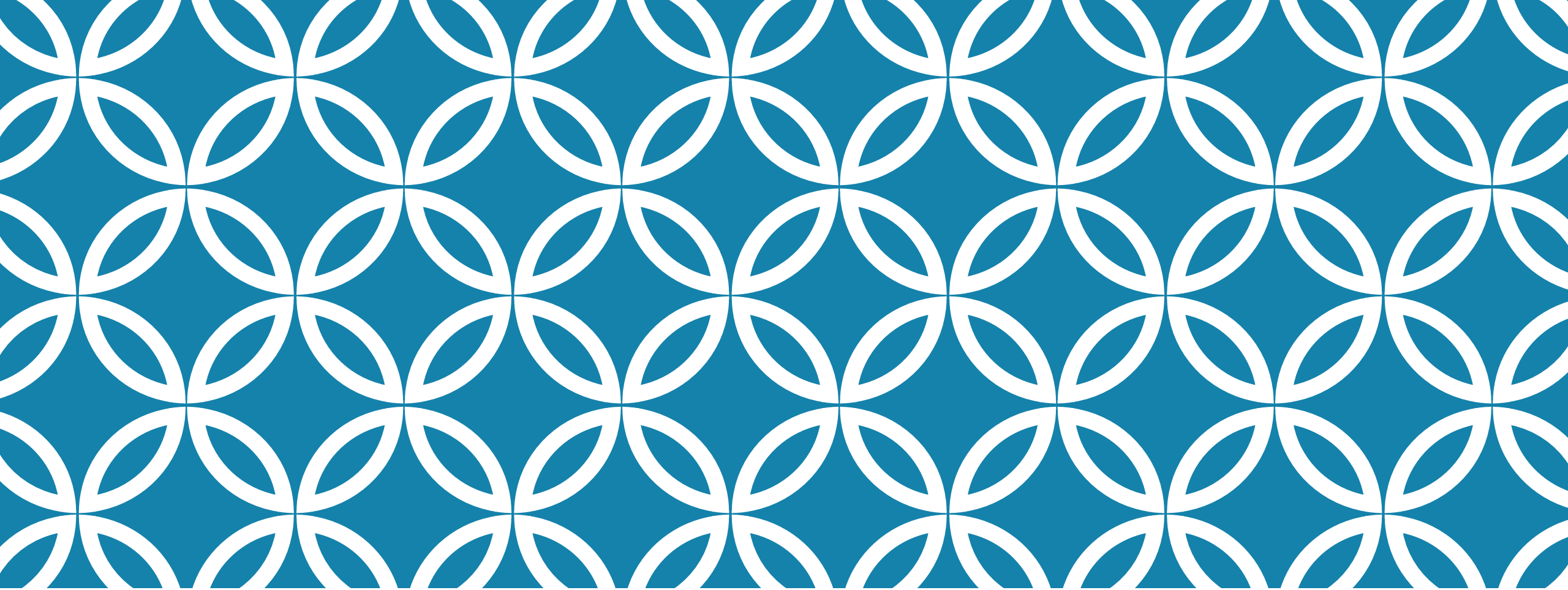


APACHE HBASE

HBASE

HBASE

Apache HBase est la base de donnée de Hadoop. Elle est distribuée et elle passe l'échelle horizontalement. C'est une base de données de type « NoSQL ». Cela signifie que HBase n'offre pas toutes les fonctionnalités des « Base de données relationnelle ». Par exemple, le langage SQL n'est pas supporté et les colonnes ne sont pas typées … HBase s'inspire très fortement de la base de données de Google appelé BigTable.

- Base de donnée non relationnelle distribuée
- Orienté colonne
- Multi dimensionnelle
- Haute disponibilité
- Haute performance

HBASE histoire

- 2006 : Article de Google sur les BigTable.
- 2007 : Première version utilisable de Hbase
- 2010 : HBase devient un projet phare de Apache
- …
- 2016 : HBase 1.2.3

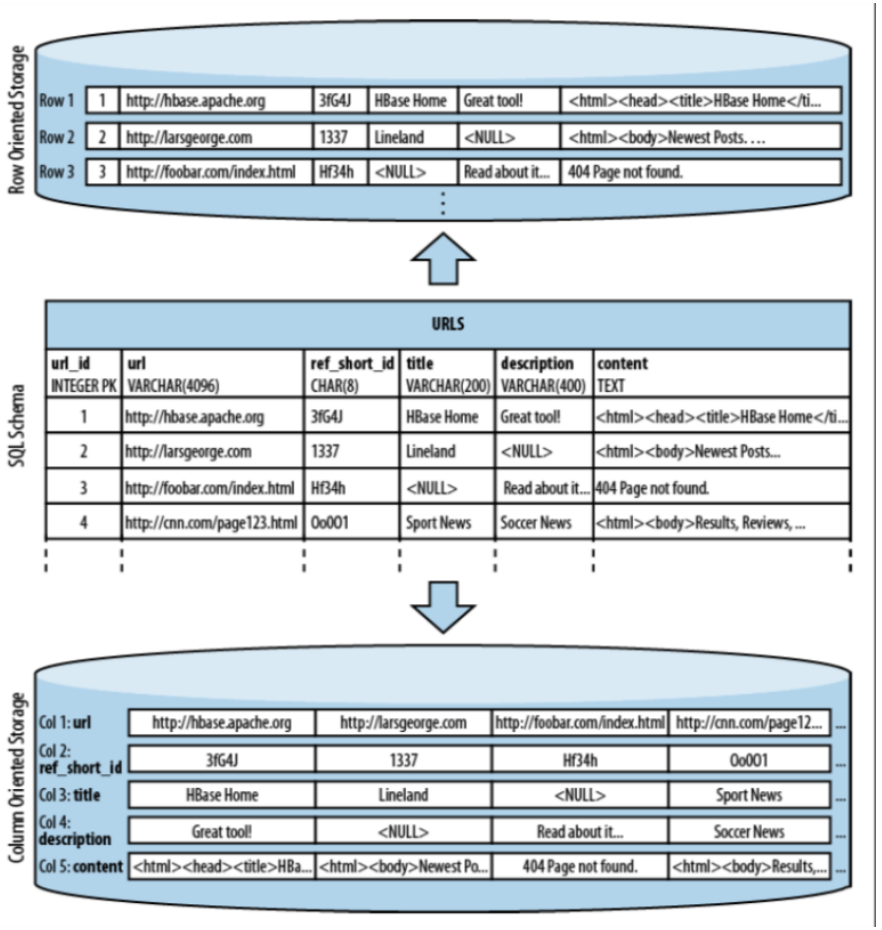
Référence :
Bigtable: A Distributed Storage System for Structured Data. (Operating system design and implementation 2006)

HBase versus HDFS

HDFS est un système de fichier distribué. Cependant il ne permet pas de stocker d'énorme quantité de fichiers et il ne permet pas non plus de lire et écrire efficacement n'importe où dans un fichier existant. HBase va permettre d'ajouter la possibilité de faire des lectures et des écritures rapides dans des tables qui seront stockées sur système de fichier distribué comme HDFS, GVFS ou encore amazon S3.

Modèle de donnée de HBASE

On retrouve dans HBase les concepts de table de ligne ou bien de colonne comme dans les bases de données relationnelles classiques. Cependant c'est terme sont un peu trompeur et il vaut mieux penser à des map ou dictionnaire quand on travaille avec HBase ou BigTable.



BigTable Vs HBASE

Dans le papier « BigTable » on trouve la définition suivant pour le model de donnée :

« A BigTable is a sparse, distributed, persistent multidimensional sorted map »

Puis:

“The map is indexed by a row key, column key and a timestamp; each value in the map is an un-interpreted array of bytes”

BigTable Vs HBASE

Chez HBase on trouve ceci:

« HBase uses a data model very similar to that of BigTable, Users store data rows in labelled tables. A data row has a sortable key and an arbitrary number of columns. The table is stored sparsely, so that rows in the same table can have crazily-varying columns, if the user likes.”

Si on lit bien tout est là. Les mots clé à retenir sont : MAP, PERSISTENT, DISTRIBUTED, SORTED, MULTIDIMENSIONNAL, et SPARSE.

HBASE : MAP

Le cœur de HBase est un système qui permet de stocker efficacement des couples <clé, valeur>. Cela fonctionne exactement comme des array (PHP), des dictionnaires (Python) ou encore des map (C++).

Les clés sont des bytes et les valeurs sont des bytes.

Exemple de map :

{
« paris » : « 1024 »,
« london » : « 2048 »,
« marseille » : « 23 »
}

HBASE : Persistant

La persistance signifie que les données restent quand votre programme est terminé. En d'autre terme les données sont stockées sur disque et on peut y accéder ultérieurement.

HBASE : Distribué

Tous les fichiers de HBase sont stockés sur HDFS, la base de données est donc distribuée. C'est le même principe pour BigTable de Google qui utiliser GFS.

HBASE : Trié

Les map HBase sont ordonnées. L'ordre est basé sur les clés. C'est un ordre lexicographique. Le fichier précédent va donc être stocké comme suit :
{
 « london » : « 2048 »,
« marseille » : « 23 »
« paris » : « 1024 »,
}

L'ordre à un impact important sur les performances. Sachant les map sont stockées dans des blocs il est préférable que les données soient dans le même bloc. Vu le fonctionnement de HDFS, HBase va devoir lire tous les blocs pour donner accès aux données. Il est donc primordial de bien étudier ses clés.

HBASE : Multidimensionnelle

Pour être plus précis, HBASE ne stocke les données en utilisant
map< Row , map<ColumnFamily, map< Column, map< TimeStamp, Value> >

Dans Hbase, les colonnes sont regroupées en famille de colunnes. Toutes les colonnes de la même famille ont le même prefix. Les famille de colonnes doivent utiliser des caractère imprimable, les noms de colonnes eux sont des champs d'octects. L'adresse d'une colonne est donc de la forme: NomDeFamiille:NomColonne. Les ":" séparent le nom de famille et le nom de la colonne (attribut)
Les familles doivent être décalrées avant de les utiliser, les attributs eux sont créés à la volée.
Techniquement, toutes les colonnes de la même famille sont stockées dans le même fichier (HDFS). Les optimisations de stockage sont donc faites au niveau famille. Il est donc important de mettre dans la même famille des « attibuts » similaires (nombre de valeur, accès simultané…)

TimeStamp correspond à un marquage temporel des données.

HBASE : Opérations

Put

L'opération « put » permet d'ajouter une nouvelle ligne dans une table. La ligne est ajouté si elle n'existe pas sinon est mise à jours. L'operation est effectué via une instance de Table avec les commandes table.put (writeBuffer) ou table.batch (non-writeBuffer). Les modifications sont estampillées mais il est possible de fixer le TimeStamp.

```
public static final byte[] CF = "cf".getBytes();
public static final byte[] ATTR = "attr".getBytes(); ...
Put put = new Put(Bytes.toBytes(row));
put.add(CF, ATTR, Bytes.toBytes( data));
table.put(put);
```

```
public static final byte[] CF = "cf".getBytes();
public static final byte[] ATTR = "attr".getBytes(); ...
Put put = new Put( Bytes.toBytes(row));
long explicitTimeInMs = 555; // just an example
put.add(CF, ATTR, explicitTimeInMs, Bytes.toBytes(data));
table.put(put);
```

HBASE : Opérations

Get

Get permet de récupérer les attibuts d'une ligne à partir de sa clé. L'operation get est appellee sur une instance de Table via table.get. Par défaut le get retourne la dernière version de la ligne mais il est possible d'obtenir toutes les versions.

```
public static final byte[] CF = "cf".getBytes();
 public static final byte[] ATTR = "attr".getBytes(); ...
Get get = new Get(Bytes.toBytes("row1"));
Result r = table.get(get);
byte[] b = r.getValue(CF, ATTR);
```

```
public static final byte[] CF = "cf".getBytes();
public static final byte[] ATTR = "attr".getBytes(); ...
Get get = new Get(Bytes.toBytes("row1"));
get.setMaxVersions(3); // will return last 3 versions of row
Result r = table.get(get);
byte[] b = r.getValue(CF, ATTR); // returns current version of value
List<KeyValue> kv = r.getColumn(CF, ATTR); // returns all versions of this column
```

HBASE : Scans

Scans:

Les scanners pemerttent d'itérer sur plusieurs lignes d'une table et sur des attributs spécifique. Il integer un mécanisme de filter permettant de ne lire que des lignes vérifant le filter.

Delete
L'operation delete permet d'enlever des lignes d'une tabler.

```
public static final byte[] CF = "cf".getBytes();
public static final byte[] ATTR =
"attr".getBytes(); ...
Table table = ... // instantiate a Table instance
Scan scan = new Scan();
scan.addColumn(CF, ATTR);
scan.setRowPrefixFilter(Bytes.toBytes("row"));
ResultScanner rs = table.getScanner(scan);
try {
  for (Result r = rs.next(); r != null; r = rs.next())
    { // process result... }
}
finally {
 rs.close(); // always close the ResultScanner!
}
```

HBASE : Shell

Hbase fournit une interface en ligne de commande pour administrer et interoger une base. On y accède en lançant la commande

>hbase shell

Liste des commandes détaillées:
https://learnhbase.wordpress.com/2013/03/02/hbase-shell-commands/

create	Create table; pass table name, a dictionary of specifications per column family, and optionally a dictionary of table configuration. **hbase> create 't1', {NAME => 'f1', VERSIONS => 5}** **hbase> create 't1', {NAME => 'f1'}, {NAME => 'f2'}, {NAME => 'f3'}** **hbase> # The above in shorthand would be the following:** **hbase> create 't1', 'f1', 'f2', 'f3'** **hbase> create 't1', {NAME => 'f1', VERSIONS => 1, TTL => 2592000, BLOCKCACHE => true}** **hbase> create 't1', {NAME => 'f1', CONFIGURATION => {'hbase.hstore.blockingStoreFiles' => '10'}}**
put	Put a cell 'value' at specified table/row/column and optionally timestamp coordinates. To put a cell value into table 't1' at row 'r1' under column 'c1' marked with the time 'ts1', do: **hbase> put 't1', 'r1', 'c1', 'value', ts1**

HBASE : MinimalSkeleton

Toutes les opérations accessible via le shell sot aussi accessible via l'API Java.

Le ToolRunner peut être utilisé pour récupérer les configurations

```
public class TPHBase extends Configured implements Tool {

private static final byte[] FAMILY = Bytes.toBytes("AAA");
private static final byte[] ROW    = Bytes.toBytes("BBB");
private static final byte[] TABLE_NAME = Bytes.toBytes("CCC");

public static void createOrOverwrite(HBaseAdmin admin, HTableDescriptor table) throws IOException {
if (admin.tableExists(table.getNameAsString())) {
admin.disableTable(table.getNameAsString());
admin.deleteTable(table.getNameAsString());
}
admin.createTable(table);
}

public static void createTable(Configuration config) {
try {
final HBaseAdmin admin = new HBaseAdmin(config);
HTableDescriptor tableDescriptor = new HTableDescriptor(TableName.valueOf(TABLE_NAME));
HColumnDescriptor famLoc = new HColumnDescriptor(FAMILY);
//famLoc.set...
tableDescriptor.addFamily(famLoc);
createOrOverwrite(admin, tableDescriptor);
admin.close();
} catch (Exception e) {
e.printStackTrace();
System.exit(-1);
}
}

public int run(String[] args) throws IOException {
createTable(getConf());
Connection connection = ConnectionFactory.createConnection(getConf());
Table table = connection.getTable(TableName.valueOf(TABLE_NAME));
Put put = new Put(Bytes.toBytes("KEY"));
table.put(put);
return 0;
}

public static void main(String[] args) throws Exception {
int exitCode = ToolRunner.run(HBaseConfiguration.create(), new TPHBase(), args);
System.exit(exitCode);
}
}
```

HBASE : Map Reduce Integration

Hbase permet une integration directe avec MapReduce de hadoop. Pour cela on peut utilser les TableInputFormat et TableOutPutFormat. Hbase fournit aussi deux classes les TableMapper et TableReducer.

```
Configuration config = HBaseConfiguration.create();
Job job = new Job(config,"ExampleReadWrite");
job.setJarByClass(MyReadWriteJob.class); // class that contains mapper
Scan scan = new Scan(); scan.setCaching(500); // 1 is the default in Scan, which will be bad for MapReduce jobs
scan.setCacheBlocks(false); // don't set to true for MR jobs // set other scan attrs
TableMapReduceUtil.initTableMapperJob( sourceTable, // input table scan, // Scan instance to control CF and attribute selection
MyMapper.class, // mapper class null, // mapper output key null, // mapper output value job);
TableMapReduceUtil.initTableReducerJob( targetTable, // output table null, // reducer class job);
 job.setNumReduceTasks(0);
boolean b = job.waitForCompletion(true);
if (!b) {
  throw new IOException("error with job!");
}
```