Rapport de projet de Programmation Large Echelle

Amelozara MAHAZOASY, Benoit FAGET, Carlos NEZOUT

January 29, 2019

Contents

1 Contexte du projet

1.1 Existant

Nous disposons d'un jeu de données composé de fichiers dont les noms indiquent des géolocalisations de type "(N or S)XX(W or E)XXX" (avec 'N' North, 'S' South, W 'West' et 'E' East, 'XX' pour la latitude et 'XXX' pour la longitude). Ces fichiers contiennent chacun un tableau 1D de taille 1201*1201 contenant l'ensemble des niveaux de sol (hauteurs) pour une localisation donnée.

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.

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.

3 Solutions proposées

Nous avons décidé de travailler directement sur les SequenceFile générés par Marie Pascal et Marc Clervaux. On récupère à partir de ces SequenceFile : en clés, les paths des fichier '.hgt' fournis ; en valeurs, les tableaux 1D contenant l'ensemble des points de hauteurs correspondant aux fichiers originels en clés. On obtient un JavaPairRDD<Text, IntArrayWritable>.
Dans les parties suivantes, on se sert des classes ColorHandler pour obtenir une couleur à partir d'une altitude et HBasePictures pour se connecter à HBase et stocker les images obtenues dans la table que nous avons définie. La méthode computePictureData permet de préparer nos données avant de les passer à storePictureData de HBasePictures.

3.1 Images 1201x1201

```
JavaPairRDD<Text, IntArrayWritable> colorPairRDD = pairRDD.mapValues((IntArrayWritable x) -> {
    int[] list = new int[x.getArray().length];
    for (int i = 0; i < x.getArray().length; ++i) {
        list[i] = ColorHandler.getColor(x.getArray()[i]);
    }
    return new IntArrayWritable(list);
});
```

Figure 1: Opération mapValues remplaçant pour chaque localisation l'ensemble des altitudes par des dégradés de couleurs correspondants.

```
colorPairRDD.foreach((Tuple2<Text, IntArrayWritable> pair) -> {
    int rowIndex = 0;
    int columnIndex = 0;
    BufferedImage bufferedImage = new BufferedImage(INITIAL_SIDE_SIZE, INITIAL_SIDE_SIZE, BufferedImage.TYPE_INT_ARGB);
    for (int i = 0; i < pair._2.getArray().length; ++i) {
        bufferedImage.setRGB(rowIndex, columnIndex, new Color(pair._2.getArray()[i]).getRGB());
        if ((i + 1) % INITIAL_SIDE_SIZE == 0) {
            columnIndex = 0;
            rowIndex++;
        }
        else {
            columnIndex++;
        }
    }
    int zoom = 1;
    String location = computeLocation(pair._1.toString());
    int[] coordinates = computeCoordinates(location);
    String row = zoom + "/" + (coordinates[1] + LATITUDE_RANGE) + "/" + (coordinates[0] + LONGITUDE_RANGE);
    byte[][] data = computePictureData(row, bufferedImage);
    HBasePictures.storePictureData(data);
});
```

Figure 2: Boucle foreach qui construit les images 1201x1201 à partir des couleurs attribuées précédemment, puis stocke les images 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 donc 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 de conserver des int étant donné que les valeurs d'altitudes sont comprises entre 0 et 8000, diminuant 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);
    int[] coordinates = computeCoordinates(location);
    int y0 = (coordinates[1] + LATITUDE_RANGE) * INITIAL_SIDE_SIZE;
    int x0 = (coordinates[0] + LONGITUDE_RANGE) * INITIAL_SIDE_SIZE;
    Map<String, short[][]> map = new HashMap<>();
    for (int i = 0; i < pair._2.length; ++i) {
        int keyX = (x0 + i % INITIAL_SIDE_SIZE) / FINAL_SIDE_SIZE;
        int keyY = (y0 + i / INITIAL_SIDE_SIZE) / FINAL_SIDE_SIZE;
        String key = keyY + "/" + keyX;
        short coordX = (short) ((x0 + i % INITIAL_SIDE_SIZE) % FINAL_SIDE_SIZE);
        short coordY = (short) ((y0 + i / INITIAL_SIDE_SIZE) % FINAL_SIDE_SIZE);
        short altitude = (short) pair._2[i];
        short[][] values = map.get(key);
        if (values != null) {
            values[0] = ArrayUtils.add(values[0], coordX);
            values[0] = ArrayUtils.add(values[0], coordY);
            values[1] = ArrayUtils.add(values[1], altitude);
            map.replace(key, values);
        }
        else {
            values = new short [2][0];
            values[0] = ArrayUtils.add(values[0], coordX);
            values[0] = ArrayUtils.add(values[0], coordY);
            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 permettant 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) {
        bufferedImage.setRGB(pair._2[0][2 * i] % FINAL_SIDE_SIZE, pair._2[0][2 * i + 1] / FINAL_SIDE_SIZE, ColorHandler.getCol
    }
    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). 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 (travail sur des zones dont la taille est un multiple par puissance de 2 de 256). On peut ainsi facilement diviser les zones, construire et ramener à la taille 256x256 les images résultantes entre chaque niveau de zoom.

```
JavaPairRDD<String, short[][]> finalPairRDD = tmpPairRDD.flatMapToPair((Tuple2<String, short[]> pair) -> {
    List<Tuple2<String, short[][]>> list = new ArrayList<>();
    String location = computeLocation(pair._1);
    int[] coordinates = computeCoordinates(location);
    int y0 = (coordinates[1] + LATITUDE_RANGE) * INITIAL_SIDE_SIZE;
    int x0 = (coordinates[0] + LONGITUDE_RANGE) * INITIAL_SIDE_SIZE;
    Map<String, short[][]> map = new HashMap<>();
    for (int z = 1; z <= NB_ZOOMS; ++z) {
        for (int i = 0; i < pair._2.length; ++i) {
            int zoom = (int) Math.pow(2, z);
            int keyX = (x0 + i % INITIAL_SIDE_SIZE) / (FINAL_SIDE_SIZE * zoom);
            int keyY = (y0 + i / INITIAL_SIDE_SIZE) / (FINAL_SIDE_SIZE * zoom);
            String key = z + "/" + keyY + "/" + keyX;
            short coordX = (short) ((x0 + i % INITIAL_SIDE_SIZE) % (FINAL_SIDE_SIZE * zoom));
            short coordY = (short) ((y0 + i / INITIAL_SIDE_SIZE) % (FINAL_SIDE_SIZE * zoom));
            short altitude = (short) pair._2[i];
            short[][] values = map.get(key);
            if (values != null) {
                values[0] = ArrayUtils.add(values[0], coordX);
                values[0] = ArrayUtils.add(values[0], coordY);
                values[1] = ArrayUtils.add(values[1], altitude);
                map.replace(key, values);
            }
            else {
                values = new short [2][0];
                values[0] = ArrayUtils.add(values[0], coordX);
                values[0] = ArrayUtils.add(values[0], coordY);
                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) -> {
    String[] tokens = pair._1.split("/");
    int z = Integer.parseInt(tokens[0]);
    int zoom = (int) Math.pow(2, z);
    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) {
        bufferedImage.setRGB(pair._2[0][2 * i] % (FINAL_SIDE_SIZE * zoom), pair._2[0][2 * i + 1] / (FINAL_SIDE_SIZE * zoom), ColorHandler.getColor(pair._2[1][i]));
    }
    if (z > 1) {
        Image tmpImage = bufferedImage.getScaledInstance(FINAL_SIDE_SIZE * zoom, FINAL_SIDE_SIZE * zoom, Image.SCALE_FAST);
        BufferedImage resizedImage = new BufferedImage(FINAL_SIDE_SIZE, FINAL_SIDE_SIZE, BufferedImage.TYPE_INT_ARGB);
        Graphics2D g2d = resizedImage.createGraphics();
        g2d.drawImage(tmpImage, 0, 0, null);
        g2d.dispose();
        bufferedImage = resizedImage;
    }
    byte[][] data = computePictureData(pair._1, 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 une table où nous stockons les images 1201 x 1201 (la table 'famane') 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 ".../.../..." (le premier '...' le niveau zoom, le deuxième '...' la position en ordonnée, le troisième '...' la position en abscisse, valeurs entières). Il y a deux column families : 'image' avec la column 'png' ; 'location' avec les columns 'x' et 'y'.

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 Specific image from HBASE TABLE DB */
router.get('/image/:z/:x/:y', (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": parseInt(req.params.z),
            "x": parseInt(req.params.x),
            "y": parseInt(req.params.y)};

        rowKey =key.z+'/'+key.y+'/'+key.x
        get = new hbase.Get(rowKey)
        key = { "z": parseInt(req.params.z),
            "x": parseInt(req.params.x),
            "y": parseInt(req.params.y)};

        client.get('famane', get,function (err,value1) {
            if (err) {
                res.sendFile(path.join(__dirname, '../public/default.png'))
            }
            else {
                if(value1 !== null){
                    let val = value1.columns[0].value;
                    let data = new Buffer(val, 'base64');
                    res.contentType('image/jpeg');
                    res.send(data);
                }
            }

        });
    }
});
```

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

```
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 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.
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.