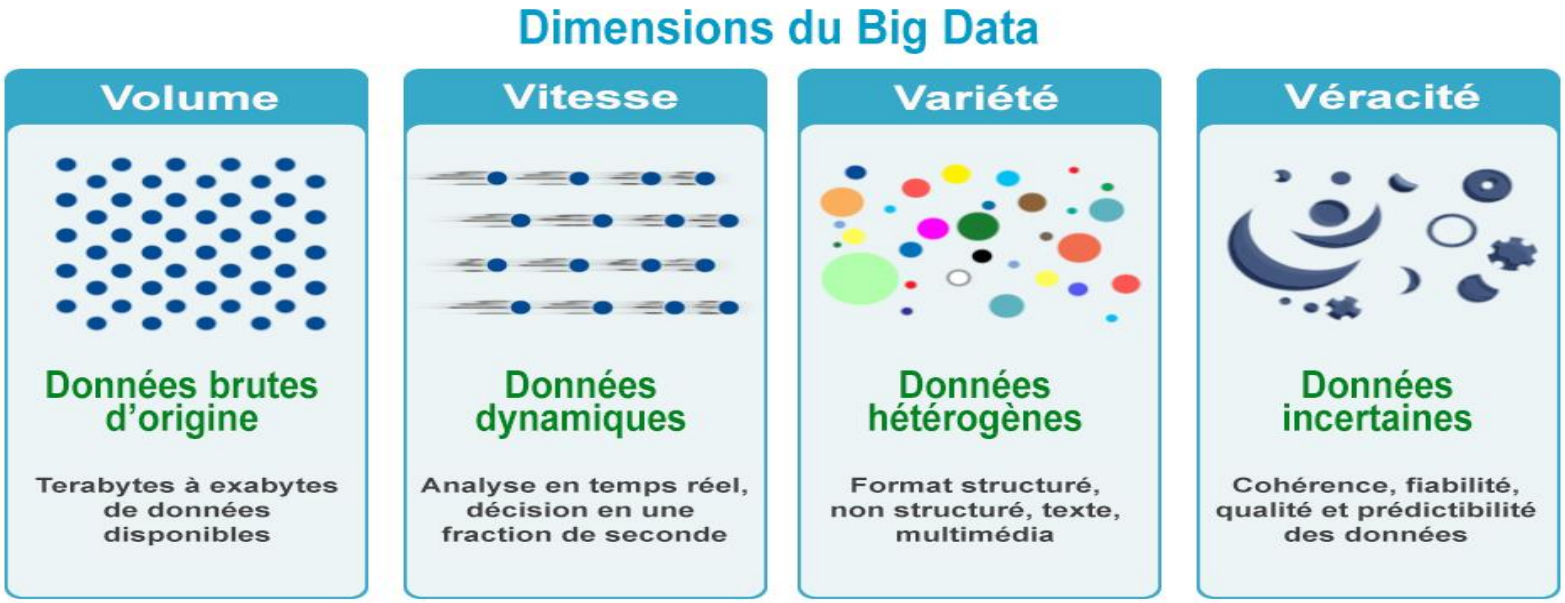
Module MLBDA
Master Informatique
Spécialité DAC

COURS 10 – NOSQL

Les 4+V du Big Data



Nouvelles architectures de stockage et d'interopérabilité

Nouveaux modèles de calcul sur des flux (Défi pour les réseaux de communication)

Nouveaux modèles, nouveaux outils de nettoyage, transformation et fusion de données

Nouveaux modèles de qualité (données & processus de traitement)

http://www.astrosurf.com/luxorion/big-data-mining.htm

Document IBM et T. Lombry

Systèmes NoSQL (not only SQL)

Systèmes qui abandonnent certaines propriétés des SGBDR (« one size does not fit all »):

- Le modèle de données définissant des tables avec des attributs atomiques
- Le rôle du schéma comme système de contraintes sur les données
- La notion de transaction avec isolation et concurrence d'accès

Pourquoi ?

- Nouvelles applications : jeux, réseaux sociaux, OLAP, machine learning
- Besoin de « passage à l'échelle » et de flexibilité

Comment ?

- Nouveaux modèles de données et langages
- Nouvelles architectures avec une forte distribution des données et des traitements (cluster)
- Adaptation élastique à la charge et au volume (cloud)

Flexibilité et Passage à l'échelle

Données
- Web2.0 : réseaux sociaux, news, blogs,…
- Graphes, ontologies
- Flux de données : capteurs, GPS,…

→ Très gros volumes, données pas ou peu structurées

Traitements
- Moteurs de recherche
- Extraction, analyse
- Recommandation, filtrage collaboratif

→ Transformation, agrégation, indexation

Infrastructures
- Clusters, réseaux mobiles, data centers, microprocesseurs multicoeurs

→ Distribution, redondance, parallélisation

Théorème CAP

Aucun système distribué **ne peut garantir simultanément plus de deux** des trois propriétés suivantes :

- Consistency (cohérence) : un service est exécuté entièrement ou pas du tout (propriétés ACID). Tous les nœuds voient les mêmes données en même temps.
- Availability (disponibilité) : le service est toujours accessible, même en cas de panne d'un nœud.
- Partition tolerance (tolérance au partitionnement) : aucune panne autre que la rupture totale du réseau ne doit empêcher le système de fonctionner.

Toutes ces propriétés sont souhaitables, mais il n'est pas possible de les avoir simultanément.

Systèmes noSQL

Abandon des caractéristiques des SGBD SQL:

- Architecture serveur-client centralisée → Architecture cluster distribuée
- Cohérence → Tolérance aux partitionnement
- Modèle de données structurées → Modèles de données « semi-structurées »

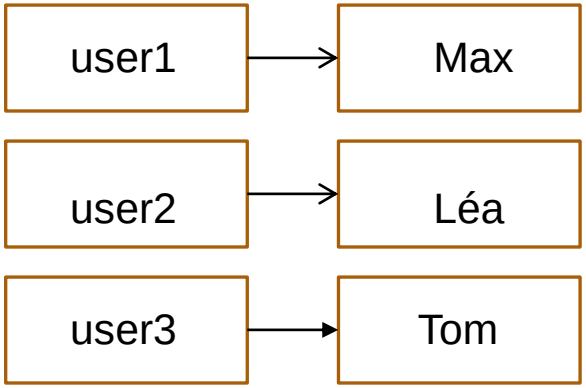
4 catégories de systèmes :

- **Clés-valeurs**
- **Documents**
- **Tables orientées colonnes**
- **Graphes**

noSQL **Clef-valeur**

La base est une ***collection d'objets***

Chaque objet est représenté par un couple ***clef-valeur*** (Ki,Vi).



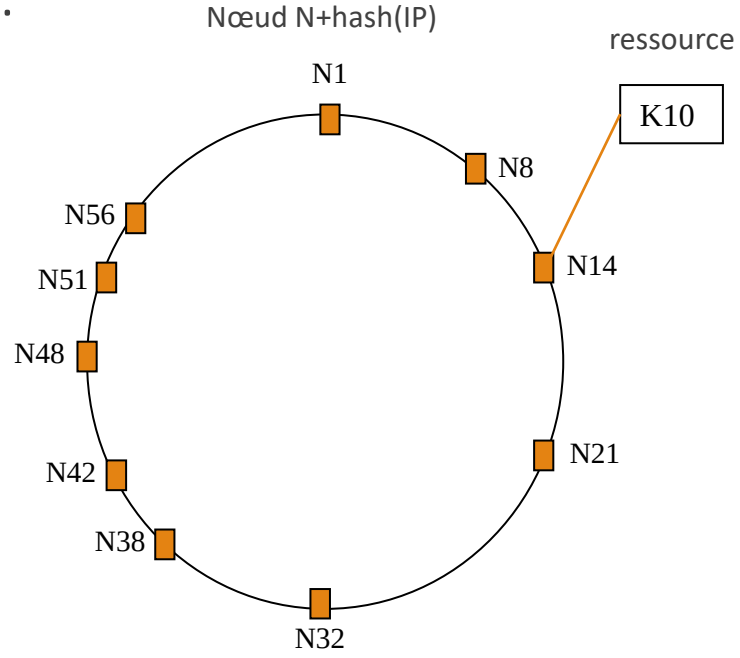
Exemples : S3, Amazon EC2, Voldemort, DynamoDB, Riak, Redis …

Système Clef-Valeur : stockage

Les objets sont distribuées sur des nœuds connectés en anneau

Les ressources sont réparties uniformément en utilisant une **table de hachage distribuée** (consistent hashing) :

- Fonction de hachage **hash** : Int →[0,Max]
- A chaque nœud **Ni** est alloué une position sur l'anneau en fonction d'une clé de hachage **hash(IP)=i** sur son adresse IP
- Placement d'un objet (C,V):
  - **hash(C)=Kj**
  - L 'objet est placée sur le nœud avec la clé **hash(IP)=Ni** ou *i est immédiatement supérieur à j*.
- **Recherche par propagation**



Clef-valeur : opérations

- Create(clef, valeur) : crée et écrit un nouveau objet (clef, valeur)
- Read(clef) → valeur : lit un objet à partir de sa clef
- Update(clef, valeur) : met à jour la valeur d'un objet à partir de sa clef
- Delete(clef) : supprime un objet à partir de sa clef

➔ Fonction principale : **recherche(hash(clef))**

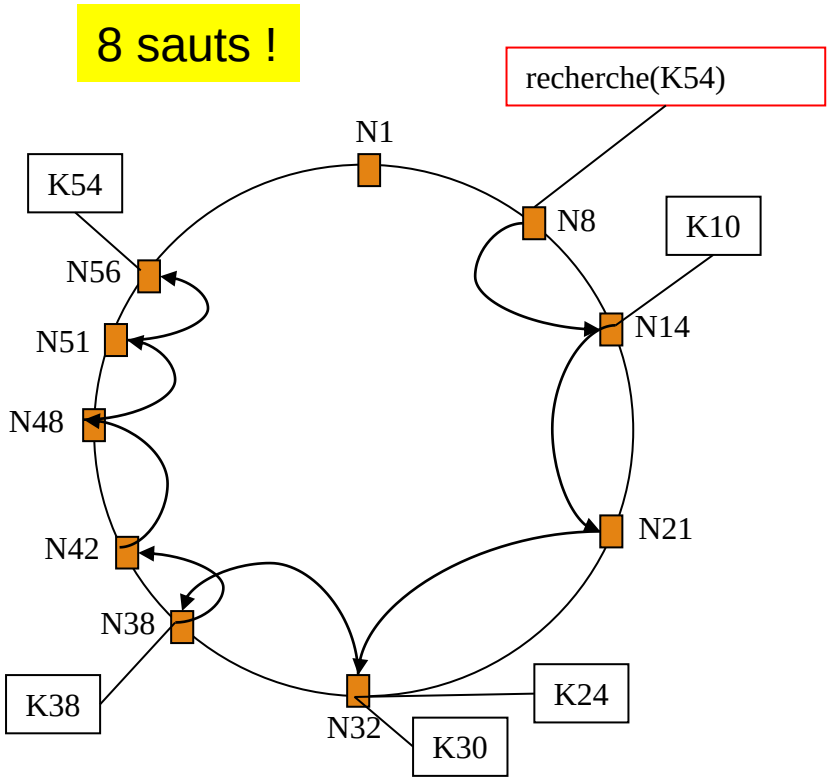
Clef-valeur : Recherche (naïve) d'une ressource

Chaque nœud connaît uniquement l'IP/port de son successeur (pas de table de routage globale)

Chaque nœud peut recevoir une opération sur une clé.

Sur le nœud Ni on reçoit la requête : **recherche(Kj)**

- Si Ni possède Kj, il retourne (Kj,Vj)
- Sinon, il propage la requête à son successeur *et ainsi de suite*.
- Le résultat suit le chemin dans l'ordre inverse
- Nombre d'étapes linéaire en nombre de nœuds = *1/2 nombre de nœuds en moyenne*



Clef-valeur : Exemple de recherche

- Chaque nœud Ni possède une **table de routage locale Succ_i avec m entrées**
- **Succ_i[k]=IP/port** du premier nœud sur l'anneau dont la clé vérifie l'équation $(i + 2^{k-1}) \bmod 2^m, 1 <= k <= m$

3 sauts au lieu de 8 sauts !

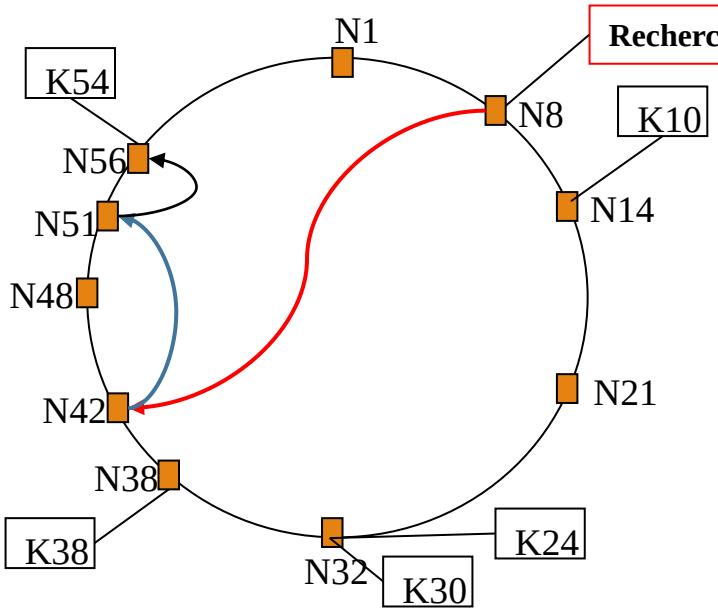


Table routage N8

N8+1	9	N14
N8+2	10	N14
N8+4	12	N14
N8+8	16	N21
N8+16	24	N32
N8+32	40	N42

Table routage N42

N42+1	43	N48
N42+2	44	N48
N42+4	46	N48
N42+8	50	N51
N42+16	58	N1
N42+32	74	N1

$m = 6$

noSQL **Clef-valeur**

+ Modèle de données très simple
+ Passage à l'échelle : évolutivité, disponibilité
+ Recherche par clef efficace et simple (API avec requêtes HTTP)
- Recherche par valeur impossible → indexation supplémentaire
- Recherche par intervalle inefficace
- Pas de langage d'interrogation de données complexes (une clé par objet)

Absence de schéma et de langage de requête : les systèmes **clef-valeur** peuvent plus être considérés comme de *couches physiques de stockage* que des bases de données.

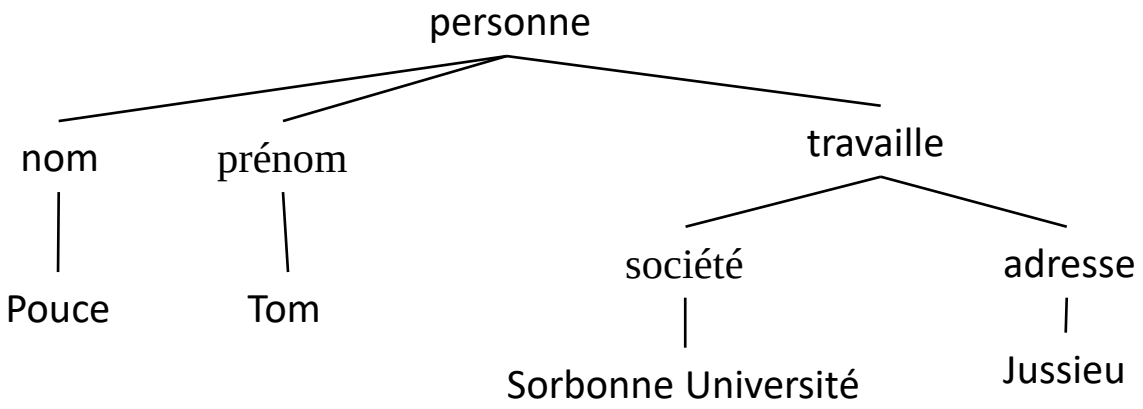
noSQL **Document**

La base est une ***collection de documents***.

Un **document** est composé de **champs** et de **valeurs** associées

Les **valeurs** sont des **structures ensemblistes imbriquées :** *listes*, *array*, *enregistrements*

Schémas **optionnels** utilisés pour la *validation* des données.



Ex : MongoDB, **CouchDB (JSON),** Xbase (**XML**), …

Document : modèle JSON

Exemple d'un document JSON

{ nom: "Alan", tél: 2157786, email: "agb@abc.com"}

Extension naturelle : les valeurs sont elles-mêmes des structures (documents) :

{ nom: { prénom: "Alan", famille: "Black« },

tél: 2157786,

email: "agb@abc.com"}

Exemple Couchbase / JSON

Modèle de données: JSON (JavaScript Object Notation)

Objet (document) :

- collection non-ordonnée de clés/valeurs
- { cle1: valeur1, clé2: valeur2, .. }

Clés:

- Chaînes de caractères

Valeurs:

- String
- Number
- *true*, *false*
- *null*
- **Objet** ou **Array**

Array:

- Liste ordonné de valeurs
- [valeur1, valeur2, …]

Exemple Objet JSON

```
{
 "firstName": "John",
 "lastName": "Smith",
 "isAlive": true,
 "age": 27,
 "address": {
 "streetAddress": "21 2nd Street",
 "city": "New York",
 "state": "NY",
 "postalCode": "10021-3100"
 },
 "phoneNumbers": [
  {
  "type": "home",
  "number": "212 555-1234"
  },
  {
  "type": "office",
  "number": "646 555-4567"
  }
  ],
  "children": [],
  "spouse": null
}
```

objet

array (contient deux objets)

Exemple Couchbase / N1QL

N1QL

- Syntaxe SQL : SELECT – FROM – WHERE
- Similarités avec XQuery
- DDL : create index, alter index, drop index
- DML: select, insert, update, delete et *upsert (insert & update)*

- Table = **bucket** = array JSON
- Attribut = concaténation de clés (à la « SQL3 »)

- SELECT : projection sur les fragments d'objets à afficher
- FROM : buckets à interroger et combiner (JOIN, NEST, UNNEST, …)
- WHERE : filtres

Exemple Couchbase / N1QL

Bucket **country**:


```
[ {
   "@car_code": "AL",
   "@area": "28750",
   "@capital": "cty-Albania-Tirane",
   "name": { "#": "Albania" },
   "unemployment": {  "#": "16.9" },
   "indep_date": {
      "#": "1912-11-28",
      "@from": "Ottoman Empire«  },
   "government": {
      "#": "parliamentary democracy«   },
    "encompassed": {
      "@continent": "europe",
      "@percentage": "100"  },
   "border": [  {
      "@country": "GR",
      "@length": "282"
   }, {
      "@country": "MK",
      "@length": "151"  } ],
  "city": [  {
      "@id": "cty-Albania-Tirane",
      "@country": "AL",
      "name": [
           { "#": "Tirana" },
          {  "#": "Tirane" } ],
      "latitude": { "#": "41.33"  },
      "longitude": { "#": "19.82" },
      "elevation": { "#": "110" },
      "population": [  {
         "#": "418495",
         "@year": "2011",
         "@measured": "census"
         }, {
         "#": "398887",
         "@year": "2005",
         "@measured": "census"
         }
...
    ]
...
  }
....
]
```

Couchbase / N1QL : SELECT FROM WHERE

```
select c.name,
   c.`@area`,
   c.`@capital`,
   c.population
from `country` c
where c.name.`#`="France"
```

- `country`: bucket
- Caractères spéciaux : `…`
- c.name.`#` : on peut “traverser” les objets, mais pas les arrays (relation 1:N)
- **c.population** est un **array** !


```
[ { "@area": "547030", "name": { "#": "France"
}, "population": [ {   "#": "40502513",
"@measured": "census",   "@year": "1946" }, {
"#": "42777162",   "@measured": "census",
"@year": "1954" }, {   "#": "46520271",
"@measured": "census",   "@year": "1962" }, {
"#": "49778540",   "@measured": "census",
"@year": "1968" }, {   "#": "52591584",
"@measured": "census",   "@year": "1990" }, {
"#": "58518395",   "@measured": "census",
"@year": "1999" }, {   "#": "61795238",
"@measured": "est.",   "@year": "2007" }, {
"#": "64300821",   "@measured": "est.",
"@year": "2015" } ] }]
```

Couchbase / N1QL : ORDER BY

Requête simple:

```
select r.name, r.length, r.`@country`
from `river` r
order by tonumber(r.length.`#`) desc
limit 2
```

Bucket : `river`


```
[ {
  "@country": "CN",
  "length": {  "#": "6380"  },
  "name": {  "#": "Yangtze"  }
  },
  {
  "@country": "CN",
  "length": {  "#": "4845"  },
  "name": {  "#": "Hwangho"  }
  }
]
```

N1QL : JOIN

Joindre plusieurs buckets (il faut définir des index):

```
select r.`name` as river,
       c.`name` as country
from `river`r join `country`c
  on r.`@country` = c.`@car_code`
limit 2
```

```
[  {
  "country": {
  "#": "United States"  },
  "river": {  "#": "Missouri"  }  },
  {
  "country": {
  "#": "China"  },
  "river": {
   "#": "Hwangho"  }  }]
```

Couchbase / N1QL : UNNEST

Itérer sur les élements d'un array (1:N):

```
select c.name, c.`@area`,
   c.`@capital`, p.`#` as population
  from country c unnest c.population p
where c.name.`#`="France"
  and p.`@year`= "2015"
```

```
[ {  "@area": "547030",
"@capital": "cty-France-Paris",
"name": {  "#": "France"  },
"population": "64300821"  }]
```

- c.population est un **array**
- N1QL**: unnest** c.population p
- SQL3**: table**(c.population) p

Couhbase / N1QL : NEST

Regrouper les éléments dans un array (inverse de unnest)

```
select c.name, montagnes
from `country`c  nest `mountain` montagnes
                on c.`@car_code`=montagnes.`@country`
                and tonumber(montagnes.elevation.`#`)>8000
limit 2
```


```
[
{   "montagnes": [   {      "#": "",      "@country": "CN",      "@id": "mount-Shishapangma",      "elevation": {      "#": "8027"      },
"latitude": {      "#": "28.55"      },      "located": {      "#": ""      "@country": "CN",      "@province": "prov-China-29"      },
"longitude": {      "#": "85.8"      },      "mountains": {      "#": "Himalaya"      },      "name": {      "#": "Shishapangma"      }    }
],    "name": {    "#": "China"    } },
{   "montagnes": [   {      "#": ""      "@country": "NEP",      "@id": "mount-Manaslu",      "elevation": {      "#": "8163"      },
"latitude": {      "#": "28.55"      },      "longitude": {      "#": "84.6"      },      "mountains": {      "#": "Himalaya"      },
"name": {      "#": "Manaslu"      }    },    {      "#": "",      "@country": "NEP",      "@id": "mount-Dhaulagiri",      "elevation": {
"#": "8167"      },      "latitude": {      "#": "28.7"      },      "longitude": {      "#": "83.5"      },      "mountains": {      "#":
"Himalaya"      },      "name": {      "#": "Dhaulagiri"      }    },    {      "#": "",      "@country": "NEP",      "@id": "mount-
Annapurna",      "elevation": {      "#": "8091"      },      "latitude": {      "#": "28.6"      },      "longitude": {      "#": "83.8"      },
"mountains": {      "#": "Himalaya"      },      "name": {      "#": "Annapurna"      }    }  ],    "name": {    "#": "Nepal"    } },
{   "montagnes": [   {      "#": ""      "@country": "PK",      "@id": "mount-NangaParbat",      "elevation": {      "#": "8125"      },
"latitude": {      "#": "35.2"      },      "located": {      "#": ""      "@country": "PK",      "@province": "prov-PK-5"      },
"longitude": {      "#": "74.6"      },      "mountains": {      "#": "Himalaya"      },      "name": {      "#": "Nanga Parbat"      }    }
],    "name": {    "#": "Pakistan"    } }]
```

N1QL : NEST versus JOIN

```
select c.name, montagne
from `country`c join `mountain` montagne
        on c.`@car_code`=montagne.`@country`
        and tonumber(montagne.elevation.`#`)>8000
```


```
[ [
{  "montagne": {  "#": "",  "@country": "CN",  "@id": "mount-Shishapangma",  "elevation": {  "#": "8027"  },
"latitude": {  "#": "28.55"  },  "located": {  "#": "",  "@country": "CN",  "@province": "prov-China-29"  },
"longitude": {  "#": "85.8"  },  "mountains": {  "#": "Himalaya"  },  "name": {  "#": "Shishapangma"  }  },
"name": {  "#": "China"  } },
{  "montagne": {  "#": "",  "@country": "NEP",  "@id": "mount-Manaslu",  "elevation": {  "#": "8163"  },
"latitude": {  "#": "28.55"  },  "longitude": {  "#": "84.6"  },  "mountains": {  "#": "Himalaya"  },
"name": {  "#": "Manaslu"  }  },  "name": {  "#": "Nepal"  } },
{  "montagne": {  "#": "",  "@country": "NEP",  "@id": "mount-Dhaulagiri",  "elevation": {  "#": "8167"  },
"latitude": {  "#": "28.7"  },  "longitude": {  "#": "83.5"  },  "mountains": {  "#": "Himalaya"  },
"name": {  "#": "Dhaulagiri"  }  },  "name": {  "#": "Nepal"  } },
{  "montagne": {  "#": "",  "@country": "NEP",  "@id": "mount-Annapurna",  "elevation": {  "#": "8091"  },
"latitude": {  "#": "28.6"  },  "longitude": {  "#": "83.8"  },  "mountains": {  "#": "Himalaya"  },
"name": {  "#": "Annapurna"  }  },  "name": {  "#": "Nepal"  } },
{  "montagne": {  "#": "",  "@country": "PK",  "@id": "mount-NangaParbat",  "elevation": {  "#": "8125"  },
"latitude": {  "#": "35.2"  },  "located": {  "#": "",  "@country": "PK",  "@province": "prov-PK-5"  },
"longitude": {  "#": "74.6"  },  "mountains": {  "#": "Himalaya"  },  "name": {  "#": "Nanga Parbat"  }  },
"name": {  "#": "Pakistan"  } }]
```

JOIN

#	@country	@id	elevation	latitude	located			longitude	mountains	name	#
	CN	mount-Shishapangma	#	#	#	@country	@province	#	#	#	China
			8027	28.55		CN	prov-China-29	85.8	Himalaya	Shishapangma	
#	@country	@id	elevation	latitude	located			longitude	mountains	name	#
	NEP	mount-Manaslu	#	#				#	#	#	Nepal
			8163	28.55				84.6	Himalaya	Manaslu	
#	@country	@id	elevation	latitude	located			longitude	mountains	name	#
	NEP	mount-Dhaulagiri	#	#				#	#	#	Nepal
			8167	28.7				83.5	Himalaya	Dhaulagiri	
#	@country	@id	elevation	latitude	located			longitude	mountains	name	#
	NEP	mount-Annapurna	#	#				#	#	#	Nepal
			8091	28.6				83.8	Himalaya	Annapurna	
#	@country	@id	elevation	latitude	located			longitude	mountains	name	#
	PK	mount-NangaParbat	#	#	#	@country	@province	#	#	#	Pakistan
			8125	35.2		PK	prov-PK-5	74.6	Himalaya	Nanga Parbat	

NEST

#	@country	@id	elevation	latitude	located			longitude	mountains	name	#
	CN	mount-Shishapangma	#	#	#	@country	@province	#	#	#	China
			8027	28.55		CN	prov-China-29	85.8	Himalaya	Shishapangma	
#	@country	@id	elevation	latitude	located			longitude	mountains	name	#
	NEP	mount-Manaslu	#	#				#	#	#	Nepal
			8163	28.55				84.6	Himalaya	Manaslu	
	NEP	mount-Dhaulagiri	#	#				#	#	#	
			8167	28.7				83.5	Himalaya	Dhaulagiri	
	NEP	mount-Annapurna	#	#				#	#	#	
			8091	28.6				83.8	Himalaya	Annapurna	
#	@country	@id	elevation	latitude	located			longitude	mountains	name	#
	PK	mount-NangaParbat	#	#	#	@country	@province	#	#	#	Pakistan
			8125	35.2		PK	prov-PK-5	74.6	Himalaya	Nanga Parbat	

N1QL : value

```
select value r.`name`.`#`
from `river`r
where r.`@country` = “F")
```

```
[ "Saone", "Seine", "Loire",
"Marne", "Ill", "Isere",
"Durance"]
```

- **value** : retourne les valeurs des objets au lieu des objets
- **value**, **raw** et **element** sont des synonymes

```
select r.`name`.`#` as river
from `river`r
where r.`@country`
```

```
[ { "river": "Saone" },
  { "river": "Seine" },
  { "river": "Loire" },
  { "river": "Marne" },
  { "river": "Ill" },
  { "river": "Isere" },
  { "river": "Durance" } ]
```

Couchbase / N1QL

```
select r.`name`.`#` as river
from `river`r
where r.`@country` in (select value c.`@car_code`
                       from `country`c
                       where c.name.`#`="France")
```

```
[ { "river": "Saone" }, { "river": "Seine" },
  { "river": "Loire" }, { "river": "Marne" },
  { "river": "Ill" }, { "river": "Isere" },
  { "river": "Durance" } ]
```

N1QL : ANY / EVERY

```
select raw c.name.`#`
from `country`c
where any b in c.border
satisfies tonumber(b.`@length`) > 7000 end;
```

```
[  "United States"]
```

```
select raw c.name.`#`
from `country`c
where every b in c.border
satisfies tonumber(b.`@length`) < 70 end;
```

```
[  "Liechtenstein",
"Andorra",  "Gaza Strip"]
```

N1QL : ANY/EVERY

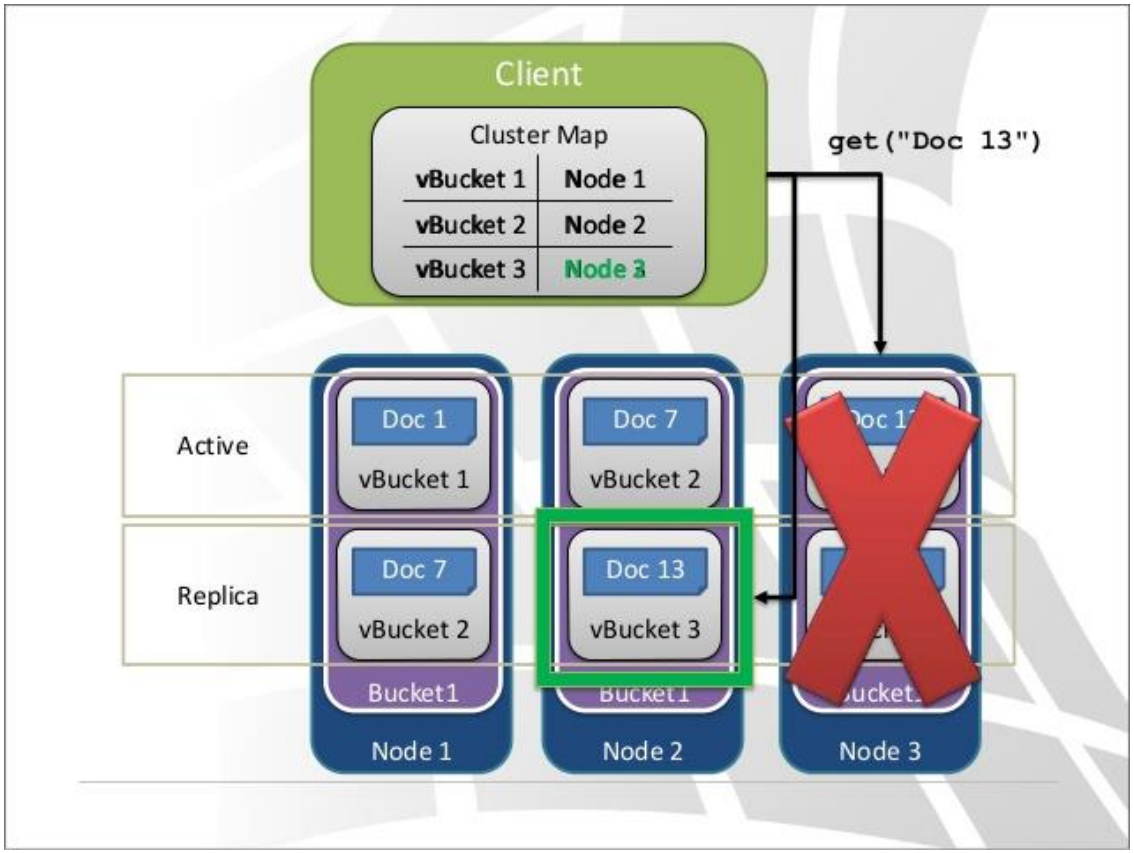
XQuery : ANY/ALL

Expressions N1QL

- Aggregate : AVG, COUNT, MAX, MIN, SUM, VARIANCE, …
  - Modes : ALL et DISINCT
- Arithmetic : +, -, *, /, %
- Collection : ANY, EVERY, EXISTS, IN, ARRAY, WITHIN
- Comparison : =/==, !=/<>, <, >, >=, <=, BETWEEN, LIKE, IS NULL, …
- Conditional : CASE WHEN … IS …
- Construction : { expr1, expr2, … }
- Functions : fonctions diverse
- Identifier : clés → valeurs
- Literal-value : string, number, Boolean, null, *missing*
- Logical : AND, OR, NOT
- Nested expression : expressions de chemin
- String : concat, contains, length, …
- Subquery : sous-requêtes

https://docs.couchbase.com/server/current/n1ql/n1ql-language-reference/index.html
http://docs.couchbase.com/files/Couchbase-N1QL-CheatSheet.pdf

Couchbase : distribution et réplication



noSQL **Document**

+ Modèle de données simple et puissant

+ Passage à l'échelle

+ Maintenance faible

+ Forte expressivité du langage de requêtes : requêtes complexes sur structures imbriquées.

+ Efficace pour les interrogations par clef mais

- Limité pour les interrogations par le contenu des documents,

- Limité aux données hiérarchiques (arbres) → graphes ???

noSQL **Colonne**

La base est une collection d'objets (lignes) avec des attributs (table universelle).

Les **lignes** sont fragmentées et stockées dans des tables avec des **colonnes.**

Chaque **colonne** est définie par un couple (***clef, valeur***).

Nom	Prénom	Parcours
Dupont	Max	UE1
Dupont	Max	UE2
Durand	Léa	UE2
Martin	Tom	UE2

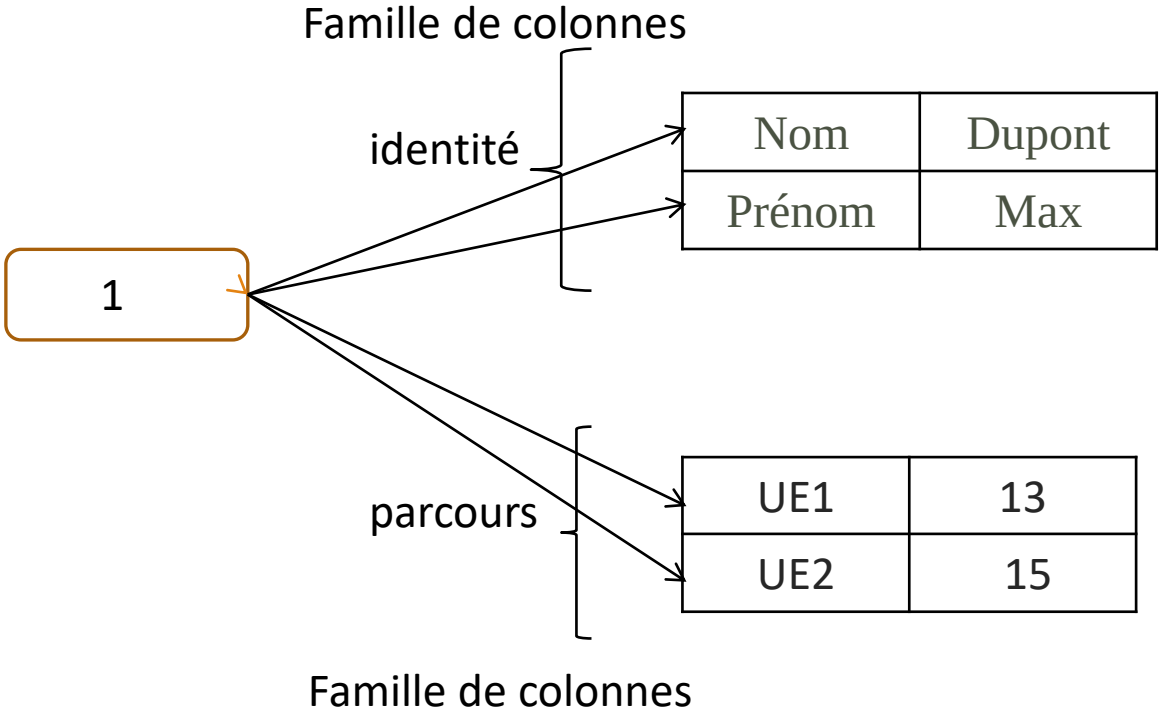
Clé	Nom
1	Dupont
2	Durand
3	Martin

Clé	Parcours
1	UE1, UE2
2	UE2
3	UE1

Clé	Prénom
1	Max
2	Léa
3	Tom

Ex : MonetDB, BigTable, Cassandra, Hadoop/Hbase…

Exemple de base colonne



Les colonnes peuvent être regroupées par objet en « *supercolonnes* » et en « ***famille de colonnes*** »

noSQL **Colonne**

+ Permet d'avoir un très grand nombre de valeurs sur une même ligne et de stocker des relations 1:N.

+ Ajout facile de colonnes, facilité de partitionnement, de compression, d'indexation.

+ Modèle efficace avec indexation sur les colonnes

+ Bien adapté à l'OLAP

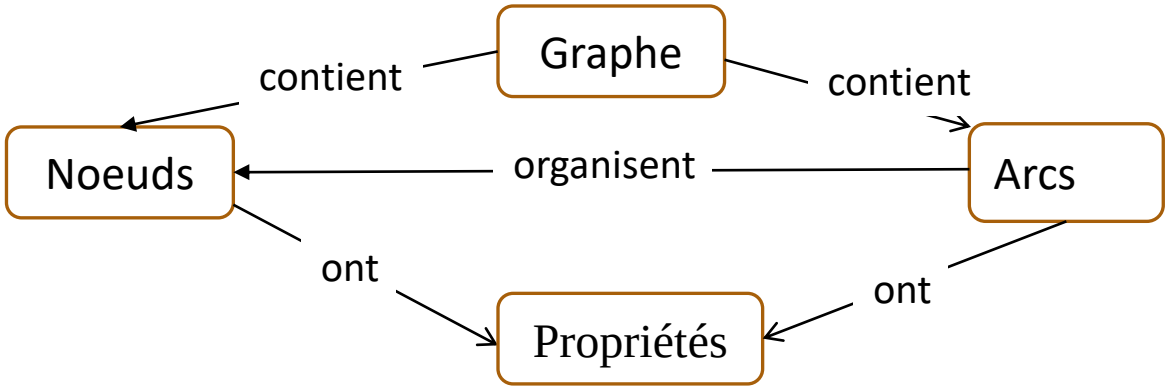
- Ne supporte pas bien les **données structurées complexes**

noSQL **Graphe**

La base est une **collection de graphes dirigés** composés de **nœuds** et **arcs**

Les nœuds et les arcs peuvent avoir des **propriétés** (*property graphs*)

Les arcs *organisent* les nœuds (associations).



Ex : **Neo4J**, Amazon Neptune, OrientDB, Jena (RDF), Cassandra, Apache Giraph

noSQL **Graphe : Exemples**

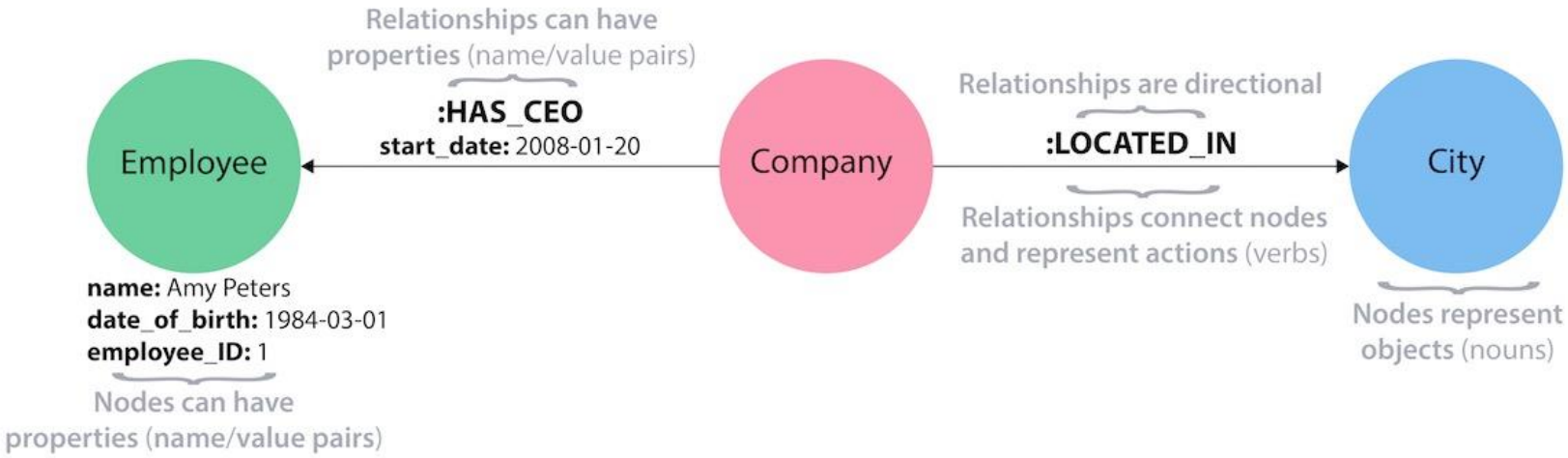
Graphes RDF :

- Nœuds : ressources RDF
- Arcs: propriétés RDF reliant deux ressources
- Propriétés : propriétés littéraux (uniquement nœuds)

Graphes Neo4J

- Nœuds: entités étiquetés (labels)
- Arcs: associations typés entre entités
- Propriétés : attributs clés-valeurs (nœuds et arcs)
- Exemple : CREATE (p:Person)-[:LIKES]->(t:Technology)

Exemple Neo4J



https://neo4j.com/developer/graph-database/

Langages de requêtes graph

Principe : **motifs de graphes**

SPARQL:

- Standard pour l'interrogation de graphes RDF
- { ?p a Person; :name 'Jennifer'; :LIKES ?g . ?g a :Technology; :type 'Graphs' . }

Cypher:

- Interrogation de property graphs
- (p:Person {name: "Jennifer"})-[rel:LIKES]->(g:Technology {type: "Graphs"})

openCypher

- projet open-source pour le développement du langage Cypher
- http://www.opencypher.org/
- Neo4J, SAP HANA, RedisGraph, Gremlin

noSQL **Graphe**

+ Bien adapté à la manipulation d'objets complexes organisés en réseaux : réseaux sociaux, cartographie, web sémantique…

+ Langages de requêtes expressives (motifs de graphes, récursion)

- Implémentation distribuée complexe (scalabilité)

NoSQL et MapReduce

noSQL: Assurer les propriétés A (availability) et P (partition tolerance)

Forte distribution des données avec des techniques de partitionnement (consistent hashing, sharding..)

Paradigme de **calcul parallèle adapté aux masses de données**

Map-reduce

- Distribution **des données et des calculs sur les nœuds d'un cluster**
- Défis: développer des **algorithmes distribués** qui évitent l'échange des données entre les nœuds du cluster (maximiser les traitements locaux)
- Conçu par Google, mais basé sur des modèles connus de programmation parallèle

Programme MapReduce

MAP & SHUFFFLE

Fonction « map » *F* :

- Transformation d'un couple x=(c,v) en un ou plusieurs couples y=(c1, v1)

Map*(F)*:

- Application de *F* à un *ensemble de couples* X={(c,v), …}
- Résultat: liste de couples Y={(c1, v1a), (c1, v1b), … (c2, v2a), …}
- Y est trié par les clé et **peut contenir plusieurs couples avec la même clé**

Shuffle:

- regrouper toutes les valeurs pour la même clé
- Y'={(c1, [v1a, v1b, …], (c2, [v2a, …], …}

REDUCE

Fonction *« reduce » R*:

- Réduction d'un couple (c1, [v1a, v1b, …]), *avec la même clé c1 en un seul couple* (c1, v1')

Reduce*(R)*:

- Reduce : Application de *R* à la liste Y'={(c1, [v1a, v1b, …], (c2, [v2a, …], …}
- Résultat une liste Z={(c1, v1'), (c2, v2'), …}
- **Z contient un seul couple par clé**

Exemple

Calcul du nombre d'occurrences des mots dans les documents du Web

Map :

- Chaque nœud **Map** applique la fonction *F* à ses données et produit pour chacun de ses documents la liste des couples (mots, occurrences)
- Y=[(cle1 : val1a), …(cle1: val1b), ... (clek : valk)]

Shuffle :

- Les listes Y reçues des différents nœuds Map sont regroupées et triées par la clé produire des listes Y'=[(cle1, {val1a, val1b, …}, …]
- Y'=((base, 1, 5, 2, 2,…), .., (données, 3, 4, 2,..), …, (modèle, 1, 5, 2, ..),..)

Reduce:

- Chaque nœud Reduce reçoit un sous-ensemble de Y' et applique la fonction *R*
- Sortie : liste des mots avec leur nombre d'occurrence total

Exemple

Calcul du nombre d'occurrences des mots dans les documents du Web

Fonction map F :

- Entrée : identifiant d'un document, contenu d'un document
- Sortie : liste des mots avec leur nombre d'occurrence
- Y = ((modèle : 1), (modèle : 5), (modèle : 2), (données : 3), … (base : 5))

Fonction reduce R :

- Entrée : Y
- Calcul pour chaque clé (mot) la somme des occurrences
- Sortie : liste des mots avec leur nombre d'occurrence total
- Z = ((modèle : 8), (données : 13), … (base : 8))

Exemple

Calcul du nombre d'occurrences des mots dans les documents du Web

Map :

- Chaque nœud Map applique la fonction *F* et produit pour chacun de ses documents la liste des couples (mots, occurrences)
- Y=((modèle : 2), …(données : 3), .. (base : 5))

Shuffle :

- Les listes Y reçues des différents nœuds Map sont regroupées et triées par mot pour produire des listes Y'=(mots, [occurrences])
- Y'=((base, 1, 5, 2, 2,…), .., (données, 3, 4, 2,..), …, (modèle, 1, 5, 2, ..),..)

Reduce:

- Chaque nœud Reduce reçoit un sous-ensemble de Y' et applique la fonction *R*
- Sortie : liste des mots avec leur nombre d'occurrence total

SQL vs NoSQL

Cohérence forte
- Logique : schémas → contraintes
- Physique : transactions ACID

Distribution des données
- Transactions distribuées

Ressources limitées
- Optimisation de requêtes

Langage standard : SQL

Cohérence faible
- Schémas → validation, inférence
- Cohérence « à terme » (CAP)

Distribution des traitements
- Traitements batch
- mapReduce

Ressources « illimitées »
- Passage à l'échelle horizontal

Langages spécialisés, API

Conclusion

4 caractéristiques de NoSQL :

- schéma flexible, API simple, relâchement des propriétés ACID et passage à l'échelle

4 catégories :

- Clef-valeur, document, colonnes, graphes

Les aspects à retenir :

- modèle, stockage, schéma de partitionnement, sémantique des transactions

SGBD de la nouvelle génération : non-relationnel, répartis, open-source, passant à l'échelle.

- Au départ (2009) : développer des SGBD modernes passant à l'échelle du Web.
- Développement important depuis 2009 : pas de schéma, support pour la réplication, API simples, éventuellement la cohérence (mais pas ACID)

Approche complémentaire de SQL et du relationnel (approfondissement en M2)

La suite

M1 S2 : SAM (Stockage et Accès aux Mégadonnées)

- Méthodes d'accès : index, arbres B+, hachage
- Optimisation de requêtes et algorithmes de jointures
- Conception et interrogation de BD réparties
- Transactions réparties
- SGBD parallèles

M2 S1 : BDLE (Bases de données à large échelle)

- Bases de données multidimensionnelles
- Map Reduce - Spark et Scala
- Interrogation de données semi-structurées en Spark – Optimisation
- Stockage à l'échelle du Web
- Grands graphes de données

M2SI : Linked Open Data et Apprentissage Symbolique

- Web sémantique : RDF, Sparql, OWL

Bibliographie

Un lien intéressant : http://nosql-database.org/ qui donne une liste classée de toutes les BD NoSQL existantes.

A. Foucret : Livre blanc sur NoSQL (http://www.smile.fr/Livres-blancs/Culture-du-web/NoSQL)

NoSQL Data Stores in Internet-scale Computing (tutoriel de Wei Tan, IBM, à la conférence ICWS) : http://researcher.watson.ibm.com/researcher/files/us-wtan/NoSQL-Tutorial-ICWS-2014-public-WeiTan.pdf