Rapport de projet de Programmation Large Echelle

Amelozara MAHAZOASY, Benoit FAGET, Carlos NEZOUT

January 30, 2019

Contents

1 Contexte du projet

1.1 Existant

Nous disposons d'un jeu de données composé de fichiers ".hgt" dont les noms indiquent des géolocalisations de type "(N/S)XX(W/E)XXX" ('N' North, 'S' South, 'W' West, 'E' East, 'XX' latitude (entre 0 et 90), 'XXX' longitude (entre 0 et 180)). Ces fichiers contiennent chacun un tableau 1D de taille 1201*1201 contenant l'ensemble des niveaux de sol (hauteurs) pour une localisation donnée. Cependant, ils ne contiennent pas toutes les géolocalisations de la planète, celles correspondant à des altitudes négatives sont considérées comme désignant des zones d'eau, et il n'est pas intéressant dans le cadre de notre projet de les traiter.

1.2 Objectif

Il s'agit dans ce projet de proposer des visualisations de ces données sous la forme d'une carte de chaleur interactive en se servant des outils de traitement Big Data et des clusters fournis.

2 Technologies employées

Pour ce projet, nous avons utilisé Spark pour les calculs distribués, HBase pour stocker les données traitées, ainsi que NodeJS + Express + (npm hbase-rpc-client) pour la visualisation avec Leaflet. Nous avons principalement travaillé sur le cluster de la salle 005.

3 Solutions proposées

Nous avons travaillé sur les TextFile ".hgt" fournis. On récupère à partir de ces TextFile : en clés, les paths des fichiers ; en valeurs, les flux de bits correspondant à des tableaux 1D contenant les hauteurs des zones correspondantes aux clés. On obtient un JavaPairRDD$< String, PortableDataStream >$.
Nous avons aussi travaillé sur les SequenceFile générés par Marie Pascal et Marc Clervaux. On récupère à partir de ces SequenceFile : en clés, les paths des fichiers '.hgt' fournis ; en valeurs, les tableaux 1D contenant les hauteurs correspondantes. On obtient un JavaPairRDD<Text, IntArrayWritable>.
Dans les parties suivantes, on se sert des classes : ColorHandler pour obtenir des dégradés de couleur à partir des altitudes ; ImageHandler pour redimensionner des images ; HBasePictures pour se connecter à HBase et stocker les images obtenues dans la table que nous avons définie. Pour les couleurs : l'eau est en bleu ; les zones de faibles altitudes (˜0m) en vert foncé ; les zones de moyennes altitudes en jaune clair (˜4000m) ; les zones de hautes altitudes (˜8000m) en marron foncé; les altitudes incorrectes (par exemple 9000m) en rose.

3.1 Images 1201x1201

3.1.1 TextFile

```
JavaPairRDD<String, int[]> colorPairRDD = pairRDD.mapToPair((Tuple2<String, PortableDataStream> pair) -> {
    DataInputStream dis = pair._2.open();
    byte[] data = new byte[dis.available()];
    dis.readFully(data);
    IntBuffer ib = ByteBuffer.wrap(data).order(ByteOrder.BIG_ENDIAN).asIntBuffer();
    int[] array = new int[ib.remaining()];
    ib.get(array);
    int[] list = new int[array.length];
    for (int i = 0; i < array.length; ++i) {
        list[i] = ColorHandler.getColor(array[i]);
    }
    return new Tuple2<String, int[]>(pair._1.toString(), list);
});
```

Figure 1: Opération mapToPair reconstituant pour chaque clé/localisation les tableaux 1D d'altitudes, puis remplaçant les altitudes par les dégradés de couleurs correspondants.

```
    colorPairRDD.foreach((Tuple2<String, int[]> pair) -> {
        BufferedImage bufferedImage = new BufferedImage(INITIAL_SIDE_SIZE, INITIAL_SIDE_SIZE, BufferedImage.TYPE_INT_ARGB);
        for (int i = 0; i < pair._2.length; ++i) {
            int y = i % INITIAL_SIDE_SIZE;
            int x = i / INITIAL_SIDE_SIZE;
            int color = pair._2[i];
            bufferedImage.setRGB(x, y, color);
        }
        String location = computeLocation(pair._1.toString());
        int[] coordinates = computeCoordinates(location);
        int posY = coordinates[0];
        int posX = coordinates[1];
        int zoom = 1;
        String row = zoom + "/" + posY + "/" + posX;
        byte[][] data = computePictureData(row, bufferedImage);
        HBasePictures.storePictureData(data);
    });
```

Figure 2: Boucle foreach construisant les images 1201x1201 à partir des couleurs attribuées précédemment avant de les stocker dans HBase.

3.1.2 SequenceFile

```
JavaPairRDD<Text, IntArrayWritable> pairRDD = context.sequenceFile("hdfs://young:9000/user/pascal/dem3seq/", Text.class, IntArrayWritable.cla

/** Images 1201x1201 (SequenceFile) **/

JavaPairRDD<String, int[]> colorPairRDD = pairRDD.mapToPair((Tuple2<Text, IntArrayWritable> pair) -> {
    int[] list = new int[pair._2.getArray().length];
    for (int i = 0; i < pair._2.getArray().length; ++i) {
        list[i] = ColorHandler.getColor(pair._2.getArray()[i]);
    }
    return new Tuple2<String, int[]>(pair._1.toString(), list);
});
int numPartitions = conf.getInt("spark.executor.instances", 10);
colorPairRDD = colorPairRDD.repartition(numPartitions);
colorPairRDD.foreach((Tuple2<String, int[]> pair) -> {
    BufferedImage bufferedImage = new BufferedImage(INITIAL_SIDE_SIZE, INITIAL_SIDE_SIZE, BufferedImage.TYPE_INT_ARGB);
    for (int i = 0; i < pair._2.length; ++i) {
        int y = i % INITIAL_SIDE_SIZE;
        int x = i / INITIAL_SIDE_SIZE;
        int color = pair._2[i];
        bufferedImage.setRGB(x, y, color);
    }
    String location = computeLocation(pair._1.toString());
    int[] coordinates = computeCoordinates(location);
    int posY = coordinates[0];
    int posX = coordinates[1];
    int zoom = 1;
    String row = zoom + "/" + posY + "/" + posX;
    byte[][] data = computePictureData(row, bufferedImage);
    HBasePictures.storePictureData(data);
});
```

On a une première opération mapToPair remplaçant pour chaque clé/localisation les altitudes par les dégradés de couleurs correspondants. Ensuite on modifie la répartition des blocs entre les exécuteurs, puis on boucle sur un foreach construisant les images 1201x1201 à partir des couleurs attribuées précédemment avant de les stocker dans HBase.

3.2 Images 256x256

```
JavaPairRDD<String, short[]> tmpPairRDD = pairRDD.mapToPair((Tuple2<Text, IntArrayWritable> pair) -> {
    int[] intArray = pair._2.getArray();
    short[] shortArray = new short[intArray.length];
    for(int i = 0; i < intArray.length; ++i) {
        shortArray[i] = (short)intArray[i];
    }
    return new Tuple2<>(pair._1.toString(), shortArray);
});
```

Figure 3: Conversion des données en types sérialisables.

Les types Text et IntArrayWritable (classe écrite par Marie Pascal et Marc Clervaux) n'étant pas sérialisables en Spark, nous avons produit un premier JavaPairRDD en faisant un mapToPair pour convertir les clés Text en String et les valeurs IntArrayWritable en short[]. Nous avons choisi d'utiliser des short plutôt que des int étant donné que les valeurs des altitudes sont comprises entre 0 et 8000, cela diminue ainsi la taille de nos données.

Il nous faut maintenant faire en sorte de traiter des zones de 256x256 au lieu de celles de 1201x1201. Pour chaque altitude, nous devons regarder sa position absolue dans l'ensemble des données et définir : dans quelle zone 256x256 elle se situe ; sa position relative dans cette zone (afin de pouvoir construire l'image par la suite).

```
JavaPairRDD<String, short[][]> finalPairRDD = tmpPairRDD.flatMapToPair((Tuple2<String, short[]> pair) -> {
    List<Tuple2<String, short[][]>> list = new ArrayList<>();
    String location = computeLocation(pair._1.toString());
    int[] coordinates = computeCoordinates(location);
    int y0 = coordinates[0] * INITIAL_SIDE_SIZE;
    int x0 = coordinates[1] * INITIAL_SIDE_SIZE;
    Map<String, short[][]> map = new HashMap<>();
    for (int i = 0; i < pair._2.length; ++i) {
        int posY = y0 + i / INITIAL_SIDE_SIZE;
        int posX = x0 + i % INITIAL_SIDE_SIZE;
        short coordY = (short) (posY % FINAL_SIDE_SIZE);
        short coordX = (short) (posX % FINAL_SIDE_SIZE);
        short altitude = (short) pair._2[i];
        int keyY = posY / FINAL_SIDE_SIZE;
        int keyX = posX / FINAL_SIDE_SIZE;
        String key = keyY + "/" + keyX;
        short[][] values = map.get(key);
        if (values != null) {
            values[0] = ArrayUtils.add(values[0], coordY);
            values[0] = ArrayUtils.add(values[0], coordX);
            values[1] = ArrayUtils.add(values[1], altitude);
            map.replace(key, values);
        }
        else {
            values = new short [2][0];
            values[0] = ArrayUtils.add(values[0], coordY);
            values[0] = ArrayUtils.add(values[0], coordX);
            values[1] = ArrayUtils.add(values[1], altitude);
            map.put(key, values);
        }
    }
    for (String key: map.keySet()) {
        list.add(new Tuple2<>(key.toString(), map.get(key)));
    }
    return list.iterator();
});
```

Figure 4: Opération flatMapToPair qui attribue de nouvelles clés de type correspondant aux localisations des zones 256x256, et ajoute les coordonnées relatives aux localisations des altitudes aux valeurs.

On obtient un JavaPairRDD<String, short[][]>. Les clés contiennent des localisations de type "1/.../..." (avec 1 la valeur par défaut pour le zoom, le premier '...' la position en ordonnée, le deuxième '...' la position en abscisse, valeurs entières). La valeur 1201 n'étant pas un multiple de 256, on se retrouve avec des instances de clés identiques contenant chacune une partie des 256*256 coordonnées/altitudes. On souhaite donc rassembler ces informations sous une seule et unique clé.

```
finalPairRDD = finalPairRDD.reduceByKey((short[][] values1, short[][] values2) -> {
    values1[0] = ArrayUtils.addAll(values1[0], values2[0]);
    values1[1] = ArrayUtils.addAll(values1[1], values2[1]);
    return values1;
});
```

Figure 5: Opération reduceByKey qui permet de rassembler les informations de chaque zone 256x256.

```
finalPairRDD.foreach((Tuple2<String, short[][]> pair) -> {
    BufferedImage bufferedImage = new BufferedImage(FINAL_SIDE_SIZE, FINAL_SIDE_SIZE, BufferedImage.TYPE_INT_ARGB
    for (int i = 0; i < pair._2[1].length; ++i) {
        int y = pair._2[0][2 * i];
        int x = pair._2[0][2 * i + 1];
        int altitude = pair._2[1][i];
        int color = ColorHandler.getColor(altitude);
        bufferedImage.setRGB(x, y, color);
    }
    int zoom = 1;
    String row = zoom + "/" + pair._1;
    byte[][] data = computePictureData(row, bufferedImage);
    HBasePictures.storePictureData(data);
});
```

Figure 6: Boucle foreach qui construit les images 256x256 à partir des coordonnées et des altitudes (calcul des dégradés de couleurs correspondant), puis stocke les images dans HBase.

3.3 Images 256x256 avec niveaux de zoom

On réitère la conversion des données en types sérialisables (cf. Figure 3 mapToPair). On retrouve ci-après une démarche similaire à celle de la partie précédente permettant toujours d'obtenir des images 256x256, mais pour différents niveaux de zoom.
Une autre solution que celle présentée ici aurait été de faire des flatMapToPair et des reduceByKey à la chaîne pour chaque niveau de zoom, en commençant par le plus grand. On peut ainsi facilement diviser les zones, construire et ramener à la taille 256x256 les images résultantes entre chaque niveau de zoom.

```
for (int zl = 1; zl <= ZOOM_LEVELS; ++zl) {
    final int zoom_level = zl;
    JavaPairRDD<String, short[][]> finalPairRDD = tmpPairRDD.flatMapToPair((Tuple2<String, short[]> pair) -> {
        List<Tuple2<String, short[][]>> list = new ArrayList<>();
        String location = computeLocation(pair._1.toString());
        int[] coordinates = computeCoordinates(location);
        int y0 = coordinates[0] * INITIAL_SIDE_SIZE;
        int x0 = coordinates[1] * INITIAL_SIDE_SIZE;
        int zoom = (int) Math.pow(2, zoom_level);
        Map<String, short[][]> map = new HashMap<>();
        for (int i = 0; i < pair._2.length; ++i) {
            int posY = y0 + i / INITIAL_SIDE_SIZE;
            int posX = x0 + i % INITIAL_SIDE_SIZE;
            short coordY = (short) (posY % (FINAL_SIDE_SIZE * zoom));
            short coordX = (short) (posX % (FINAL_SIDE_SIZE * zoom));
            short altitude = (short) pair._2[i];
            int keyY = posY / (int) (FINAL_SIDE_SIZE * zoom);
            int keyX = posX / (int) (FINAL_SIDE_SIZE * zoom);
            String key = keyY + "/" + keyX;
            short[][] values = map.get(key);
            if (values != null) {
                values[0] = ArrayUtils.add(values[0], coordY);
                values[0] = ArrayUtils.add(values[0], coordX);
                values[1] = ArrayUtils.add(values[1], altitude);
                map.replace(key, values);
            }
            else {
                values = new short [2][0];
                values[0] = ArrayUtils.add(values[0], coordY);
                values[0] = ArrayUtils.add(values[0], coordX);
                values[1] = ArrayUtils.add(values[1], altitude);
                map.put(key, values);
            }
        }
        for (String key: map.keySet()) {
            list.add(new Tuple2<>(key.toString(), map.get(key)));
        }
        return list.iterator();
    });
```

Figure 7: Opération flatMapToPair qui, pour chaque Z, attribue de nouvelles clés de type correspondant aux localisations des zones $(256 * 2^Z)x(256 * 2^Z)$, et ajoute les coordonnées relatives aux localisations des altitudes aux valeurs.

```
finalPairRDD = finalPairRDD.reduceByKey((short[][] values1, short[][] values2) -> {
    values1[0] = ArrayUtils.addAll(values1[0], values2[0]);
    values1[1] = ArrayUtils.addAll(values1[1], values2[1]);
    return values1;
});
```

Figure 8: Opération reduceByKey permettant de rassembler les informations par zone et par zoom.

```
finalPairRDD.foreach((Tuple2<String, short[][]> pair) -> {
    int zoom = (int) Math.pow(2, zoom_level);
    BufferedImage bufferedImage = new BufferedImage(FINAL_SIDE_SIZE * zoom, FINAL_SIDE_SIZE * zoom, BufferedImage.TYPE_INT_ARGB);
    for (int i = 0; i < pair._2[1].length; ++i) {
        int y = pair._2[0][2 * i];
        int x = pair._2[0][2 * i + 1];
        int altitude = pair._2[1][i];
        int color = ColorHandler.getColor(altitude);
        bufferedImage.setRGB(x, y, color);
    }
    if (zoom_level > 1) {
        bufferedImage = ImageHandler.resizeImage(bufferedImage, zoom);
    }
    String row = zoom_level + "/" + pair._1;
    byte[][] data = computePictureData(row, bufferedImage);
    HBasePictures.storePictureData(data);
});
```

Figure 9: Boucle foreach qui construit les images $(256 * 2^Z)x(256 * 2^Z)$ à partir des coordonnées et des altitudes (calcul des dégradés de couleurs correspondant), les ramène à une taille de 256x256, puis les stocke dans HBase.

3.4 Stockage des images avec HBase

Nous avons des tables où nous stockons les images 1201 x 1201 ('famane1201' et 'famane1201_hgt') et 256x256 ('famane256'). Nous n'avons pu générer des images sur des petits jeux de données seulement et n'avons rien pu produire pour les zooms (cf. Problèmes rencontrés).
Les rows sont remplies avec des informations sur le niveau de zoom et la localisation ".../.../..." (premier '...' niveau zoom, deuxième '...' position en ordonnée, troisième '...' position en abscisse, valeurs entières). Il y a une column family : 'image' avec la column 'png'.

3.5 Récupération des images pour Leaflet

Pour récupérer les images, nous utilisons le module hbase-rpc-client pour la connexion à HBase. Le serveur node est lancé sur le port 3000.

```
let hbase = require('hbase-rpc-client');

//instantiate client
const client = hbase({
    zookeeperHosts: ['young'],
    zookeeperRoot: '/hbase'
});
```

Figure 10: Configuration du client hbase

Nous avons défini les route suivantes :

- */map* : pour la visualisation de la map avec Leaflet.
- */image/z/x/y* : une route pour envoyer toutes les images sur la map.

```
/* GET the map page */
router.get('/map', function(req, res, next) {
  res.render('map', { title: 'Map Page' });
});

/* GET Specific image from HBASE TABLE DB */
router.get('/image/:z/:y/:x', (req, res, next) => {
    if (req.params.x && req.params.y && req.params.z) {

      //let rowID = req.params.x +','+ (180-Number(req.params.y));
        let key = { "z": 1,

            "y": 180 - Number(req.params.y),
            "x": Number(req.params.x)};

        rowKey =key.z+'/'+key.y+'/'+key.x
        console.log(rowKey);
        //rowKey= '1/170/200'
        get = new hbase.Get(rowKey)

        client.get('famane1201_seq', get,function(error,value1) {

           try {
            if(value1 !== null){
                let val = value1.columns[0].value;
                console.log(val);
                let data = Buffer.from(val, 'base64');
                res.contentType('image/png');
                res.status(200).send(val);
            }
            else {
                res.status(400).sendFile(path.join(__dirname, '../public/default.jpg'));
            }
           } catch (error) {

           }


        });
    }
```

Figure 11: Récupération de l'image en fonction des coordonnées et du zoom.

On fixe le paramètre zoom à 1 quelle que soit l'entrée reçu, mais pour avoir accès à l'ensemble des coordonnées courantes qui fournissent le bon accès aux images, nous l'avons fixé à 9 dans Leaflet. Ce qui donne des valeurs de 0 à 360.

```
extends layout
block content
    #map
    script.
        var map = L.map('map', {center: [0,0], zoom: 1, minZoom:0, maxZoom: 3});
        L.TileLayer.HeightMap = L.TileLayer.extend({
            getTileUrl: function(coords) {
                return "/image/"+coords.z+"/"+coords.x+"/"+coords.y;
            }
        });
        L.tileLayer.heightMap = function() {
            return new L.TileLayer.HeightMap();
        }
        L.tileLayer.heightMap().addTo(map);
```

Figure 12: Vue de la map (initialisation et appel des images en fonction des coordonnées courantes).

4 Résultats

Nous avons réussi à générer les images par 1201, toutefois l'écriture n'est pas arrivé jusqu'au bout à cause des différentes coupures sur HBase. Nous sommes en mesure d'afficher une image avec notre API comme vous pouvez le voir en dessous en requêtant à l'adresse suivante : [machine]:9000/image/z/y/x

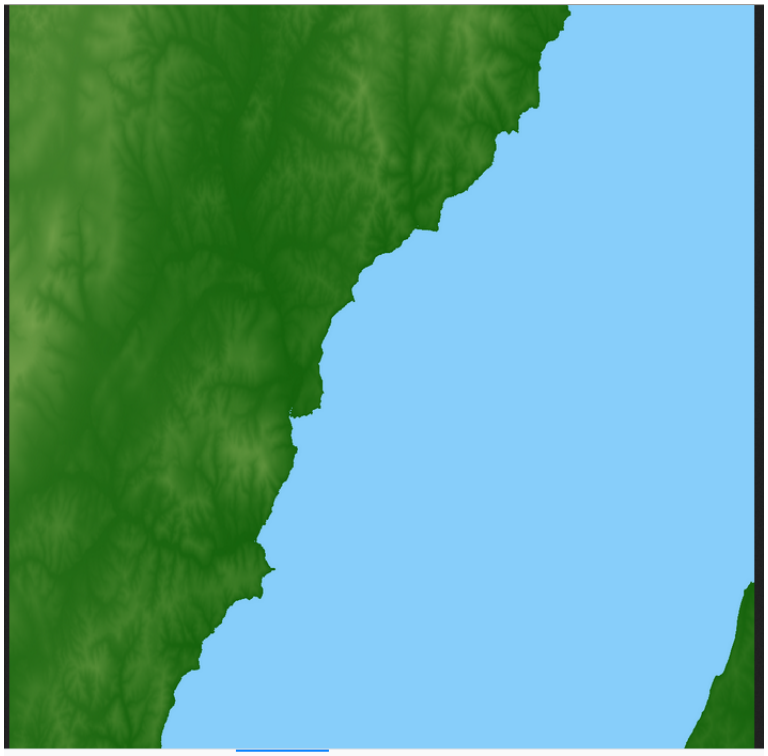

Figure 13: Excemple de requête GET sur une image de la table.

Sur l'image ci-dessus, on voit une partie de la map générée étant donnée que nous n'avons pas toutes les images en base.

Figure 14: Récupération d'images sur la map dynamiquement

5 Problèmes rencontrés

Nous avons rencontré des problèmes sur la configuration de HBase qui nécessitait de source deux fois le fichier user-env.sh afin d'éviter toute exception sur l'accès àux librairies permettant d'utiliser HBase pour intéragir avec la base depuis les traitements Spark.
Problèmes sur le nombre de connexions à la table HBase, les accès avant écriture font une connexion à chaque fois ce qui fait que l'on n'arrive pas à écrire toutes les entrées dans la table HBase, le nombre de connexions simultannée à HBase est limitée. L'idée aurait été de n'avoir q'une seule connexion au moment de l'écriture et de la fermer lorsqu'il n'y a plus d'entrées à sauvegarder.
Les principaux problèmes sont ceux que nous avons rencontrés avec l'utilisation du cluster, et plus particulièrement la répartition des ressources entre les différents groupes (pas toujours très fair-play). Entre extinctions ou surcharges des machines et plantages de HBase, il a été assez difficile de lancer et faire tourner même nos jobs les moins gourmands. Difficile donc de travailler dans de bonnes conditions.