NoSQL - SGD distribués et HBase

I. Mougenot isabelle.mougenot@umontpellier.fr

Faculté des Sciences Université Montpellier

2015

UNIVERSITÉ DE MONTPELLIER

Les motivations autour du NoSQL (Not Only SQL)

Recouvre différentes initiatives qui se targuent d'être complémentaires aux modèles relationnels et relationnels objets

1. évolution du Web, projets sources de données ouvertes (Open Linked Data), impulsion Google, Facebook, Amazon, Twitter, ...
2. ↗ volume des données ↗ interconnexion des données
3. limites des bases de données relationnelles face à de nouveaux besoins :
   - flexibilité : schémas très ouverts : nombreuses entités et nombreuses associations entre ces entités
   - adaptabilité : évolutions très fréquentes des schémas
   - des milliers voire des millions d'usagers

UNIVERSITÉ DE MONTPELLIER

Quand passer par un système NoSQL ?

Alternatives au relationnel dans des cas de figure ciblés

- recours fréquent à de l'évolution de schémas
- entités munies de nombreuses caractéristiques souvent non renseignées
- de multiples associations avec des multiplicités 1..* aux extrémités
- des attributs organisés naturellement sous forme d'arborescences
- un flux transactionnel très élevé

NoSQL : se démarquer des SGBD relationnels

Critiques ouvertes

- prépondérance du schéma et poids fort mis sur la représentation du domaine d'intérêt : s'affranchir de schémas normalisés vus comme des sophistications inutiles au détriment de l'efficacité
- modèle transactionnel et propriétés ACID : proposer une alternative moins exigeante : CAP (comprenant BASE)
- passage à l'échelle ou scalabilité (scalability) par ajout de serveurs au niveau de l'architecture physique : diminuer le temps de réactivité lors de l'afflux de nouveaux usagers, de nouvelles transactions à servir
- systèmes distribués et mécanismes de tolérance aux pannes : fragmentation des schémas et réplication, médiateur, entrepôt de données . . .

UNIVERSITÉ DE MONTPELLIER

Passage à l'échelle ou scalabilité

Capacité de l'architecture à s'adapter à une montée en charge (nouveaux usagers, nouvelles transactions) sans besoin de refonte des applications

- scalabilité horizontale (scaling out) : ajouter des serveurs (noeuds) avec des mécanismes de répartition de charge $\Leftarrow$ NoSQL
- scalabilité verticale (scaling up) : rendre plus performant un serveur : ajout de processeurs (CPU), barrettes mémoire (RAM), disques secondaires, cartes réseaux ...

Scalabilité horizontale

Etablir une relation linéaire entre les ressources ajoutées et l'accroissement des performances

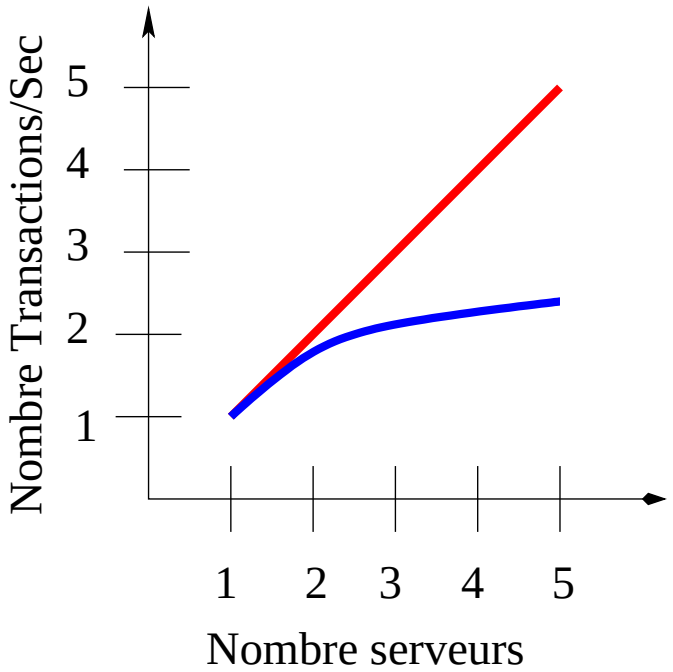


Figure: 1 serveur : 100 transactions/s ; 2 serveurs : 200 transactions/s . . .

SGBDR et besoins applicatifs à large échelle

Limites face aux besoins des applications à large échelle sur le Web (à partir Web 2.0)

- partionnement : les schémas fragmentés (fragmentations horizontale, verticale, hybride) distribués sur l'ensemble des partitions doivent être des fragments d'un seul schéma de données initial
- réplication sur différents noeuds : les fondements OLTP (On Line transactional processing) imposent de maintenir une intégrité forte sur les données, dans une application faisant appel à de nombreux noeuds, la disponibilité des données va être pénalisée (surtout si les transactions impliquent de nombreuses écritures).

systèmes NoSQL : grands principes

Pensés comme des systèmes de données distribués (distributed data stores)

- **Simplicité**
- **Flexibilité**
- **Efficacité**
- **Passage à l'échelle** : gros volumes de données distribués et interconnectés
  - partitionnement dynamique - sharding (partitionnement horizontal + plusieurs co-occurrences de schémas)
  - réplication à large échelle
  - architecture décentralisée

Complémentarité des systèmes NoSQL

One size doesn't fit all

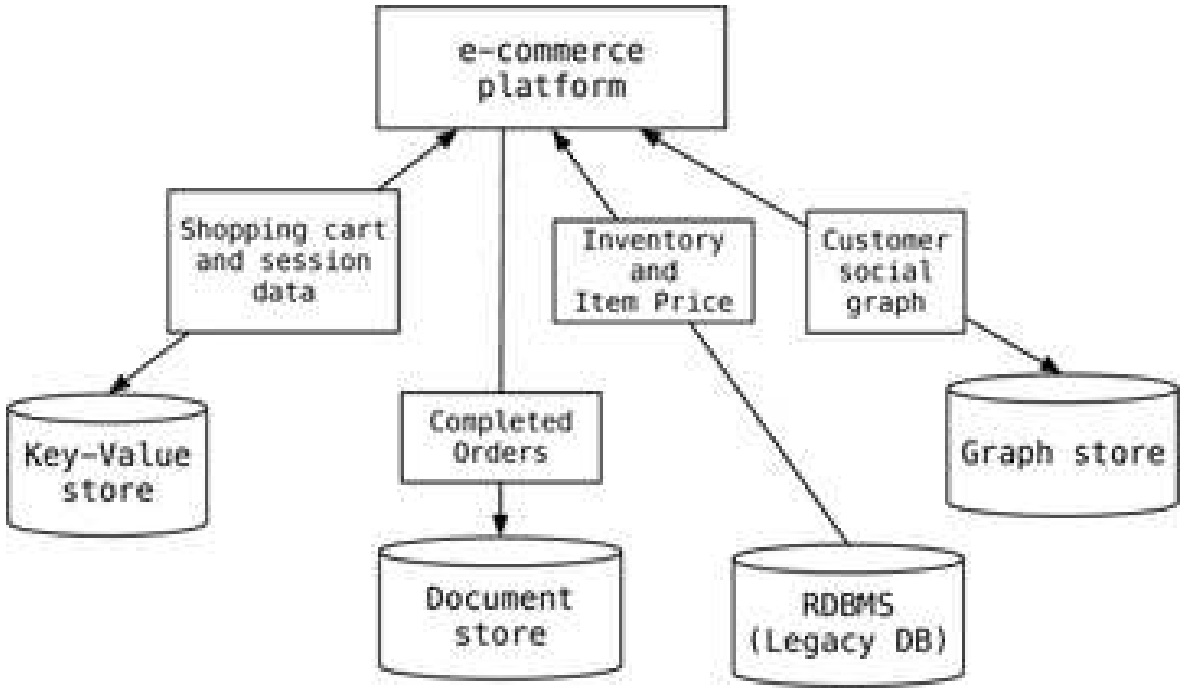


Figure: Persistance dite polyglotte (extrait de NoSQL distilled)

Théorème CAP

Constat de Brewer : aucun des systèmes distribués n'est à même de satisfaire en même temps les principes C, A et P :

1. **Consistency ou cohérence des données** : toute modification de donnée est suivie d'effet pour tous les noeuds du système
2. **Availability ou disponibilité des données** : toute requête émise et traitée par un noeud du système, reçoit une réponse (même en situation d'échec à produire une réponse)
3. **Partition tolerance ou recouvrement des noeuds** assurer une continuité du fonctionnement en cas d'ajout/suppression de noeud (ou partition) du système distribué

Un système distribué va satisfaire deux des trois points mais ne va pouvoir satisfaire les trois - Brewer. Towards robust distributed systems - ACM 2000

Théorème CAP

Considérations SGBDR / Systèmes NoSQL

1. **SGBDR** : Cohérence et haute disponibilité (pas ou peu de P, cad de différents noeuds système)
2. **Systèmes NoSQL** : Choix du P (système naturellement distribué) et sélection soit du C, soit du A
   1. abandon du A $\Leftarrow$ Accepte d'attendre que les données soient cohérentes
   2. abandon du C $\Leftarrow$ Accepte de recevoir des données parfois incohérentes

Positionnement des systèmes / CAP

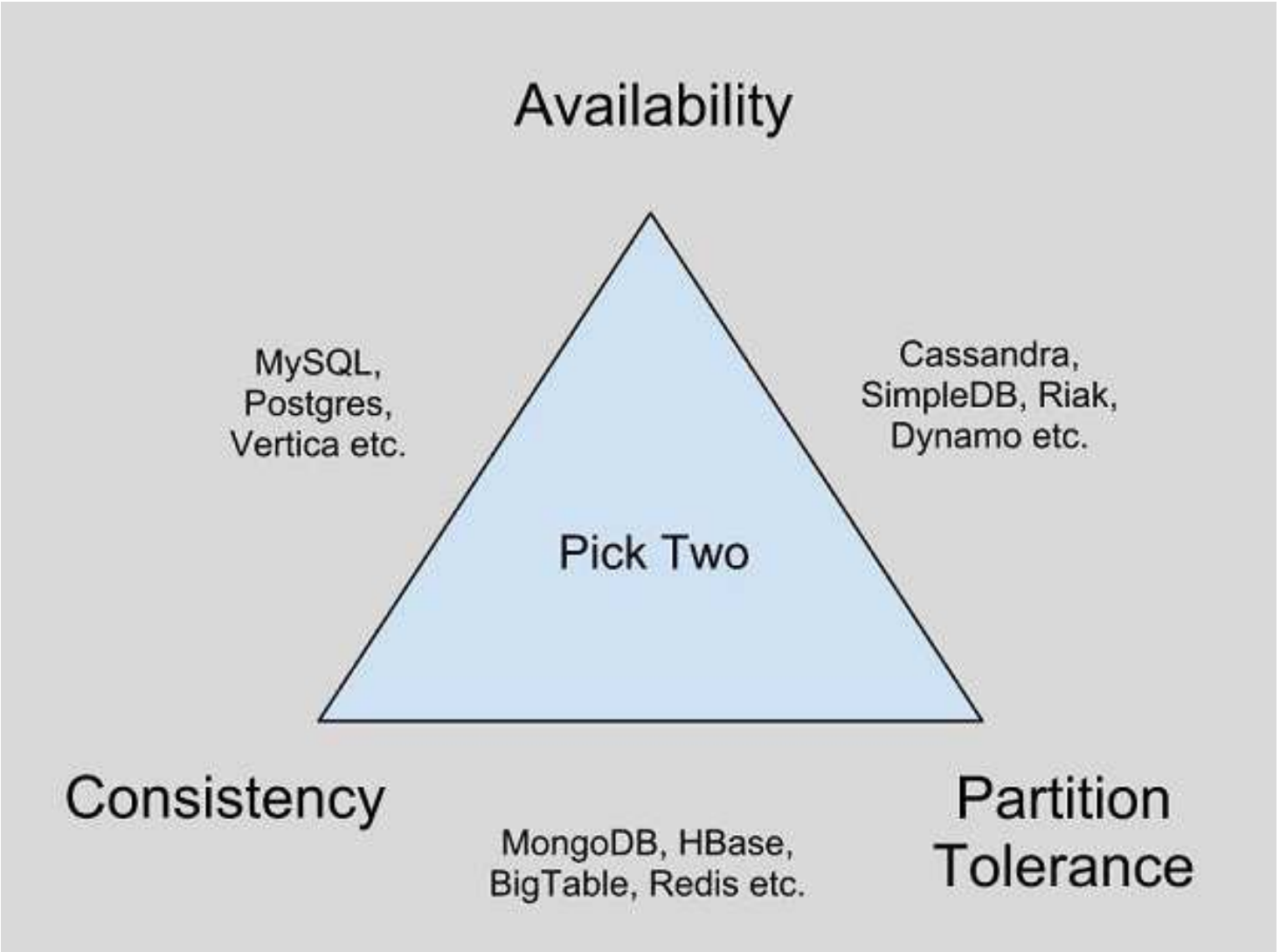


Figure: Synthèse CAP

Parti-pris Base (issu de CAP) versus propriétés ACID

BASE : **B**asically **A**vailable, **S**oft state, **E**ventual consistency

- **Modèle transactionnel** : les propriétés ACID (Atomique, Cohérent, Isolé, Durable) des transactions des SGBDRs ne sont pas complètement respectées au profit des performances et du passage à l'échelle
- **BASE** :
  - réplication et partitionnement horizontal/sharding pour aller vers de la haute disponibilité des données distribuées sur les différents noeuds du système (faire en sorte de diminuer l'impact de pannes éventuelles). Le résultat en est un système hautement disponible même si des sous-ensembles de données peuvent devenir indisponibles sur de de courtes périodes $\Leftarrow$ "disponible à court terme"

Parti-pris Base (issu de CAP) versus propriétés ACID

BASE : **B**asically **A**vailable, **S**oft state, **E**ventual consistency

- dans l'idée les systèmes NoSQL garantissent que les données deviennent cohérentes non pas en instantané mais au travers du temps. Les propriétés ACID nécessitent un verrouillage pessimiste et obligent à vérifier la cohérence des données à chaque fin de transaction. BASE propose une vision optimiste en reportant à plus tard la vérification de la cohérence de la base de données $\Leftarrow$ "cohérente à terme".
- L'état du système peut changer au travers du temps et cela sans nouvelle mise à jour en raison du modèle "cohérence à court terme" $\Leftarrow$ "Etat lâche"

Typologie des systèmes NoSQL

Au regard du mode de représentation choisi

- principe de base : clé/valeur
  - **Systèmes clé/valeur distribués**
  - **Systèmes orientés colonne**
  - **Systèmes orientés document**
- **Systèmes orientés graphe**
- dans un certaine mesure les triples stores et les SGBDOO

Illustration typologie NoSQL

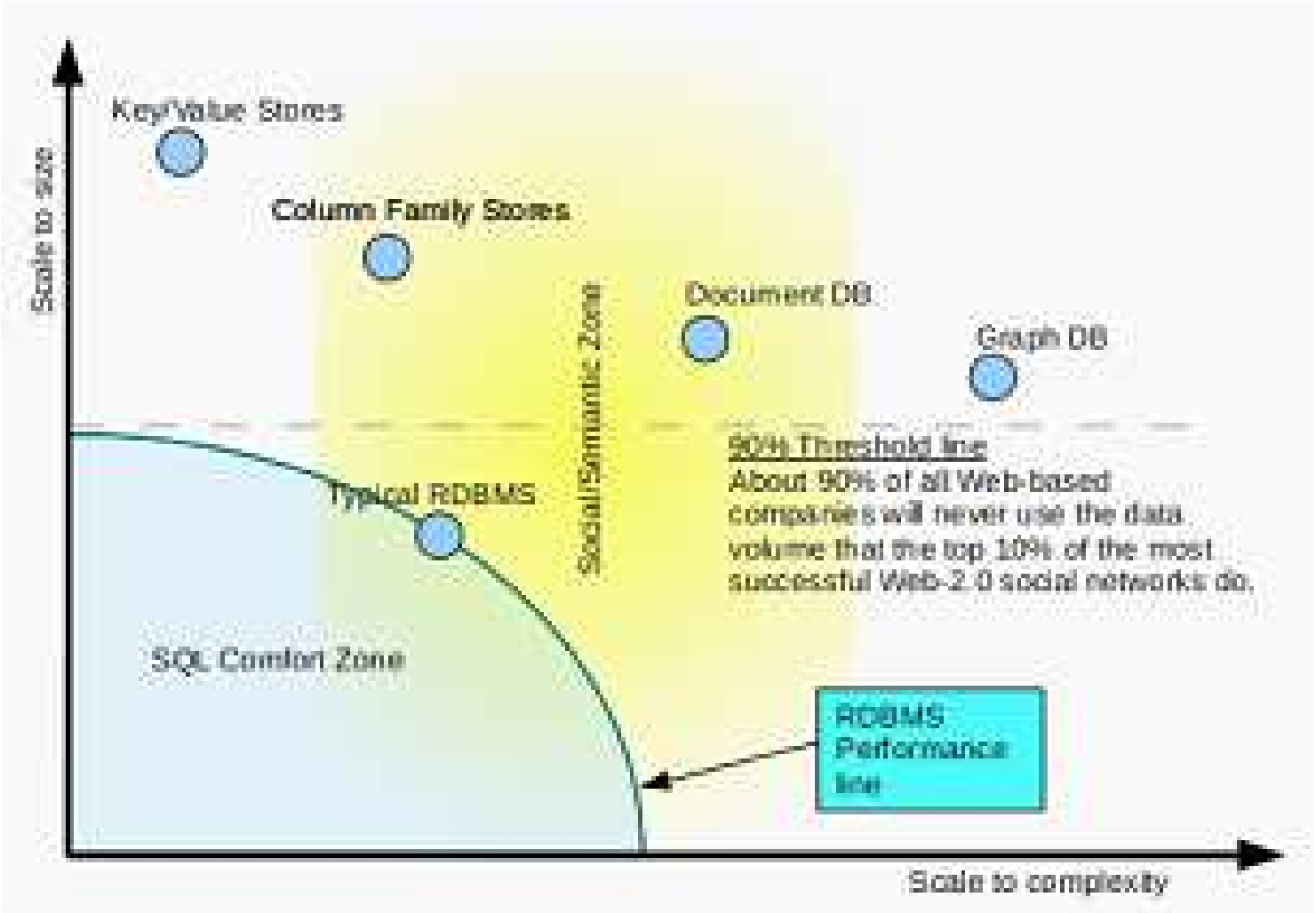


Figure:

Difficulté : absence de standards

Au regard du mode de représentation comme du système choisis

1. **APIs spécifiques**
2. **Terminologies propriétaires**
3. **Mécanismes de requêtage à géométrie variable**

1. **Systèmes ayant fait école (”proofs of concept”)**
   1. BigTable
   2. Memcached
   3. Amazon’s Dynamo

Systèmes existants

Table: Quelques systèmes et leurs modes de représentation

Name	Mode représentation	CAP
CouchDB	Document	AP
MongoDB	Document	CP
Neo4j	Graph	CA
GraphDB	Graph	unknown
Hbase	Column	CP
Memcachedb	Key-Value	unknown
Riak	Key-Value	CP
Project Voldemort	Key-Value	AP
Cassandra	Column	AP
Hypertable	Column	unknown

Systèmes existants

Table: Applications communautaires sur le Web

Name	Système NoSQL	Mode
Google	BigTable, LevelDB	Column
LinkedIn	Voldemort	Key-Value
Facebook	Cassandra	Column
Twitter	Hadoop/Hbase, Cassandra	Column
Netflix	SimpleDB, Hadoop/HBase, Cassandra	Column
CERN	CouchDB	Document
Amazon	Dynamo	Key-Value

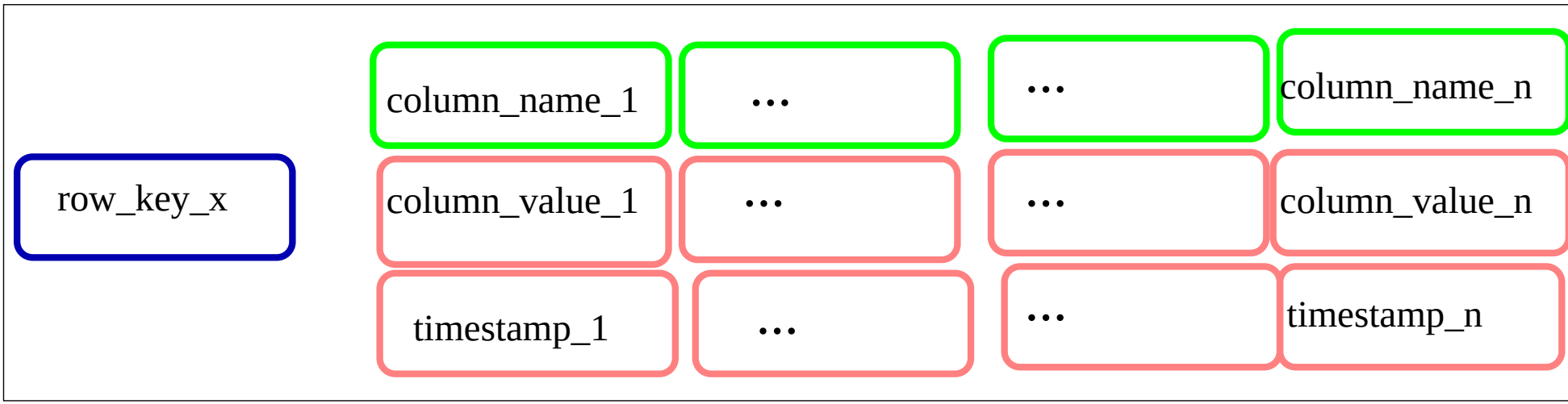
Systèmes NoSQL à colonnes

Confusion possible : le modèle physique est à colonnes mais pas le modèle de données

- **paradigmes** clé/valeur avec ”clé d’adressage” composite
- **Colonne** : triplet : adresse de colonne / valeur de colonne / estampille (gérer les versions et les conflits)
- **Famille de colonnes** : regrouper les colonnes qui sont partagées par un ensemble d’individus
- **Familles de super colonnes** : extension du modèle avec la notion de ”super colonnes” qui sont des collections de colonnes (poser des index à différents niveaux)

Systèmes NoSQL à colonnes

Illustration colonnes (qualifier)




```
row_key_x => {
                column_name_1: column_value_1,
                ...
                column_name_n: column_value_n,
                }
```

Figure: Vision générale

Systèmes NoSQL à colonnes

Illustration `colonnes` et "taille" variable d'un tuple

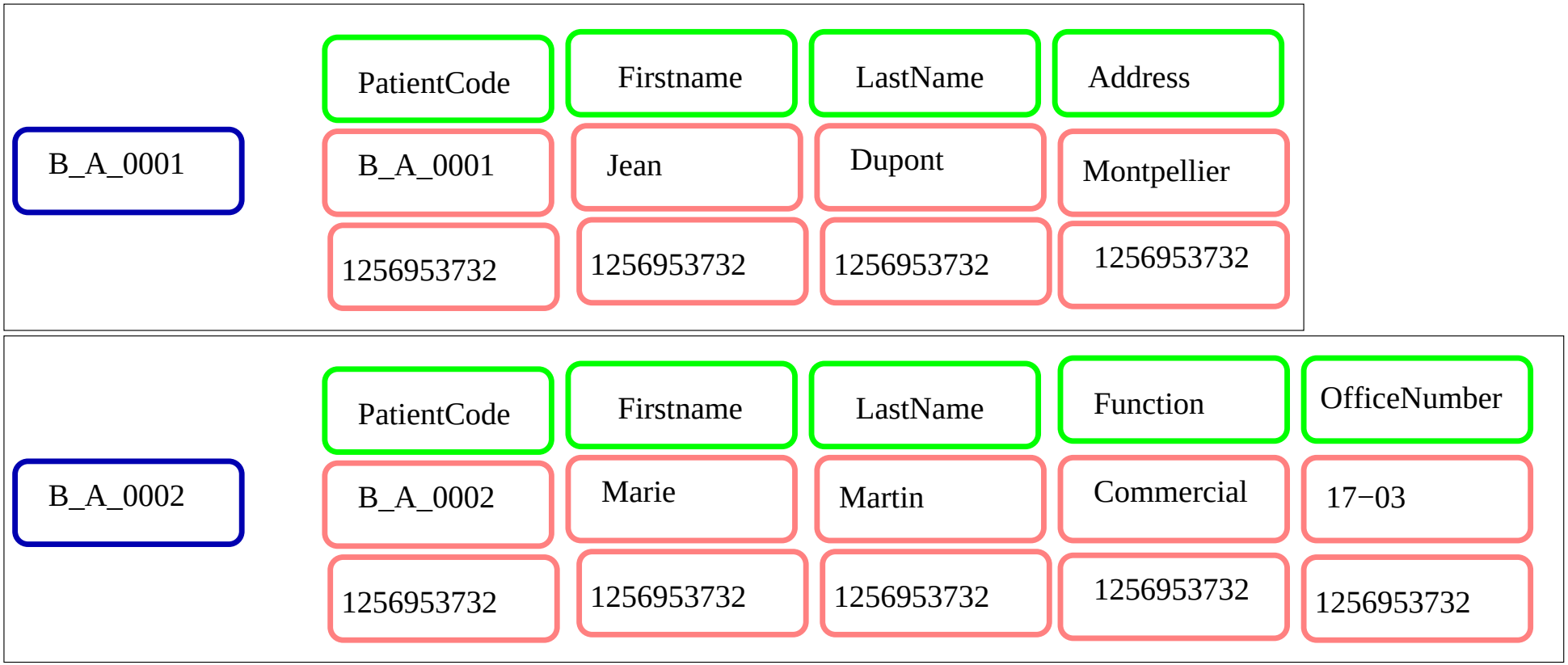


Figure: Exemple de deux tuples d'une famille

Systèmes NoSQL à colonnes

Ecriture inspirée de la notation JSON

```
B_A_0001 => {
Gal_Infos:PatientCode:"B_A_0001",
Gal_Infos:Firstname: "Jean",
Gal_Infos:LastName: "Dupont",
Gal_Infos:Address: "Montpellier",
Allergy_Infos:Sneezing: "mild",
Allergy_Infos:Itchy_Troat: "severe",
Allergy_Infos:Snuffy_Nose: "moderate",
Allergy_Infos:Watery_Eyes: "mild",
Allergy_Infos:Itchy_Nose: "severe"
}
```

Listing 1: Un tuple

Systèmes NoSQL à colonnes

Illustration famille de colonnes

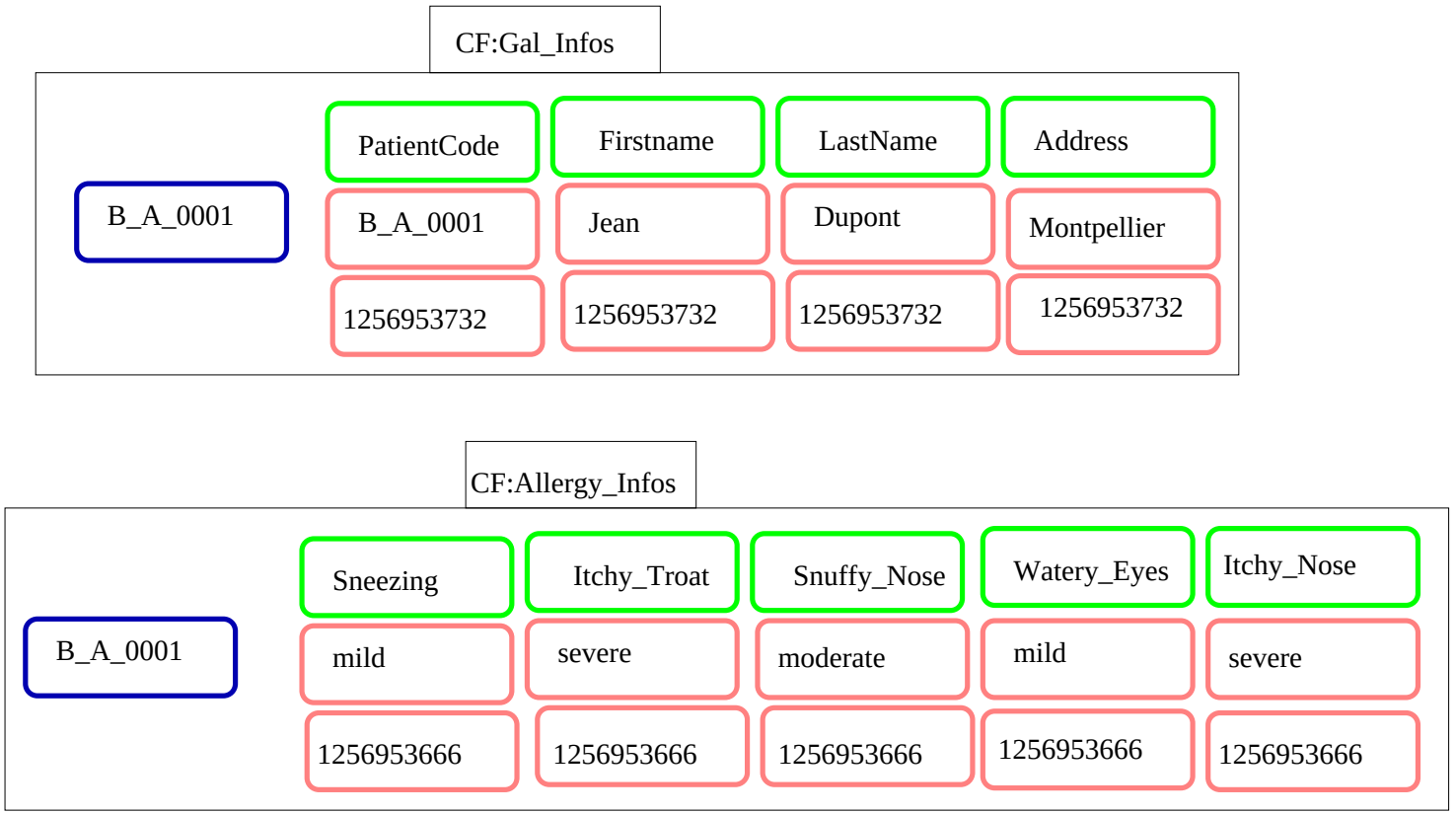


Figure: Exemple de deux familles de colonnes

NoSQL : systèmes

S'inspirent largement de Google BigTable(1)

- **Cassandra**
- **Hbase**

(1) BigTable - A distributed storage system for distributed data - Chang et al, 2006

HBAse : système ”column family”

Distribué, privilégie la cohérence et la disponibilité des données sans oublier les performances

S’appuie sur Hadoop (projet open source Apache) qui facilite le traitement distribué de larges jeux de données et ses composants

Hadoop Core =

- **HDFS pour le stockage**
  - **MasterServer** : namenode (mode master/slave)
  - **RegionServer** : datanode
- **Zookeeper** : infrastructure centralisée et services pour gérer un ”cluster” de serveurs : parmi les activités : synchronization, choix du serveur maître et vérification de la disponibilité des serveurs
- **MapReduce** : modèle de programmation distribuée

UNIVERSITÉ DE MONTPELLIER

Architecture de HBase

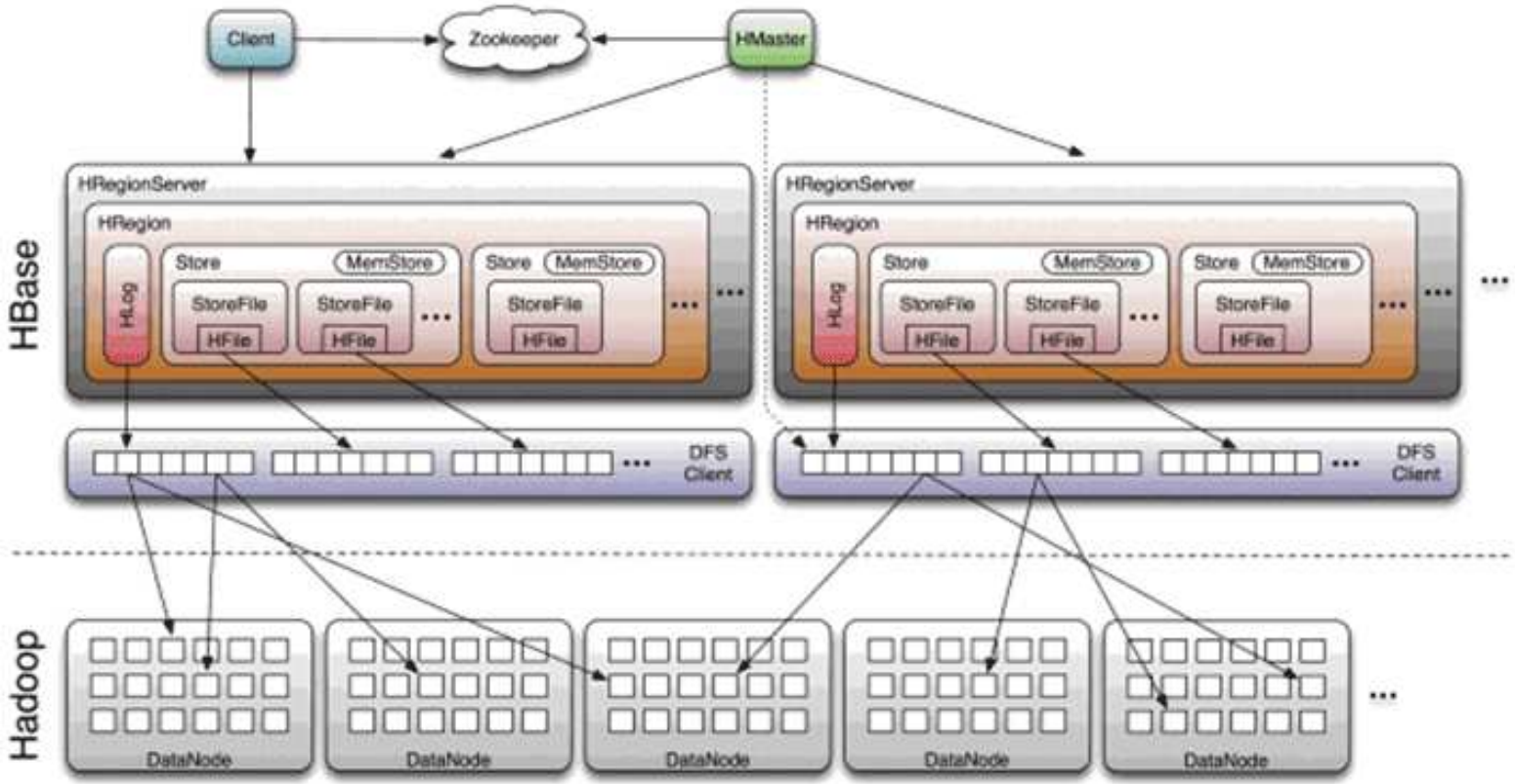


Figure: Vision générale

Structurellement parlant :

unité de base : table (keyspace) fragmentée en parties égales = régions (intervalles de valeurs de clés)

- tables triées sur la valeur de la clé
- familles : nombre quelconque de colonnes
- colonne (qualifier) : dont les valeurs peuvent être en nombre quelconque de versions (horodatage)

- (table, row, column family, column qualifier, timestamp) -> value (la valeur de la donnée est stockée avec l'ensemble de ses coordonnées)
- à noter : pas de super colonne avec HBase

Organisation de la donnée

```
SortedMap (
    RowKey, List (
        SortedMap (
            Column, List (
                Value, Timestamp
                )
            )
        )
    )
```

Figure: Organisation sous-jacente à tout tuple

Aspects internes à l'architecture de Hbase

Structures mises en jeu

1. familles de colonnes (CF) dont les colonnes sont stockées dans les mêmes fichiers bas niveau = HFile
2. une table est associée à une ou à plusieurs régions (partition de valeurs) selon les besoins en matière de place
3. HStore : une zone tampon par région associée à une table :
4. MemStore : une mémoire assignée au tri des tuples et à l'écriture séquentielle du flux de données dans les fichiers de données (HFile) - Sort & Flush
5. CachingBlock : tampon de données en mémoire vive
6. HFile : 1 à plusieurs fichiers de données par région
7. HLog : 1 fichier journal par RegionServer
8. Block : unité d'échange entre les mémoires vive et de masse (64 Ko à l'ordinaire)

Orchestration de différents composants

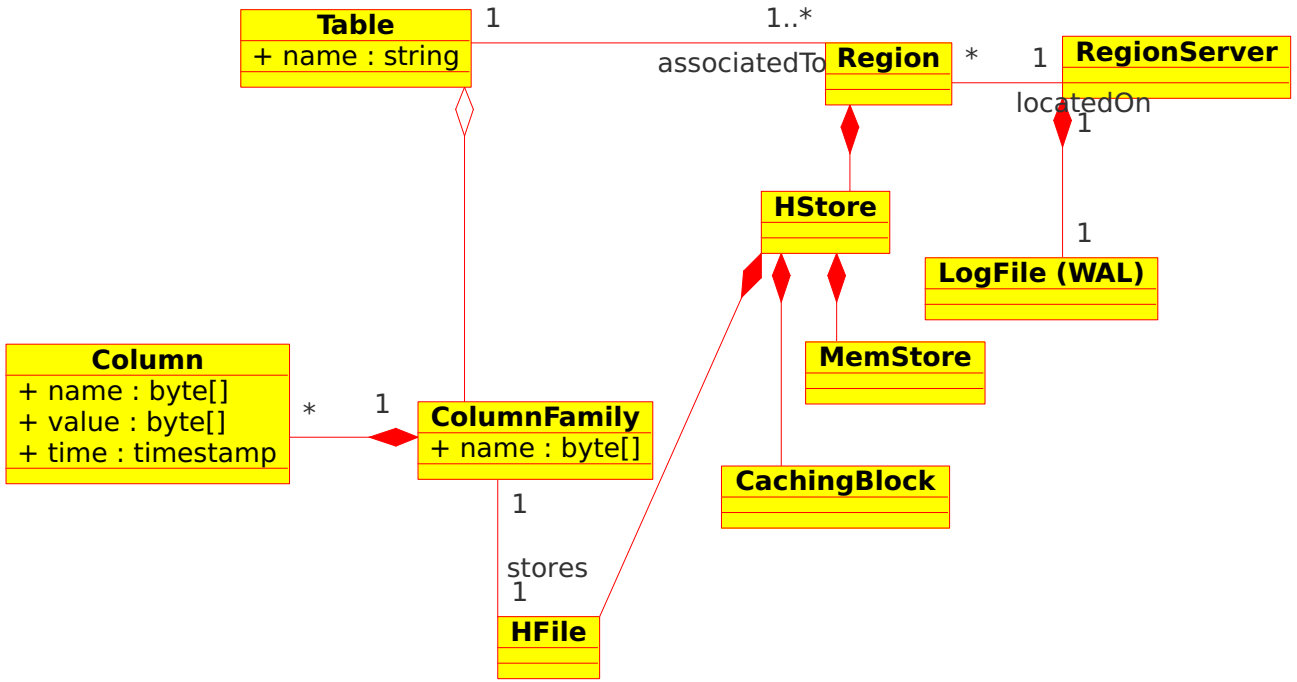


Figure: Diagramme de classes : structures internes

Orchestration de différents composants

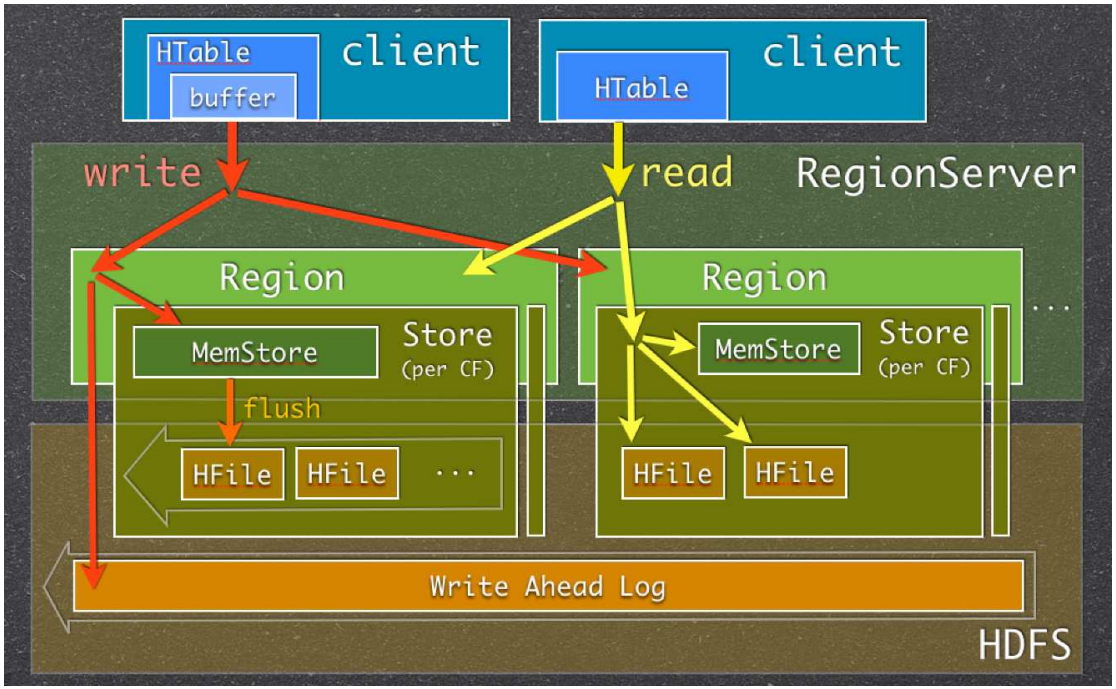


Figure: Extrait de HBase internals and schema design presentation

Mécanisme de reprise

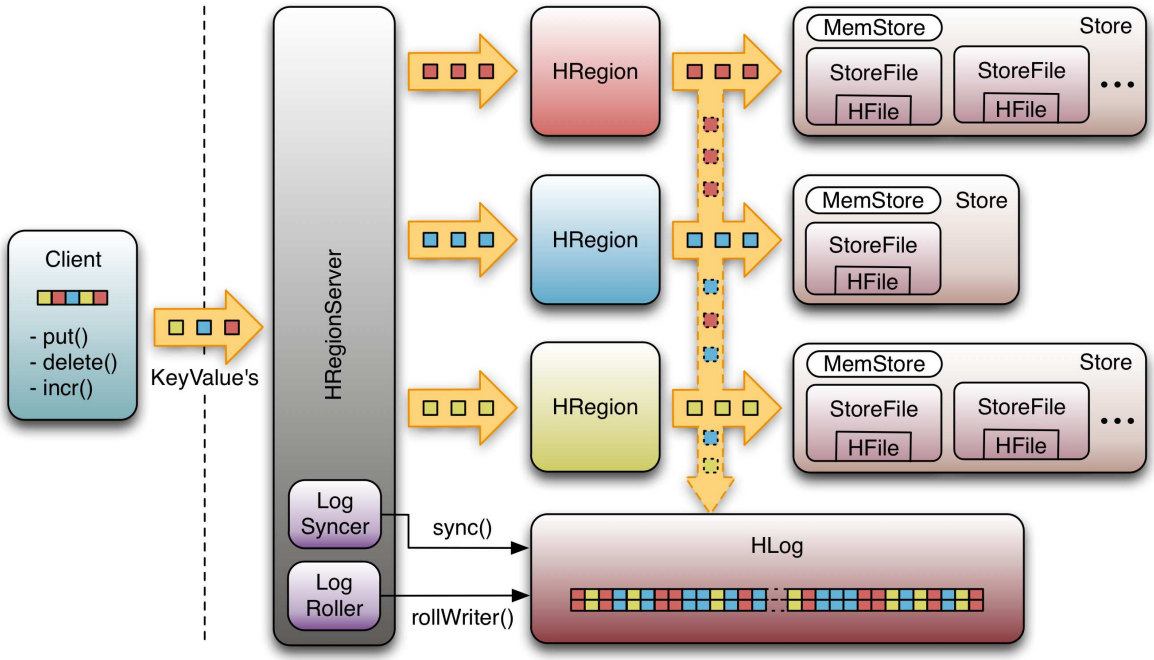


Figure: Ecriture dans fichier journal pour restauration éventuelle

Fichiers de données

Structure d'index : Log Structured Merge (LSM) Tree optimisée pour les accès séquentiels

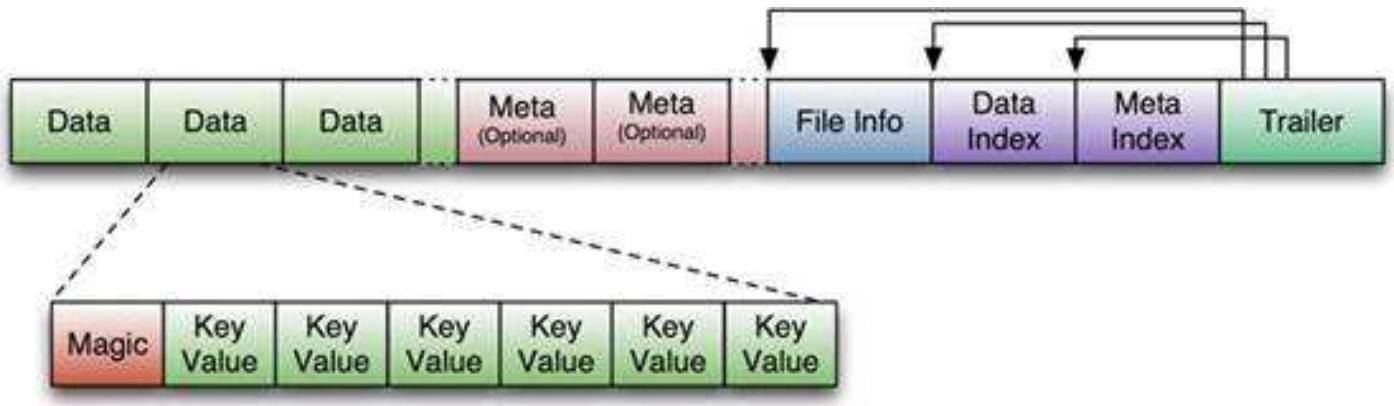


Figure: Ordonnancement des valeurs des colonnes sur la base de la clé du tuple

Page d'accueil (WebApp)

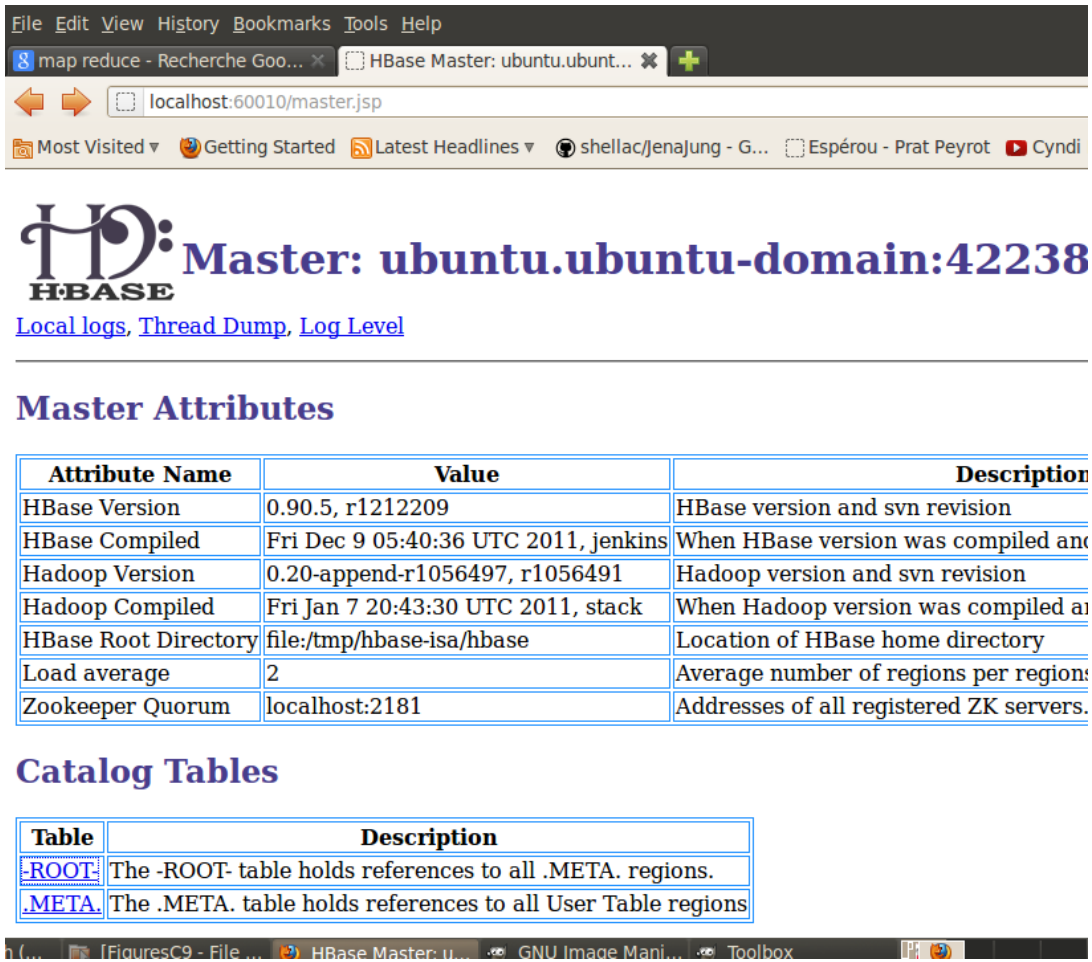


Figure: Tables du métaschéma : .META. (infos sur toutes les régions) et -ROOT- (localisation .META.)

Accès et traitement des données

Accès plus impératif que déclaratif

1. pas de langage DSL à l'exemple de SQL pour requêter les données
2. accès imperatif au travers d'API clientes : Java mais recours possible à d'autres langages JRuby, Clojure, Scala, Jython, ...
3. notion de coprocessor (procédure stockée) pour traiter directement les données au niveau d'un noeud de données (RegionServer)
4. complémentarité avec le framework MapReduce qui fournit des wrappers pour convertir les tables en collection de paires clé/valeur en entrée comme en sortie de diverses tâches d'analyse

En pratique

Clients HBase

- API Java
- Shell HBase (JRuby)
- Clients non-java
  - serveurs Thrifts (Ruby, C++, Erlang, . . .)
  - serveurs Rest (Stargate)

Construire une table et deux familles de colonnes avec JRuby

```
create 'ville', 'iG', 'ip10'
put 'ville', '01024', 'iG:codeInsee', '01024'
put 'ville', '01024', 'iG:nom', 'Attignat'
put 'ville', '01024', 'iG:popMun','2850'
put 'ville', '34172', 'iG:codeInsee', '34172'
put 'ville', '34172', 'iG:nom', 'Montpellier'
put 'ville', '34172', 'iG:popMun', '255080'
put 'ville', '34172', 'ip10:nbreRedevables', '1913'
--
put 'ville', '34172', 'ip10:nbreRedevables', '1914'
delete 'ville', '34172', 'ip10:nbreRedevables'

scan 'ville'

_________
ROW                 COLUMN+CELL
 01024  column=iG:codeInsee, timestamp=1354564480805,
     value=01024
 01024  column=iG:nom, timestamp=1354564480902, value=Attignat
```

Exemples API Java

Opérations élémentaires

1. opérations sur une table : create, scan, disable, drop
2. opérations sur un tuple : put, delete, get
3. notions de filtre à partir du parcours des tuples d'une table

Création - Insertion (create, put)

```
Configuration hc = HBaseConfiguration.create();
HTableDescriptor ht = new HTableDescriptor( "patient" );
ht.addFamily( new HColumnDescriptor( "allergy" ) );
Put pierrePut = new Put(new String("P_M_001").getBytes());
pierrePut.add(new String("allergy").getBytes(),
new String("sneezing").getBytes(),
new String("mild").getBytes());

HBaseAdmin hba = new HBaseAdmin( hc );
System.out.println( "creating table...patient " );
hba.createTable( ht );
HTable table = new HTable(hc, "patient");
System.out.println( "creating row...Pierre " );
table.put(pierrePut);
```

Listing 3: table Patient et un tuple Pierre

Affichage du contenu d'une table (scan)

```
 Configuration conf = HBaseConfiguration.create();
  HTable table = new HTable(conf, "personne");
  byte[] general = Bytes.toBytes("general");
  byte[] nom = Bytes.toBytes("nom");

  Scan scan = new Scan();
  scan.addColumn(general, nom);
  ResultScanner scanner = table.getScanner(scan);
      for (Result result = scanner.next(); (result != null);
          result = scanner.next()) {
      for(KeyValue keyValue : result.list()) {
      System.out.println("Qualifier : " +
          keyValue.getKeyString() +
//System.out.println("Qualifier : " +
   Bytes.toString(keyValue.getQualifier()) +
       " : Value : " + Bytes.toString(keyValue.getValue()));
     } }}}
```

Listing 4: Scan sur colonne

Filtre : patients âgés d'au moins 26 ans

```
Configuration conf = HBaseConfiguration.create();
HTable table = new HTable(conf, "patient");
List<Filter> filters = new ArrayList<Filter>();
Filter famFilter = new FamilyFilter(CompareFilter.
CompareOp.EQUAL,
           new BinaryComparator(Bytes.toBytes("iG")));
filters.add(famFilter);
Filter colFilter = new QualifierFilter(
CompareFilter.CompareOp.EQUAL,
    new BinaryComparator(Bytes.toBytes("age")));
filters.add(colFilter);
Filter valFilter = new ValueFilter(CompareFilter.
CompareOp.GREATER_OR_EQUAL,
           new BinaryComparator(Bytes.toBytes("26")));
filters.add(valFilter);
```

Listing 5: Exemple de filtre

Filtre Suite

```
FilterList fl = new FilterList( FilterList.Operator.
MUST_PASS_ALL, filters);
Scan scan = new Scan();
scan.setFilter(fl);
ResultScanner scanner = table.getScanner(scan);
   System.out.println("Scanning table... ");
   for (Result result : scanner) {
      for (KeyValue kv : result.raw()) {
System.out.println("kv:"+kv +", Key: " +
Bytes.toString(kv.getRow())
+ ", Value: " +Bytes.toString(kv.getValue()));
      }
   }
scanner.close();
```

Listing 6: Exemple de filtre

Notion de ”Coprocessor”

Analogie soit avec les déclencheurs, soit avec les procédures stockées en relationnel

Coprocesseur : fonctionnalités définies par l'usager pour étendre ou surcharger les fonctionnalités inhérentes au système

- Observer (triggers) : au niveau DML (ex. prePut/postPut), DDL (ex. avant ou après création d'une table), journalisation
- EndPoint (procédures stockées en relationnel) : ex. fonctions de calcul liées à un regroupement (AggregationClient) sur une colonne ou sur un tuple : count, min, max, sum, avg . . .

Rendre AggregationClient opérationnel

Deux façons de faire

```
Ajout dans le fichier hbase-site.xml :
<property>
<name>hbase.coprocessor.user.region.classes</name>
<value>
org.apache.hadoop.hbase.coprocessor.AggregateImplementation
</value>
</property>

Modification de la table dans le shell HBase
1. hbase> disable 'Commune'
2. ajouter le programme
hbase> alter 'Commune', METHOD => 'table_att',
'coprocessor'=>'
|org.apache.hadoop.hbase.coprocessor.AggregateImplementation||'
3. hbase> enable 'Commune'
```

Listing 7: AggregationClient

Compter le nombre de communes avec AggregationClient

```
public class AgregationCommunes {
  private static final byte[] TABLE_NAME =
      Bytes.toBytes("CommuneJ");
  private static final byte[] General =
      Bytes.toBytes("general");
  public static void main(String[] args) throws Throwable {
  Configuration configuration = HBaseConfiguration.create();
  AggregationClient aggregationClient = new AggregationClient(
           configuration);
  Scan scan = new Scan();
  scan.addFamily(General);
  long rowCount = aggregationClient.rowCount(TABLE_NAME,
      null, scan);
  System.out.println("row count is " + rowCount);
  }}
```

Listing 8: Utiliser AggregationClient

Compter le nombre de communes sans AggregationClient

Ecrire votre propre code au risque d'une inefficacité du calcul

```
public static void getCount (String tableName, Configuration
    conf) {
      try{
          HTable table = new HTable(conf, tableName);
          Scan s = new Scan();
          ResultScanner ss = table.getScanner(s);
          int Compteur = 0;
         for(Result r:ss){ Compteur++; }
               System.out.print("nombre de tuples "+Compteur);
      } catch (IOException e){
          e.printStackTrace();
      }
   }
```

Listing 9: Dans la classe ParcoursScanCommune

Attributs multivalués

Avec JRuby

```
put 'CommuneJ', '34392', 'population:2015', 110000
put 'CommuneJ', '34392', 'population:2015', 111000
put 'CommuneJ', '34392', 'population:2015', 112000

get 'CommuneJ', '34392',
    {COLUMN=>'population:2015',VERSIONS=>3}
```

Listing 10: Dans la classe ParcoursScanCommune

Attributs multivalués

Avec Java

```
Configuration hc = HBaseConfiguration.create();
    HTable tabl = new HTable(hc,"CommuneJ");
    Get get = new Get(Bytes.toBytes("34392"));
    get.addFamily(Bytes.toBytes("population"));
    get.setMaxVersions(Integer.MAX_VALUE);
    Result result = tabl.get(get);
    NavigableMap<byte[],NavigableMap<byte[],
NavigableMap<Long,byte[]>>> map =
    result.getMap();
.... parcourir ensuite les valeurs
```

Listing 11: Dans la classe ParcoursScanCommune

Combiner l'information provenant de plusieurs tables

Par exemple relevant d'une jointure

- **Programmation Map/Reduce** des classes facilitatrices de l'utilisation de plusieurs tables (côté Map) à l'exemple de `MultiTableInputFormat` (à partir de HBase 0.94.16))
- **Implanter les algorithmes de jointure** boucles imbriquées, tri-fusion, jointure par hachage
- **Autres idées ?**

Boucles imbriquées : le plus simple

L'intuition : parcourir les valeurs (clé ou d'une colonne d'un tuple) et les comparer

```
     while (rset1.next())
{
   System.out.print("value "+rset1.get(colX));

   while (rset2.next())
   {
      if (rset1.get(colX)==(rset2.get(colY)))
      {
         // action
      }
   }
}
```

Listing 12: Exemple Boucles

Charger des données depuis des fichiers tabulés

Plusieurs choix possibles

- **Programmation Map/Reduce** lorsque gros volumes de données
- **Migration depuis une BDR** Exploiter JDBC pour insérer des tuples dans une table HBase (exemple donné)
- **Définir un parseur** (exemple extrait de la doc HBase)

Synthèse accès et manipulation des données

Les mêmes actions qu'avec une BD traditionnelle

- **Sélection, Projection** avec scan, filter
- **Jointure** avec MultiTableInputFormat, réécriture des opérateurs de jointure en s'appuyant sur les algorithmes existants
- **Fonctions de calcul** : notion de Coprocessor