin des Loges",  
    "Ville": "VERSAILLES"  
  }  
}
```

Exemple avec MongoDB

- En shell : dans une BD vente la collection nommée **clients** :
 - > use vente // Sélectionne la base de données "vente"
 - > db.clients.find(); // Cherche et affiche tous les documents de la collection "clients".
- Le résultat s'imprime à l'écran :
 - { "_id": 28974, "Nom": "ID Technologies", "Adresse" : "7 Rue de la Paix, Paris" }
 - { "_id": 89136, "Nom": "Yoyodine", "Adresse" : "8 Rue de la Reine, Versailles" }

Exemple avec MongoDB

```
Mongo m = new Mongo("localhost", 27017);
DB db = m.getDB("mydb");
DBCollection coll = db.getCollection("testCollection")

BasicDBObject doc = new BasicDBObject();
doc.put("name", "NoSQL Europe");
doc.put("type", "Conference");
doc.put("attendees", 100);
coll.insert(doc);

// Affiche le nombre de documents dans la collection
System.out.println(coll.getCount());
```

Exemple avec MongoDB

```
// Définition d'une requête portant sur tous les documents avec attendees > 80
BasicDBObject query = new BasicDBObject();
query.put("attendees", new BasicDBObject("$gt", 80));

DBCursor cur = coll.find(query);
while(cur.hasNext()) {
    System.out.println(cur.next());
}

// Une requête portant sur les documents avec 20 <attendees <= 100
query = new BasicDBObject();
query.put("attendees", new BasicDBObject("$gt", 20).append("$lte", 100));

DBCursor cur = coll.find(query);
while(cur.hasNext()) {
    System.out.println(cur.next());
}
```

Exemple avec Jongo/MongoDB

- Jongo offre une désérialisation des résultats en objets Java (*avec Jackson*).

```
// Mongo shell
```

```
db.peoples.find({age: {$gt: 18}})
```

```
// Jongo
```

```
Iterable<People> adults = peoples.find("{age: {$gt: 18}}").as(People.class);
```

```
// Java driver
```

```
Iterable<DBObject> adults = peoples.find(new BasicDBObject("age", new BasicDBObject("$gt", 18)));
```

```
// Morphia
```

```
Iterable<People> adults =  
    ds.createQuery(People.class).field("age").greaterThan(18)  
    ;
```

BDD orientée Colonnes

*A chaque ID de ligne correspond
une liste de couples clé-valeur*

	A	B	C	D	E
1	foo	bar	hello		
2		Tom			
3			java	scala	cobol

BDD relationnelle

1	<div><div>A foo</div><div>B bar</div><div>C hello</div></div>				
2	<div><div>B Tom</div></div>				
3	<div><div>C java</div><div>D scala</div><div>E cobol</div></div>				

BDD orientée colonnes

BDD orientée Colonnes

- Stockage des données par colonnes
 - SGBD : tuples (lignes)
- Facile d'ajouter une colonne (pas une ligne!)
 - Schéma peut être dynamique (d'un tuple à l'autre)
- BigTable/Hbase (Google), Cassandra (Facebook&Apache), SimpleDB (Amazon)

BDD orientée Colonnes

□ **Avantages :**

- Supporte le semi-structuré (XML, JSon) donc multi-valué et *sparsité*
- Indexation de chaque colonne
- Passage à l'échelle horizontal
- Compression possible si les données d'une colonne proches
- Possibilité de regrouper des colonnes en super-colonnes

□ **Inconvénients :**

- Lecture de données complexes difficile
- Difficulté de relier les données (distance, trajectoires, temps)
- Requêtes pré-définies (pas à la volée)

Exemple avec Cassandra

```
TTransport tr = new TSocket("192.168.216.128", 9160);

TProtocol proto = new TBinaryProtocol(tr);
tr.open();
Cassandra.Client cassandraClient = new Cassandra.Client(proto);

Map<String, List<ColumnOrSuperColumn>> insertClientDataMap =
    new HashMap<String, List<ColumnOrSuperColumn>>();
List<ColumnOrSuperColumn> clientRowData =
    new ArrayList<ColumnOrSuperColumn>();

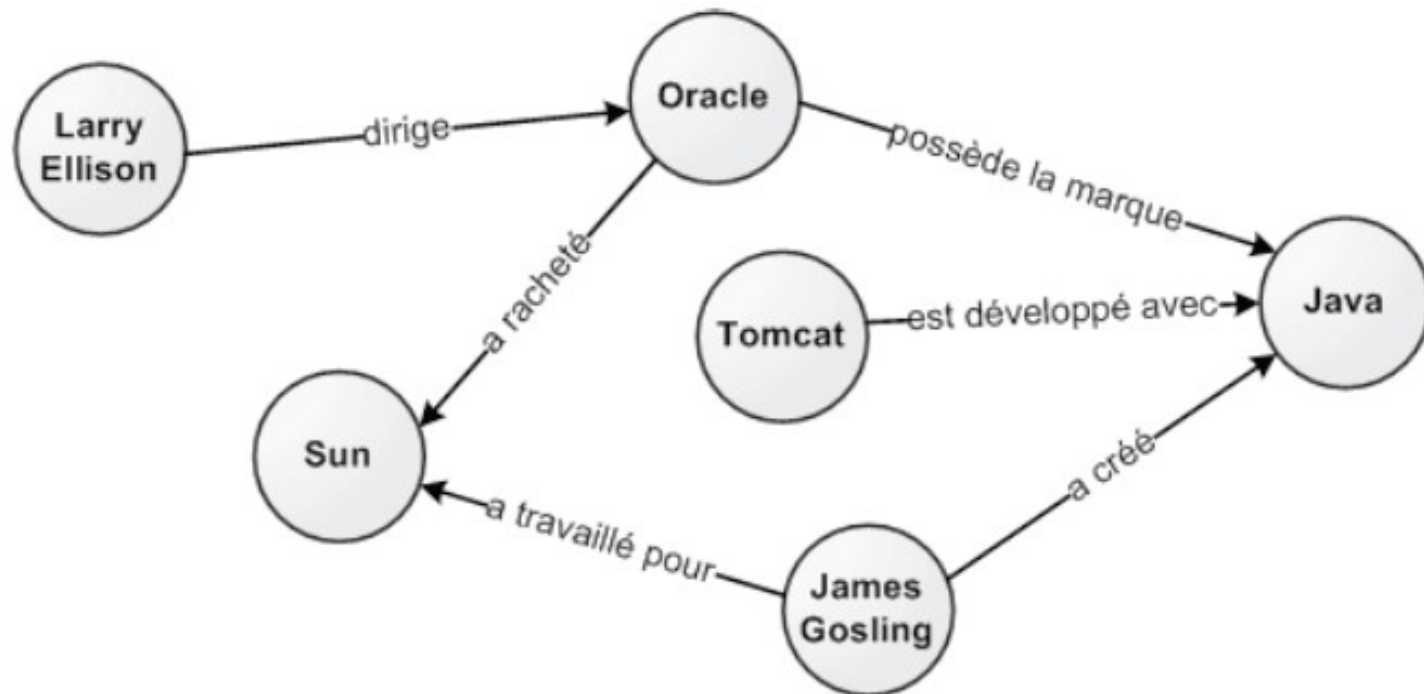
ColumnOrSuperColumn columnOrSuperColumn = new ColumnOrSuperColumn();
columnOrSuperColumn.setColumn(new Column("fullName".getBytes(UTF8),
aCustomer.getName().getBytes(UTF8), timestamp));
clientRowData.add(columnOrSuperColumn);

insertClientDataMap.put("customers", clientRowData);
cassandraClient.batch_insert("myBank", aCustomer.getName(),
insertClientDataMap, ConsistencyLevel.DCQUORUM);
```

Cassandra

The logo for digg, featuring the word "digg" in a bold, blue, sans-serif font. The letters are slightly offset, giving it a 3D or layered appearance.The logo for Twitter, featuring the word "twitter" in a light blue, rounded, lowercase font with a white outline.The logo for Facebook, featuring the word "facebook" in a white, sans-serif font inside a dark blue rectangular box.

BDD orientées Graphes



BDD orientées Graphes

- Stockage des noeuds, relations et propriétés
 - Théorie des graphes
 - Interrogation par traversées de graphe
 - Appel des données sur demande (parcours performants)
 - Modélisation non triviale
- Neo4j, OrientDB (Apache), FlockDB (Twitter)

BDD orientées Graphes

□ Avantages

- Modèle adapté pour le stockage de grands graphes
- Offre des fonctionnalités de calculs sur grands graphes
- les objets (cf. documents)
- les arcs (avec propriétés)

□ Inconvénients:

- Sharding/partitionnement difficile (impossible ?)

Exemple avec Neo4J

```
GraphDatabaseService graphDb = new EmbeddedGraphDatabase("/var/graphdb");

// Toute lecture et écriture doit se faire dans une transaction avec Neo4J
Transaction tx = graphDb.beginTx();
try
{
    Node jamesGosling = graphDb.createNode();
    Node java = graphDb.createNode();
    Relationship relationship =
        jamesGosling.createRelationshipTo(java, CustomRelationships.CREATED);

    jamesGosling.setProperty("nom", "James Gosling");
    java.setProperty("date", "1995");
    relationship.setProperty("type", "a créé");

    tx.success();
}
finally
{
    tx.finish();
}

graphDb.shutdown();
```

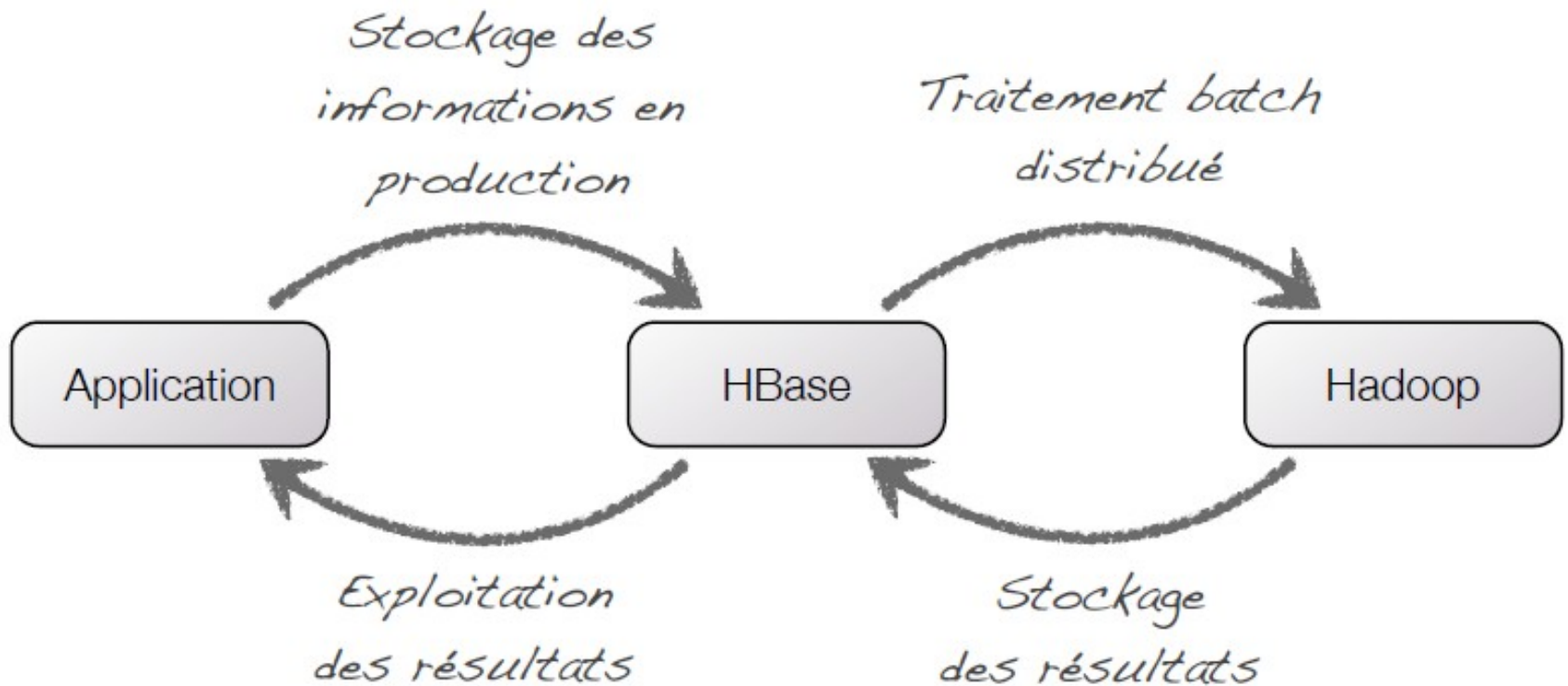

Intérêts des BD NoSQL

- Stockage polyglotte : une meilleure adéquation entre la BDD et les données
- Scalabilité linéaire : être à même de répondre aux besoins les plus gourmands
- Haute disponibilité : du multi-serveurs au multi-datacenters
- Elasticité : une intégration naturelle à la logique du Cloud Computing
- Curseur pour s'adapter : + de consistance ou + de fiabilité ($R + W > N$)
- Et finalement... la possibilité crée le besoin !

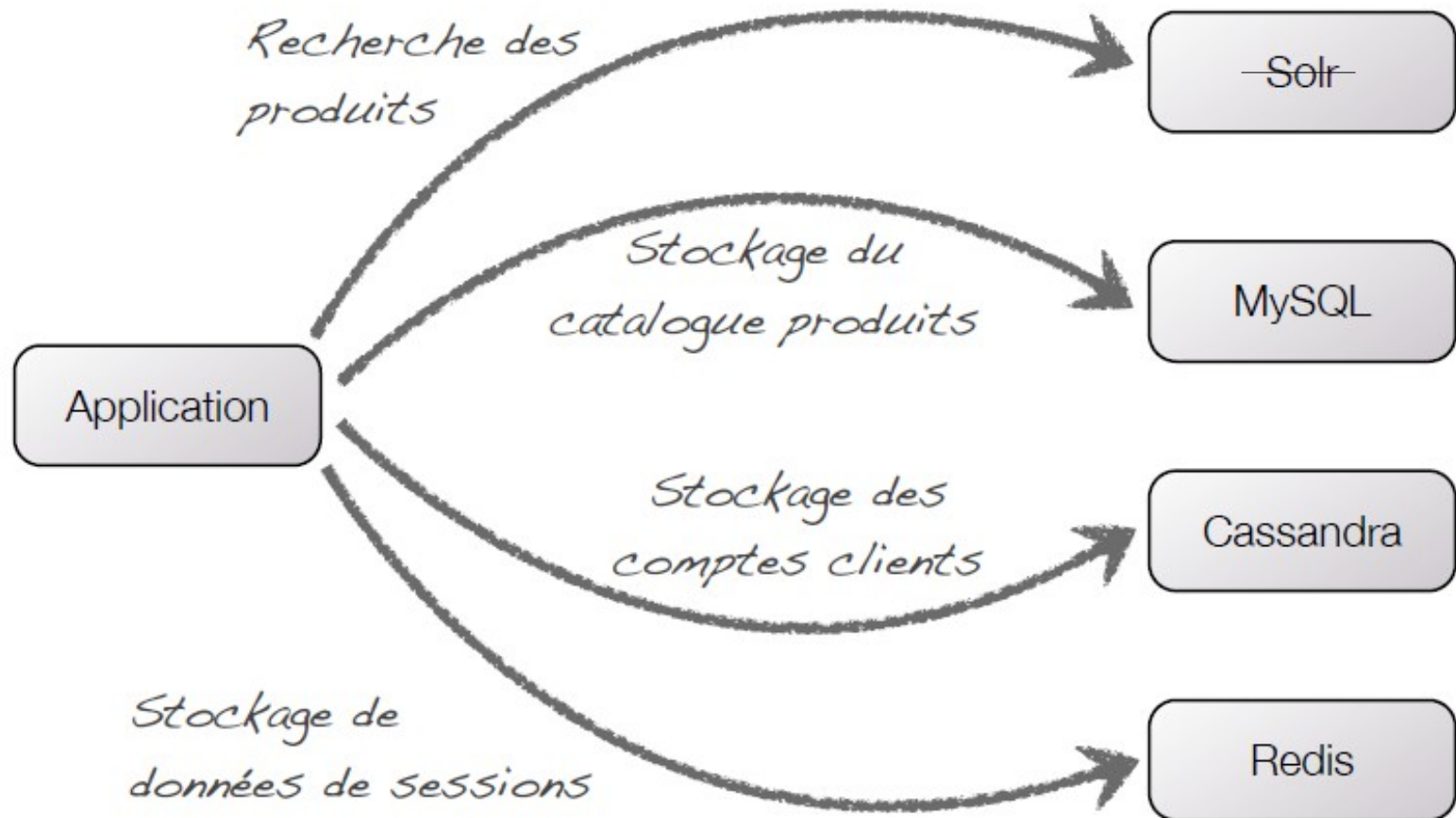
NoSQL en prod ?

- En production chez de nombreux « Grands du Web »
- Outillage encore réduit
- Monitoring par JMX
- Backups peuvent être problématiques avec des volumes importants

Ucase : BI sur la BDD de production



Stockage polyglotte



DB engine ranking

358 systems in ranking, September 2020

Rank			DBMS	Database Model	Score		
Sep 2020	Aug 2020	Sep 2019			Sep 2020	Aug 2020	Sep 2019
1.	1.	1.	Oracle	Relational, Multi-model	1369.36	+14.21	+22.71
2.	2.	2.	MySQL	Relational, Multi-model	1264.25	+2.67	-14.83
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1062.76	-13.12	-22.30
4.	4.	4.	PostgreSQL	Relational, Multi-model	542.29	+5.52	+60.04
5.	5.	5.	MongoDB	Document, Multi-model	446.48	+2.92	+36.42
6.	6.	6.	IBM Db2	Relational, Multi-model	161.24	-1.21	-10.32
7.	7.	8.	Redis	Key-value, Multi-model	151.86	-1.02	+9.95
8.	8.	7.	Elasticsearch	Search engine, Multi-model	150.50	-1.82	+1.23
9.	9.	11.	SQLite	Relational	126.68	-0.14	+3.31
10.	11.	10.	Cassandra	Wide column	119.18	-0.66	-4.22
11.	10.	9.	Microsoft Access	Relational	118.45	-1.41	-14.26
12.	12.	13.	MariaDB	Relational, Multi-model	91.61	+0.69	+5.54
13.	13.	12.	Splunk	Search engine	87.90	-2.01	+0.89
14.	14.	15.	Teradata	Relational, Multi-model	76.39	-0.39	-0.57
15.	15.	14.	Hive	Relational	71.17	-4.12	-11.93
16.	16.	18.	Amazon DynamoDB	Multi-model	66.18	+1.43	+8.36
17.	17.	25.	Microsoft Azure SQL Database	Relational, Multi-model	60.45	+3.60	+32.91
18.	18.	19.	SAP Adaptive Server	Relational	54.01	+0.05	-2.09
19.	19.	21.	SAP HANA	Relational, Multi-model	52.86	-0.26	-2.53
20.	20.	16.	Solr	Search engine	51.62	-0.08	-7.35
21.	21.	22.	Neo4j	Graph	50.63	+0.44	+2.41
22.	22.	20.	HBase	Wide column	48.35	-0.76	-7.37
23.	23.	17.	FileMaker	Relational	47.58	-0.46	-10.57
24.	24.	28.	Google BigQuery	Relational	33.32	+0.72	+8.77
25.	25.	24.	Microsoft Azure Cosmos DB	Multi-model	31.67	+0.94	+0.80

Conclusion

- NoSQL :
 - Dédié à un contexte extrêmement distribué
 - Calcul fortement distribué
 - 4 types de calculs complexes (clé-valeur, document, colonnes, graphes)
- Ne doit pas remplacer automatiquement un SGBD
 - Propriétés ACID
 - Requêtes complexes
 - Performance des jointures !!!