HMIN313 : Systèmes à colonnes et HBAse

I. Mougenot isabelle.mougenot@umontpellier.fr

FDS - Université Montpellier

Semestre 1 2018

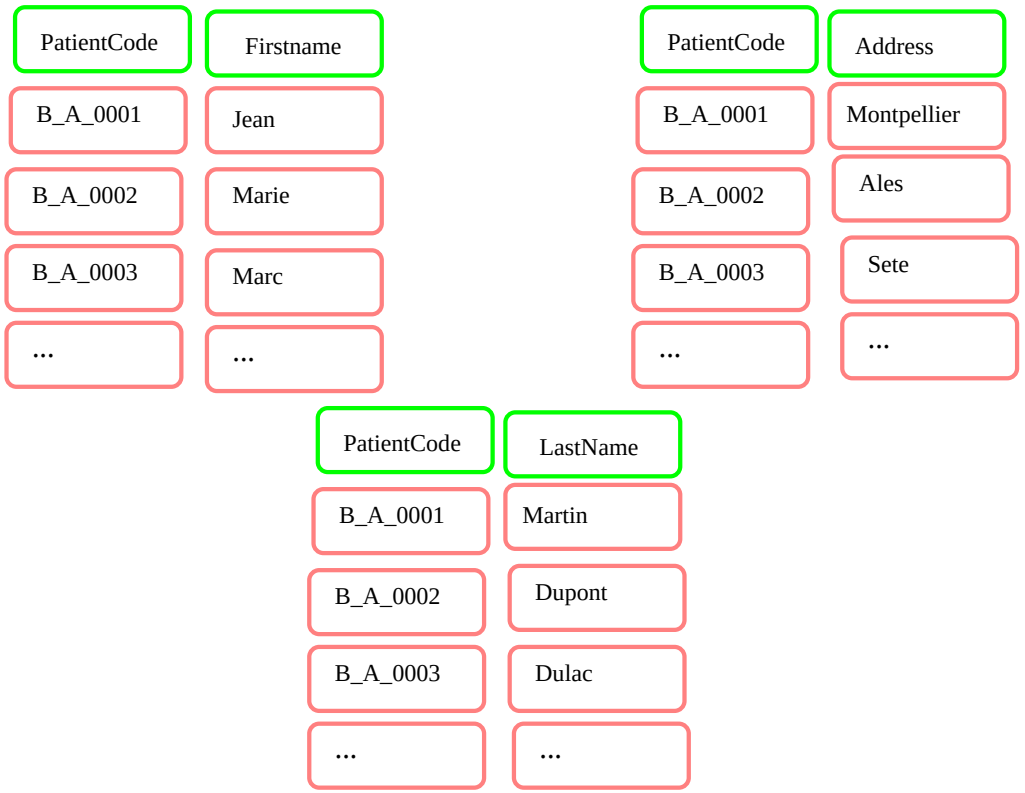
Colonnes : quelques principes pour dépasser les limites du relationnel

Données complexes, multivaluées et éparses

- modèle orienté colonne : DSM (Decomposition Storage Model), partitionnement vertical
- modèle NF2 Non First Normal Form : s'affranchir de la contrainte de la première forme normale et gérer des données multi-valuées (collections) et composites (types complexes)

Systèmes orientés colonne : partitionnement vertical

∼ "schema Aware" des triple stores



Colonnes : quelques exemples de systèmes

Decomposition Storage Model ; SBGDR sous-jacents (usage de vues matérialisées)

- Académiques
  - C-Store (MIT Cambridge MA) : depuis 2005
  - MonetDB (CWI Amsterdam) : pionnier en la matière
- Commerciaux
  - Vertica (C-Store)
  - Sybase IQ
  - InfoBright MySQL

Domaines d'application phare pour les systèmes orientés colonnes

Exploités ou source d'inspiration pour :

- Entrepôts de données
- Fouille de données
- Jeux de données scientifiques : santé, écologie, astrophysique, génomique fonctionnelle ...
- Modèles RDF (triple stores)
- **Google Big Table** pour le modèle physique

Modèle NF2 : inspiration TAD (SQL3)

Données composites et multi-valuées (collection)

PatientCode	Firstname	LastName	DOB	Symptoms		
				name	value	date
B_A_0001	Jean	Martin	5/5/55	sneezing	severe	3/3/3
	Ernest			itchy_troat	severe	3/3/3
B_A_0002	Marie	Dupont	6/6/66	...		
B_A_0003	Marc	Dulac	7/7/77	...		
	Antoine					
...	...	...	...	...		

Figure: Illustration modèle NF2

Modèle NF2 : SGBD Objet-Relationnel

Exemples de types de données abstraits avec Oracle

```
Create type Tsymptom as object
      (name varchar(15), value varchar(10)
      diag_date date, )
/
Create type coll_symptoms as Table of Tsymptom
/
Create table Patient (code varchar(8),
   firstname coll_fsn, lastname varchar(20),
   symptoms coll_symptoms);
Nested table symptoms Store As allSymptoms
```

Listing 1: one row

NoSQL : systèmes orientés colonnes (wide-column store)

S'inspirent largement de Google BigTable(1) et se démarquent de la sphère relationnelle

- **Cassandra (Apache)**
- **Hbase (Apache)**
- **Google Cloud Bigtable**

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

HBAse : brique de l'écosystème Hadoop

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 volumes de données

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

Ecosystème Hadoop

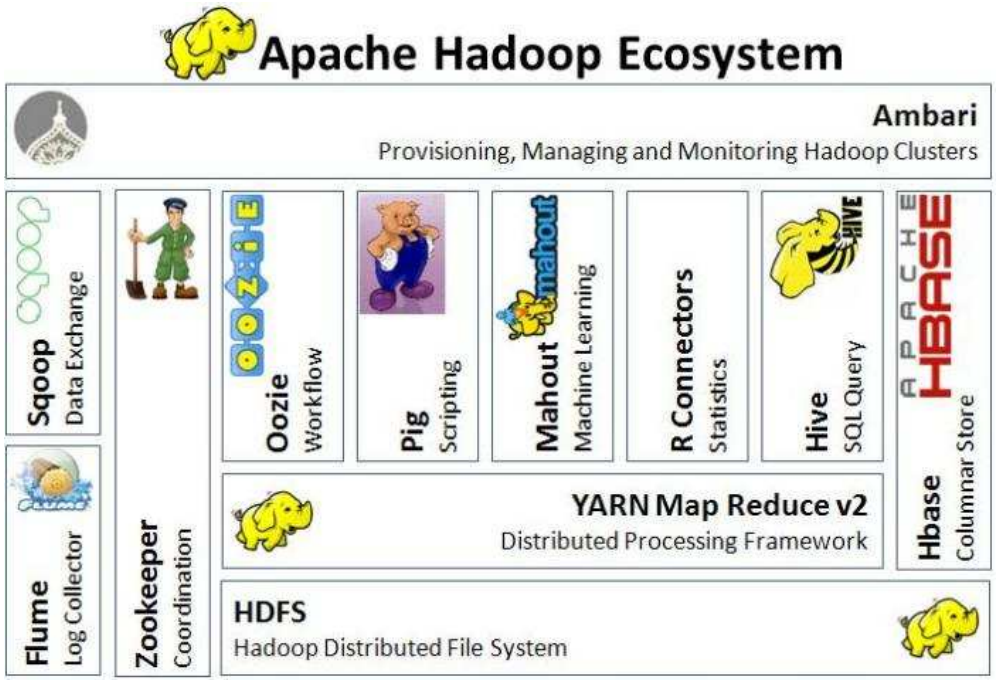


Figure: Orienté BigData

Architecture de HBase

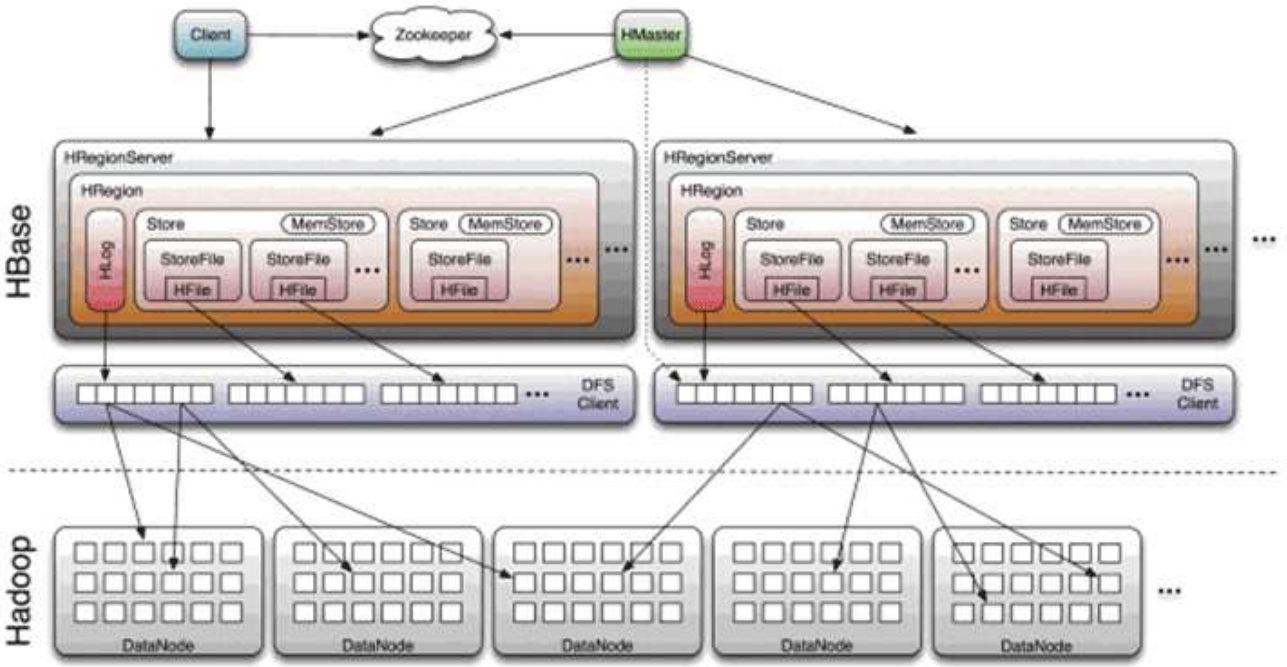


Figure: Vision générale

Structurellement parlant :

unité de base : table contenant des tuples clé/valeur

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

- à noter : super colonne absente dans HBase

Un tuple visuellement parlant

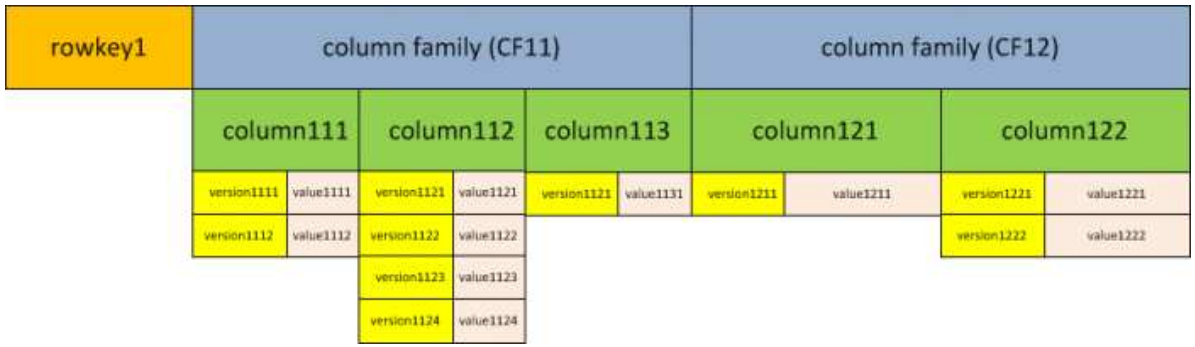


Figure: Pas de valeur "null", colonnes multi-valuées, colonnes éparses

Systèmes NoSQL à colonnes

Illustration Column family

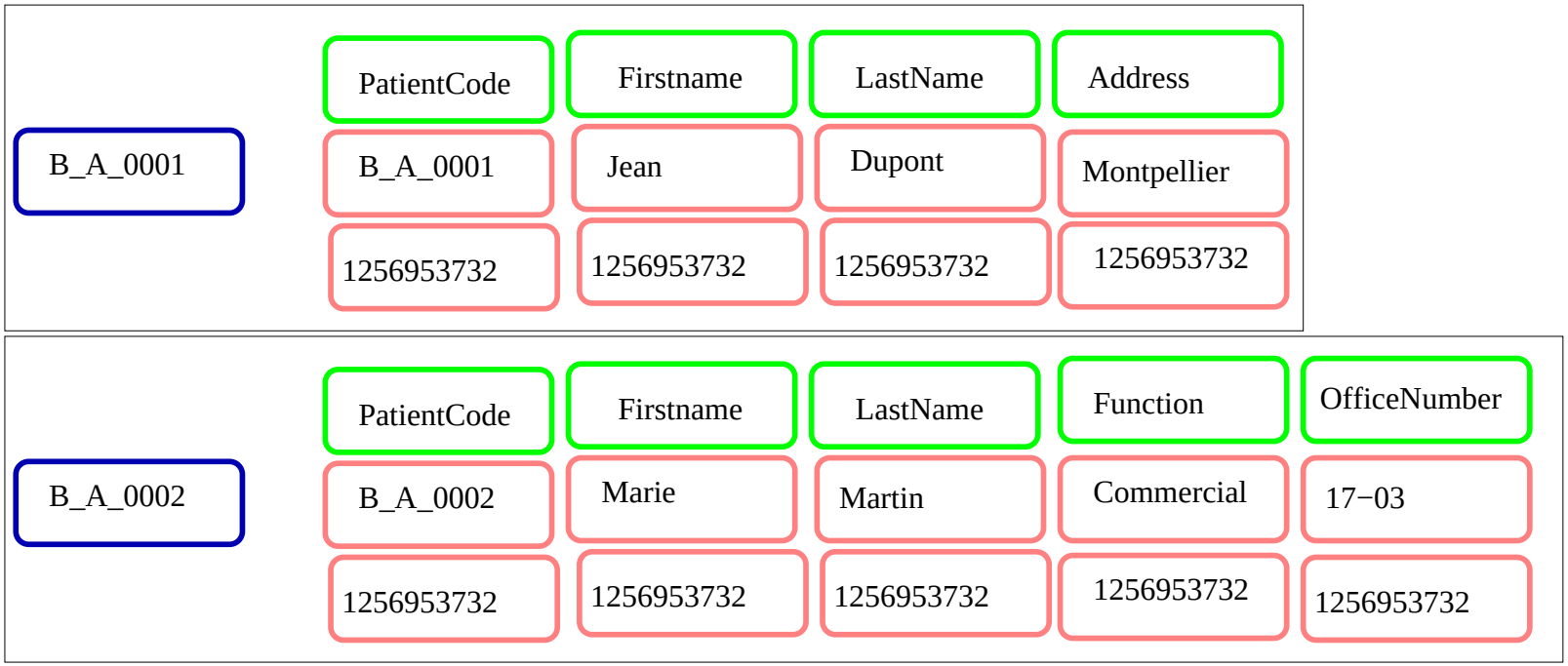


Figure: Exemple de deux tuples d'une famille

Un tuple d'une table en JSON

plusieurs dimensions : famille, colonne, estampille

```
{
   "R_00001": {
      "personal": {
         "Firstname": {
            "Timestamp1": "Jean",
            "Timestamp2": "Pierre" },
         "LastName": {
            "Timestamp1": "Dupont"},
         "Address": {
            "Timestamp1": "Montpellier"}
      },
      "clinical": {
         "Sneezing": {
            "Timestamp4": "severe",
```

Retour sur la notion d'agrégat

Différentes dimensions : CF, Qualifier, Timestamp et Cell

```
{ "R_00001": {                                              key
 "personal" :                                               cf
   {"Firstname": { "Timestamp1":"Jean",
     "Timestamp2":"Pierre"}
     "LastName": {"Timestamp1":"Dupont"},                   qualifier
    "Address": { "Timestamp1":"Montpellier",                ts
      "Timestamp3":"Palavas"}                               value
 }
"clinical":
 {"Sneezing": { "Timestamp4":"severe",
     "Timestamp5":"mild"}
 "Itchy_Troat": : { "Timestamp4":"severe",
     "Timestamp5":"mild",
     "Timestamp6":"moderate"}
 }
}
```

Retour sur la notion d'agrégat

Deux façons d'envisager la clé (entité racine de l'agrégat)

```
{ "R_00001": {
 "personal" :
    {"Firstname": { "Timestamp1":"Jean",
      "Timestamp2":"Pierre"}
      "LastName": {"Timestamp1":"Dupont"},
     "Address": { "Timestamp1":"Montpellier",
        "Timestamp3":"Palavas"}
  }
 "clinical":
  {"Sneezing": { "Timestamp4":"severe",
      "Timestamp5":"mild"}
  "Itchy_Troat": : { "Timestamp4":"severe",
      "Timestamp5":"mild",
      "Timestamp6":"moderate"}
  }
}
```

Figure: clé = Row Key & value = CF/Qualifier/TS/Cell

Retour sur la notion d'agrégat

Deux façons d'envisager la clé (entité racine de l'agrégat)

```
{ "R_00001": {
 "personal" :
   {"Firstname": { "Timestamp1":"Jean",
     "Timestamp2":"Pierre"}
     "LastName": {"Timestamp1":"Dupont"},
    "Address": { "Timestamp1":"Montpellier",
      "Timestamp3":"Palavas"}
 }
"clinical":
 {"Sneezing": { "Timestamp4":"severe",
     "Timestamp5":"mild"}
 "Itchy_Troat": : { "Timestamp4":"severe",
     "Timestamp5":"mild",
     "Timestamp6":"moderate"}
 }
}
```

Figure: clé = Key/CF/Qualifier/TS & value = Cell

Représentation interne HBase

Deuxième manière de voir les choses

```
(R_00001, personal:Address, Timestamp1) => Montpellier
(R_00001, personal:Address, Timestamp3) => Palavas
(R_00001, clinical:Sneezing, Timestamp4) => severe
```

Listing 3: key/value

Organisation physique en colonne

Répercussion sur l'architecture physique de Hbase

Table triée sur la valeur de la clé (ordre lexicographique) et fragmentée en parties égales = régions (intervalles de valeurs de clés)

- 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
- Caching Block (+/- équivalent à databuffer pour Oracle) : tampon de données pour les opérations de lecture (algorithme LRU)
- HFile : 1 à plusieurs fichiers de données par région
- HLog : 1 fichier journal par RegionServer
- Block : unité d'échange entre les mémoires vive et de masse (64 Ko à l'ordinaire)

Organisation structures mémoires et fichiers

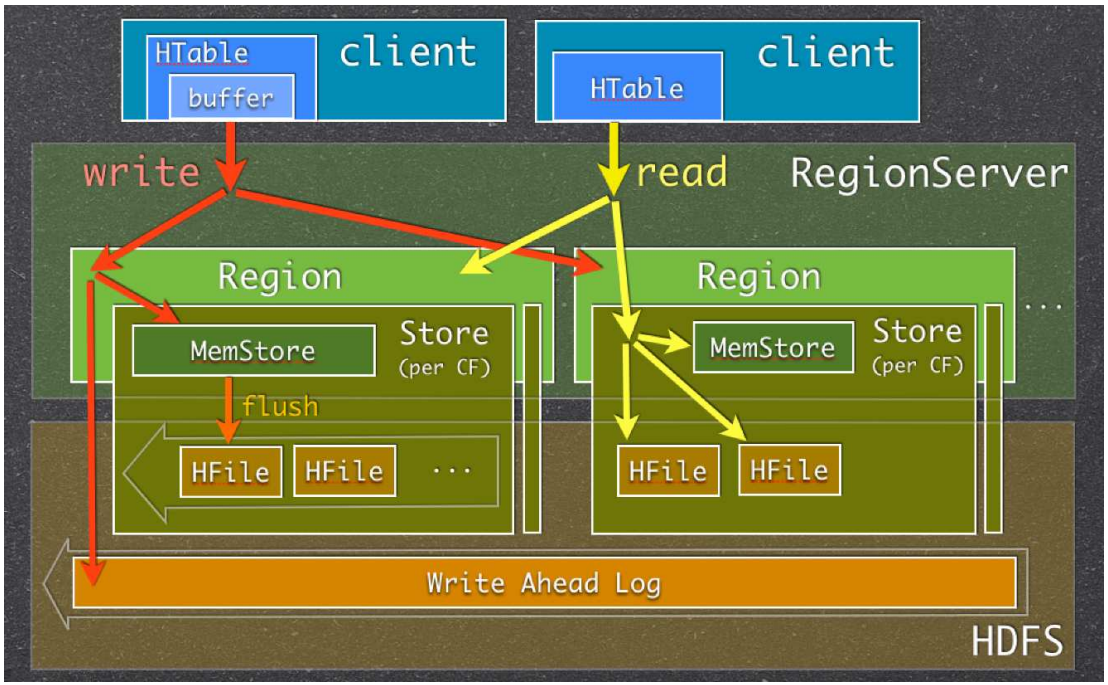


Figure: Détails RegionServer

Diagramme de classes des grandes notions mises en jeu

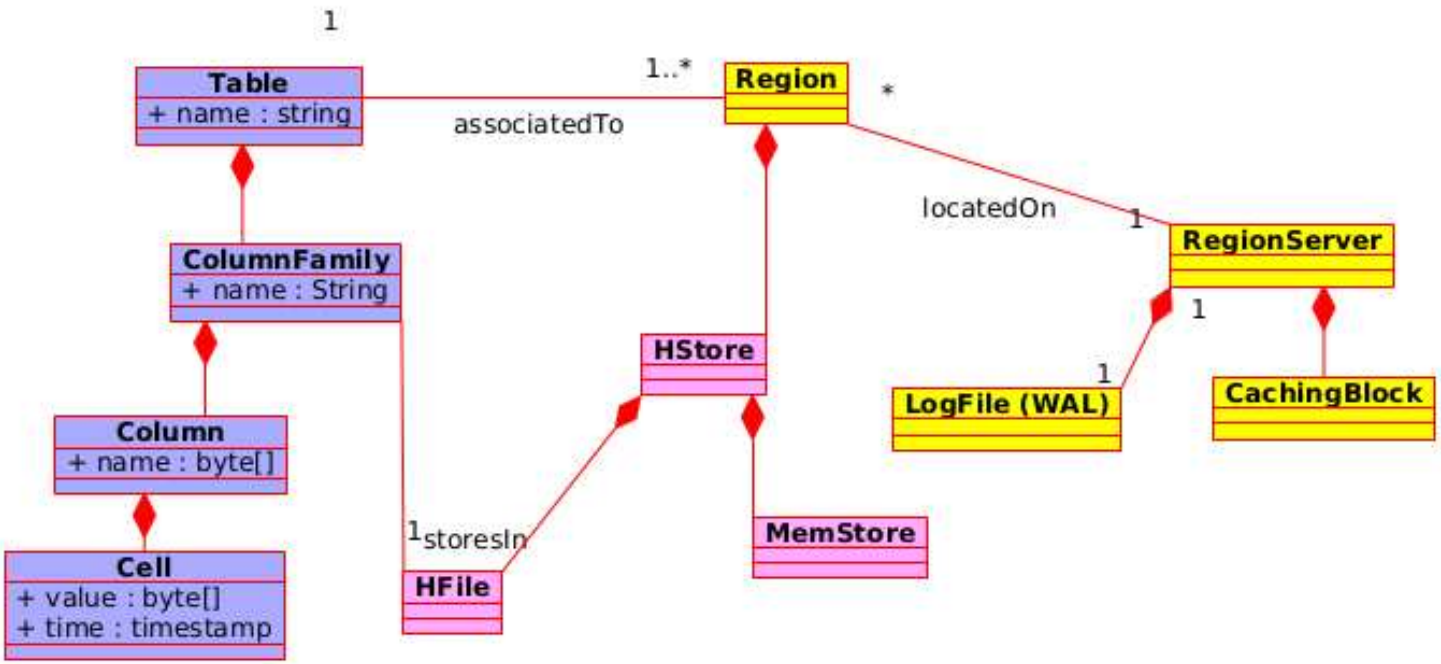


Figure: Diagramme de classes UML

Diagramme de séquence : accès en écriture

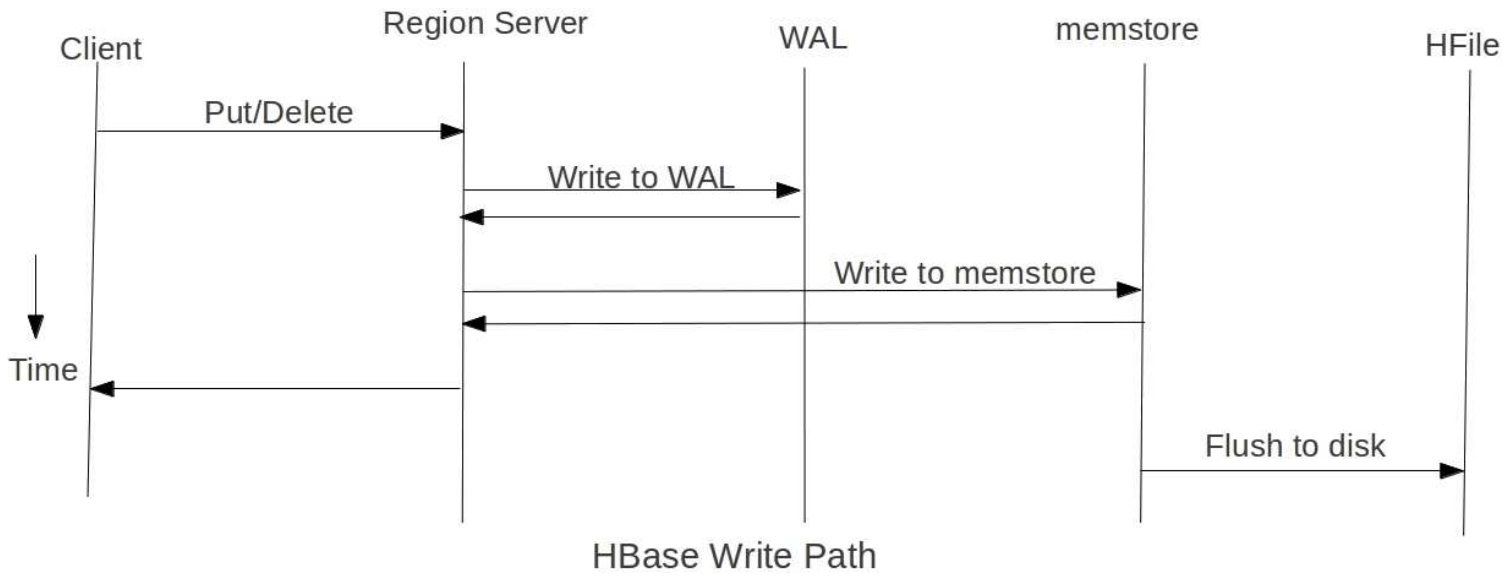
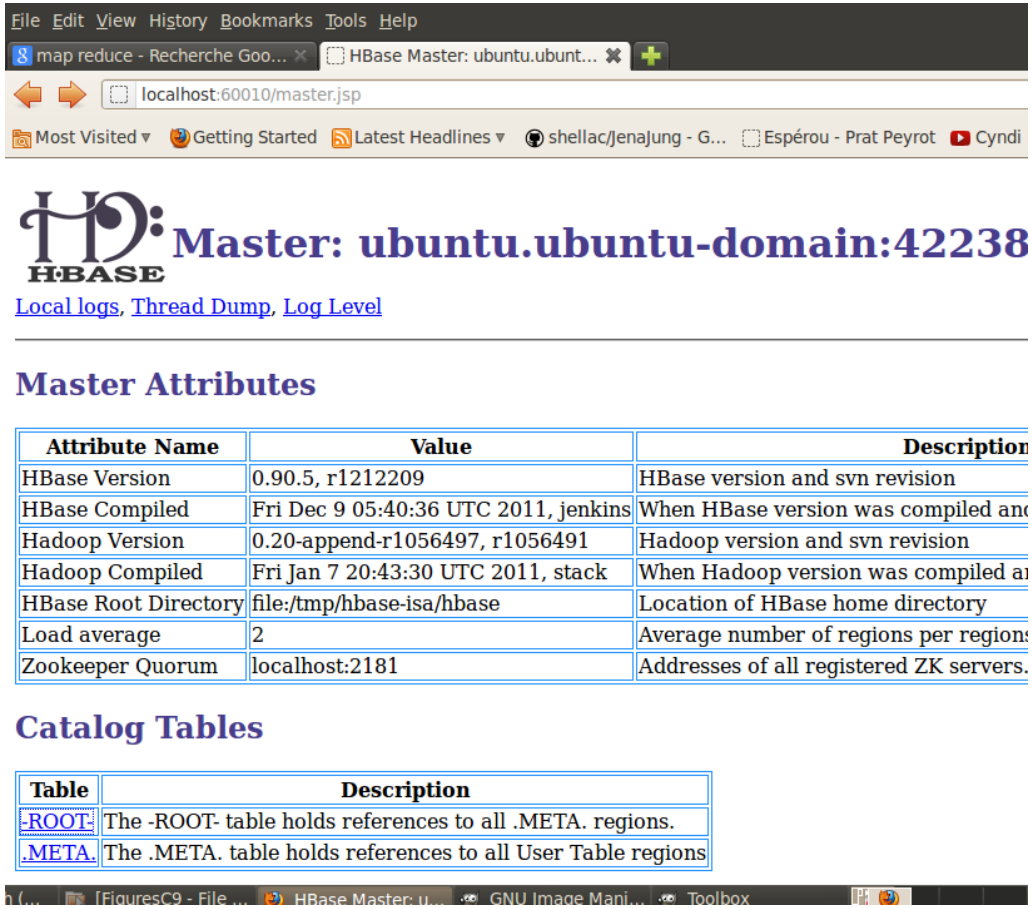


Figure: Diagramme de classes UML

Page d'accueil (WebApp)



Catalogues de métadonnées root et meta

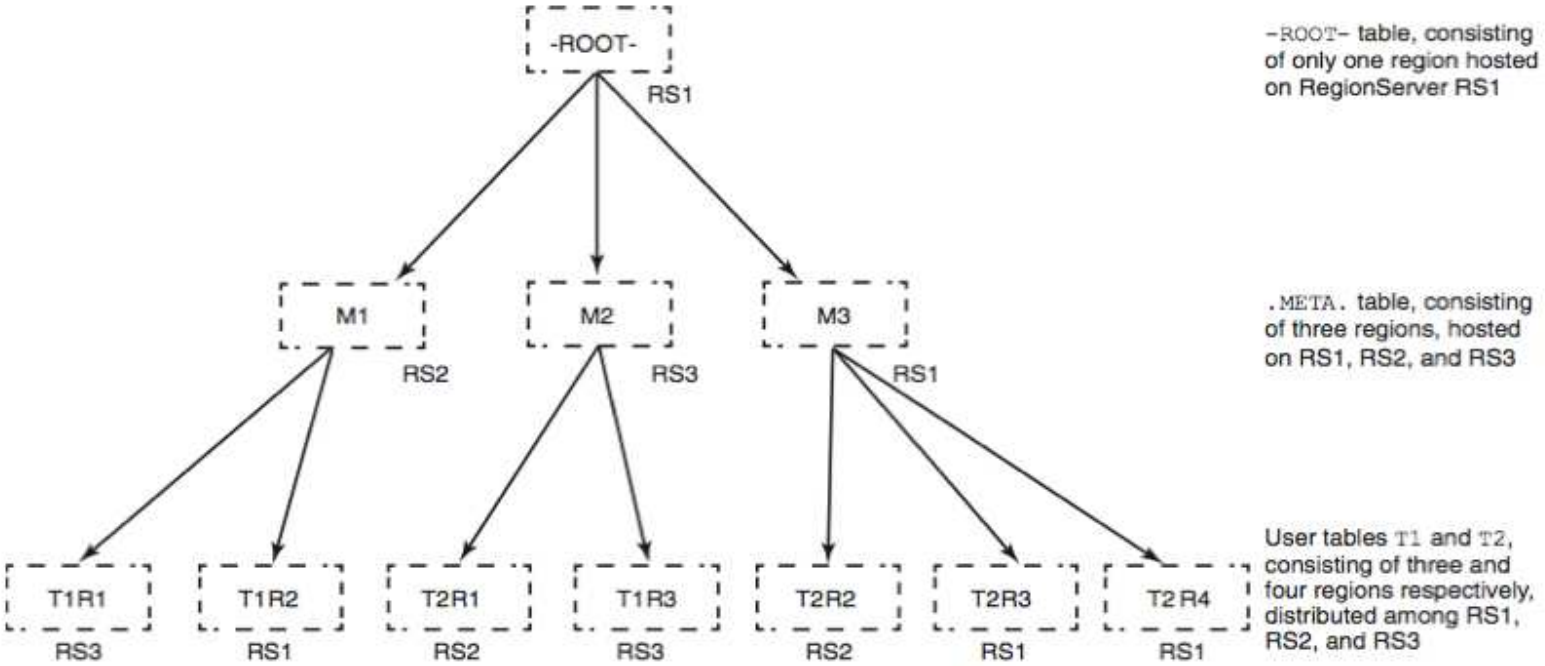


Figure: Organisation des informations de synthèse

Accès et traitement des données

Accès plus impératif que déclaratif

1. absence 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 Clojure, Scala, Jython, PHP, C++...
3. notion de coprocessor 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 collections de paires clé/valeur en entrée comme en sortie de diverses tâches d'analyse

Construire une table et deux colonnes avec JRuby

```
create 'ville', 'iG', 'ip10'
put 'ville', '01024', 'iG:codeInsee', '01024'
put 'ville', '01024', 'iG:nom', 'Attignat'
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'
resultats

_________
ROW                  COLUMN+CELL
 01024  column=iG:codeInsee, timestamp=1354564480805,
```

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

Construire une table et insertion de tuples

```
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 5: tuple Pierre dans table patient

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 6: patients de moins de 26 ans

Filtre Suite et affichage

```
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 7: retourner les jeunes patients

Mise en œuvre autour du traitement distribué

HBase : données massives dans un environnement distribué

- Mapreduce : environnement d'exécution avec une visée analytique
- Coprocessor : même idée mais pour des tâches plus élémentaires

Principes Partitionnement

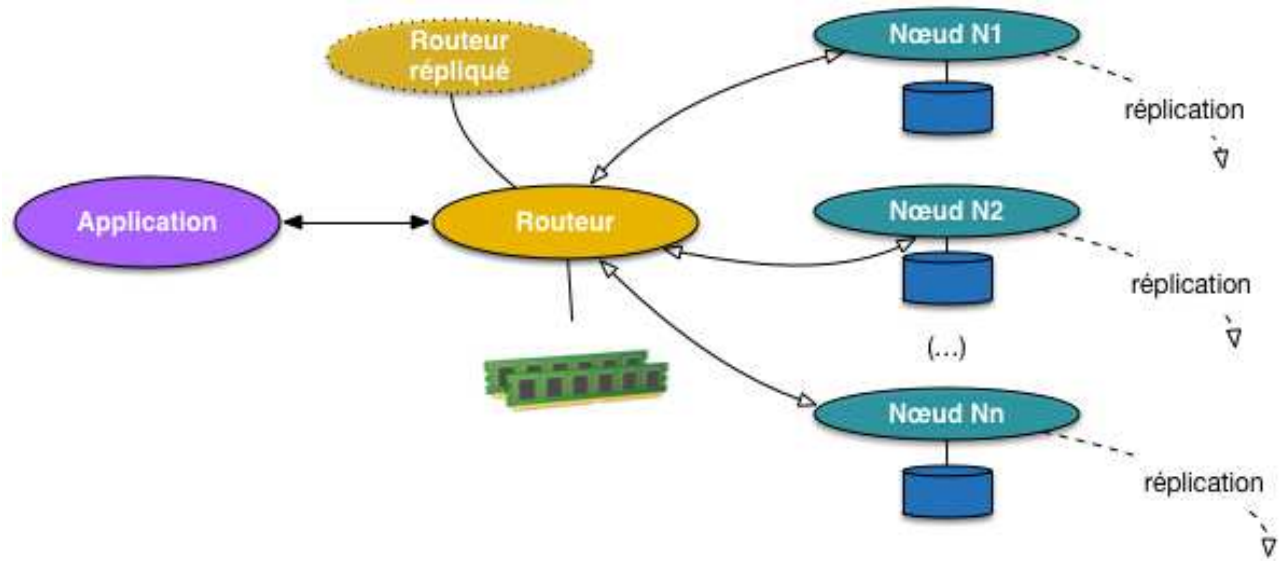


Figure: Extrait de http://b3d.bdpedia.fr/sharding.html

HBase, MongoDB partitionnement par intervalles / Cassandra, Riak, Redis partitionnement par hachage

Partitionnement horizontal des tables sur valeur triée de la clé

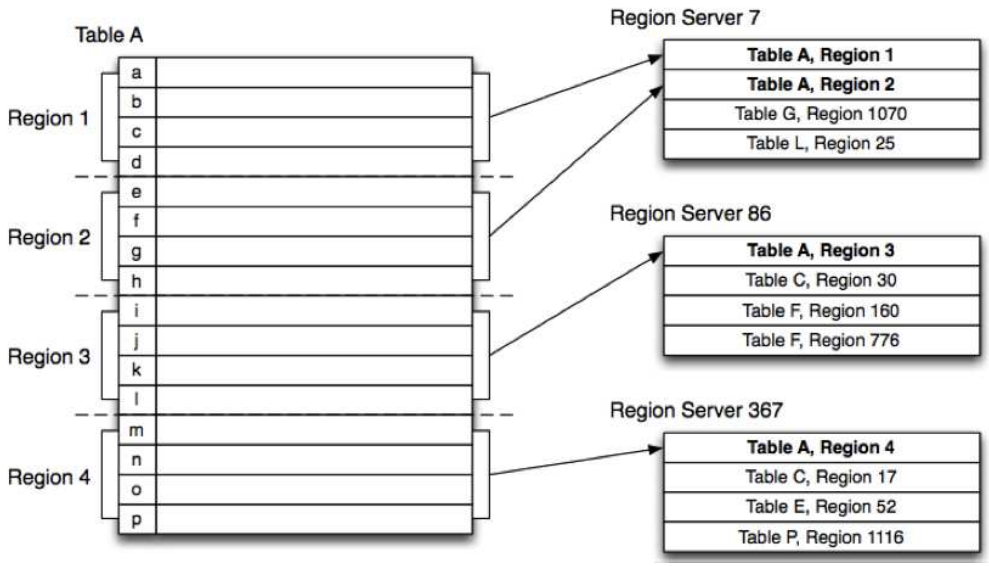


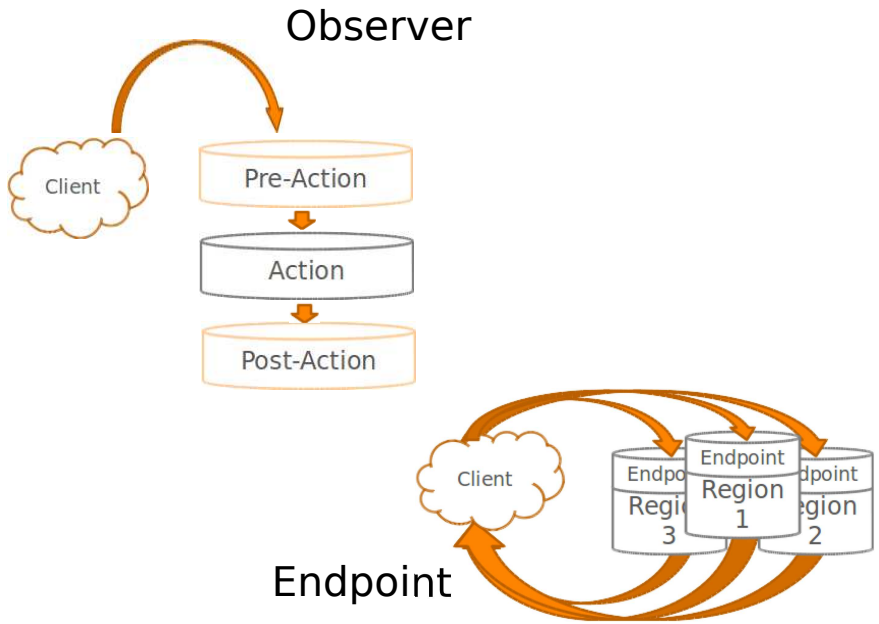
Figure: "Sharding" (partitionnement horizontal)

Notion de Coprocessor

permettre d'exécuter du code au niveau du nœud RegionServer (en local sur chaque noeud RegionServer du cluster)

- étendre les fonctionnalités de HBase
- principes identiques ceux rencontrés avec les surcouches procédurales PL/SQL d'Oracle ou PL/PgSQL de Postgres = traiter les données directement là ou elles sont et ne pas chercher à transférer de gros volumes de données
- Défini dans : J. Dean, "Designs, Lessons and Advice from Building Large Distributed Systems", LADIS '09

Deux familles de Coprocessor



(extrait de J. Weatherford HBase Conf. 2013)

Figure: Observer et Endpoint

Deux familles de Coprocessor

qui sont construites au travers de deux API distinctes

- Observer : plus ou moins des déclencheurs qui vont s'exécuter à partir d'un évènement : put, get ou delete par exemple
- EndPoint (à rapprocher des procédures stockées), par exemple des calculs sur des agrégats comme le calcul portant sur les populations municipales pour retourner la somme d'habitants pour un département donné et une année donnée

Rôles joués par les Coprocessors

Eventail assez large

- accès et sécurité : empêcher certaines écritures, contrôler les connexions clientes
- définition d'index secondaire et optimisation des accès
- gérer de la cohérence entre données : mise à jour/suppression en cascade par exemple (intégrité référentielle)
- vues synthétiques sur les données (agrégation et calcul)
- analyse temps réel ...

Interfaces et classes d'intérêt

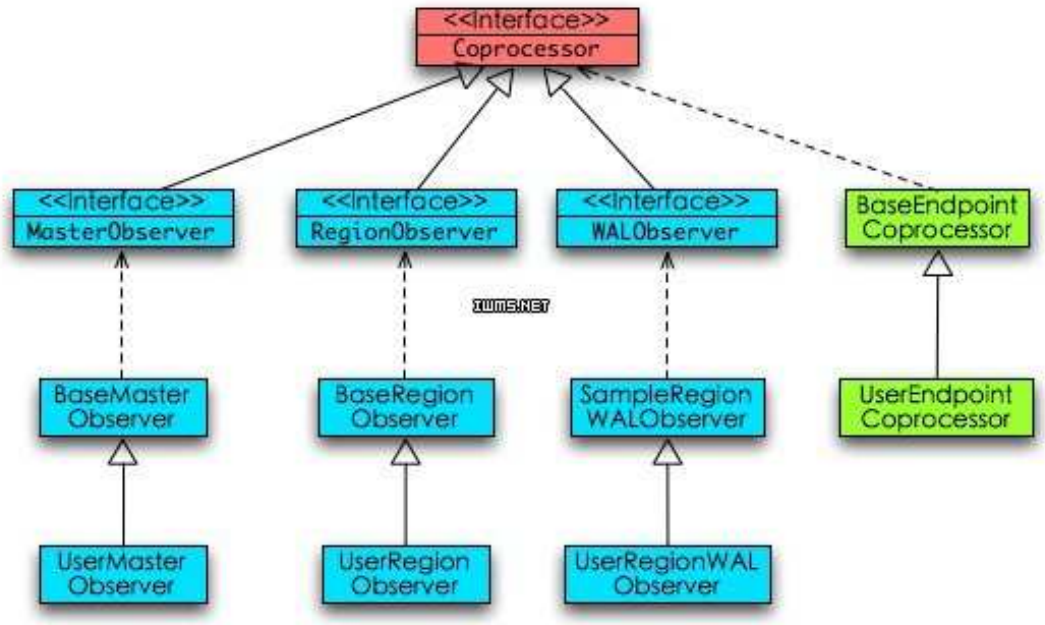
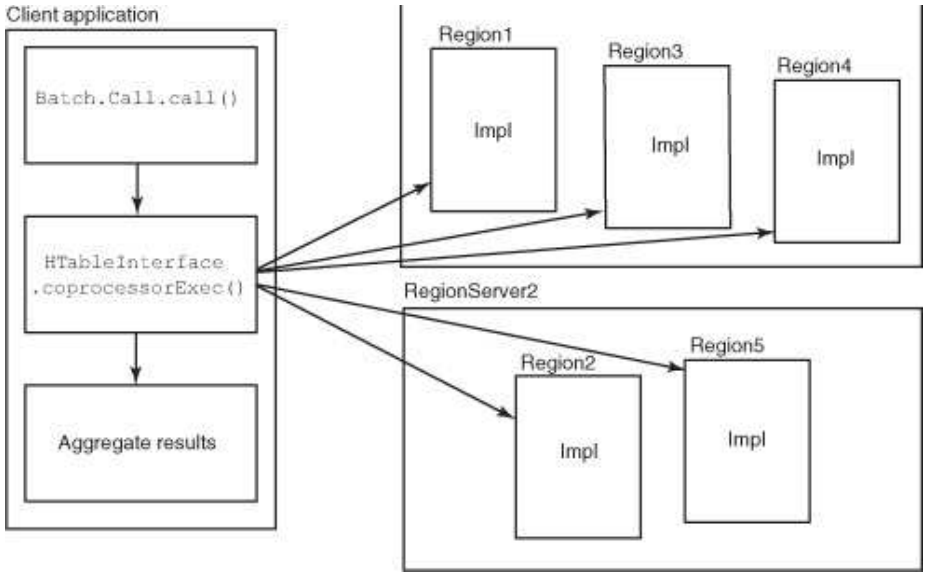


Figure: Dériver une classe ou bien implémenter une interface

Endpoint : procédure stockée

- Assurer des tâches de calcul comme une moyenne ou une somme qui restent performantes sur de très grosses tables partitionnées sur des régions multiples.
- L'idée ici est de disposer comme dans le cas de procédures stockées en PL/SQL d'une extension générique permettant de publier de nouvelles méthodes adaptées au besoin des applications clientes. Les Coprocessors de type endpoint s'executent aussi au niveau des noeuds esclaves (RegionServer donc et co-localisés avec les données)
- Pour le client, l'invocation se fait de manière transparente. La requête est transmise à l'interface HTableInterface qui l'exécute sur le cluster et les résultats sont ensuite collectés pour être retournés au client.

Exemple de la distribution d'une calcul portant sur une agrégation



Instance de Batch.Call encapsule invocation de méthode, méthode coprocessorExec() gère le partage. Après achèvement de chaque sous-requête, les résultats sont retournés au client puis agrégés (ex. somme ou moyenne).

Endpoint : étapes de construction

- définir une classe qui implémente une interface Protocol
- surcharger les méthodes voulues
- Exporter le code dans une archive Java (jar)
- Etendre BaseRegionEndpoint
- Charger le Coprocessor dans la table
- Ecrire un programme de test

Exemple avec un coprocessor prédéfini : AggregationClient

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

Charger au préalable le Coprocessor

dans hbase-site.xml

```
<property>
  <name>hbase.coprocessor.user.region.classes</name>
  <value>
org.apache.hadoop.hbase.coprocessor.AggregateImplementation
</value>
 </property>
```

Listing 9: exemple chargement

Observer (Trigger) : niveaux d'application

- `RegionObserver` : déclencher des actions relatives à des évènements sur une region : par exemple 'prePut' et 'postPut' pour les opérations d'insertion.
- `RegionServerObserver` : déclencher des actions relatives à un nœud esclave tel que l'arrêt du noeud ou la réalisation d'opérations avant ou après des fusions, des validations ou des annulations de transaction (attention aux arrêts malicieux)
- `WAL Observer` : activités de type avant ou après écriture dans les structures de journalisation : deux méthodes 'preWALWrite()' and 'postWALWrite()'.
- `Master Observer` : déclencheurs pour les opérations DDL : création, suppression, ou modification de tables.

Exemple de la table sous surveillance

	ColumnFamily: personalDet			ColumnFamily: salaryDet		
Rowkey	name	surname	dob	gross	net	allowance
admin	Admin	Admin				
chard	Charles	Dickens	06/21/83	8000	7000	2000
johnm	John	Milton	03/16/79	9000	8000	2500

Observer : exemples de méthodes d'intérêt de BaseRegionObserver

Déclencher des évènements sur une `Region` : preGet, postGet, prePut, postPut, preDelete, postDelete

déclenchement de l'action avant l'évènement intercepté

```
@Override
public void prePut (final ObserverContext e, final Put
    put, final WALEdit edit,final Durability durab)
throws IOException {
        }
```

Listing 10: surcharger prePut

Exemple : ne pas afficher les détails de l'admin

action avant l'évènement intercepté : preGetOp et preGet

```
public class RegionObserverExample extends
    BaseRegionObserver {
private static final byte[] admin =
    Bytes.toBytes("admin");
private static final byte[] COL_F =
    Bytes.toBytes("personal_det");
private static final byte[] COL =
    Bytes.toBytes("Admin_det");
private static final byte[] VAL = Bytes.toBytes("no
    access to admin details");
  public void preGetOp(final ObserverContext e, final
      Get get, final List results)
   throws IOException {
       ...
```

Exemple suite

```
      ...
  if (Bytes.equals(get.getRow(),admin)) {
     Cell c = CellUtil.createCell(get.getRow(),COL_F,
        COL, System.currentTimeMillis(), (byte)4, VAL);
     results.add(c);
     e.bypass();
    }
      List kvs = new ArrayList(results.size());
      for (Cell c : results) {
         kvs.add(KeyValueUtil.ensureKeyValue(c));}
      preGet(e, get, kvs);
      results.clear();
      results.addAll(kvs);}
}
```

Listing 12: overriding prePut

Principes BaseRegionObserver

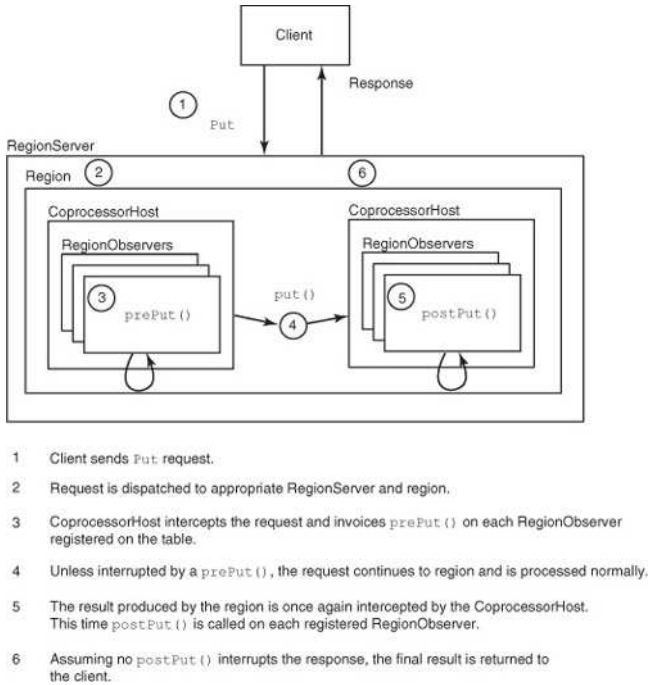


Figure: Illustration Evènement-Action

Observer : étapes de construction

- définir une classe qui implémente une interface Observer
- surcharger la méthode correspondant à l'action voulue ici 'preGetOp()' (ou 'preGet()' pour les versions passées, Op pour Open). Dans l'exemple les détails des comptes sont affichés sauf pour Admin.
- Exporter le code dans une archive Java (jar)
- Placer the jar dans le système HDFS ou HBase va pouvoir le localiser
- Charger le Coprocessor
- Ecrire un programme de test

Principes MapReduce

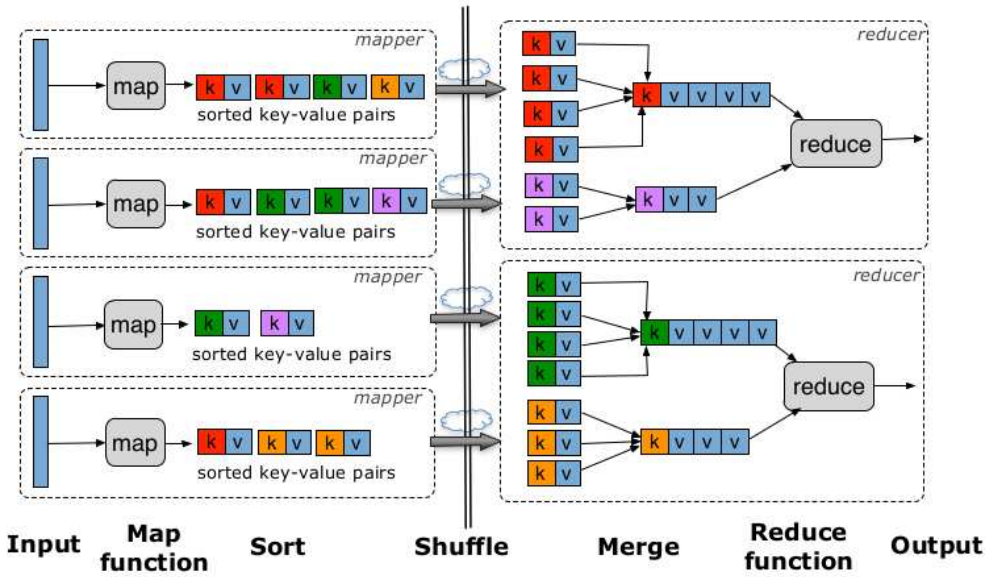


Figure: Extrait de I. Manolescu IN2P3 Roscoff 2013

Mise en œuvre avec HBase : data "source" et data "sink"

- TableInputFormat (en entrée du processus MR) avec
- processus Map retourne un "Writable" qui correspond à la clé et un Result avec les valeurs des colonnes pour le tuple
- en sortie du processus de Map les couples clé/valeur souhaités en fonction de la logique métier
- processus Reduce construit les résultats et émet les clés sous forme d'un ImmutableBytesWritable et une commande Put pour sauvegarder les résultats dans HBase. (TableOutputFormat)
- éventuellement le concours de TableMapReduceUtil (pouvoir exploiter les tuples au travers d'une HTable)

Exemple de comptage d'occurrences (compteur de visite de pages)

- Une class Mapper avec une méthode map, une class Reducer avec une méthode reduce, et un programme principal qui configure l'ensemble de la tâche (job).
- Extrait de http://sujee.net/2011/04/10/hbase-map-reduce-example

Table origine

```
userID_timestamp =>
{details => {page}}

row       details:page
user1_t1   a.html
user2_t2   b.html
user3_t4   a.html
user1_t5   c.html
```

Listing 13: Table access_logs

Table résultat

```
userID =>
{
details => {frequency}
}

user  count (frequency)
user1  2
user2  1
user3  1
```

Listing 14: Table summary_user

Etapes intermédiaires : en sortie de map

```
(user1, 1)
(user2, 1)
(user1, 1)
(user3, 1)
```

Listing 15: After map

Etapes intermédiaires : en entrée de reduce

```
(user1, [1, 1])
(user2, [1])
(user3, [1])
```

Listing 16: After shuffle/sort

Etapes intermédiaires : en sortie de reduce

```
(user1, 2)
(user2, 1)
(user3, 1)
```

Listing 17: After reduce

Création de tables préalable

```
create 'access_logs', 'details'
   create 'summary_user', {NAME=>'details', VERSIONS=>1}
```

Listing 18: shell HBase

Corps du programme principal intégrant deux "inner class"

```
public class MRCount {
   static class Mapper1 extends
       TableMapper<ImmutableBytesWritable, IntWritable> {
      public void map(ImmutableBytesWritable row,
          Result values, Context context) { } }
   public static class Reducer1 extends
       TableReducer<ImmutableBytesWritable, IntWritable,
       ImmutableBytesWritable> {
   @Override
      public void reduce(ImmutableBytesWritable key,
          Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
                } }
   public static void main(String[] args) throws
       Exception { }
```

Classe qui dérive de TableMapper

```
public class MRCount {
   static class Mapper1 extends
       TableMapper<ImmutableBytesWritable, IntWritable> {
      private int numRecords = 0;
      private static final IntWritable one = new
          IntWritable(1);
      public void map(ImmutableBytesWritable row,
          Result values, Context context) throws
          IOException {
         // extract userKey from the compositeKey
             (userId + counter)
         ImmutableBytesWritable userKey = new
             ImmutableBytesWritable(row.get(), 0,
             Bytes.SIZEOF_INT);
         try {
            context.write(userKey, one);
```

Classe qui dérive de TableReducer

```
public class MRCount { ...
   public static class Reducer1 extends
       TableReducer<ImmutableBytesWritable, IntWritable,
       ImmutableBytesWritable> {
      public void reduce(ImmutableBytesWritable key,
          Iterable<IntWritable> values, Context context)
          {
         int sum = 0;
         for (IntWritable val : values) {
            sum += val.get();}
         Put put = new Put(key.get());
         put.add(Bytes.toBytes("details"),
             Bytes.toBytes("total"), Bytes.toBytes(sum));
         System.out.println(String.format("stats : key
             : %d, count : %d", Bytes.toInt(key.get()),
             sum));
```

Corps du programme principal intégrant deux "inner class"

```
public class MRCount {...
  public static void main(String[] args) throws
      Exception {
      Configuration conf = HBaseConfiguration.create();
      Job job = new Job(conf, "MRCount");
      job.setJarByClass(MRCount.class);
      Scan scan = new Scan();
      scan.addFamily(Bytes.toBytes("contact"));
      TableMapReduceUtil.initTableMapperJob("personne",
          scan, Mapper1.class,
          ImmutableBytesWritable.class,
            IntWritable.class, job);
      TableMapReduceUtil.initTableReducerJob("summary_user",
          Reducer1.class, job);
      System.exit(job.waitForCompletion(true) ? 0 : 1);
```