

Projet PLE

Louis Leduc - Manon Philippot

Janvier 2019

1 Introduction

L'objectif du projet est de travailler à partir de données géolocalisées de grande taille pour fournir des visualisations interactives via une interface Web ou une application desktop connectée à l'infrastructure Big Data.

Les données sont stockées sur HDFS et nous avons choisi d'utiliser Spark et HBASE pour mener à bien la partie "BigData" de ce projet.

Le jeu de donnée que nous allons utiliser sont des relevés satellitaires de la NASA indiquant le niveau du sol (hauteur) sur l'ensemble du globe. Nous devons aussi les présenter sur une carte interactive LeafLet disposant de niveaux de zooms différents pour pouvoir afficher nos données. Le jeu de donnée est composé d'images de 1201 par 1201 pixels stockés en .hgt.

Notre travail consiste donc à découper ces tuiles de 1201 * 1201, à les coloriser afin d'avoir un rendu allant de bleu pour la mer à marron pour les montagnes en passant par le vert et le jaune, à calculer les différents niveaux de zooms et à les stocker dans HBase, à mettre en place une API desservant ces tuiles et ensuite les intégrer dans LeafLet.

2 Analyse du code

2.1 Importation des données

L'ensemble des données est chargé dans un RDD $\langle String, PortableDataStream \rangle$ avec la fonction `.binaryFiles()` de spark. Chaque élément du RDD est donc un tuple constitué du nom du fichier associé à son flux de bytes.

2.2 Calcul du zoom maximal (Zoom 11 leaflet)

2.2.1 Mappeur

La première étape consiste à convertir chaque élément du RDD chargé en un tuple $\langle Tuple2 \langle Integer, Integer \rangle, short[] \rangle$, le tuple d'Integer correspondant aux coordonnées de la tuile dans un repère dont l'origine se trouve en haut à gauche de la map et le tableau de shorts correspondant à la conversion du

tableau de bytes obtenu à partir du PortableDataStream en entrée en tableau de shorts représentant les hauteurs.

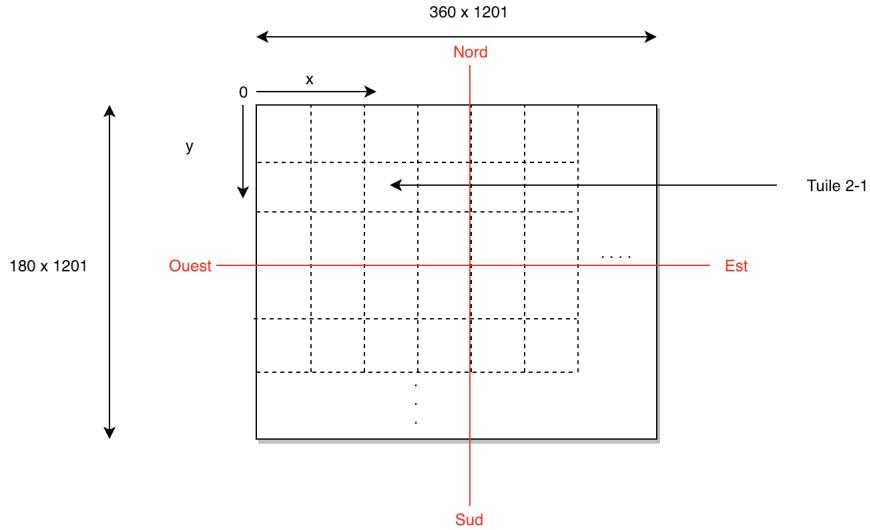


FIGURE 1 – Nouveau repère ayant pour origine le coin en haut à gauche de la map pour désigner chaque tuile.

Pour obtenir les coordonnées de la tuile dans ce nouveau repère, il suffit de parser le nom du fichier et d'évaluer sa position grâce aux coordonnées géographiques contenues dans son nom. Ainsi, une tuile de coordonnées N43W001 aura pour coordonnées dans le nouveau repère ($pos_y = 46; pos_x = 179$).

Pour obtenir le tableau de shorts nous vérifions d'abord que la validité du tableau de bytes en entrée, puis nous utilisons un ByteBuffer pour convertir le tableau de bytes en tableau de shorts (en Big Endian). Si la valeur d'une hauteur est inférieure à 0, elle est remplacée par 0.

2.2.2 Calcul des sous-images pour les tuiles

Leaflet affichant par défaut des tuiles de 256 * 256, nous avons décidé de découper nos tuiles de 1201 * 1201 en sous-tuile de 256 * 256. Faire un agrandissement ou un rétrécissement des tuiles à un multiple de 256 pour ensuite faire notre découpage aurait occasionné une perte d'information, de ce fait nous avons décidé de découper nos tuiles comme si nous découpons la grande map de 360 * 180 tuiles de 1201 * 1201 en sous-tuiles de 256 * 256. Nous avons tout d'abord besoin de savoir comment tombent ces sous-tuiles dans les grandes tuiles afin de savoir comment les découper. Nous voulons ensuite uniquement produire des sous-tuiles de 256 * 256 : si une sous-tuile couvre plusieurs grandes

tuiles il faut que chacune des grandes tuiles concernées en prennent compte et produisent quand même des images de $256 * 256$, quitte à devoir produire des sous-tuiles incomplètes et à les fusionner plus tard.

Soit $(\text{pos_y} ; \text{pos_x})$ la position d'une grande tuile dans la grande map. Nous obtenons à partir de cette position la position relative au début de la map de la première sous-tuile couvrant le haut à gauche de la grande tuile ($\text{new_pos_y} = (\text{pos_y} * 1201) / 256$; $\text{new_pos_x} = (\text{pos_x} * 1201) / 256$) et les coordonnées de l'origine de la grande tuile dans cette première sous-tuile ($\text{new_coord_y} = (\text{pos_y} * 1201) \% 256$; $\text{new_coord_x} = (\text{pos_x} * 1201) \% 256$).

A partir de ces coordonnées, nous pouvons déterminer le nombre de sous-tuiles couvrant la grande tuile en y et en x. En y, cela revient à $\text{nb_sous_tuiles_y} = \text{plus_petit_multiple_de_256_supérieur_à}(1201 + \text{new_coord_y}) / 256$. En x, cela revient à $\text{x} = \text{plus_petit_multiple_de_256_supérieur_à}(1201 + \text{new_coord_x}) / 256$.

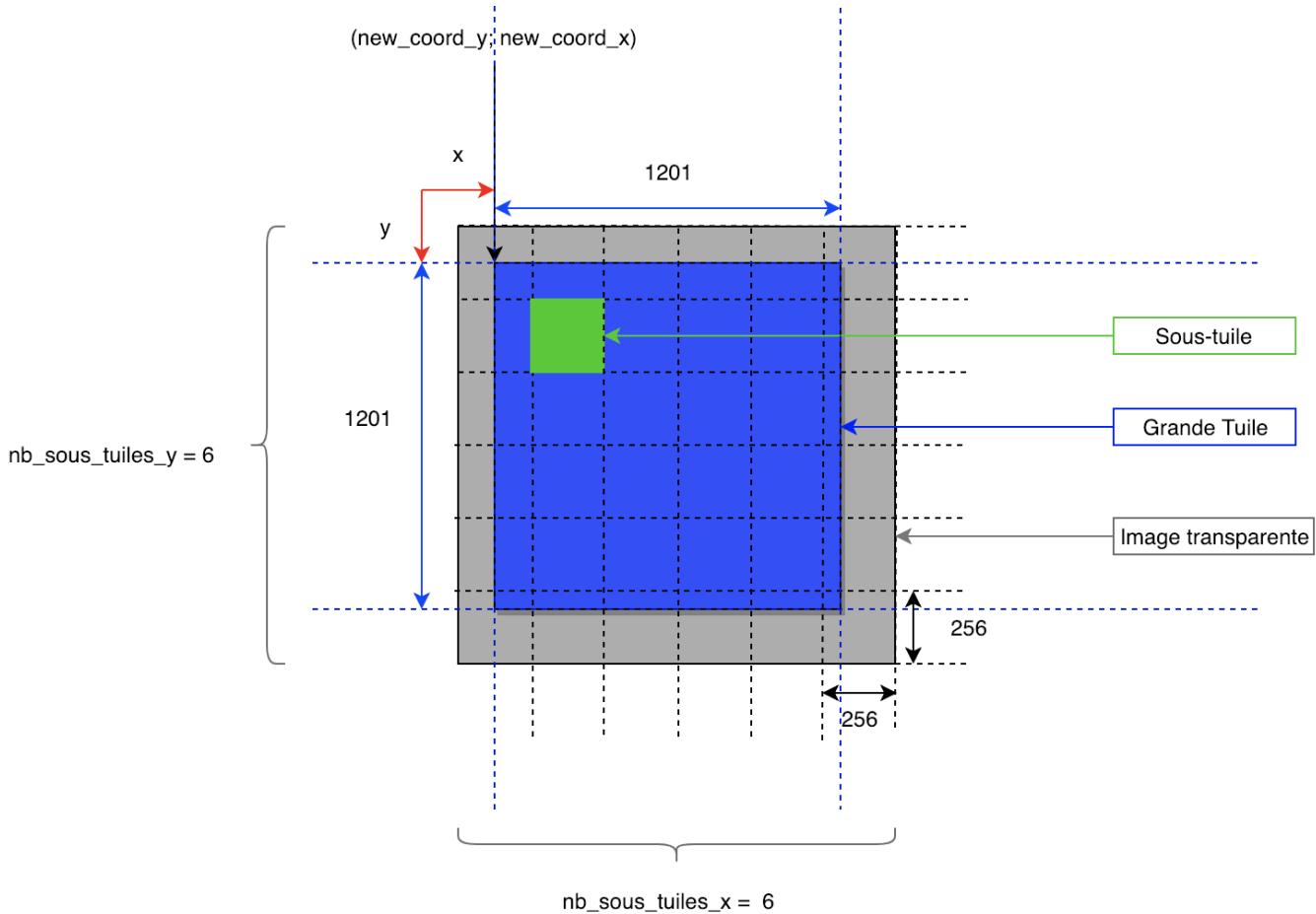


FIGURE 2 – Découpage de la grande tuile en sous-tuiles.

Nous créons ensuite une image transparente de taille (`nb_sous_tuiles_y*256`) * (`nb_sous_tuiles_x*256`) et nous copions la grande tuile dans l'image transparente en commençant à la coordonnée (`new_coord_y` ; `new_coord_x`). Pour ce faire, chaque short (hauteur) de la grande tuile est convertie en une couleur dont la valeur est fonction de la valeur du short. L'image résultante est ensuite divisée en `nb_sous_tuiles_y * nb_sous_tuiles_x` sous-tuiles de `256*256`. Ces sous-tuiles sont ensuite enregistrées dans le RDD résultat, chacune ayant pour clé sa position relative au début de la map sous forme de String ("`new_pos_y-new_pos_x`") et pour contenu associé la sous-tuile sous forme de tableau de bytes.

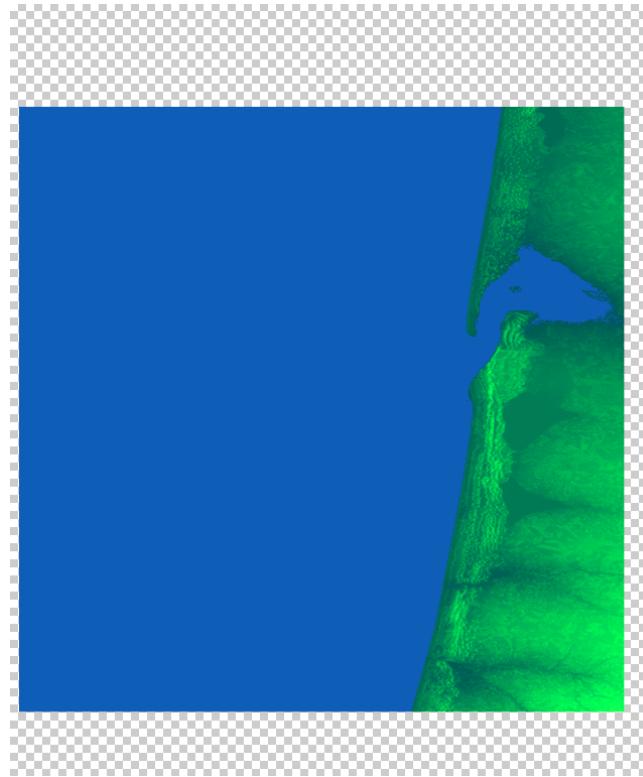


FIGURE 3 – Grande tuile dans l'image transparente multiple de 256 avant le découpage.

2.2.3 Reduceur

Les sous-tuiles de `256 * 256` chevauchant plusieurs grandes tuiles ont la même clé. Par conséquent nous effectuons un `reduceByKey` afin de les combiner et de former une seule image.

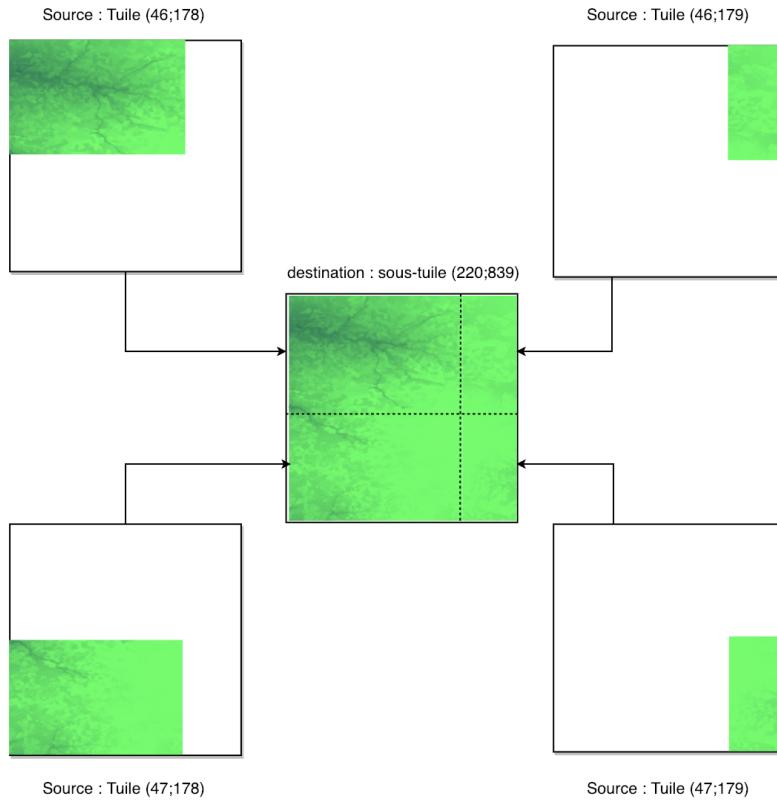


FIGURE 4 – ReduceByKey pour la combinaison de sous-tuiles. Les 4 sous-tuiles ont la même clé (220 ; 839) et sont donc réunies pour former une seule et unique sous-tuile de clé (220 ; 839).

2.3 Calcul des zooms inférieurs

Faire un zoom consiste à réunir 4 tuiles de 256 * 256 qui sont côte à côte sur la map pour former une image de 512 * 512, et à la redimensionner pour obtenir une nouvelle tuile de 256 * 256.

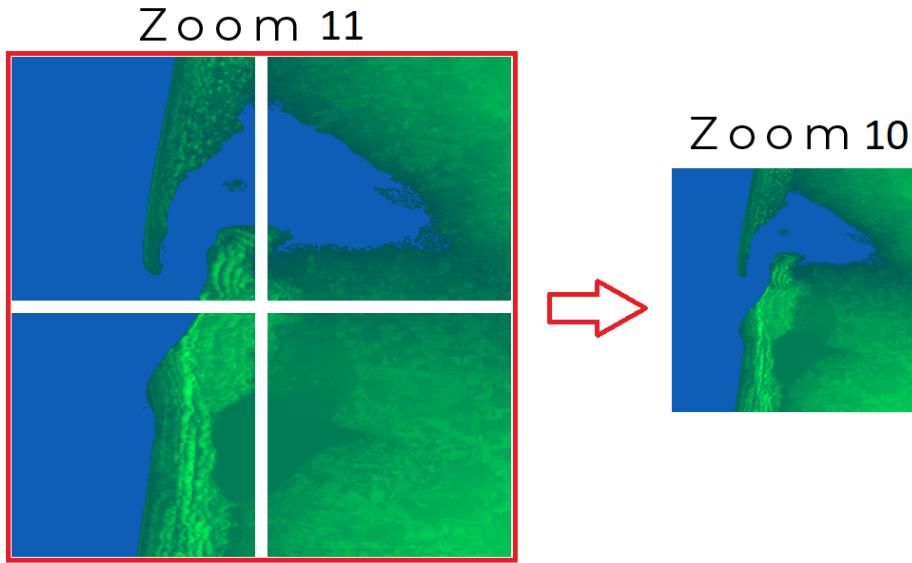


FIGURE 5 – Calcul d'un zoom.

Le calcul des zooms est un processus itératif. Ainsi les étapes décrites ci-dessous sont à répéter pour chaque itération de zoom.

2.3.1 Mappeur

Pour chaque tuile d'un zoom n , le mappeur va calculer la position de la tuile du zoom $n-1$ auquelle elle appartient. Puisque nous rassemblons 2 images en y et en x , il suffit de diviser la position de la tuile au zoom n par 2 pour obtenir la position de la tuile au zoom $n-1$. Le mappeur va ensuite calculer la coordonnée du pixel à partir de laquelle la tuile au niveau n devra être à l'intérieur de la tuile au niveau $n-1$. Cette coordonnée est soit $(0;0)$ pour la tuile en haut à gauche, $(0;256)$ pour la tuile en bas à gauche, $(256;0)$ pour la tuile en haut à droite ou $(256;256)$ pour la tuile en bas à droite. Ces informations sont stockées dans un objet `ZoomTile` décrit ci-dessous. Pour chaque tuile de zoom n en entrée, le mappeur renvoie un tuple avec pour clé la position de la tuile au niveau $n-1$ et pour valeur le `ZoomTile` correspondant.

2.3.2 Classe `ZoomTile`

Une classe `ZoomTile` est créée afin de stocker les informations des tuiles devant être combinées pour former un niveau de zoom inférieur. Chaque `ZoomTile` contient l'image au zoom n sous la forme d'un tableau de bytes, la position de la tuile à laquelle elle doit appartenir au niveau $n-1$, la coordonnée du pixel à partir de laquelle la tuile au niveau n devra être à l'intérieur de la tuile au niveau $n-1$, et un booléen pour savoir si la tuile au zoom n a déjà été transformé

dans le réduceur. En effet certaines tuiles ne passent pas dans le réduceur car ils sont les seules à avoir leur clé, et donc ne sont pas traités et redimensionnés en 512 * 512. Cette classe doit être sérialisable afin de pouvoir l'utiliser en tant que valeur dans un RDD.

2.3.3 Reduceur

Le réduceur récupère toutes les tuiles au zoom n ayant la même clé pour les combiner et former une grande tuile de 512 * 512 regroupant 4 tuiles de 256 * 256. Pour cela il crée une tuile transparente de 512 * 512 et copie les tuiles de zoom n à l'intérieur de celle-ci aux coordonnées spécifiées dans le ZoomTile. Il renvoie ainsi un tuple avec pour clé la position de la tuile au niveau n-1 et pour valeur la tuile de taille 512 * 512 sous la forme d'un tableau de bytes.

2.3.4 Redimensionner l'image

Une fonction map est ensuite appelée sur le reducer afin de redimensionner chaque tuile de taille 512 * 512 en tuiles de taille 256 * 256. Elle donne un rdd avec pour clé la position de la tuile au niveau n-1 et pour valeur la nouvelle tuile de taille 256 * 256 sous forme de tableau de bytes.

2.4 Cache

Afin de ne pas recalculer pour chaque zoom l'intégralité des RDD, nous utilisons la fonction .cache() et .unpersist() de Spark. Lors du calcul d'un zoom n, le RDD résultat est mis en cache afin que le calcul du zoom n-1 puisse s'en resservir. Lorsque le calcul du zoom n-1 est terminé, celui-ci est mis en cache et le précédent RDD est enlevé du cache. Cela nous permet d'obtenir un gain de temps considérable.

2.5 HBase

Une table 'team-rocket-ml' est créée dans HBase (young). Chaque column-family correspond à un zoom (de 'zoom_0' à 'zoom_11'). Chaque row a pour nom la concaténation des coordonnées y et x de la tuile stockée ('y-x'), et chaque cell a pour contenu la tuile correspondante (tableau de bytes).

team-rocket				
row	zoom_0	zoom_1	zoom_11
			

FIGURE 6 – Schéma de la table HBase.

A chaque calcul d'un zoom, les tuiles correspondantes sont stockées dans cette table. A noter qu'ici 'zoom_0' correspond au zoom maximal et 'zoom_11' au zoom minimal. C'est le contraire sur LeafLet où le zoom 0 correspond au zoom minimal et le zoom maximal correspond au zoom maximal. Cette différence sera gérée dans l'API.

2.6 API

Notre API nous permet de récupérer une image sur HBase via un GET :URL de type : /Project/webapi/tile/x/y/z où x et y sont les coordonnées de la tuile et z est le zoom. Ces paramètres sont déterminés automatiquement par LeafLet en fonction des mouvements détectés sur la map.

La route de notre API récupère un tableau de byte depuis HBase et renvoie une image au format .png. Si la tuile cherchée n'existe pas, on renvoie une tuile bleue correspondant à de l'eau (mer / océan).

Cette API est un servlet en java et est déployée avec Tomcat 8.5.

2.7 LeafLet

Pour le front de notre application, nous avons simplement un conteneur avec une map LeafLet à l'intérieur. Celle ci fait appel à notre API pour afficher les différentes tuiles dont elle a besoin pour afficher correctement la map.

Nous avons configuré LeafLet pour avoir un zoom minimum de 0 et un zoom maximum de 11 (plus grand niveau de détail dont nous disposons et correspondant à celui offert par les .hgt).

Lors du calcul des zooms, les fichiers .hgt manquants sont interprétés par de la transparence. De ce fait sur les tuiles générées, les mers et océans ne sont pas bleus mais transparents.

Pour avoir un affichage convenable des mers et océans, nous avons fixé la propriété background-color de LeafLet à la même couleur que la couleur représentant l'eau dans nos tuiles (bleu).

3 Résultats

Après avoir testé notre programme Spark sur un sous-ensemble de données, nous avons calculé une première fois le zoom 11 (zoom maximal) et 10 (zoom précédent) sur la totalité des fichiers, avec succès. Le jour d'après, nous avons lancé de nouveau notre programme pour calculer cette fois-ci l'intégralité des zooms, cependant nous avons eu beaucoup de mal à terminer le job avec succès. Une erreur de connexion avec HBase sur plusieurs machines provoquait une boucle (le programme essayait de se reconnecter à HBase mais il n'y arrivait pas, puis réessayait de se reconnecter à HBase, etc) et empêchait la fin du job. Cette erreur s'est manifestée chez plusieurs autres groupes. Nous ne savons pas ce qui est réellement responsable de cette erreur, mais pensant qu'elle pouvait venir d'une erreur sur une ou plusieurs machines, nous avons lancé notre programme avec seulement 6 executors en espérant ne pas tomber sur ces machines. Après 1h49, le job s'est terminé avec succès et a rempli la table 'team-rocket-ml' sur HBase avec succès.

Les tuiles calculées pour tous les zooms semblent correctes : elles représentent bien la surface de la terre et on peut observer une cohérence entre elles. La coloration est elle aussi cohérente. En raison de notre découpage en tuiles de $256 * 256$, l'image réelle de la surface de la terre n'occupe pas toute la surface de la tuile sur les tout premiers zooms, mais cela ne pose pas de problèmes plus tard avec LeafLet. HBase stocke dans sa totalité 519426 lignes, et l'intégralité des tuiles stockées pèse 17,9 Go. L'API desservant les tuiles est fonctionnel et gère les cas où les coordonnées de la tuile saisie n'existe pas. En faisant appel à notre API, leaflet arrive à afficher avec succès la surface de la terre, ainsi que les zooms calculés.

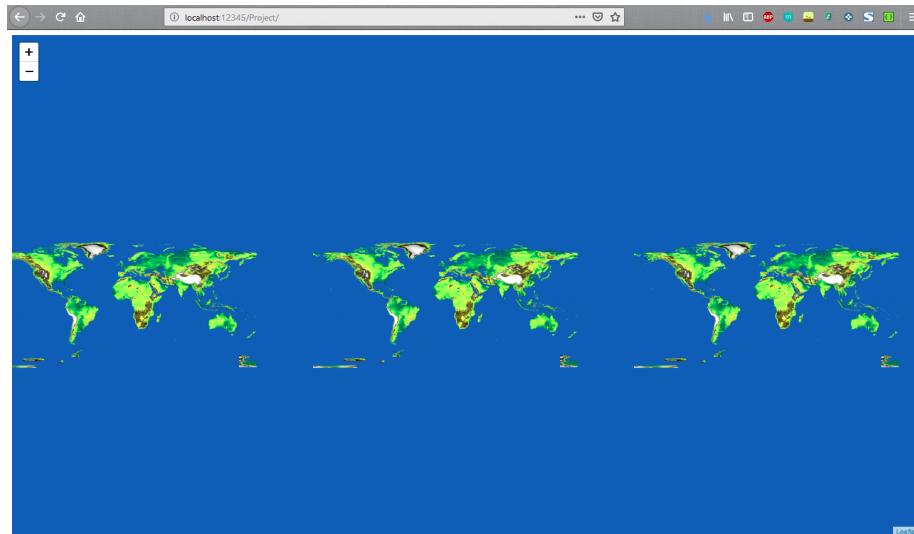


FIGURE 7 – Zoom 1, dans LeafLet, on peut voir la répétition de la map.

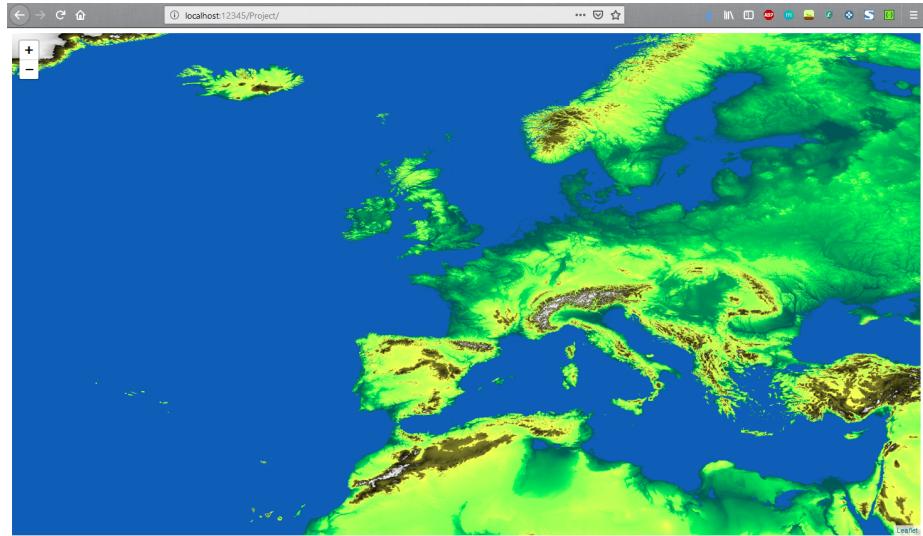


FIGURE 8 – Zoom 5, plus précis

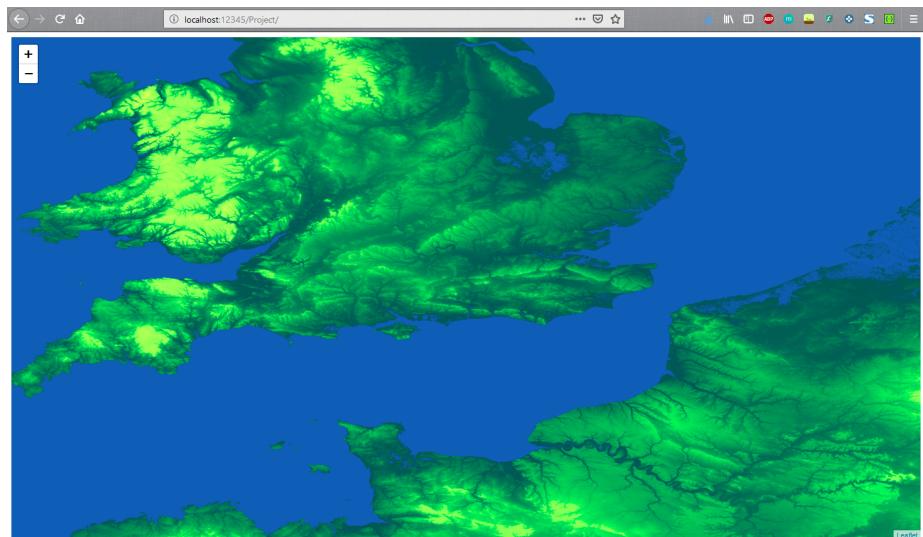


FIGURE 9 – Zoom 8

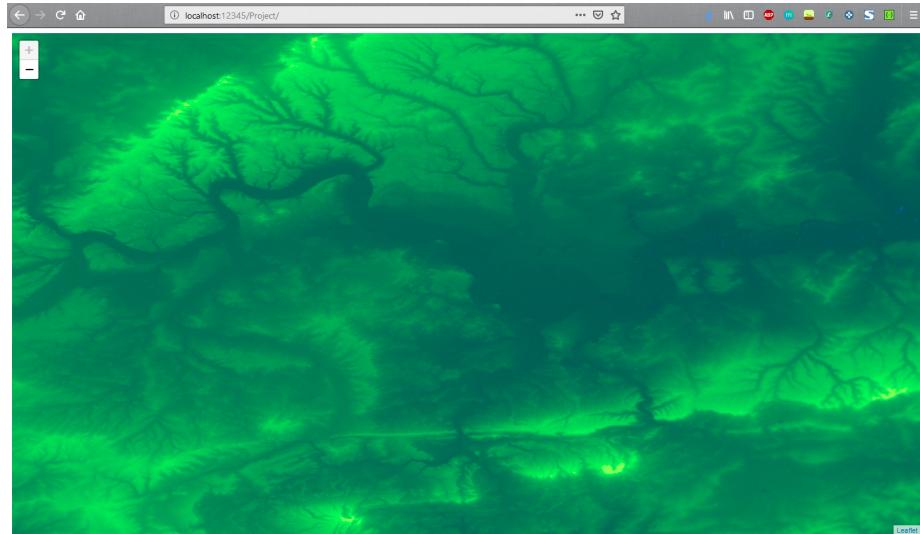


FIGURE 10 – Zoom 11, maximum

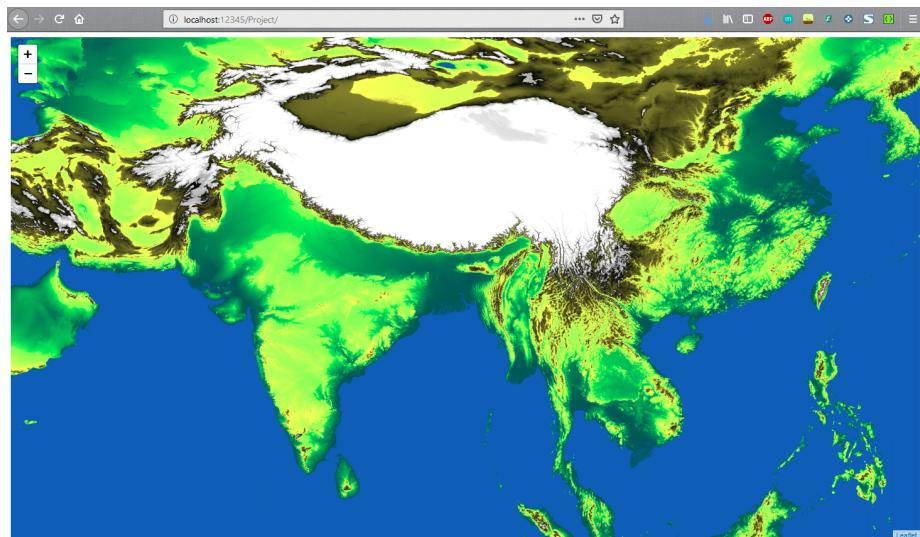


FIGURE 11 – Exemple de coloration.

4 Conclusion

A la suite de ce projet, nous avons un Job Spark qui remplit bien HBase avec tous les niveaux de zooms de manière cohérente pour toutes les tuiles, nous

avons une API qui communique bien avec HBase et qui permet à notre LeafLet de créer une carte sur laquelle nous pouvons effectuer 11 niveaux de zooms (+ zoom initial) différents sans erreur.

A la suite de ce projet nous avons rencontré de nombreux problèmes liés au cluster comme l'échec d'un job à cause d'une machine du cluster qui est corrompue, la concurrence avec les autres élèves pour avoir des executors ou encore l'extinction d'une ou plusieurs machines du cluster par des tiers. Cela a grandement ralenti notre progression pour tester notre code, continuer le projet, ou alors essayer d'y apporter une hausse de performance. Cependant, malgré tout cela nous avons réussi à proposer une solution fonctionnelle.