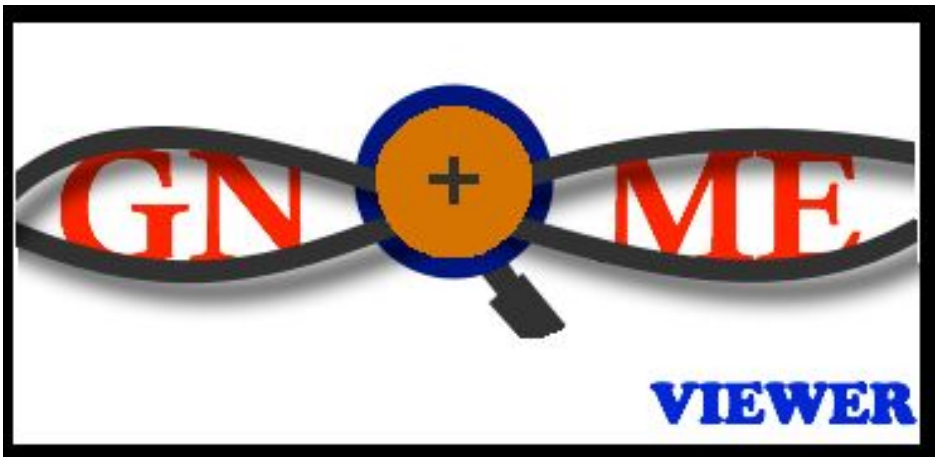
RAPPORT TECHNIQUE
PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
DANS LE CADRE DU COURS GTI795 - LOG795

GOAT - Genetic Output Analysis Tool
GenomeViewer - *An interactive Somatic Mutation Visualizer*



David Guay - GUAD21118902
Émile Filteau-Tessier - FILE05039202
Raphaël Papillon - PAPR17119109
Max St-Onge - STOM15129206

DÉPARTEMENT DE GÉNIE LOGICIEL ET DES TI

Professeur-superviseur

Alain April

MONTRÉAL, 13 AVRIL 2017
HIVER 2017

REMERCIEMENTS

Nous tenons à remercier tous les membres ayant permis de créer les premières itérations de l'application GOAT dont Béatriz Kanzki, Cédric Urvoy et Victor Dupuy.
Tout au long de ce projet, nous avons eu de l'aide de plusieurs personnes que nous tenons à remercier:

Alain APRIL;

Christian DESROSIERS;

Béatriz KANZKI

GOAT - Genetic Output Analysis Tool
GenomeViewer - *An interactive Somatic Mutation Visualizer*

David Guay - GUAD21118902
Émile Filteau-Tessier - FILE05039202
Raphaël Papillon - PAPR17119109
Max St-Onge - STOM15129206

RÉSUMÉ

Les médecins chercheurs en génétique ont besoin d'un outil pertinent et efficace leur permettant de visualiser facilement et rapidement les données les intéressants. Pour chaque visualisation, la quantité de données à traiter est très imposante. D'autant plus, les données sont complexes à représenter. Pour combler à ce manque, le logiciel Genetic Output Analysis Tool (*GOAT*) a été développé. Ce projet est donc la suite de la troisième itération de ce projet, aussi intitulée *GOAT_V3*, réalisée par Victor Dupuy lors de l'année 2016. Lors de cette itération, M. Dupuy a grandement amélioré la maintenabilité et l'évolutivité du projet en séparant ce dernier en plusieurs couches applicatives distinctes.

Cependant, ce faisant, la pile d'éléments logiciels nécessaires à l'exécution du projet a été grandement complexifiée et trop peu de documentation a été faite à ce sujet. Ce manque de documentation rend difficile l'exécution du logiciel, ce qui complique grandement la tâche aux futurs développeurs qui souhaiteraient poursuivre le projet.

Afin de remédier aux nombreux problèmes de la version 3, notre proposition est de séparer le projet en 2 parties. La version 3 de GOAT est utilisable et maintenable, elle sera donc améliorée et pendant ce temps, un nouveau back-end utilisant la technologie ADAM sera développé afin de faire des calculs plus rapide et efficace.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

GOAT : Genetic Output Analysis Tool
JSON : JavaScript Object Notation
SPA : Single-Page application
ORM : Object-relational mapping
MVC : Model-View-Controller
VCF : Variant Call Format
JSX : Java Serialization to XML
CSV : Comma-separated values
CSS : Cascading style sheet
HDFS : Hadoop File System
RSID : Biomarqueur génétique
API : Application programming interface
HTTP : Hypertext Transfer Protocol
SQL: Structured Query Language

Introduction

La technologie étant de plus en plus présente dans nos vies, les chercheurs sont toujours à la recherche de nouvelles façons d'intégrer les dernières tendances dans notre quotidien. Dans certains cas, la technologie s'intègre bien dans la vie de tous les jours d'une personne normale, mais dans d'autres cas elle tente d'améliorer et d'accélérer certains processus dans le but de rendre l'humain plus efficace. Dans le cadre de la médecine moderne, on peut clairement voir comment les dernières technologies permettent de prendre des décisions plus éclairées et même de sauver des vies. Des milliers de chercheurs tentent chaque année de trouver de nouvelle façon de diagnostiquer et de traiter toutes sortes de maladies. Une des maladies les plus suivies de nos jours est bien sûr le cancer. Cette maladie cause en moyenne 216 morts chaque jour, et ce, seulement au Canada. À regarder ces chiffres, il est clair que nous pouvons faire mieux, surtout grâce à la puissance qu'offrent les ordinateurs d'aujourd'hui.

La détection du cancer intéresse beaucoup d'instituts de recherche, car plus cette maladie est détectée tôt, plus elle est facile à traiter. C'est notamment le cas du *Berkeley institutes internationales* qui tente chaque année de repousser les limites des connaissances humaines sur plusieurs sujets médicaux. Ils ne sont toutefois pas seuls, des dizaines de centres de recherche au travers le monde sont à la conquête de nouveaux processus puisqu'il n'existe pas encore de standard.

Dans le cadre de ce projet de fin d'études, notre équipe a tenté de faire le point sur les avancements technologiques des dernières années, et voir comment les nouvelles technologies peuvent améliorer un projet émergeant de Beatriz Kanzki s'appelant GOAT (Genetic Output Analysis Tool). Cet outil a pour principal but de visualiser les différents types de biomarqueurs liés au différent type de cancers. Selon Wikipédia : En génétique, le phénotype est l'ensemble des traits observables d'un individu.[1] Ces derniers permettent donc, de lier une mutation génétique à une position du génome, pour un phénotype donné. Dans ce cas-ci tous les phénotypes sont des mutations observables du génome humain lié au cancer.
Notre mandat était donc d'analyser la solution qu'offre la version trois de GOAT, d'y faire plusieurs recommandations et d'y apporter les modifications nécessaires. Ce rapport fera donc un survol des différentes étapes au travers desquelles notre

[1] https://fr.wikipedia.org/wiki/Ph%C3%A9notype

équipe a dû passer.

Nous débuterons donc par un sommaire de la situation initiale du projet, pour en faire ressortir les exigences fonctionnelles et non fonctionnelles. Nous discuterons ensuite des différentes solutions envisagées par l'équipe, et comment la solution actuelle pourrait être amélioré. Nous ferons donc une analyse approfondie des besoins et comment la solution choisie y répond. Ensuite, nous parlerons des différents outils utilisés lors du développement de notre solution. Par la suite, nous détaillerons notre conception et parlerons des différents détails d'implémentations de la nouvelle itération de GOAT. Nous terminerons donc par plusieurs recommandations pour le futur de ce projet très prometteur.

Situation initiale

Contexte

Au tout début du projet, nous avons fait face à la version #3 du projet GOAT. Ce projet, écrit en Python, permettait d'exécuter plusieurs tâches sur les données sauvegardées préalablement dans la base de données. Les différentes fonctionnalités qui étaient offertes par le logiciel GOAT v3 étaient : AreaSelection, Manhattan et SNPS Genes. Dû au départ d'une partie prenante majeure du projet (le Dr. Hamet), ces trois fonctionnalités seront retirées du projet et remplacées par une nouvelle (qui sera décrite plus loin). Pour ne pas entrer dans trop de détails inutilement, une seule de ces fonctions sera exposée plus en détail ci-bas, soit AreaSelection. Pour commencer, voici à quoi ressemblait la page d'accueil de l'application web :

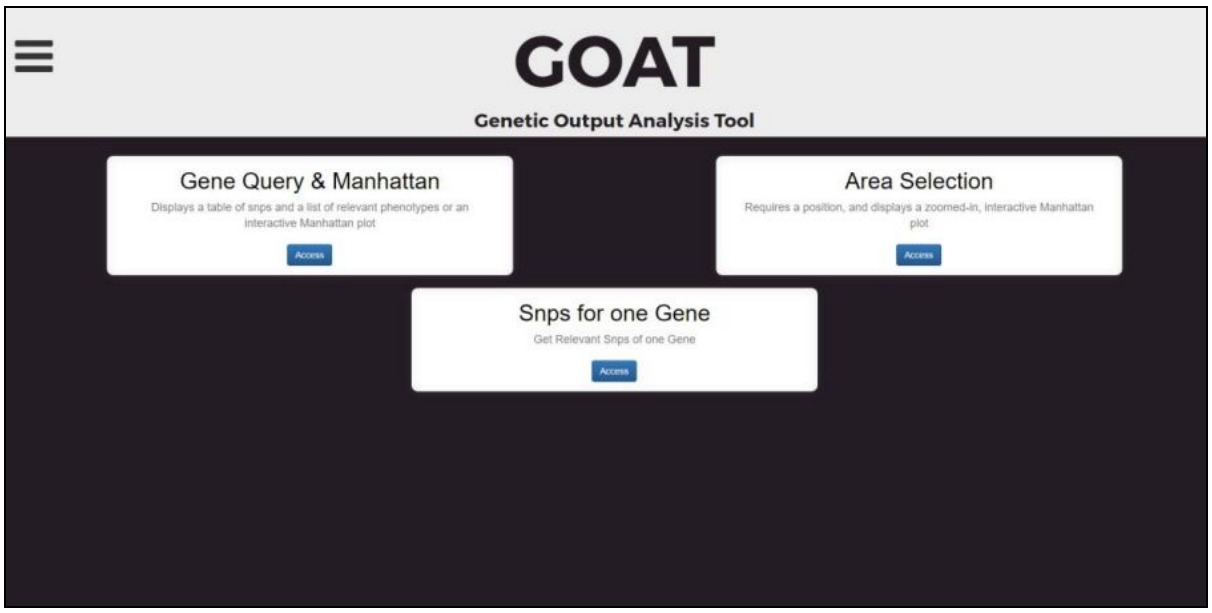


Figure 1: Page d'accueil de l'application GOAT initiale

AreaSelection

Cette fonctionnalité de l'application permettait de sélectionner plusieurs variantes en fonction d'un numéro d'identification unique (rsID), un chromosome et une position dans la séquence. Voici l'interface qui permettait de faire cette sélection :

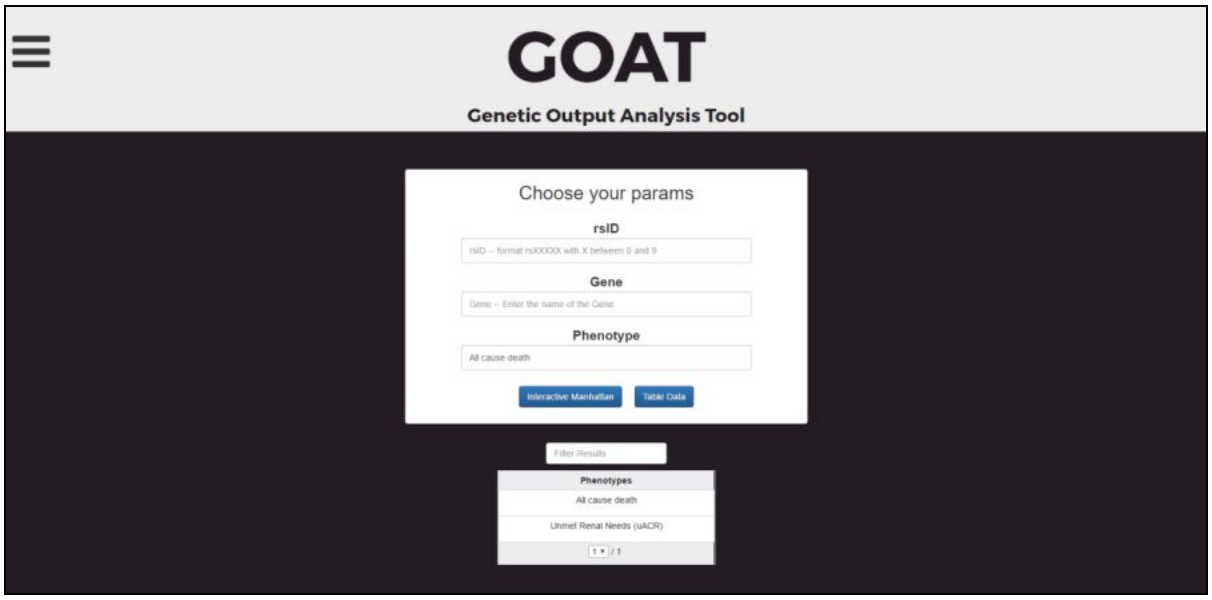


Figure 2: Page d'accueil de la fonctionnalité AreaSelection initiale

Une fois ces trois paramètres entrés et confirmés, l'application web générait le graphique suivant :

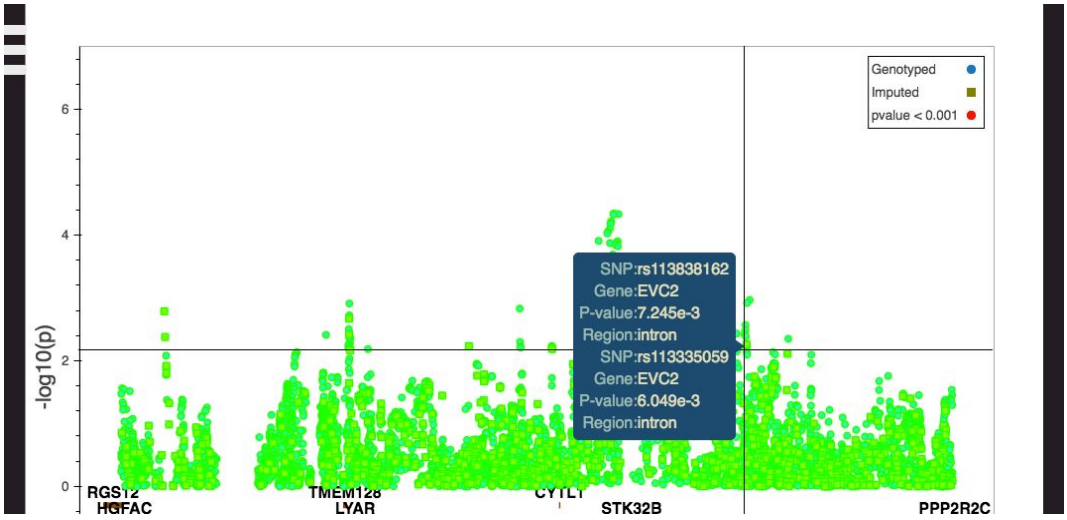


Figure 3: Graphique AreaSelection initial

Ce graphique permet aux experts de comparer visuellement plusieurs variantes présentes dans la base de données. Près de cinq minutes étaient nécessaires pour la génération de ce graphique, ce qui était plutôt problématique.

Manhattan

Cette fonctionnalité de l'application permettait de créer un graphique permettant de voir l'association d'une maladie à une position du génome. Il s'agit d'un test qui permet de comparer personne malade versus une personne saine.

SNPS Genes

Cette fonctionnalité, bien simple, permettait de sélectionner tous les variants qui appartiennent à un même gène.

Même si toutes ces fonctionnalités étaient déjà présentes dans la plus récente version de GOAT, elles présentaient quelques problèmes qui seront abordés dans la prochaine section.

Un aspect très important de la version trois de GOAT, est qu'elle était bâtie sous un format web, pour permettre à un très grand nombre d'utilisateurs d'accéder au système, et ce, sans avoir à se déplacer. C'est pourquoi le logiciel a été bâti à l'aide du cadriciel Django[2]. Ce cadriciel basé sur le langage de programmation Python permettait à la fois de bénéficier des différentes librairies de calcul avancé de python tout en ayant un cadriciel web facile d'utilisation pour faire des requêtes à la base de données.

Base de données MySQL

Toutes ces fonctionnalités utilisaient une base de données MySQL. Pour que le logiciel soit efficace à grande échelle, on doit utiliser des millions de données. En utilisant une base de données relationnelle SQL, la performance se dégrade avec autant de données et le logiciel devient très lent.

[2] https://www.djangoproject.com/

Problématiques du projet

Malgré toutes ses fonctionnalités, la dernière version de GOAT comportait plusieurs problématiques qui l'empêchaient de croître et d'atteindre les objectifs initiaux du projet.

Premièrement, le logiciel n'avait pas de procédure de déploiement. Ceci laissait donc croire que le logiciel était uniquement déployé localement sur le poste qui l'utilise. Ceci est très problématique lorsqu'on regarde l'un des buts de ce logiciel qui est d'être facilement accessible d'un peu partout. Il fallait donc trouver une solution pour que ce logiciel puisse être déployé quelque part, et également décider s'il devait être déployé sur un serveur connu ou dans le cloud.

Deuxièmement, la version trois de GOAT n'utilisait aucun cadriciel de calcul distribué. Ces cadriciels offrent habituellement la possibilité de distribuer les calculs lourds sur plusieurs machines, voir même des grappes de serveur pour accélérer les tâches coûteuses en temps de calcul. Puisque les calculs de comparaisons de phénotypes sont très lourds, le chargement d'une page pouvait donc durer plusieurs secondes, voir plusieurs minutes!

Troisièmement, toutes les données sur lesquelles étaient exécutés les calculs lourds se retrouvaient dans la base de données MySQL. MySQL est un gestionnaire de base de données transactionnelle. Ce type de gestionnaire de base de données n'est donc pas optimisé pour le chargement de très grande quantité de données, mais plutôt pour gérer plusieurs transactions sur une plus petite quantité de données. Ceci n'était donc pas non plus en faveur du chargement rapide des pages.

Quatrièmement, la conception du logiciel n'était pas très bien documentée, ce qui rendait la tâche de compréhension du code très complexe. Aucun test n'était présent dans l'application et il était donc impossible de valider le bon fonctionnement du logiciel. Les différentes logiques d'affaires n'étaient pas non plus séparées à la manière MVC. Cette séparation aurait également permis de mieux isoler les différentes tâches de l'application et de séparer le modèle de données, des requêtes et des vues. Il était donc très difficile de comprendre le fonctionnement du logiciel fourni.

Cinquièmement, avec le départ du Dr. Hamet, les fonctionnalités présentes dans l'application n'étaient plus directement alignées avec la direction du projet et pourraient donc être retirées. Ces fonctionnalités devraient donc être remplacées par les nouvelles fonctionnalités qui rempliront les besoins de la nouvelle direction.

Objectifs et exigences

Front-end

L'objectif était d'ajouter la fonctionnalité GenomeViewer au projet. Premièrement, cette fonctionnalité devait permettre de téléverser un fichier CSV contenant des données génétiques.

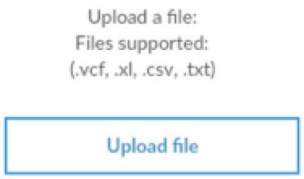


Figure 4: Maquette du téléversement de fichier

Ce fichier pouvait contenir des en-têtes dans n'importe quel ordre. Afin de traiter le fichier de la bonne façon, une correspondance devait être faite entre les en-têtes présents dans le fichier et les données attendues par l'application.

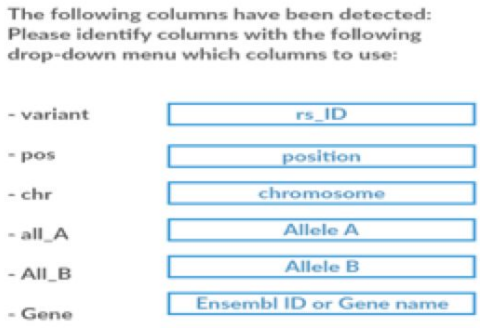


Figure 5: Maquette de la correspondance d'en-têtes de fichier

On devait aussi être en mesure d'entrer manuellement un ou plusieurs identificateurs sans utiliser de fichier.

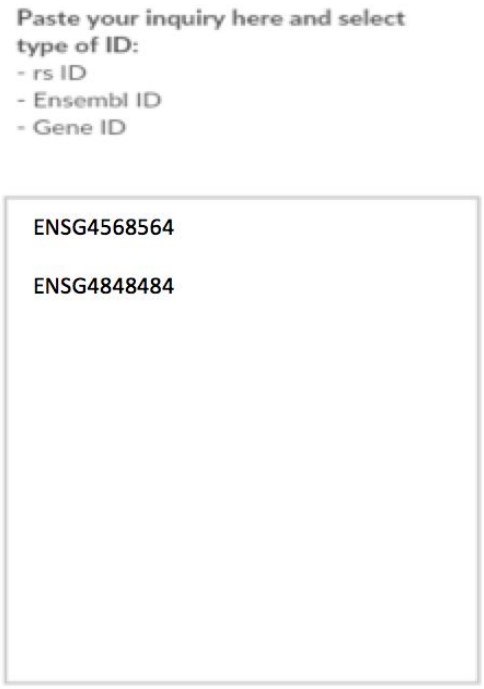


Figure 6: Maquette de l'entrée de plusieurs identificateurs

Ensuite, l'application devait être en mesure de comparer les données génétiques contenues dans le fichier avec la base de données. À partir de cette comparaison, un graphique devait être généré afin de visualiser, pour chaque chromosome, les rsid correspondants.

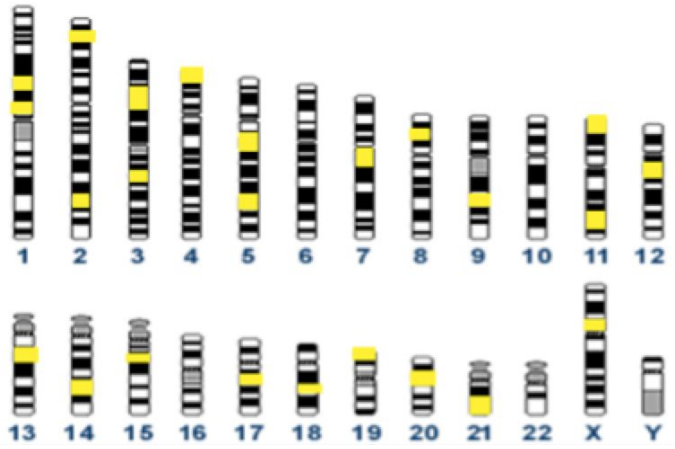


Figure 7: Maquette du graphique de GenomeViewer

Le graphique généré devait présenter les 24 chromosomes ainsi que leur longueur. Les rsid correspondants à la requête devaient se retrouver à la bonne position sur le chromosome correspondant. Il devait être possible de visualiser seulement certains phénotypes ou mutations sur le graphique.

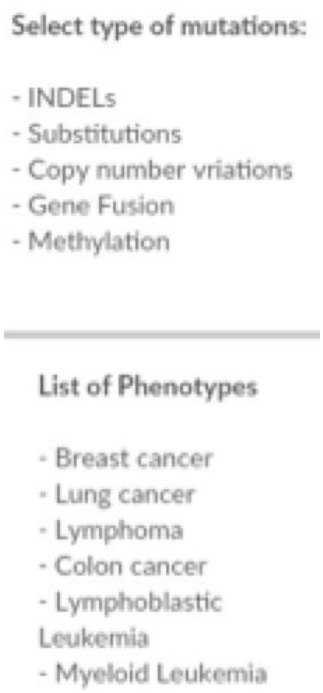


Figure 8: Maquette de la sélection de mutations et phénotypes dans GenomeViewer

Lorsque l'on clique sur un certain rsid, nous devions être en mesure d'obtenir beaucoup plus d'informations. L'interface suivante présente ce qui devrait être fait:

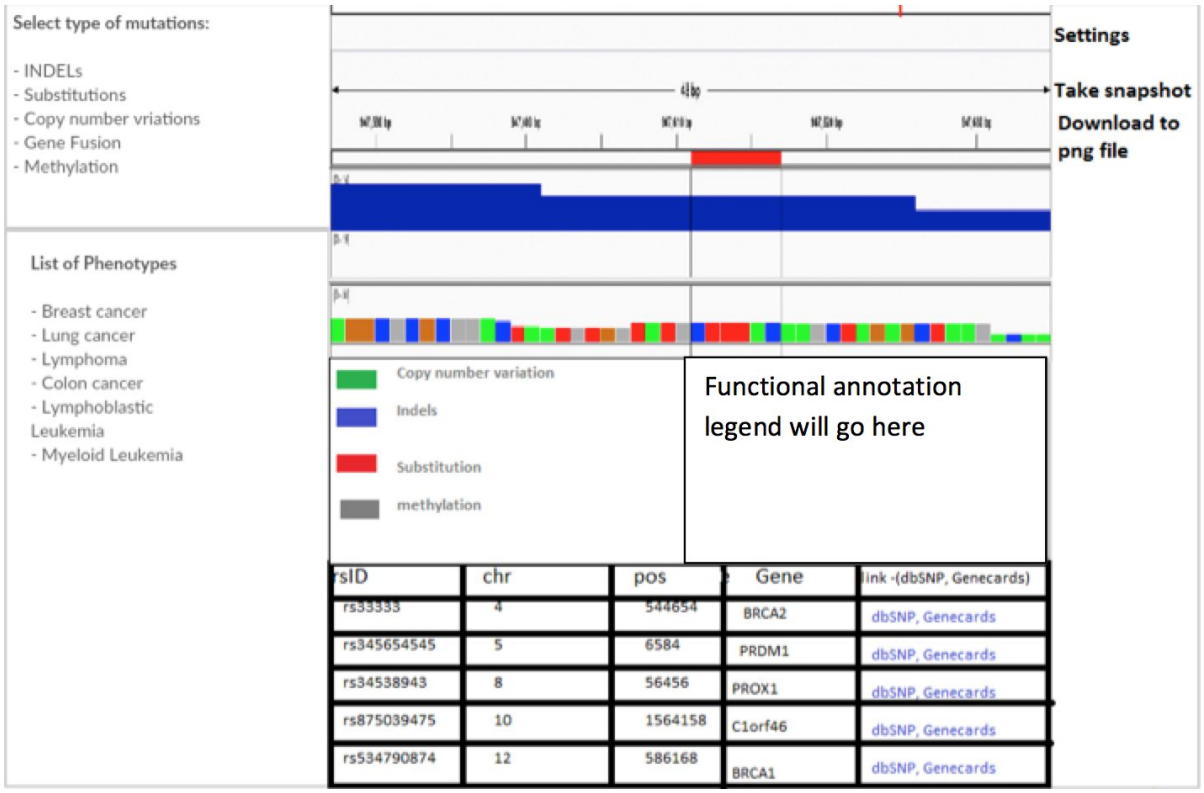


Figure 9: Maquette de la dernière étape de GenomeViewer

Puisque nous avons travaillé de manière itérative et que nous ne nous sommes point rendus à cette étape, nous ne serons pas en mesure de l'expliquer dans ce rapport. Pour avoir plus de détail sur cette partie de la fonctionnalité, il faudra communiquer avec Béatriz Kanzki.

Back-end

Du côté du back-end, plusieurs objectifs étaient fixés.

Il était premièrement question d'accélérer le temps de traitement des différentes fonctionnalités puisqu'il n'était pas viable d'attendre aussi longtemps pour un chargement de page. Pour ce faire, il fallait donc attaquer deux aspects importants de l'application, le stockage de donnée et le traitement des données. Il nous était toutefois exigé de conserver la plateforme Django comme cadriciel web.

Il était aussi important pour nous de fournir une solution clé en main. Lors du début du projet, beaucoup de problèmes sont survenus avec la plateforme GOAT. Afin d'installer le projet sur un ordinateur, cela a pris plusieurs heures et à faire perdre beaucoup de temps à l'équipe. Il était donc important pour l'équipe de créer une plateforme qui pourrait être déployée sur n'importe quel ordinateur rapidement et efficacement. Tout au long du projet, une documentation sur notre travail ainsi que les problèmes rencontrés devra être faite.

Un autre objectif était d'expérimenter avec le cadriciel de traitement de données ADAM. Ce cadriciel permet notamment d'exécuter des calculs sur des données génétiques à l'aide de l'engin de calcul SPARK reconnu pour sa rapidité d'exécution et son extensibilité sur des grappes. Plateforme misant sur l'évolutivité et la compatibilité est un atout pour ce genre de travail qui se voit "open-source".

Le deuxième grand objectif du back-end était de rendre la plateforme disponible sur le web. Nous devions donc explorer les différentes possibilités pour rendre le logiciel accessible tout en conservant certains standards de sécurité et en gérant les accès. La plateforme choisie devait à la fois être performante, résiliente, et permettre une extensibilité facile et transparente.

Afin de bien représenter la configuration souhaitée, un diagramme a été créé:

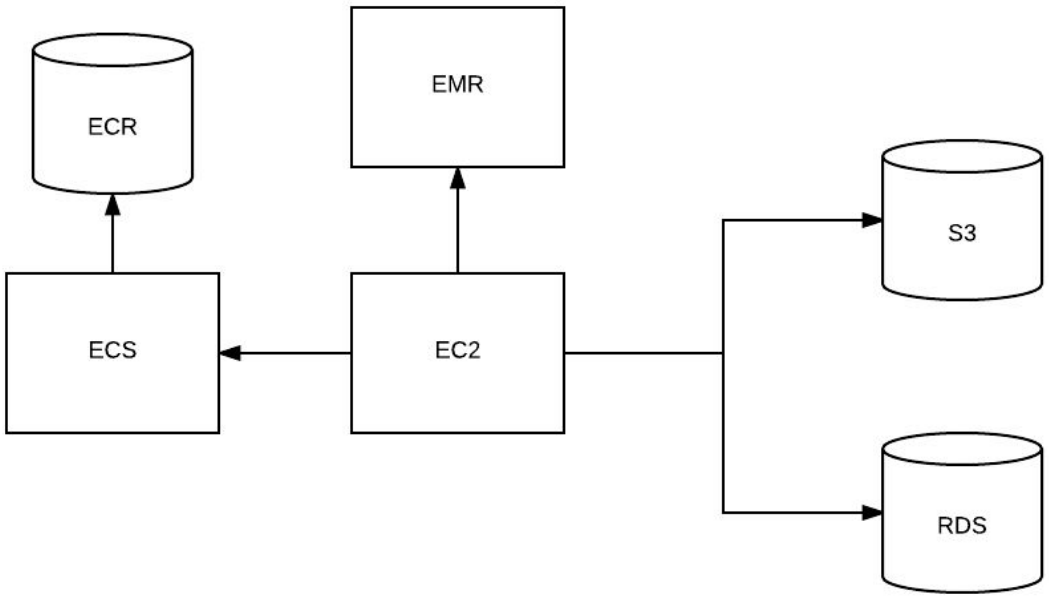


Figure 10: Diagramme d'infrastructure AWS souhaitée

Solutions

Front-end

Tel que vu dans la mise en situation, la version 3 de GOAT était déjà entièrement fonctionnelle et permettait, à partir de certains paramètres d'entrées, d'afficher un graphique. Afin d'accélérer le processus de développement de la nouvelle fonctionnalité, il est évident que le plus de code possible a été réutilisé. Ainsi, la création de la nouvelle fonctionnalité GenomeViewer fut grandement inspirée de la fonctionnalité déjà présente AreaSelection.

Initialement, nous avions l'intention de faire l'affichage du nouveau graphique avec Bokeh. Cependant, dû à des difficultés qui seront expliquées plus bas, nous avons finalement dû nous résoudre à changer de technologie d'affichage graphique. C'est pourquoi nous avons utilisé la librairie AmCharts.

Pour ce qui est du côté interface utilisateur, nous avons continué à utiliser ce qui était en place, c'est-à-dire, la librairie React. Cela nous a permis de rapidement ajouter nos fonctionnalités dans l'interface.

Back-end

Afin d'accélérer le traitement des données et rendre la plateforme disponible en ligne, plusieurs étapes ont été suivies. Nous avons commencé par faire des recherches sur la plateforme ADAM et par la suite sur les outils offert par la plateforme AWS d'Amazon.

ADAM

ADAM est une plate-forme d'analyse de la génomique. Utilisant Apache Avro, Apache Spark et Apache Parquet, il s'agit d'une des meilleures solutions sur le marché en termes d'analyse de données génétique. Afin d'utiliser ADAM, il est important d'utiliser des fichiers sous le format .adam. ADAM offre plusieurs outils permettant de convertir les données dans un format lisible par le logiciel.

Dans le cas de ce projet, les données de 1000 Genomes ont été utilisées. 1000 Genomes est un projet regroupant l'ADN de plus de 2 500 personnes issues de 26 populations différentes. Les fichiers disponibles dans ce projet sont sous le format VCF, il est cependant possible d'utiliser la commande `vcf2adam` du logiciel ADAM afin de convertir les fichiers dans le bon format et les lire avec ADAM.

Amazon Web Service

Afin de rendre l'application disponible au travers du monde, les outils d'Amazon Web Service ont été utilisé. Grâce à un partenariat entre l'ETS et Amazon, nous avions accès au service gratuitement, ce qui permettait de faire plusieurs tests avec des configurations différentes.

Afin de rendre nos différents services en ligne et utilisables pour tous, plusieurs parties d'Amazon seront utilisées:

S3: Un conteneur S3 sera utilisé afin d'emmagasiner les données utilisées ainsi que les données converties dans le format ADAM. De cette façon, tous les développeurs cherchant à travailler pourront directement avoir accès à des données déjà converties.

EC2: Sera utilisé afin de créer un serveur accessible partout sur le web et qui permettra à l'équipe du front-end de communiquer avec l'application ADAM. Ce serveur agira en tant qu'API.

Analyse

Front-end

En reproduisant le code de AreaSelection et en y injectant nos propres objectifs, nous pouvions rapidement nous attaquer à la nouvelle fonctionnalité. En effet, comme nous avions une base de code disponible assez solide pour nous aider à réaliser la nouvelle fonctionnalité, la tâche principale à réaliser lors de l'analyse était de comprendre la nature de cette dernière. Nous avons accompli ceci en quatre étapes.

Correspondance entre l'interface utilisateur et la base de données

Après avoir analysé ces deux éléments, nous avons réussi à pointer trois valeurs critiques à la réalisation de la nouvelle fonctionnalité. Comme le démontre la figure ci-dessous, ces trois champs étaient tous présents dans la même table de base de données, ce qui allait nous simplifier considérablement la tâche.

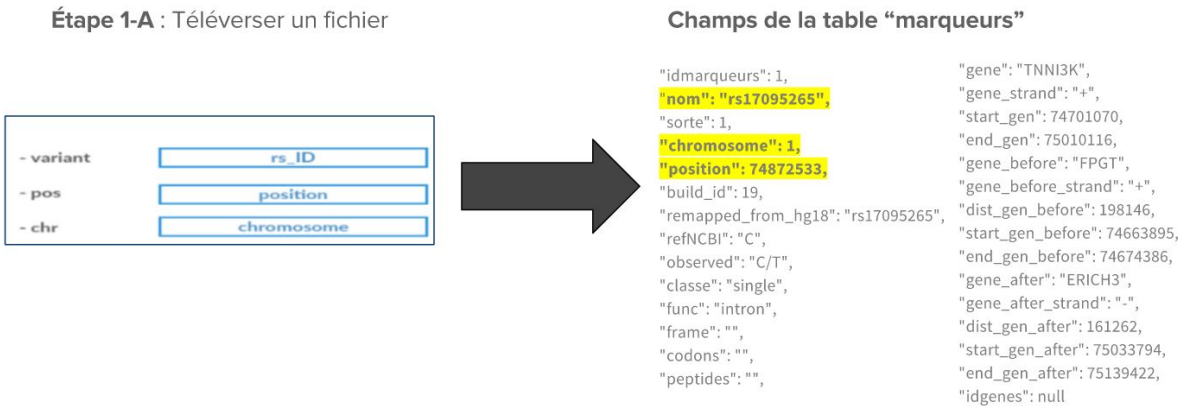


Figure 11: Correspondances entre l'interface et la base de données

Correspondance entre ces champs et le graphique

Comme le démontre la figure suivante, les trois champs peuvent être interprétés dans le graphique de GenomeViewer. Dans le graphique, les chromosomes sont représentés par les longues barres. Les rsids, pour leurs parts, sont représentés par les petites boîtes jaunes. La position détermine à quelle hauteur le rsid apparaîtra sur le chromosome. Il est important de remarquer que sur le graphique, tous les chromosomes ont des longueurs différentes. Nous devions donc trouver une

manière de gérer cette situation..

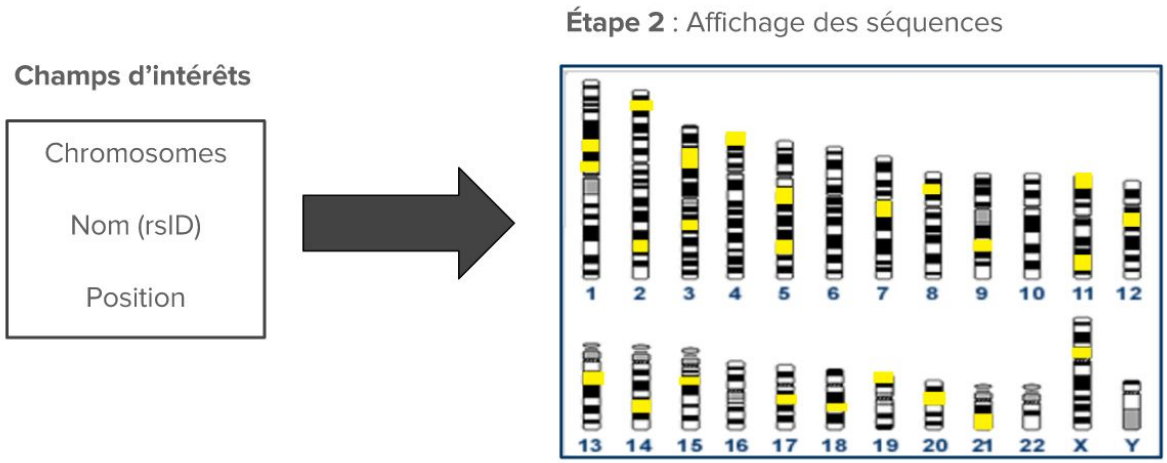


Figure 12: Correspondances entre les champs et le graphique de GenomeViewer

Interroger la base de données

Afin de pouvoir représenter les données, nous avons déterminé que deux requêtes SQL étaient nécessaires. Dans la figure suivante, la première requête permet de déterminer la longueur des chromosomes qui seront affichés dans l'interface. La deuxième requête, pour sa part, permet de simplement obtenir tous les différents détails correspondant aux trois différents paramètres : rsid, position et chromosome.

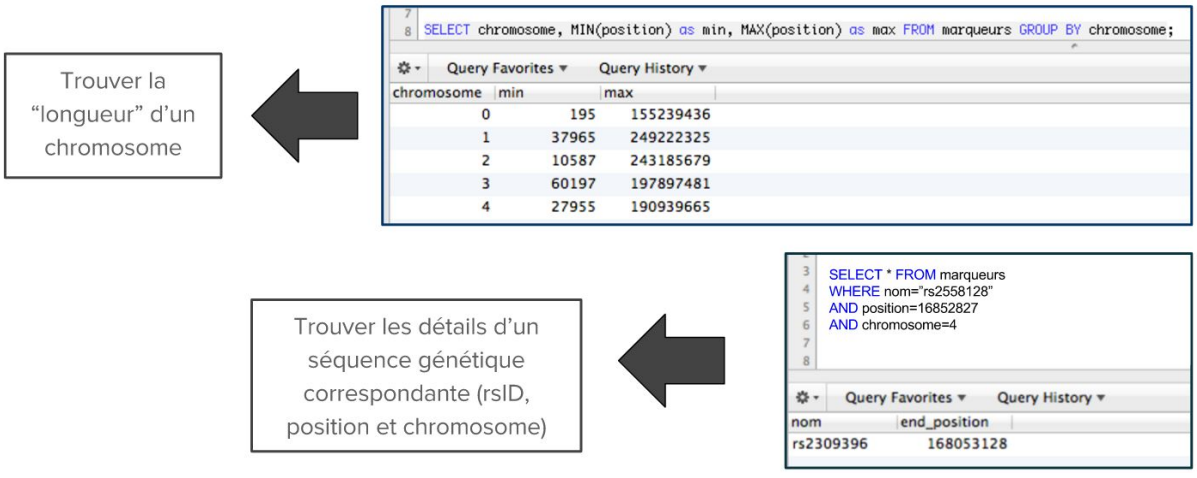


Figure 13: Requêtes SQL effectuées dans la base de données

Représentation simplifiée d'un rsid arbitraire

En utilisant les deux requêtes présentées plus haut, nous avons pu tenter de représenter un rsid choisi arbitrairement. Nous avons donc pu le représenter

manuellement grâce à un outil graphique. Le résultat était le suivant :

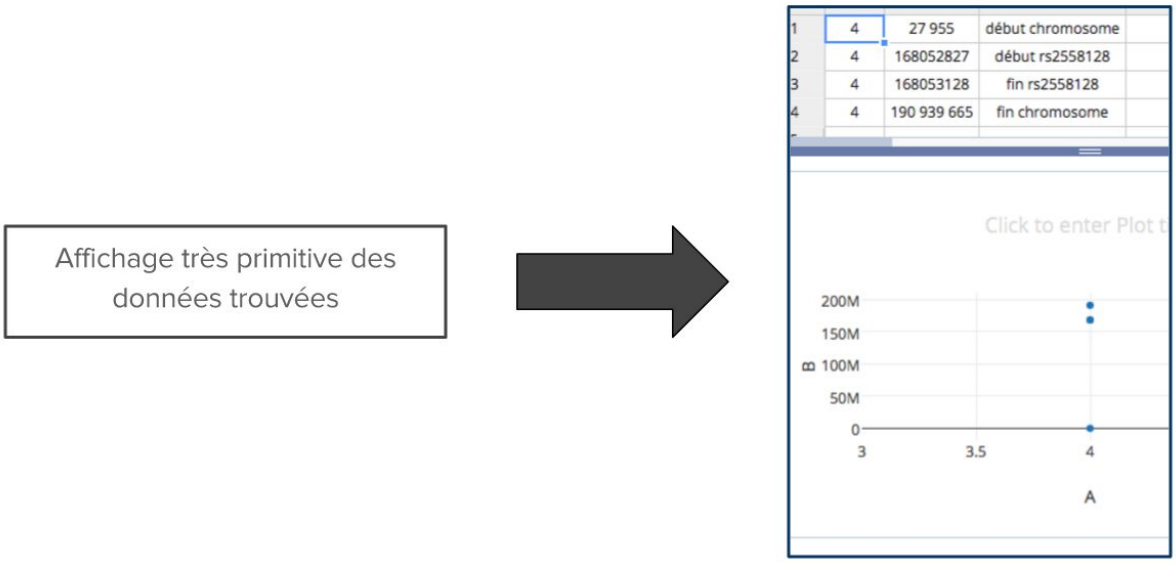


Figure 14: Affichage primitif d'un rsid choisi arbitrairement

Une fois ces quatre objectifs d'analyse atteints, nous pouvions rapidement nous attaquer à la création de la nouvelle fonctionnalité en reproduisant partiellement le code de AreaSelection.

Back-end

Fonctionnalités recherchées

Afin de créer un nouveau back-end répondant aux nouveaux besoins, il était important de créer une liste des fonctionnalités recherchées qui sont représentées ci-dessous:

Lecture des fichiers provenant de 1000 Genomes

Afin d'utiliser les donnés de la base de données de 1000 Genomes, il était important de s'assurer qu'u'il serait possible de les convertir dans le format ADAM. Une recherche sur la plateforme ADAM a donc été effectuée et une commande VCF2ADAM a été trouvée qui permettra de faire la conversion.

Lecture de donnée rapide et efficace

Afin de s'assurer de la performance d'ADAM, plusieurs recherches ont été effectuées. Comme expliqué précédemment, ADAM utilise Spark qui permet de gérer une grande quantité de donné et cela rapidement. Des recherches ont donc été faites sur la plateforme Spark et il a été possible de conclure que cela nous permettrait d'interagir avec nos données très rapidement.

Rendre les données disponibles en ligne

Afin de rendre les donnés en ligne, il existait principale une façon de faire qui s'agit de créer un serveur de donné. Par chance, Amazon offre ce type de service avec les conteneurs S3.

Site web hébergé en ligne

Ayant une équipe avec des parcours différents, il a été possible de trouver un membre ayant déployé plusieurs applications sur internet dans le passé. Il existe quelques applications répondant à ce besoin, mais grâce aux partenariats avec Amazon, il a été d'utiliser les services d'AWS.

Solution modifiable afin d’ajouter de nouvelles fonctionnalités

Afin de répondre à ce besoin, il a été décidé de mettre beaucoup de temps dans la documentation. De la même occasion, un service de gestion de code sera utilisé et l’équipe à décider d’utiliser Github puisque nous avions tous déjà de l’expérience avec le service.

Outils

Django

Cadriciel web et ORM puissant qui permet de développer des applications web complètes. Permets de faire le routage des URL et de diriger les requêtes aux bons contrôleurs. On pourrait décrire Django en trois grandes sections. Les modèles, les vues et les gabarits. Les modèles sont la définition des données utilisées dans l'application. Ils sont directement liés à la base de données. Les modèles peuvent être considérés comme des classes. Les vues sont constituées de fonction Python qui permet de gérer les requêtes web et de retourner des réponses. On pourrait les considérer comme des contrôleurs. Les gabarits sont simplement les fichiers HTML qui représentent l'interface utilisateur et peuvent utiliser des données retournées par les vues.

MySQL

Système de gestion de bases de données relationnelles.

Python

Langage de programmation interprété.

React

Librairie JavaScript développée par Facebook qui permet de développer des sites web dynamiques de type SPA. React est basé sur l'utilisation de composants réutilisables qui permettent de rendre l'application très flexible. Ces composants possèdent un état qui peut être utilisé tout au long du cycle de vie du composant. On peut aussi utiliser des données d'entrée qui aideront à personnaliser le composant. Ces données sont appelées "props". Les états et les données d'entrée peuvent être utilisés afin de mettre à jour le site web dynamiquement.

Reflux

Librairie JavaScript qui permet de simplifier le modèle MVC en adoptant un flux de données uniques. Afin d'y arriver, Reflux utilise des actions et des *Stores*. Les actions sont des événements qui peuvent être déclenchés afin de mettre à jour le site web. Ces actions sont utilisées en parallèle avec un *Store*. Simplement, les *Stores* permettent d'écouter des actions et de réagir selon l'action déclenchée.

Comme les composants React, les *Stores* ont un état et peuvent l'utiliser.

JSX

Préprocesseur qui ajoute une syntaxe XML à JavaScript afin de simplifier l'utilisation de React. Cela permet de se rapprocher de la syntaxe HTML communément utilisée. En effet, à défaut de créer des éléments React, on peut simplement utiliser la syntaxe HTML. L'utilisation d'attributs HTML est aussi possible.

Bokeh

Librairie python qui permet de générer des graphiques interactifs ou d'autres outils de visualisation. Le plus gros avantage de Bokeh est sa capacité à traiter un grand nombre de données. Cela se traduit en de meilleures performances au niveau de l'interactivité.

AmCharts

Librairie JavaScript permettant de générer des graphiques dynamiques et interactifs. Tous les graphiques de AmCharts sont très personnalisables à l'aide de configurations ou de code. Cela permet de modifier pratiquement tous les aspects des graphiques. De plus, cette librairie permet de construire des graphiques accessibles et qui s'adaptent à l'espace disponible.

Github

GitHub est un service web d'hébergement et de gestion de développement de logiciels, utilisant le logiciel de gestion de versions Git. Github permet donc d'héberger la nouvelle version de GOAT et d'ADAM et sera disponible pour tous les futurs développeurs du projet.

Spark

Développé par la fondation Apache issue de recherches à Berkeley. Apache Spark est un cadriciel de traitements Big Data. Il est utilisé afin de traiter une grande quantité de données rapidement..

ADAM

ADAM est une plate-forme d'analyse de la génétique avec des formats de fichiers spécialisés. Le but de ce projet est pour permettre aux chercheurs d'avoir une plateforme commune leur permettant d'analyser des données rapidement et

efficacement. ADAM supporte plusieurs types de fichiers, mais ses fonctions de base utilise le format .adam. Il est cependant possible de convertir plusieurs types de fichiers en format .adam avec les fonctions d'ADAM.

1000 Genomes

Il s'agit d'une recherche organisée par plusieurs scientifiques essayant de créer l'un des plus grands catalogues de la génétique humaine. Les fichiers représentent l'ADN de plus de 2 500 personnes issues de 26 populations différentes.

Amazon Web Service

Amazon Elastic Compute Cloud (EC2)

Les instances EC2 disponible sur AWS offrent une solution offrant une capacité de calcul dans le Cloud. EC2 permet donc d'exécuter les commandes offertes dans le logiciel ADAM. EC2 permet donc de créer des machines virtuelles sur demande afin d'exécuter les programmes souhaités. Il est aussi possible de modifier les caractéristiques d'une instance afin de répondre à une demande plus grandissante.

EC2 est utilisé afin d'héberger nos serveurs web pour rendre l'application disponible sur le web.

S3

Amazon S3 est un service de stockage d'AWS. Il permet de sauvegarder une quantité « virtuellement » illimitée de données avec une très haute disponibilité. (99.99%) L'instance S3 permet de sauvegarder toutes les données utilisées dans la nouvelle version de GOAT. Cela permettra aux futurs développeurs d'avoir accès à toutes les données nécessaires pour travailler sur le projet.

EMR

Amazon EMR fournit une infrastructure Hadoop qui permet de traiter de manière simple et rapide de grandes quantités de données sur des instances EC2 qui seront automatiques créés. Dans le cas d'ADAM, nous avons un cluster EMR utilisant Spark afin de calculer nos requêtes rapidement. Il est possible de voir une différence de 400% lors d'exécution sur un cluster comparativement à une exécution locale.

Conception

Front-end

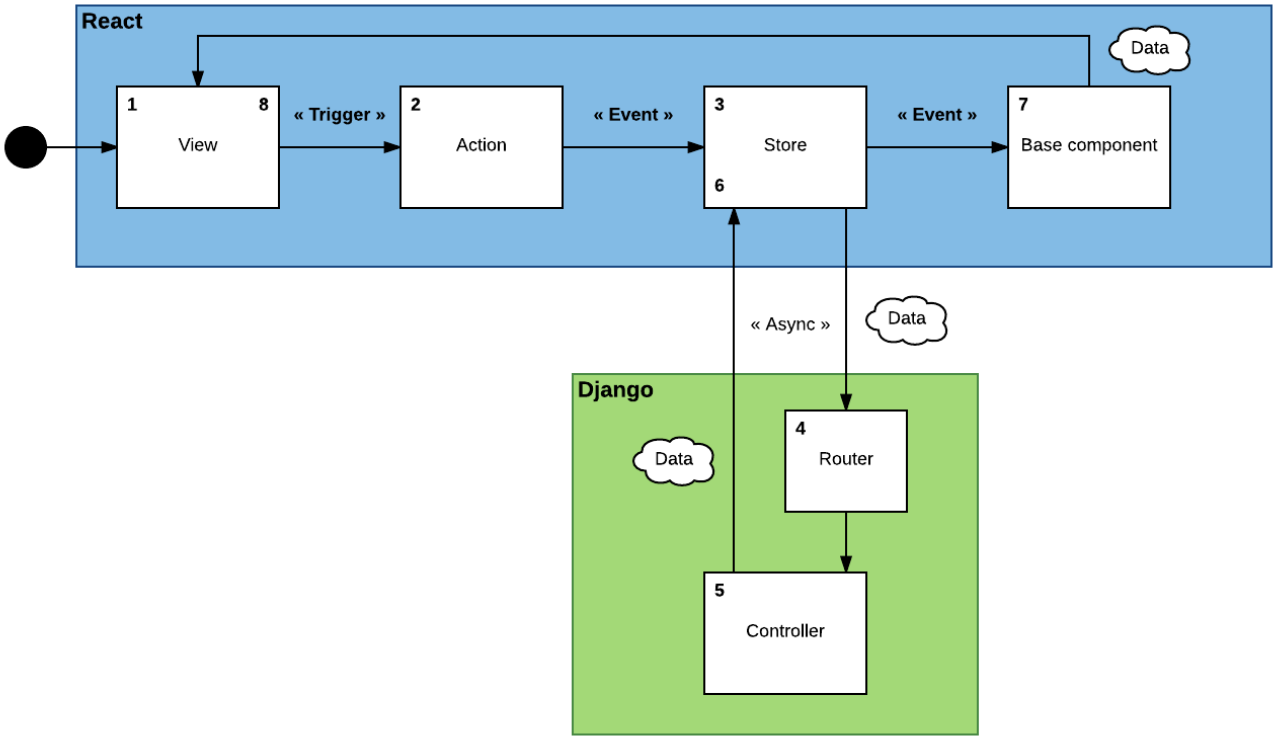


Figure 15: Diagramme représentant le flux de l'application front-end

Le diagramme ci-dessus présente la conception et le flux général de l'application front-end. On comprend rapidement que nous avons dû agencer deux technologies différentes afin de réaliser le projet. Voici l'explication en détails du flux de l'application :

1. On commence dans une vue qui correspond à un composant React permettant de représenter une fonctionnalité. Cette vue permet à l'utilisateur d'interagir avec l'application web. Puisque c'est un composant React, il est dynamique et peut utiliser des données externes afin de construire la vue correctement.

2. À partir de cette vue, lorsqu'une action est entreprise par l'utilisateur, on fait appel à une action Reflux définie auparavant. Cette action est simplement un événement qui peut être déclenché. L'action est donc invoquée et elle sera gérée par un autre composant.

3. L'événement est capté par un *Store* qui pourra gérer l'action. Il reçoit les données passées en paramètres à l'action. Pour la plupart des cas, le *Store* fait une requête GET ou POST asynchrone sur une URL en lui passant les données.

4. La requête est captée par un routeur Django grâce à l'URL. Le routeur peut ensuite faire appel à la bonne méthode dans le contrôleur approprié.

5. Le contrôleur Django reçoit la requête et la traite à l'aide des données. Le contrôleur retourne une réponse HTTP contenant le résultat généralement sous format JSON.

6. La requête asynchrone envoyée auparavant par le *Store* reçoit une réponse HTTP et déclenche un événement qui sera capté par le composant de base.

7. Le composant de base capte l'événement et modifie l'état général de l'application selon l'événement capté. L'état général de l'application est aussi géré par le composant de base qui connaît tous les états de l'application possibles. Selon l'état, il retourne le bon composant React en lui passant les données reçues du contrôleur Django auparavant.

8. Le composant React gère et utilise les données afin de rafraîchir l'interface utilisateur.

Ce flux représente la plupart des fonctionnalités nécessitant un appel serveur. La logique d'affaire se retrouve dans les contrôleurs Django.

Back-end

Afin de permettre à ce projet d'évoluer face à la demande, nous devions créer une architecture stable et facilement maintenable.Pour répondre à ce besoin, nous avons décidé d'utiliser les services d'Amazon. Voici une représentation de l'architecture souhaitée:

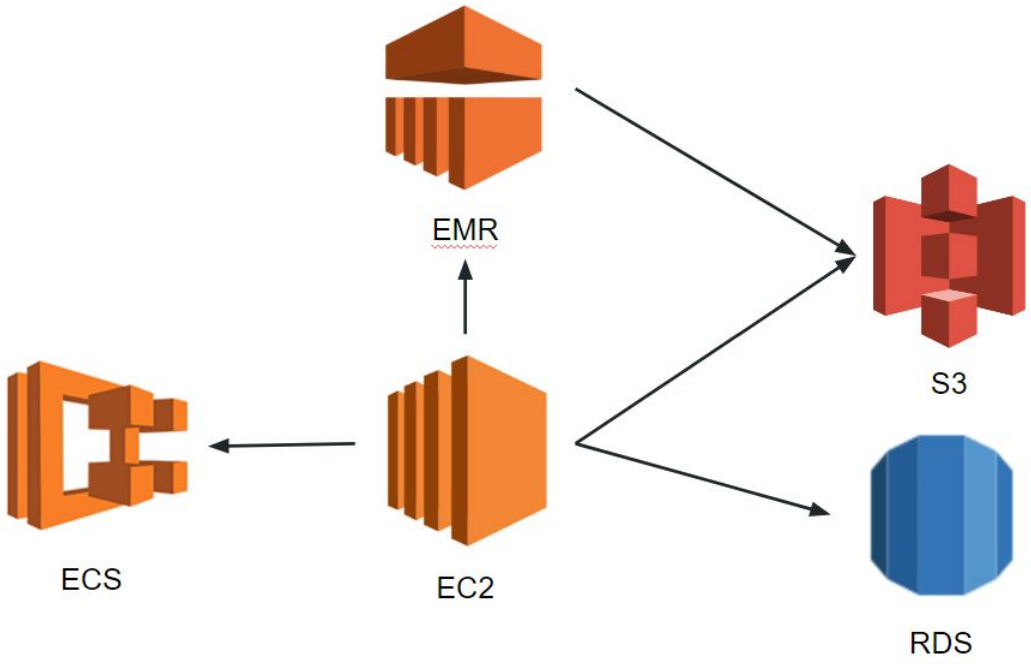


Figure 16: Diagramme AWS d'infrastructure

Comme expliqué précédemment, afin de répondre aux besoins du projet, des instances EC2, EMR et S3 seront utilisés.

Dans un premier lieu, EC2 sera utilisé afin d'héberger notre application Django qui sera utilisée comme API entre le serveur web du front-end et ADAM. Une instance EC2 est aussi utilisée pour le serveur web du front-end.

Afin d'accélérer le calcul des requêtes ADAM, une instance EMR a été créée. Cette instance pourra communiquer avec le logiciel ADAM afin de paralléliser le calcul des différentes commandes exécutées. Grâce à la modification d'ADAM, il a été possible de créer une cache pour chacune des réponses retournées.

C’est à ce moment que le compartiment S3 entre en jeu, puisque nous avons un serveur S3, il est possible d’emmagasiner chacune des réponses dans un fichier texte qui sera sauvegardé dans le compartiment S3.

Cela permet au serveur web de comparer les commandes reçues aux fichiers placés en cache, si un fichier correspond à une commande exécutée dans le passé, il sera possible de retourner une réponse en moins d’une seconde et en évitant le temps de calcul.

Dû à un manque de temps,il a été impossible d’intégrer RDS et ECS. La technologie RDS aurait permis de créer une base de données gérant les accès. De cette façon, seuls les usagés ayant des comptes dans la base de données pourront utiliser le service.

Pour ce qui est d’ECS, il serait possible de gérer les instances EC2 automatiquement. Tout au long de nos tests, cela a été fait manuellement et les instances étaient ouvertes 24 heures par jour et étaient fermées à la fin de la phase de test. ECS pourrait donc réduire les couts en désactivant des instances lors de temps mort et en créant des instances lors de phase de grands calculs.

Voici un diagramme représentant la communication entre le front-end et le back-end:

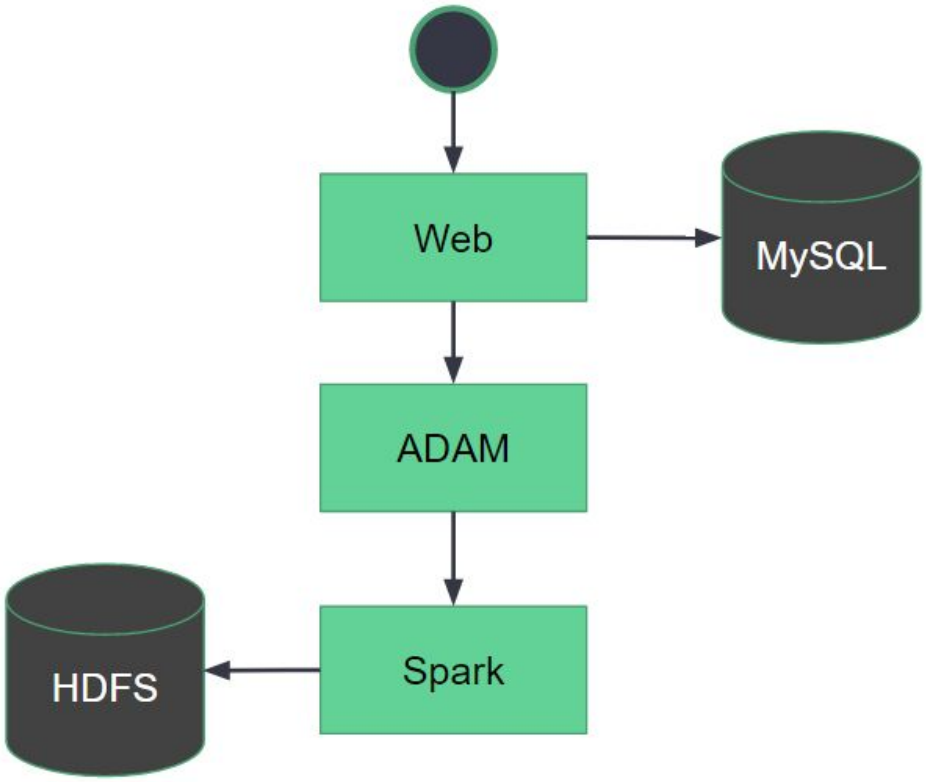


Figure 17: Relation entre le front-end et le nouveau back-end

Implémentation

Front-end

Téléversement de fichier

Téléversement d'un fichier de type CSV contenant des données génétiques afin de générer le graphique GenomeViewer.

Cette fonctionnalité permet de téléverser son propre fichier CSV contenant les informations nécessaires pour la correspondance de données avec les bases de données. Le fichier contient au minimum les rsid, chromosomes et positions. Ces

données sont extraites du fichier et sont envoyées au GenomeViewer afin de générer le graphique. Afin d'extraire les bonnes données, l'utilisateur doit faire la correspondance entre ses en-têtes dans son fichier et les informations demandées (rsid, chromosome et position). L'interface suivant a été conçue afin de faciliter le téléversement.

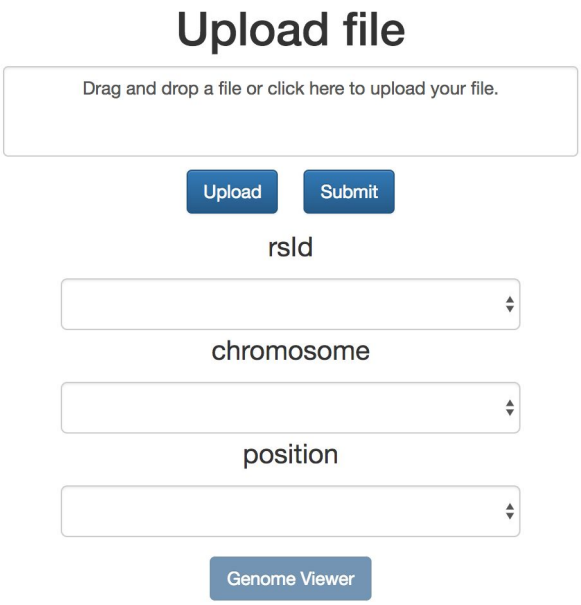


Figure 18: Interface permettant de téléverser un fichier CSV dans GenomeViewer

Après avoir téléversé le fichier, les en-têtes du fichier se retrouvent dans chacune des listes déroulantes afin que l'utilisateur puisse faire la correspondance.

Composant React permettant le téléversement d'un fichier CSV et l'extraction de ses en-têtes.

Voici une partie du code du composant React utilisé pour le téléversement du fichier. Plusieurs commentaires en rouges ont été ajoutés afin d'expliquer plus précisément le comportement du code. Certaines parties du code ont été retirées volontairement afin de faciliter la lecture.

```
// Création du composant UploadFilePage.
var UploadFilePage = React.createClass({

// Cette fonction est appelée lorsque le composant est créé. Elle permet d'initialiser
l'état du composant. On y ajoute des valeurs par défaut qui seront ajoutées aux listes
déroulantes.
    getInitialState:function() {
```

```
        return {
            rsid_header: 'rsid',
            chromosome_header: 'chr',
            position_header: 'position',
        };
    },

// Cette fonction est appelée lorsque les données d'entrée du composant vont être
modifiés. Les nouvelles données sont stockées dans l'objet nextProps. Dans notre cas, ceci
se produit après avoir faire l'extraction des en-têtes et qu'elles sont retournées et donc
contenues dans l'objet nextProps. On modifie l'état du composant afin d'y modifier les
valeurs des trois en-têtes. Puisqu'ils sont directement utilisés dans le code HTML, ils
seront mis à jour aussitôt que cette méthode est appelée.
    componentWillReceiveProps: function(nextProps){
        this.setState({
            rsid_header: nextProps.rsid,
            chromosome_header: nextProps.chr,
            position_header: nextProps.pos,
        });
    },

// Cette fonction est appelée lorsqu'on soumet le fichier CSV. On empêche la requête de se
poursuivre normalement afin d'utiliser l'action extractHeaders. Les actions seront
expliquées plus en détail ultérieurement.
    onSubmit : function(e) {
        e.preventDefault();
        GenomeViewerActions.extractHeaders(this.state.file);
    },



// Cette fonction est appelée lorsque l'utilisateur a complété la correspondance entre les
informations demandées et les en-têtes de son fichier et qu'il envoie le formulaire afin
de générer le graphique GenomeViewer. Encore une fois, on bloque la requête et on utilise
l'action fileGenomeViewer.
    OnGenomeViewer : function(e) {
        e.preventDefault();

        GenomeViewerActions.fileGenomeViewer(
            this.state.file,
            this.state.rsid_header,
            this.state.chromosome_header,
            this.state.position_header
        );
    },

// Cette fonction permet de générer l'interface HTML.
    render : function() {
        // On retourne le code HTML ici.
```

```
        return ();
    }
});
```

Téléversement du fichier et extraction des en-têtes.

Comme mentionné dans le code ci-haut, on utilise une action afin de transmettre le fichier.

```
GenomeViewerActions.extractHeaders(this.state.file);
```

Ces actions sont définies dans le fichier *GenomeViewerActions*.

```
var Actions = Reflux.createActions([
  'queryParams',
  'adamGenomeViewer',
  'uploadFile',
  'extractHeaders',
  'fileGenomeViewer',
]);
```

Voici une partie du code du *Store* utilisé pour l'extraction des en-têtes.

```
// Création du Store GenomeViewerStore.
var GenomeViewerStore = Reflux.createStore({
  // Écoute des actions de GenomeViewerActions.
  listenables : [GenomeViewerActions],

// Fonction appelée lorsque l'action extractHeaders est déclenchée.
  extractHeaders : function(file) {
    var store = this;
    var xhr = new XMLHttpRequest();
    var data = new FormData();

// On joint le fichier aux données qui seront envoyées dans la requête AJAX.
    data.append('file', file);

// Requête POST vers l'URI /extractHeaders/ qui pointe sur une views Django qui fera
l'extraction des en-têtes.
    xhr.open('POST', encodeURI("/extractHeaders/"), true);
    xhr.onload = function() {

// Si la requête réussit, on reçoit les en-têtes extraits en format JSON. On envoie le
résultat à la fonction sendHeaders contenu dans GenomeViewerStore.
      if (xhr.status==200) {
        var json = JSON.parse(xhr.responseText);
```

```
        store.sendHeaders(json.headers);
      }
    };
    xhr.send(data);
  },

// Fonction qui reçoit les en-têtes extraits et déclenche l'événement sendHeaders du
composant de base de l'application.
  sendHeaders : function(headers) {
    this.trigger('sendHeaders', [
      headers[0],
      headers[1],
      headers[2]
     ]);
  }

});
```

La *view* Django qui est appelé ouvre simplement le fichier, extrait les en-têtes du fichier et renvoie la réponse en format JSON.

```
def extractHeaders(request):
    f = request.FILES['file']
    reader = csv.reader(f)
    headers = reader.next()
    response = json.dumps({'headers': headers})
    f.close()

    return HttpResponse(response)
```

Le déclenchement des événements de *GenomeViewerStore* est capté par le composant de base de l'application. Ce composant contient tous les états possibles de l'application et peut écouter plusieurs *Store* différents.

```
// Importation du composant UploadFilePage.
var UploadFilePage = require('./UploadFilePage.jsx');

// Création du composant de base.
var Base = React.createClass({

// On écoute les événements de GenomeViewerStore.
  mixins : [
    Reflux.listenTo(GenomeViewerStore,'onChange')
  ],

// Fonction appelée lorsqu'un événement est déclenché.
  onChange : function(event, data){
```

```
    switch(event){

// Dans le cas où l'événement déclenché est sendHeaders, on modifie l'état de
l'application pour UploadFile et on ajoute les valeurs des en-têtes, contenus dans les
données envoyées, à l'état courant afin de les utiliser plus tard.
      case "sendHeaders":
          this.setState({
            appState : "UploadFile",
            rsid : data[0],
            chr : data[1],
            pos : data[2]
          });
          break;
    }
  },

// Fonction appelée lorsque l'état du composant est modifié.
  handlingState : function(){
    switch(this.state.appState){

// On affiche le composant UploadFile en lui envoyant les en-têtes extraits. Les en-têtes
seront reçus par la fonction componentWillReceiveProps du composant UploadFilePage.
      case "UploadFile":
        return <UploadFilePage rsid={this.state.rsid} chr={this.state.chr}
pos={this.state.pos}/>

// On afficher la page d'accueil par défaut.
      default :
        return <HomePage/>
    }
  }
});
```

Correspondances des en-têtes avec les informations rsid, chromosome, position et envoi des données au GenomeViewer.

Après avoir fait la correspondance, on peut envoyer les données au *GenomeViewer* en appuyant sur *Genome Viewer*.

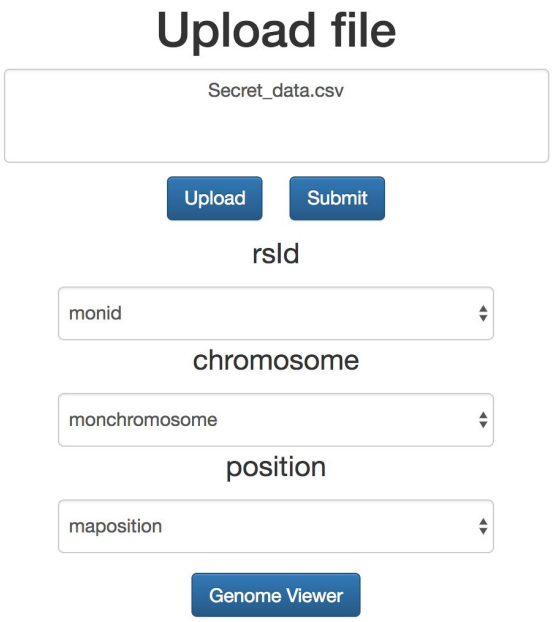


Figure 19: Interface représentant le résultat d'un téléversement de fichier dans GenomeViewer

Comme pour l'extraction des en-têtes, on utilise une action. Cette action est déclenchée lorsque l'utilisateur appuie sur *Genome Viewer*.

```
OnGenomeViewer : function(e) {
  e.preventDefault();

  GenomeViewerActions.fileGenomeViewer(
    this.state.file,
    this.state.rsid_header,
    this.state.chromosome_header,
    this.state.position_header
  );
},
```

Cette action est alors captée par *GenomeViewerStore* et sera gérée par la fonction *fileGenomeViewer*.

```
fileGenomeViewer : function(file, rsid_header, chromosome_header, position_header) {
      var store = this;
      var xhr = new XMLHttpRequest();

      var data = new FormData();
      data.append('file', file);
```

```
      data.append('rsid_header', rsid_header);
      data.append('chromosome_header', chromosome_header);
      data.append('position_header', position_header);

// Requête POST vers l'URI /fileGenomeViewer/ qui pointe sur une views Django qui fera
l'extraction des données du fichier.
      xhr.open('POST', encodeURI("/fileGenomeViewer/"), true);
      xhr.onload = function() {

// Si la requête réussit, on reçoit le contenu du fichier CSV en format JSON. On envoie le
résultat à la fonction fireUpdate.
        if (xhr.status==200) {
            result = JSON.parse(xhr.responseText);
            store.data = result.data;
            store.fireUpdate();
        } else {
          console.error("GOAT here : We couldn't get your data. Check the route, or your
connection");
        }
      };
      xhr.send(data);
  },

// Fonction qui reçoit le contenu du fichier CSV et déclenche l'événement adamGenomeViewer
du composant de base de l'application qui permet de générer le graphique.
fireUpdate : function(){
    this.trigger(adamGenomeViewer, [
      this.data,
      {
        rsid : this.rsid,
        chromosome : this.chromosome,
        position : this.position
      }
    ]);
  }
```

La *view* Django qui est appelé ouvre le fichier, extrait les données du fichier en format JSON, modifié le JSON afin de faire la correspondance entre les en-têtes du fichier choisis par l'utilisateur et les vraies valeurs demandées et appelle la fonction *sqlGenomeViewer* en lui passant le résultat sous format JSON.

```
def fileGenomeViewer(request):
    f = request.FILES['file']
    reader = csv.DictReader(f)
    data = json.dumps([ row for row in reader ])
```

```
    f.close()

    output = []

    for row in json.loads(data):
        output.append({
            'rsid': row[request.POST['rsid_header']],
            'position': row[request.POST['position_header']],
            'chromosome': row[request.POST['chromosome_header']]
        })

    return sqlGenomeViewer(output)
```

Du côté du composant de base.

```
// On gère l'événement adamGenomeViewer, on modifie l'état de l'application pour
GenomeViewer et on ajoute les informations nécessaires à l'état courant afin de les
utiliser plus tard.
case "adamGenomeViewer":
        this.setState({
          appState : "GenomeViewer",
          data : data[0],
          chromosome : data[1].chromosome,
          position : data[1].position,
          rsid : data[1].rsid
        });
        break;

handlingState : function(){
    switch(this.state.appState){

// On affiche le composant GenomeViewer en lui envoyant les données.
      case "GenomeViewer":
        return <div><GenomeViewerPage data={this.state.data}
chromosome={this.state.chromosome} position={this.state.position}
rsid={this.state.rsid}/></div>
        break;
      default :
        return <HomePage/>
    }
}
```

Dans la vue, obtention des données et envoi sous format JSON

Comme, au final, nous avons utilisé la librairie AmCharts pour représenter le graphique, c'est ce flux d'information qui sera présenté ci-bas. Dans la vue, les paramètres de requêtes sont utilisés afin d'aller chercher les données et de les

convertir dans un format Json qui sera utilisé dans la page JSX (et donc par AmCharts). C'est au sein du fichier "/GenomeViewer/views.py" que le traitement commence.

Voici à quoi devraient ressembler les données reçues en paramètres.

```
data = [
    {"rsid": "rs2558128", "position": "168052827", "chromosome": "4"},
    {"rsid": "rs2319227", "position": "68281255", "chromosome": "4"},
    {"rsid": "rs1177257", "position": "35936786", "chromosome": "14"},
    {"rsid": "rs12403445", "position": "180587560", "chromosome": "1"},
    {"rsid": "rs7539261", "position": "40050503", "chromosome": "1"},
    {"rsid": "rs2718295", "position": "88262924", "chromosome": "7"},
    {"rsid": "rs6489602", "position": "5140968", "chromosome": "12"},
    {"rsid": "rs1402337", "position": "119154183", "chromosome": "12"}
]
```

Par la suite, voici comment ces données sont traitées.

```
// Fonction principale. C'est elle qui est appelée dans le routage de l'application web.
def sqlGenomeViewer(data):

    rsidArray = data

    // Une méthode est appelée pour obtenir les "longueurs des chromosomes".
    chrBoundaries = getChromosomeBoundaries()
    // Une méthode est appelée pour obtenir les données correspondant aux paramètres
    fournis.
    validRsids = fetchValidRsids(rsidArray)

    // La réponse est formatée dans un format JSON.
    response = json.dumps({
            'data' : {
                'jsonChrBoundaries' : chrBoundaries,
                'jsonValidRsids' : validRsids
            }
        },
        sort_keys=True,
        indent=4,
        separators=(',', ': ')
    )

    return HttpResponse(response)

// Une requête SQL est effectuée afin d'obtenir, pour chaque chromosome, les valeurs
minimales et maximales.
def getChromosomeBoundaries():
    #We query the database for min and max positions for each chromosome
    sqlQuery = "SELECT chromosome, MIN(position) as min, MAX(position) as max FROM
marqueurs GROUP BY chromosome;"
    chrBoundaries = connect.fetchData(sqlQuery)
    return buildJsonData(chrBoundaries)

// Une requête SQL est effectué afin d'obtenir tous les détails d'un rsid corespondant au
```

```
bon chromosome et à la bonne position tels qu'ils ont été fournis en paramètres.
def fetchValidRsids(rows):
    validRsids = []

    // Comme plusieurs ensembles de paramètres ont pu être envoyés, il est nécessaire
    d'effectuer plusieurs requêtes et de concaténer les résultats
    for row in rows:
        sqlQuery = 'SELECT * FROM marqueurs WHERE nom="' + row['rsid'] + '" AND position='
+ row['position'] + ' AND chromosome=' + row['chromosome'] + ';'
        match = connect.fetchData(sqlQuery)

        # We do [1:-1] because it extracts the data from stringified-array. The data being
a string, it removes the first and last characters : [ and ].
        j = buildJsonData(match)[1:-1]
        # If no result was returned, the data was "[]" which both chars were stripped.
So.. empty string.
        if j != "":
            validRsids.append(json.loads(j))

    return json.dumps(validRsids)

// Cette petite fonction permet de convertir le résultat d'une requête SQL (un tableau) en
un format JSON.
def buildJsonData(data):
    jsonData = data.to_json(orient='records')#json table
    return jsonData
```

Réception de l'objet JSON et affichage du graphique

Ce code est contenu dans le fichier "/front_end/components/GenomeViewer.jsx". C'est ici que l'objet JSON envoyé précédemment est finalement traité.

```
// Initialisation du composant React.
var GenomeViewerPage = React.createClass({

  // C'est dans cette fonction de la page est effectuée. Incidemment, c'est ici que le
  graphique est initialisé.
  render : function() {
    // Dans this.props.data se trouvent nos données : jsonChrBoundaries et jsonValidRsids.
    On commence donc par les extraire.
    var jsonChrBoundaries = JSON.parse(this.props.data.jsonChrBoundaries);
    var jsonValidRsids = JSON.parse(this.props.data.jsonValidRsids);

    // Afin de créer le graphique, 2 objets devront être construits, soit dataProvide et
    graphs
    var data = [];

    // Dans le dataProvider (un array), on commence par insérer chacun des chromosome
    grâce à leur valeurs min et max.
    for (var i=0; i<jsonChrBoundaries.length; i++) {
      data.push({
        "chromosome": jsonChrBoundaries[i].chromosome,
        "min": jsonChrBoundaries[i].min,
```

```
    "max": jsonChrBoundaries[i].max
  });
}

// Nous devons ensuite indiqué dans l'array graph de quelle manière seront
représentées ces données.
var graphs = [{
  "colorField": "color",
  "fillAlphas": 0.8,
  "lineColor": "#00bfff",
  "openField": "min",
  "type": "column",
  "valueField": "max",
  "showBalloon": false
}];

   //Ensuite, pour chacun des points à affiché sur le graphique, on doit l'insérer
   dans l'array dataProvider ainsi que dans l'array graphs. Même si cela peut sembler
   étrange puisque tous les points sont représentés de la même façon, cela est
   nécessaire puisque les informations apparaissant dans la bulle de texte sont
   indiquées à l'intérieur de l'objet graph.
       for (var i=0; i<jsonValidRsids.length; i++) {
         var validRsid = jsonValidRsids[i];

         data[validRsid.chromosome][validRsid.nom] = validRsid.position;

  // On précise comment la valeur est représentée dans le graphique.
  graphs.push({
      "title": validRsid.nom,
      "bullet": "square",
      "bulletColor": "#ff1144",
      "bulletSize": "15",
      "valueField": validRsid.nom,
      "balloonText":
        "nom : " + validRsid.nom + "\n" +
        "chromosome : " + validRsid.chromosome + "\n" +
        "position : " + validRsid.position + "\n" +
        "gene_before : " + validRsid.gene_before + "\n" +
        "gene_after : " + validRsid.gene_after + "\n" +
        "idgenes : " + validRsid.idgenes + "\n" +
        "idmarqueurs : " + validRsid.idmarqueurs,
      "phenotypeLegendText" : validRsid.phenotype,
      "mutationLegendText" : validRsid.mutation
  });
}

// Cette fonction est appelée lors de l'initialisation du graphique. Elle sert à faire
la gestion de la légende permettant de montrer/cacher certains éléments du graphique.
Concrètement, la fonction permet de bâtir la liste de tous les phénotypes et une liste
de toutes les mutations ayant été retrouvées au sein de nos donnés. Ces listes seront
affichées en tant que légende pour montrer/cacher les données.
function onInit (e) {

  var chart = e.chart,
      graphs = chart.graphs,
      createLegend = function (id, property) {
        var element = document.getElementById(id),
            itemElement,
            textElement,
            matchingGraphs = [],
```

```
            referenceItems = {},
            legendReference,
            graph,
            value,
            legendText;

          for (var i = 0, iLen = graphs.length; i < iLen; i++) {

            graph = graphs[i];
            value = graph[[property, "LegendText"].join("")];
            legendText = graph[[property, "LegendText"].join("")];

            if (value && legendText) {
              matchingGraphs.push(graph);
              legendReference = [property, value].join("-");

              if (!chart.customLegends[legendReference])
                chart.customLegends[legendReference] = [];

              // Keep the graph reference per property name
              chart.customLegends[legendReference].push(graph);

              // Don't add to the DOM again if the item is already there
              if (!referenceItems[value]) {

                // Keep reference that is item is already added
                referenceItems[value] = true;

                itemElement = document.createElement("div");
                textElement = document.createElement("span");

                textElement.innerText = legendText;

                itemElement.appendChild(textElement);
                element.appendChild(itemElement);

                // Add the propery name to identify this item on click
                itemElement.legendReference = legendReference;

                itemElement.addEventListener("click", onLegendClick, false)
              }
            }
          }
        };

  // Leave a reference to the legend items in the chart object
  chart.customLegends = {};

  // Pass the legend's element id and which property to look for and group by
  createLegend("phenotypeslegend", "phenotype");

  // Pass the legend's element id and which property to look for and group by
  createLegend("mutationslegend", "mutation");
}

// Cette fonction fait en sorte qu'en cliquant sur un élément de la légende (soit
phénotype ou mutation) les rsid associés à cette valeur seront montrés ou cachés
selon leur état précédent.
function onLegendClick (e) {
  console.log(e);
```

```
      var target = e.currentTarget,
          legendReference = target.legendReference,
          chart = AmCharts.charts[0],
          selectedGraphs = chart.customLegends[legendReference];

      for (var i = 0, iLen = selectedGraphs.length; i < iLen; i++) {

        if (selectedGraphs[i].hidden)
          chart.showGraph(selectedGraphs[i]);
        else
          chart.hideGraph(selectedGraphs[i]);
      }
    }



    // Création de la configuration qui est utilisée pour le graphique. Remarquez
l'utilisation des deux objets créés précédemment : data et graphs.
    var config = {
      "type": "serial",
      "theme": "light",
      "thousandsSeparator": " ",
      "sequencedAnimation": false,
      "startDuration": 1,

      "dataProvider": data,

      "valueAxes": [ {
         "axisAlpha": 0,
         "gridAlpha": 0.1,
         "position": "left"
       } ],
      "graphs": graphs,
      "columnWidth": 0.4,

      "categoryField": "chromosome",
      "categoryAxis": {
        "gridPosition": "start",
        "gridAlpha": 0,
        "gridAlpha": 0.1
      },
      "export": {
        "enabled": true
      },
      "listeners": [{
        "event": "init",
        "method": onInit
      }]
    };


    // HTML retourné par la vue. Remarquez l'utilisation de la balise AmCharts.React ainsi
que de l'utilisation de l'objet de configuration.
    return (
      <div className="container">
        <div className="col-md-10" style={{height:'500px',
backgroundColor:"white"}}><AmCharts.React {...config} /></div>
        <div className="col-md-2" style={{backgroundColor:"white", height:"500px"}}>
          <br/>
```

```
          <strong>Phenotypes Legend</strong>
          <div id="phenotypeslegend" className="custom-legend"></div>
          <br/>
          <strong>Mutations Legend</strong>
          <div id="mutationslegend" className="custom-legend"></div>
          <br/>
          <button id="showAll" onClick={this.showAll} className="btn btn-primary">Show
all</button>
          <button id="hideAll" onClick={this.hideAll} className="btn btn-primary">Hide
all</button>
        </div>
      </div>
    );
  },

// Cette fonction permet de montrer tous les rsid présents sur le graphique.
showAll : function(e) {
    var chart = AmCharts.charts[0]

    for (var i=1, iLen=chart.graphs.length; i<iLen; i++) {
      console.log(i);
      chart.showGraph(chart.graphs[i]);
    }
  },

// Cette fonction permet de cacher tous les rsid présents sur le graphique.
hideAll : function(e) {
    var chart = AmCharts.charts[0]

    for (var i=1, iLen=chart.graphs.length; i<iLen; i++) {
      chart.hideGraph(chart.graphs[i]);
    }
  },

});
```

Résultat final de l’affichage du graphique avec AmCharts

Au final, notre graphique répond bien aux requis de GenomeVIewer. Nous avons aussi réussi à implémenter les légendes de phénotypes et de mutations permettant de filtrer l’affichage des données. Voici à quoi ressemble le graphique dans l’application web :

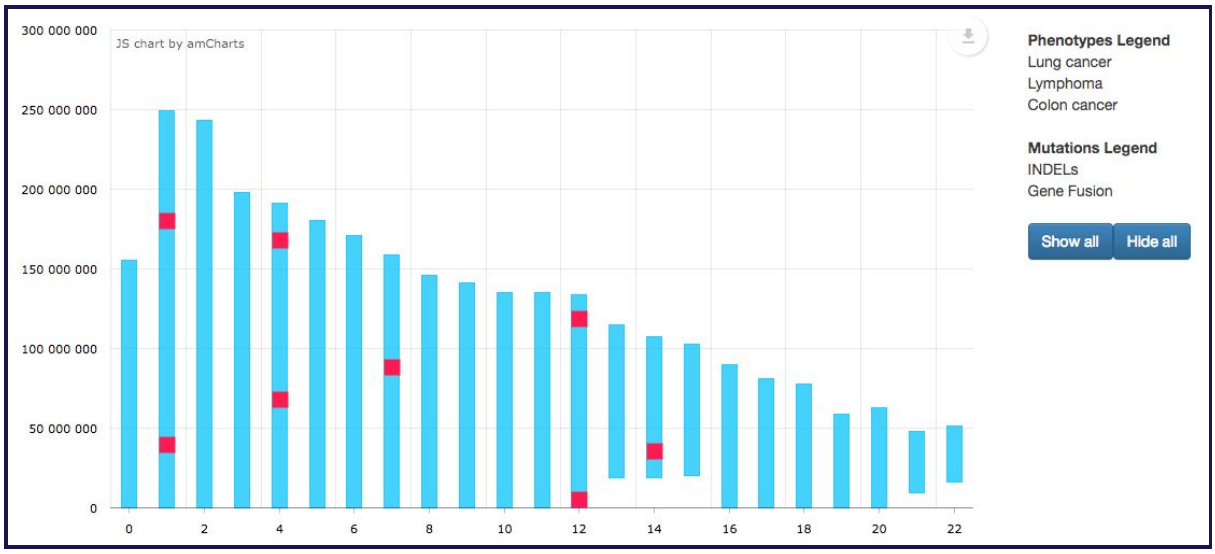


Figure 20: Résultat final du graphique AmCharts pour GenomeViewer

Back-end

Ajout d'une fonctionnalité dans ADAM

Afin de modifier ADAM, nous avons utilisé la dernière version disponible du code (https://github.com/bigdatagenomics/adam). Nous avons donc créé notre propre branche d'ADAM se retrouvant à l'adresse suivante : https://github.com/Emile-Filteau/adam. Une fois cette branche clonée localement, nous devions comprendre le fonctionnement d'ADAM et comment nous pouvions ajouter une fonctionnalité à ce dernier. Voici ce dont nous avons compris de la structure d'ADAM.

Le dossier **adam-cli** contient le code source des commandes pouvant être exécuté directement à l'aide de la commande **adam-submit**.

Le dossier **adam-core** contient tout le code source des classes représentant les différents objets de bio génétique, tels que les génotypes et les variantes. Il contient également des classes de type RDD (ex : VariantRDD) qui représente une liste d'objet sur lequel il est possible d'exécuter un calcul distribué dans Spark.

Les dossiers **adam-apis** et **adam-assembly** ne contiennent pas grand-chose et ne sont pas intéressant lors du développement de nouvelles fonctionnalités.

Un autre aspect important d'adam est qu'il est développé dans le langage Scala. Ce langage compilé qui est exécuté dans l'environnement virtuel de Java est un étrange mélange entre un langage fonctionnel et orienté objet. Puisqu'il fait partie de la famille de Java et qu'il est 100% compatible avec les librairies en Java, ce langage utilise maven[3] pour gérer les dépendances et la compilation. Il est donc nécessaire d'avoir maven installé sur la machine locale pour modifier adam.

[3] https://maven.apache.org/

Voici les étapes que nous avons suivies pour développer notre nouvelle fonctionnalité dans adam :

1. **Ajout de la commande**
   Comme mentionné plus haut, nous avons commencé par ajouter une commande dans le dossier adam-cli. Nous avons appelé notre commande *Matching.scala*, car les permet de trouver les variantes qui *match* avec le rsID passé en paramètre. Un nom plus descriptif pourrait être trouvé dans le futur.

   La base d'une commande adam est définie par un objet scala héritant de la commande de base :

```scala
object Matching extends BDGCommandCompanion {
  val commandName = "matching"
  val commandDescription = "Finds all variants with a given rsid"

  def apply(cmdLine: Array[String]) = {
    new Matching(Args4j[MatchingArgs](cmdLine))
  }
}
```

   L'objet `Matching` représente ce qui exposé lors de l'importation du fichier. Il définit le nom de la commande, une description et l'action effectuée lorsqu'elle est appelée.

   La méthode `apply` créer un objet de classe `Matching` en lui passant en paramètre un objet de type `MatchingArgs` avec les arguments de la ligne de commande.

```scala
class MatchingArgs extends Args4jBase with ParquetArgs {
  @Argument(required = true, metaVar = "INPUT", usage = "The ADAM or FASTA file to match variants from", index = 0)
  var inputPath: String = null
  @Argument(required = true, metaVar = "OUTPUT", usage = "The path to save result to", index = 1)
  var outputPath: String = null
  @Argument(required = true, metaVar = "RSID", usage = "The rsid that you want to match against the database", index = 2)
  var rsid: String = null
}
```

   La classe `MatchingArgs` hérite de la classe de base pour les arguments `Args4jBase`. Elle définit plusieurs variables d'instances et, à l'aide

d'annotations, défini de quelle façon cet argument devrait être analysé.

L'argument `inputPath` représente le fichier ou le dossier contenant les génotypes à analyser.

L'argument `outputPath` représente le fichier dans lequel nous voulons sauvegarder le résultat du matching.

L'argument `rsid` est tout simplement le rsid passé en paramètre qui servira à filtrer les résultats.

```
class Matching(protected val args: MatchingArgs) extends BDGSparkCommand[MatchingArgs] with Logging {
  val companion = Matching

  def run(sc: SparkContext) {

    // Read from disk
    val genotypes = sc.loadGenotypes(args.inputPath)

    // Filter variants by RSID
    val filteredVariants = genotypes.filterVariants(args.rsid)

    // Print the first result
    filteredVariants.take(1).foreach(println)

    // Save to text file
    filteredVariants.saveAsTextFile(args.outputPath)
  }
}
```

La classe Matching représente la commande créée dans l'objet défini préalablement. Cette classe doit définir la méthode *run* qui prend en paramètre le contexte Spark. Le contenu de cette méthode sera abordé dans les prochains points.

2. **Chargement des données**
   Pour charger les génotypes de 1000 Genomes en mémoire, nous devons utiliser une méthode définie dans `AdamContext` appelé `loadGenotypes`. Cette méthode prend en paramètre un chemin d'accès dans le HDFS qui peut être un dossier ou un fichier. Toutefois, le contenu de ce dossier (ou fichier) doit être des fichiers VCF ou parquet représentant des genotypes. https://github.com/Emile-Filteau/adam/blob/master/adam-core/src/main/scala/org/bdgenomics/adam/rdd/ADAMContext.scala#L1450

3. **Filtrage des variantes**
   Une fois les génotypes chargés en mémoire, nous devons aller chercher les variantes à l'intérieures de chacuns des génotypes et filtrer celles qui ont le rsid passé en paramètre. Pour que la tâche puissent être efficacement distribué sur le cluster, il est primordial d'extraire cette logique à l'intérieure de la classe RDD associée au calcul. Dans notre cas, puisque nous traitons des génotypes. La classe modifiée est le `GenotypeRDD`.

   Voici le code source de la méthode `filterVariants` développée :

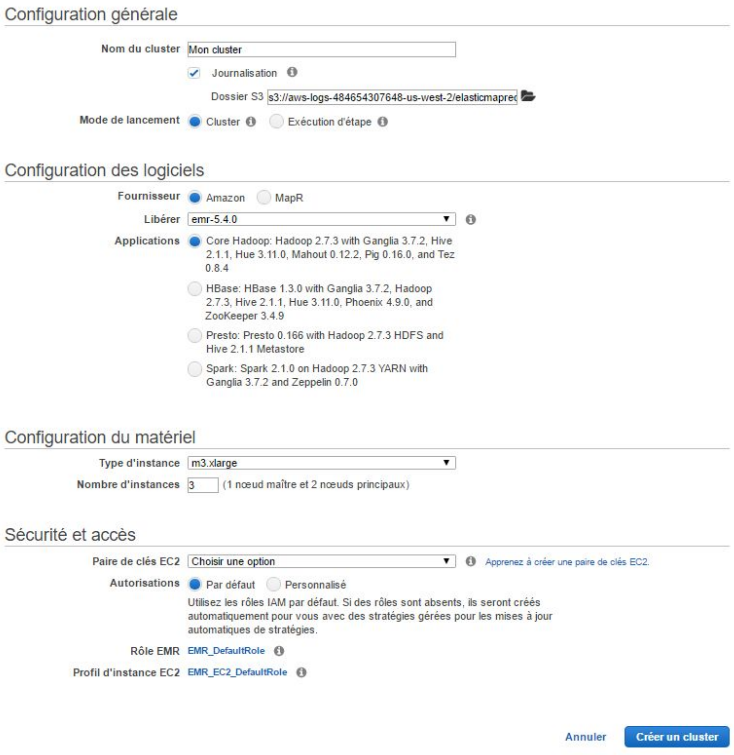
```scala
/**
 * Filters genotypes by variant names
 *
 * @param rsId The variant name to filter
 * @return Returns a VariantRDD containing the filtered variants
 */
def filterVariants(rsId: String): RDD[Variant] = {
  rdd.map(genotype => genotype.variant).filter(variant => variant.names.contains(rsId))
}
```

   Cette méthode prend en paramètre un rsid sous forme de chaîne de caractère. On commence par extraire les variantes des génotypes à l'aide d'un `map`. On filtre ensuite les variantes retournées par le map pour garder uniquement celles qui ont le rsid demandé dans leurs noms (Chaque variante peut posséder plusieurs rsid selon le modèle 1000 Genome). Cette méthode retourne un RDD d'objet Variant. Cette méthode pourrait être grandement améliorée en performance si elle retournait directement un objet de type VariantRDD.

4. **Retour des données**
   Suite au filtrage, il ne reste plus qu'à sauvegarder les résultats dans le fichier de sortie. La classe RDD permet de faire ceci très facilement grâce à la méthode `saveAsTextFile`. Toutefois, cette méthode prend beaucoup de temps puisqu'elle ne sait pas à priori comment sérialiser efficacement les variantes. C'est pourquoi l'optimisation proposée au point précédent permettrait une sauvegarde plus rapide puisque la classe `VariantRDD` possède une méthode `save` qui sauvegarde très rapidement les variantes sur le disque.

Installation d'ADAM sur un cluster EMR

1. **Création d'un cluster EMR sur AWS**.



2. **Connexion SSH sur la machine** (http://docs.aws.amazon.com/fr_fr/emr/latest/ManagementGuide/emr-connect-master-node-ssh.html)
3. **Installation de Maven**
    a. wget http://mirror.its.dal.ca/apache/maven/maven-3/3.5.0/binaries/apache-maven-3.5.0-bin.tar.gz
    b. tar -zxvf apache-maven-3.5.0-bin.tar.gz
    c. export PATH=$HOME/maven/apache-maven-3.3.9/bin:$PATH
4. **Installation d'ADAM:**
    a. git clone https://github.com/Emile-Filteau/adam.git
    b. cd ADAM
    c. export MAVEN_OPTS="-Xmx512m -XX:MaxPermSize=256m"
    d. ./scripts/move_to_spark2.sh

e. mvn clean package -DskipTests

Conversion de fichier VCF

1000 Genomes détient un compartiment S3 avec tous les chromosomes nécessaires à l'adresse suivante: http://s3.amazonaws.com/1000genomes

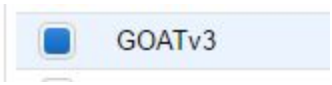
Afin de convertir un fichier VCF, voici la marche à suivre:

1. **Téléchargement d'un VCF:**
    a. aws s3 cp s3://1000genomes/phase1/analysis_results/integrated_call_sets/ALL.chr1.integrated_phase1_v3.20101123.snps_indels_svs.genotypes.vcf.gz /home/hadoop/
2. **Extraction du VCF:**
    a. gunzip ALL.chr1.integrated_phase1_v3.20101123.snps_indels_svs.genotypes.vcf.gz
3. **Déplacer le fichier dans l'espace HDFS**
    a. hadoop fs -put /home/hadoop/ ALL.chr1.integrated_phase1_v3.20101123.snps_indels_svs.genotypes.vcf /user/hadoop/
4. **Exécution de la commande vcf2adam**
    a. adam-submit vcf2adam "user/hadoop/ALL.chr1.integrated_phase1_v3.20101123.snps_indels_svs.genotypes.vcf" "/user/hadoop/adamfiles"

Déploiement de l'application sur une instance EC2

Afin de rendre l'application disponible sur l'internet, nous avons dû déployer l'application Django directement sur une instance EC2 (L'instance "master" d'EMR). Voici les démarches pour faire cela:

1. **Connexion sur AWS et création d'une instance EC2.**
    a. https://us-west-2.console.aws.amazon.com
2. **Connexion à l'instance EC2**
    a. ssh -i "key.pem" ec2-user@[serveur-amazon].com

3. **Téléchargement du projet Django**
    a. git clone git@github.com:Emile-Filteau/GOAT.git
4. **Installation des pré-requis pour Django (Pip, VirtualEnv, Gunicorn)**
    a. sudo yum install python-pip python-dev
    b. sudo pip install virtualenv
    c. cd goat
    d. virtualenv goatenv
    e. source myprojectenv/bin/activate
    f. pip install django gunicorn psycopg2
5. **Initialisation du projet**
    a. pip install -r requirements.txt
    b. ./manage.py makemigrations
    c. ./manage.py migrate
6. **Déploiement du serveur**
    a. gunicorn --bind 0.0.0.0:8000 GOAT.wsgi:application
7. **Modification du pare-feu EC2**
    a. Sélectionner l'instance EC2 utilisée sur AWS
        i. 

    b. Sélectionner le groupe de sécurité
        i. 

    c. Ajouter le port 8000 en TCP
        i. 

Type	Protocole	Plage de ports	Source
HTTP	TCP	80	0.0.0.0/0
HTTP	TCP	80	::/0
Règle TCP personnalisée	TCP	8000	0.0.0.0/0
Règle TCP personnalisée	TCP	8000	::/0

Utilisation de S3:

1. **Installation des dépendances**
    a. yum install gcc libstdc++-devel gcc-c++ curl-devel libxml2-devel openssl-devel mailcap
    b. Installation de Fuse: https://github.com/libfuse/libfuse
    c. Installation de S3FS: https://github.com/s3fs-fuse/s3fs-fuse
2. **Afin d'accéder aux instances S3, nous devons créer un fichier avec nos clés d'accès**
    a. vim /etc/secret-s3fs
        i. AWS_ACCESS_KEY_ID:AWS_SECRET_ACCESS_KEY
    b. chmod 600 /etc/secret-s3fs
3. **Ajouter le compartiment S3 sur notre instance EC2**
    a. s3fs goat-bucket /mnt/goat -o passwd_file=/etc/secret-s3fs

Grâce à cela, il sera possible d'interagir directement avec le compartiment S3 et pourrons modifier/déplacer/copier n'importe quel fichier.

Connexion front-end et back-end

Pour connecter le travail accompli par l'équipe du front-end avec celui fait par l'équipe du back-end, une petite API a été développée. Pour suivre le courant du développement actuel de GOAT, cette API a été développée en python en utilisant le cadriciel web Django. Cette API a pour but de fournir un point d'accès via requêtes HTTP qui fait le relais vers ADAM. Ce point d'accès ne fait que relayer les requêtes vers ADAM en créant un sous-processus qui exécute la commande sur le cluster. Une fois la commande ADAM terminée, le résultat est traité pour être retourné dans un format JSON plus facile à utiliser pour le front-end.

La fonction est donc disponible sous la section "ADAM Genome Viewer" du site internet:

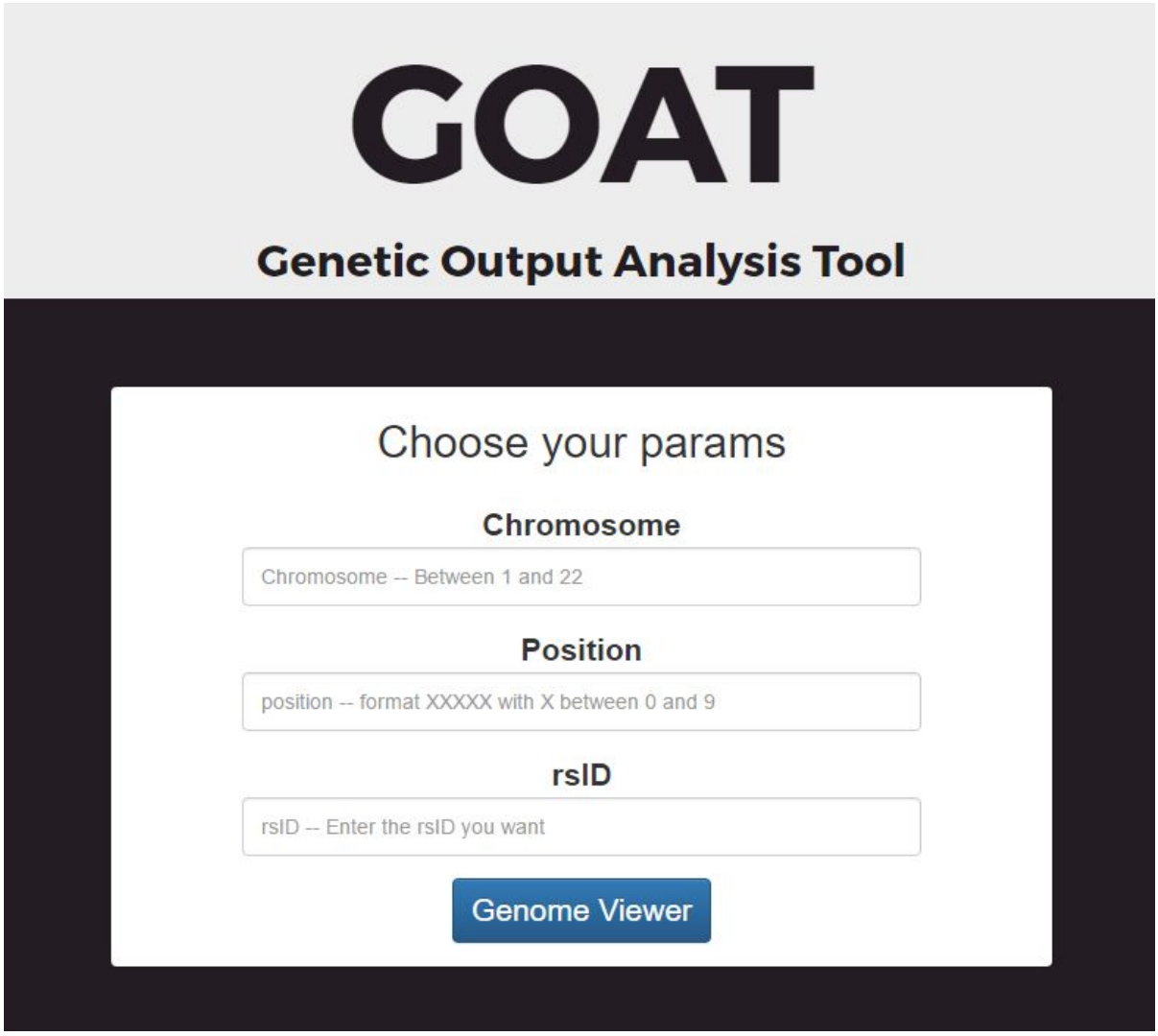


Figure 21: Interface utilisateur de la fonction ADAM dans le front-end

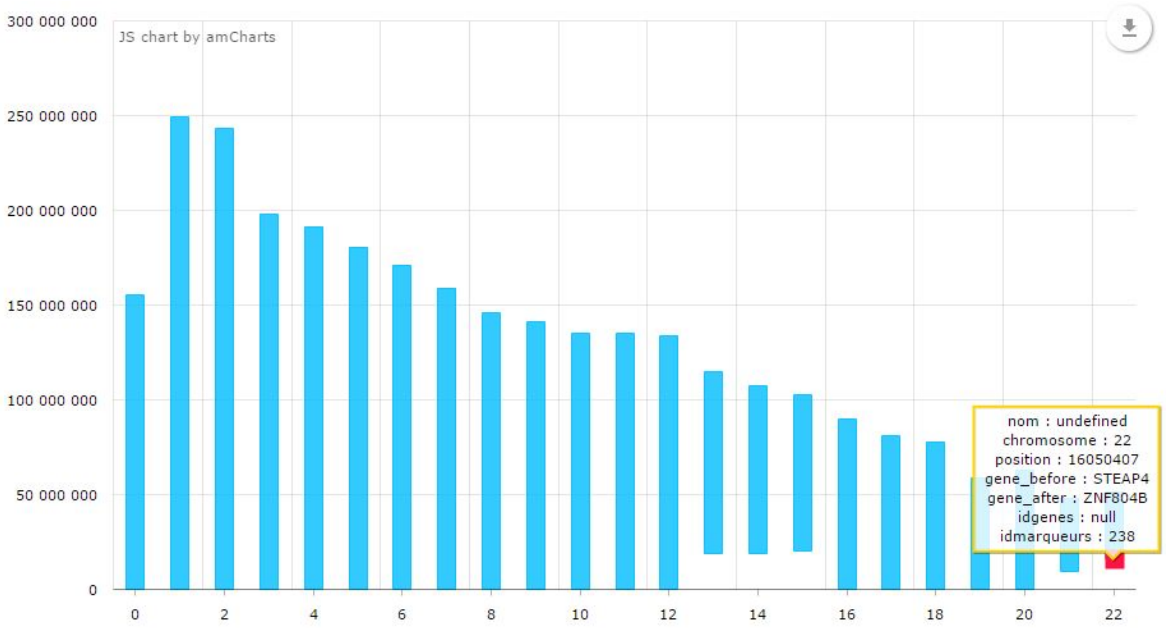


Figure 22: Graphique retourné après l'exécution de la fonction d'ADAM

Difficultés rencontrées

Front-end

Démarrage du projet

Lors du démarrage du projet, nous avons eu beaucoup de difficultés à faire fonctionner l'application. Le projet contenait peu de documentation et le fichier *requirements.txt* très important dans les projets utilisant Python était manquant. En effet, ce fichier liste toutes les librairies et versions requises afin de faire fonctionner le projet. Une seule commande peut être exécutée pour installer toutes ces librairies. N'ayant pas accès à cette ressource, nous avons découvert les librairies nécessaires par essais et erreurs.

Par contre, avec cette méthode, nous avons eu un autre problème. Sans nous en rendre compte, nous avions installé une version trop récente d'une certaine librairie. Étant donné que le projet utilisait une version antérieure, certaines fonctionnalités de la librairie n'avaient plus la même syntaxe et faisaient planter le projet. Une erreur obscure était lancée et nous avons dû travailler fort afin de comprendre ce qui se passait. Face à ces difficultés, nous avons apporté plusieurs améliorations.

Premièrement, nous avons amélioré la documentation afin de définir, étape par étape, les outils et logiciels à installer afin de démarrer le projet sans problème. Deuxièmement, nous avons évidemment ajouté le fichier *requirements.txt* au projet pour connaître rapidement les librairies à installer. Troisièmement, nous avons amélioré le processus d'installation en général en y ajoutant l'utilisation de l'outil *Virtualenv*. Cet outil permet d'avoir un environnement virtuel vierge et d'y installer les librairies nécessaires. Cela évite d'avoir des conflits avec d'autres librairies possiblement installées sur l'ordinateur et permet de gérer les versions facilement pour différents projets.

Processus de développement

Étant donné que nous avions à utiliser une base de données MySQL contenant plusieurs millions de données pour le développement des fonctionnalités front-end, les requêtes prenaient entre 3 et 5 minutes à s'exécuter. Cela ralentissait énormément le temps de développement et rendait la tâche de comprendre le code et le flux de l'application très fastidieuse.

À ce point-ci du projet, il était inutile d'essayer d'optimiser les requêtes ou la base de données puisque la deuxième partie du projet consistait à développer un back-end complètement différent et plus performant.

Afin d'accélérer le processus de développement, nous avons ciblé la table contenant les données génétiques nécessaires au développement des fonctionnalités demandées et nous avons tout simplement tronqué cette table. En ayant seulement environ 50 entrées dans notre nouvelle table, cela nous a permis de développer rapidement et plus efficacement.

React

N'ayant jamais travaillé avec la librairie React auparavant, il a fallu s'apprivoiser avec celle-ci et comprendre le flux du code déjà en place. Étant une librairie 100% asynchrone qui permet de faire des SPA, il n'était pas toujours évident de suivre le flux de l'application. La courbe d'apprentissage était parfois élevée.

De plus, la relation avec le cadriciel Django utilisé était différente de ce que nous avions expérimenté dans le passé et amenait des questionnements dans certains cas.

Génération de graphique avec Bokeh

Nous avons passé une partie importante de la durée du projet à travailler avec Bokeh pour représenter notre graphique. Nous étions même plutôt avancés étant parvenus au résultat suivant :

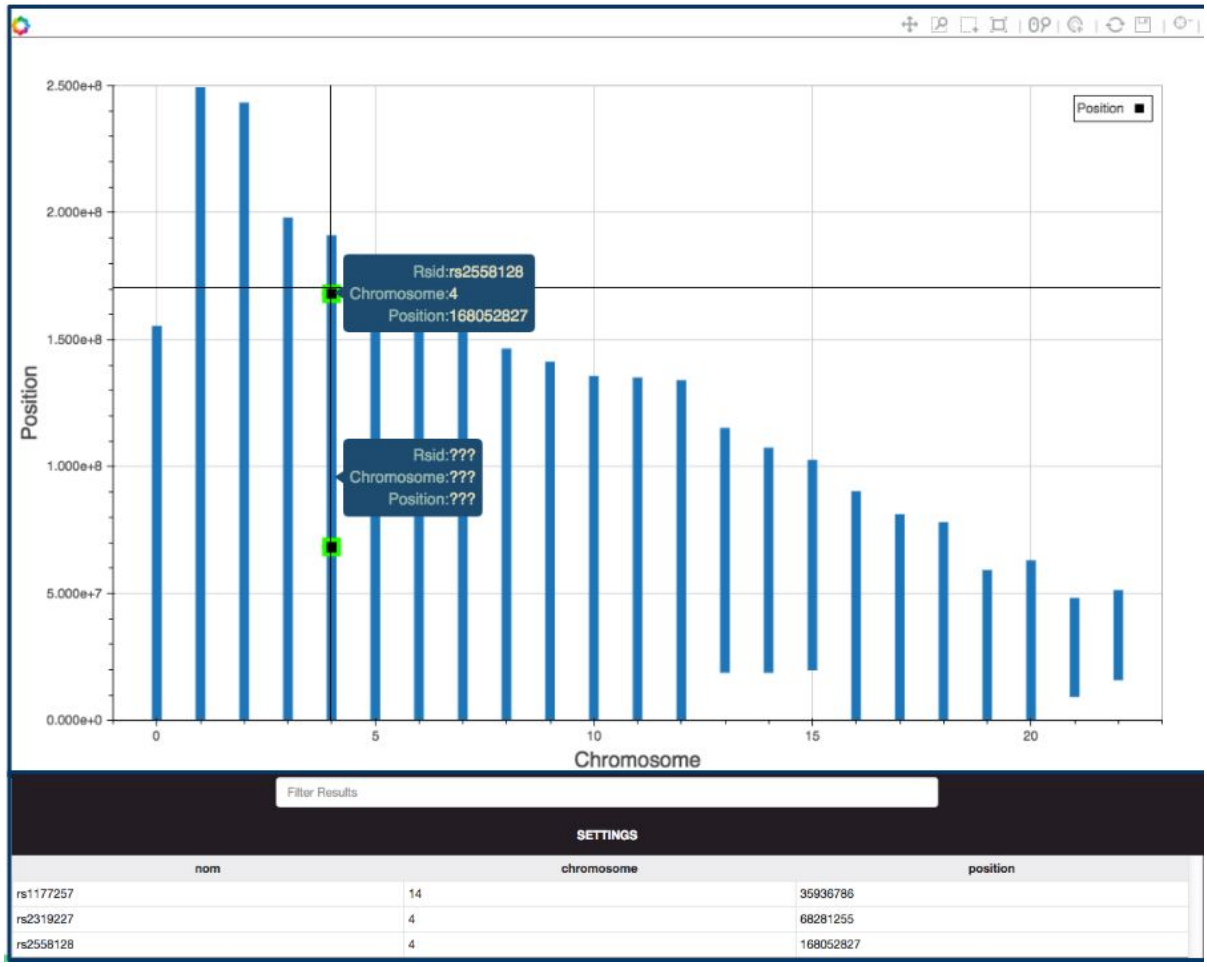


Figure 23: Affichage d'un graphique GenomeViewer avec Bokeh (retiré de l'application)

Cependant, parvenus à ce point, l'étape suivante était de faire l'affichage des légendes des phénotypes et des mutations afin de permettre de filtrer l'affichage des rsids selon ces associations. C'est à ce moment que nous avons encontré un problème majeur : l'ajout d'intéractivité dans le graphique Bokeh. Dû à une documentation de faible qualité, nous avons été incapables de surmonter ce défi. Au final, il fut décidé de changer de librairie et de se tourner vers AmCharts, une librarie javascript.

Génération de graphique avec AmCharts

Ce fut un immense retour en arrière que de changer complètement de librairie d'affichage graphique. D'autant plus que cela changeait grandement le flux de l'information dans l'application. Avec Bokeh, les données étaient traitées du côté serveur en python. Avec AmCharts, du côté serveur en python, on doit bâtir un objet Json qui est envoyé à la page JSX. Ensuite, en javascript, cet objet Json est utilisé

afin d'initialiser le graphique.

Malgré tout celà, ce fut une bonne décision car AmCharts était beaucoup plus facile à utiliser et le documentation était de loin supérieure.

Nous sommes éventuellement arrivés au résultat suivant :

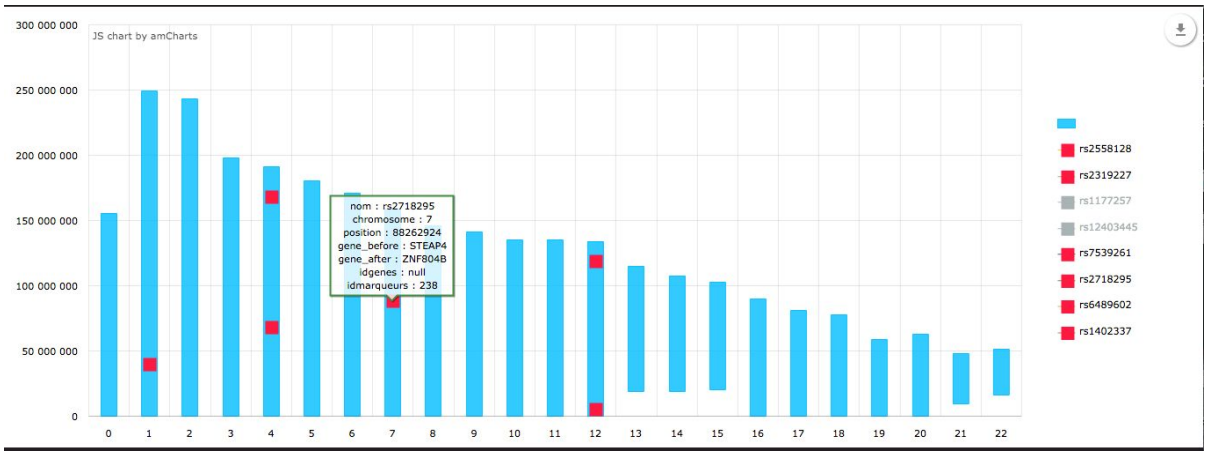


Figure 24: État du graphique GenmeViewer reproduit avec AmCharts

Cependant, nous avions encore la même difficulté à créer des légendes pour filtrer l'affichage de phénotypes et de mutations. À cette étape très avancée du projet, ce qui a sauvé la mise fut une demande de support à l'équipe d'AmCharts. Ces derniers ont été phénoménaux et nous ont rapidement créé un exemple entièrement personnalisé reproduisant nos données afin de nous montrer comment parvenir à nos fins. Cet exemple fut utilisé et nous à permis de réussir l'ajout des légendes, tel que présenté plus haut dans le rapport.

Back-end

Installation d'ADAM

Lors de l'installation d'ADAM sur une instance EC2, nous avons encouru un problème face à la version de Spark installé. ADAM est par défaut compilé avec la version 1.6 de Spark.

Après quelque temps, il a été possible de retrouver un cas semblable aux nôtres qui

nous a dirigé vers le script “move_to_spark2.sh” qui permet de modifier ADAM afin de permettre une compilation avec Spark 2.

Modification d’ADAM

La modification d’ADAM nous a donné beaucoup de fil à retordre. Premièrement, ADAM est développé en Scala. Il s’agit d’un très bon langage qui rencontre les exigences du projet, mais nous n’avions aucune expertise dans ce langage au sein de l’équipe. Ceci a grandement ralenti notre rythme de développement. Un autre aspect problématique était qu’après chaque petite modification, nous devions recompiler le projet au complet pour la tester. La compilation prenant 5 minutes à chaque fois, les séances de débogages pouvaient s’étirer sur plusieurs heures. Finalement, nous avons mis du temps à comprendre comment la sérialisation des tâches fonctionne pour être en mesure d’exécuter une commande sur un cluster. Une fois cette étape surmontée, le développement s’est très bien passé.

1000 Genomes

Lors des premiers tests, il était nécessaire de télécharger des fichiers VCF de 1000 Genomes. Cependant, les fichiers extraits de 1000 Genomes sont plus de 100GB une fois extraits.

L’instance EC2 a cependant réglé ce problème puisqu’il était possible d’agrandir l’espace disque des instances sans problème.

Instance EC2

Les instances EC2 sont instanciées avec une version de CentOS modifié appelé Amazon Linux. Cette version de Linux ne fonctionne pas comme une version de CentOS originale. Cela a causé plusieurs problèmes au niveau de l’installation de logiciel, mais cela a été résolu en lisant la documentation fournie par Amazon.

Afin de se connecter sur une instance EC2, il est demandé d’utiliser un certificat PEM. Ce certificat est fourni lors de la création d’une instance. Une fois perdu, il sera impossible de se connecter à la machine. Il était donc difficile de partager un tel certificat entre les membres de l’équipe.

Afin de résoudre ce problème, nous avons abaissé les paramètres sécurité afin de

permettre une simple connexion SSH avec mot de passe.

Recommandations

Front-end

Quatre recommandations ont été identifiées pour le frontend de l'application. Il s'agit de fonctionnalités présentes qui devraient être améliorées, ou de fonctionnalités manquantes qui devront être réalisées.

Téléversement de fichiers VCF
Malheureusement, par manque de temps, cette fonctionnalité n'a pas pu être achevée. Elle présentait un niveau de complexité beaucoup plus élevé que pour le téléversement des fichiers CSV. Comme il serait important d'offrir de la flexibilité aux docteurs qui utiliseront l'application, il serait important de permettre, dans le futur, le téléversement de différent formats de fichiers.

Entrée de plusieurs rsid
L'application devait permettre d'entrer manuellement plusieurs rsid à la fois. Dans cette section de notre application, il n'est possible que d'entrer seulement un rsid. En réalité, cette section fut utilisée pour connecter avec le nouveau back-end ADAM dans le but de faire une preuve du concept. L'interface pourrait facilement être modifiée pour inclure une sélection multiple de rsids. Encore une fois, cette fonctionnalité serait importante afin d'offrir une plus grande flexibilité aux docteurs utilisant l'application et ainsi de leur permettre une plus grande efficacité.

Meilleur filtrage des rsid selon les légendes
En ce moment, bien que fonctionnel, le filtrage n'est pas idéal. Un clic sur un des éléments de la légende inverse l'état des rsids correspondants. Ce serait mieux si les légendes étaient faites sous forms de boîtes cochables afin de déterminer précisément l'état de l'affichage. Cela permettrait au docteur de visualiser plus efficacement le contenu du graphique.

Étape 3 de GenomeViewer
N'ayant pas eu le temps de nous attaquer à cette fonctionnalité, elle devra être réalisée ultérieurement. Il s'agit sans aucun doute d'une fonctionnalité cruciale au succès futur de l'application. Cette étape devrait être réalisée lors de la prochaine

itération du projet afin de prouver son potentiel.

Back-end

Malgré tout le travail accompli, il ne s'agit quand même d'une seule itération sur cet immense projet. Il reste beaucoup de travail à faire pour arriver à une solution idéale et voici quelques points qui pourraient être améliorés dans un futur proche.

Tout d'abord, la configuration du cluster EMR et des instances EC2 pourrait être automatisée. Cette automatisation pourrait facilement se faire grâce à l'outil Docker qui permet de créer un conteneur ayant tout ce qui est nécessaire à l'application et qui peut être déployé rapidement sur une instance amazon.

De plus, l'utilisation du service ECS (EC2 Container Service) d'amazon permettrait de créer et détruire les instances EC2 automatiquement dépendamment de l'utilisation. Ceci permettrait de réduire grandement les coûts associés à EC2 et de ne plus avoir à se soucier de la création et destruction des instances.

La fonctionnalité ADAM développée fonctionne bien, mais prend beaucoup de temps à sauvegarder les variantes qui ne sont pas présentes en cache. Ceci est dû au fait que la sauvegarde est effectuée au niveau de la commande et non au niveau du RDD. Comme mentionnée dans la section Implémentation, cette partie de la fonctionnalité pourrait être optimisé grandement. De plus, aucun test pour notre nouvelle fonctionnalité n'a été développé ce qui rend la tâche de test longue et ardue. Nous suggérons de développer les tests au fur et à mesure du développement pour le futur du projet.

L'une des prochaines étapes logiques serait aussi de développer une fonctionnalité qui permet de sélectionner plusieurs rsid à la fois. Cette fonctionnalité est présentement utilisée dans le front-end dans la page "SQL Genome Viewer". Cette fonctionnalité permettrait de détruire la base de données MySQL et continuer seulement le travail dans le nouveau back-end. De cette façon, une transition complète pourrait être faite et seulement utiliser ADAM et les données de 1000 Genomes.

Finalement, considérant que la conversion des fichiers de 1000 Genomes de VCF vers adam à prit plusieurs heures, nous recommandons de demander à Amazon

d’ajouter la version adam des VCF dans leur bucket gratuit pour éviter à tout le monde de prendre des heures à tout convertir.

Conclusion

En conclusion, le projet GOAT est un projet très prometteur qui a une portée très large. Une fois les objectifs atteints, ce projet permettrait à des médecins et des chercheurs de facilement comparer le génome humain et le tout, avec un outil graphique facile à utiliser. À terme, cet outil permettrait même de sauver des vies. Toutefois, la portée étant très large, notre équipe s'est penchée sur comprendre et analyser l'état actuel de l'application pour en faire ressortir les points à améliorer. Une fois cette tâche accomplie, nous avons pu détailler un plan d'action permettant d'amener le projet GOAT là où il devait aller. Ce plan d'action contenait notamment des recommandations sur la structure même de l'application, mais aussi sur la façon que les fonctionnalités devraient être implémentées dans le frontend.

Notre équipe s'est donc attaquée à cette liste de recommandation, en reprenant le travail pratiquement du début. En revenant au fondement même de l'application, nous avons pu trouver des solutions efficaces pour atteindre les objectifs du projet. L'ajout des nouvelles fonctionnalités, le déploiement sur Amazon et la connexion avec le cadriciel ADAM ne sont que quelques exemples du travail qui a été accompli par notre équipe tout au long de la session.

Pour terminer, nous sommes tous fiers d'avoir pu participer à un tel projet et nous espérons que le travail qui a été accompli par notre équipe dans le cadre de ce projet de fin d'études a permis de remettre le projet sur les rails et qu'il a un futur brillant devant lui.

Bibliographie

"Django." The Web Framework for Perfectionists with Deadlines | Django. https://www.djangoproject.com/.

"Introducing JSX." React. https://facebook.github.io/react/docs/introducing-jsx.html.

"JavaScript Charts & Maps." AmCharts. https://www.amcharts.com/.

"A JavaScript Library for Building User Interfaces - React." https://facebook.github.io/react/.

Okonet. "Okonet/react-dropzone." GitHub. https://github.com/okonet/react-dropzone.

Reflux. "Reflux/refluxjs." GitHub. https://github.com/reflux/refluxjs.

Contributors, Bokeh. "Welcome to Bokeh." Welcome to Bokeh — Bokeh 0.12.5 Documentation. http://bokeh.pydata.org/en/latest/.

"IGSR: The International Genome Sample Resource." 1000 Genomes | A Deep Catalog of Human Genetic Variation. http://www.internationalgenome.org/

ADAM. “bigdatagenomics/ADAM” Github. https://github.com/bigdatagenomics/adam

Spark, “Apache Spark Documentation” Documentation: http://spark.apache.org/documentation.html

Scala, “Community-driven documentation for Scala.”

Documentation: http://docs.scala-lang.org/

Maven, “Welcome to Apache Maven.”

Documentation: https://maven.apache.org/

AWS Big Data, “AWS Big Data Blog”

Blog: https://aws.amazon.com/fr/blogs/big-data/

ADAM, “Genome analysis with adam”

Documentation:

http://ampcamp.berkeley.edu/5/exercises/genome-analysis-with-adam.html

S3, “How to Mount S3 Bucket on CentOS and Ubuntu using S3FS”

Documentation:

https://www.interserver.net/tips/kb/mount-s3-bucket-centos-ubuntu-using-s3fs/

Déploiement sur EC2, “How To Set Up Django with Postgres, Nginx, and Gunicorn on Ubuntu 14.04”

Documentation:

https://www.digitalocean.com/community/tutorials/how-to-set-up-django-with-postgres-nginx-and-gunicorn-on-ubuntu-14-04

Donnés 1000 Genomes, “Projet 1000 Genomes et AWS”

https://aws.amazon.com/fr/1000genomes/

Annexes

Rencontres SCRUM - GOAT Hiver 2017

19 Janvier 2017

Rencontre:

- Lieu : ÉTS 9h
- Personnes présentes
    - Émile Filteau, ÉTS
    - Max St-Onge, ÉTS
    - David Guay, ÉTS
    - Raphaël Papillon, ÉTS
    - Beatriz Kanzki, Hangout

Ce qui a été fait (heures travaillées : 12 heures total) :
Recherche et documentation sur les différentes technologies reliées au projet.

- **Spark :** Engin de calcul développé par apache permettant d'effectuer du traitement big data.
- **Parquet :** Format de fichier compressé optimisé pour le big data.
- **Avro** : Système de sérialisation utilisé par Spark pour sauvegarder les documents avec Parquet.
- **Adam :** Framework développé par dessus Apache Spark pour effectuer du traitement spécifique aux séquences d'ADN écrit en scala.

Recherche sur les différents services d'amazon qui pourrait être utilisé conjointement avec les technologies.

- **EC2 :** Système de cloud computing permettant d'exécuter des logiciels sur des conteneurs prédéfinis.
- **ECS :** Système de gestion des conteneurs pour EC2. Ce service permet de "scaler" l'application en créant des instances EC2 à la demande.
- **ECR :** Gestionnaire d'image Docker utilisé par ECS.
- **EMR :** Système permettant d'héberger notre instance d'Apache Spark.
- **S3 :** Système de fichier dans le cloud accessible par les conteneurs EC2.

Discussion avec Beatriz sur les prochaines étapes du développement du logiciel. Nous avons demandé à Beatriz de nous préparer une démo pour mieux saisir les requis du

système. Nous avons également demandé à Beatriz de nous présenter une liste des fonctionnalités requises pour la prochaine version de GOAT.

Éléments bloquants :

- Les membres de l'équipe n'ont pas tous accès à Amazon AWS.

Plan d'action :

- Rencontre lundi prochain entre les membres de l'équipe ayant pour but d'exécuter le logiciel existant et d'en sortir les fonctionnalités et défauts.
- Déployer une instance d'Apache Spark sur EMR.

26 Janvier 2017

Rencontre:

- Lieu : Bibliothèque ÉTS
- Personnes présentes
    - Alain April, ÉTS
    - Raphaël Papillon, ÉTS
    - David Guay, ÉTS
    - Beatriz Kanzki, Hangout

Ce qui a été fait (heures travaillées : 12 heures total):

- Rencontre entre les membres de l'équipe le lundi 23 janvier.
    - Test du logiciel actuel
    - Ébauche de l'architecture pour la version 3 de GOAT
    - Roadmap de ce que nous devrons faire dans les prochaines semaines

Éléments bloquants :

- Charger la base de données.
- Problème de permission sur AWS.

Ordre du jour SCRUM :

- Voir image 1 et 2

Plan d'action de la semaine:

- Beatriz : Envoyer les spécifications fonctionnelles pour modifier GOAT.
- Équipe : Charger GOAT avec des données et tester AreaSelection seulement.
- Alain: Donner accès et les droits sur AWS à toute l'équipe.
- Équipe : Faire les recommandations pour le projet.
- Équipe : Lire le rapport de maîtrise de Cédric et Victor Dupuy.

2 février 2017

Rencontre:

- Lieu : Hangout
- Personnes présentes
    - Raphaël Papillon, ÉTS
    - David Guay, ÉTS
    - Émile Filteau, ÉTS
    - Max St-Onge, ÉTS

Ce qui a été fait (heures travaillées : 12 heures total) :

- Essayer d'exécuter GOAT avec la base de données MySQL.
- Élaborer des recommandations pour le projet.
- Lecture des rapports de maîtrise de Victor et Cédric.

Recommandations :

Priorité	Recommendations	Objectifs
1	Migrer les données vers le format Adam et les déployer sur le Cloud ● Documenter le processus ● Documenter le résultat	● Avoir un format de données intéressant et utilisable
2	Rendre la plateforme de calcul open-source (sur Github) ● Documenter le projet (issues, etc)	● Permettre des itérations futures du projet ● Contribuer à la communauté open-source ● Bonne visibilité pour l'ETS
3	Faire un ménage dans le projet GOAT actuel ● Enlever fonctionnalités non utilisées ● Enlever le côté "MySQL"	● Avoir du code à jour
4	Si possible, connecter la fonctionnalité "Area Selection" de GOAT avec les données avec Spark-Adam	● Faire une preuve de concept de la scalabilité de la plateforme et du projet

Éléments bloquants :

- Incapable de faire fonctionner GOAT.

Ordre du jour SCRUM :

- Présenter les recommandations à Beatriz.

- Se renseigner sur ce que Phaudyl peut nous donner.

Ce qui s'est dit pendant la rencontre :

- C'est probablement beaucoup plus complexe que l'on pensait de migrer les données vers ADAM-Spark
- Pour avancer le projet, avoir deux personnes qui travaillent sur GOAT_v2 afin d'ajouter la fonctionnalité à Beatriz.
- Décider si on veut déployer ADAM-Spark en local ou sur le cloud, et si on veut un petit subset de données ou l'entièreté (par Phaudyl).

Plan d'action de la semaine:

- Raphaël et David vont installer GOAT_v2 sur leur poste
- Ils vont évaluer les modifications à apporter dans l'interface pour ajouter la fonctionnalité demandée par Beatriz.
- Émile et Max vont fournir la liste des étapes à faire par Alain afin de poursuivre le travail fait pour avoir les données dans un format ADAM et Spark.

9 février 2017

Ce qui a été fait (heures travaillées : x heures totales):

- Installation de GOAT_v2 sur les ordinateurs individuels (15 heures)
- Faire fonctionner GOAT_v2 sans succès même avec la base de données (10 heures)
- Installation complète de v3 et tests d'utilisations sur l'ordinateur à Raphaël (6 heures)
- Discussion sur la méthode à utiliser pour convertir les données vers le format ADAM. (1 heure)
- Installation de ADAM et Sparks (2 heures)
- Familiarisation avec ADAM et les différentes fonctionnalités offertes (2 heures)
- Lecture sur le Framework Play (2 heures)
- Installation du Framework Play avec Scala (2 heures)
- Élaboration d'un concept pour l'exécution d'ADAM au sein du framework Scala (2 heures)
- Début d'un site web de base avec le framework Play (2 heures)

Difficultés rencontrées :

- Trouver les bonnes versions de framework/librairies à installer
- Problèmes d'URL avec le site web
- Problèmes de variables globales dans le logiciel
- Path vers les fichiers utilisés pour générer le graphique d'AreaSelection sont "hardcodée"
- La logique pour Area Selection semble ne jamais être appelée dans le code (placée après un retour)
- Une seule vue avec un seul URL
- La courbe d'apprentissage du langage Scala est plus abrupte que prévue

Éléments bloquants :

- On ne trouve pas la valeur de threshold que Beatriz nous avait dit de mettre à 0 (ça semble être seulement dans v3)
- Incapable de générer un graphique dynamique dans AreaSelection
- Éléments de Goat_v3 à réutiliser pour créer la nouvelle fonctionnalité

Ordre du jour SCRUM :

- Présenter l'avancement de la partie back-end
- Présenter l'avancement de la nouvelle fonctionnalité de Goat_v3

Ce qui s'est dit pendant la rencontre :

- Prendre le même query que Manhattan pour faire le graphique de AreaSelection

- Dans la nouvelle fonctionnalité
    - Il faudrait pouvoir sélectionner les Rsid à traiter parmi ceux présents dans le fichier téléverser.
    - Quand il y aura un match, il faudra afficher le phénotype dans une boîte à gauche
    - Le tableau en bas correspond aux données entrées par le chercheur.
    - Pour commencer, utiliser les champs, Rsid, position et chromosome seront suffisants
    - Une barre noire représente un exon. Les barres blanches représentent les “non-exon”.
- Dans AreaSelection, on pourra enlever le graphique vert et seulement afficher les boites en bas.

Plan d'action de la semaine:

- Essayer d’avancer un peu les 2 parties malgré notre semaine d’examens.
- Nouvelle fonctionnalité :
    - Faire le routage et créer de nouvelles vues dans l’application
- ADAM + Spark

16 février 2017

Ce qui a été fait:

- Familiarisation avec React.js (2 heures)
- Tentatives initiales afin d'amener la nouvelle fonctionnalité (valider le concept avec Beatriz) (2 heures)
- Installation de gulp et ses dépendances (30 minutes)
- Création d'un fichier package.json pour pouvoir installer les dépendances Node nécessaires au projet
- Installation d'ADAM (1 heure)
- Familiarisation avec ADAM afin de pouvoir ajouter une fonctionnalité (2 heures)
- Installation et configuration d'une nouvelle application Django
- Ajout d'une fonctionnalité dans l'application Django afin d'appeler ADAM

Requête qui retourne les positions minimales et maximales d'un chromosome. Permettrait de créer la barre complète d'un chromosome et d'ainsi échelonner correctement les matchs qui seront faits par la suite. Le range du chromosome 4 est de

190 939 665 - 27 955 = **190911710**

```
SELECT chromosome, MIN(position) as min, MAX(position) as max FROM marqueurs GROUP BY chromosome;
```

Query Favorites ▾ Query History ▾

chromosome	min	max
0	195	155239436
1	37965	249222325
2	10587	243185679
3	60197	197897481
4	27955	190939665

Requête que retourne le Rsid avec la position supérieure la plus proche d'un rsid connu et de sa position Permettrait de représenter la barre jaune dans le graphique. Dans ce cas, rs2558128 et position=168052827 est suivi de rs2309396 et position=168053128. La longueur à afficher serait donc de

168053128 - 168052827 = **301.**

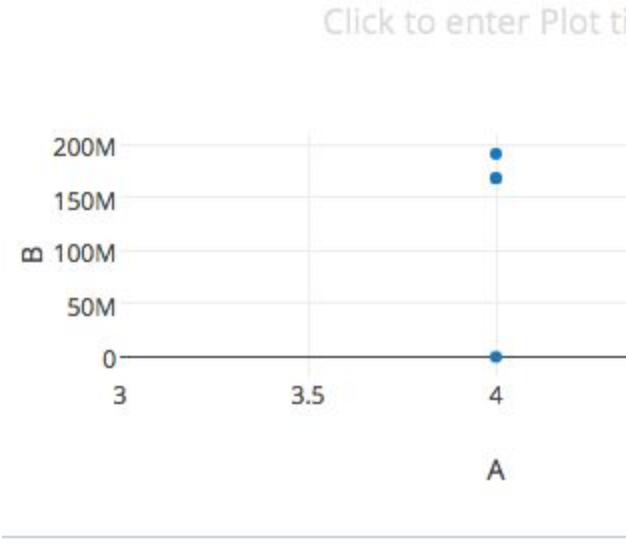
```
select nom, min(position) as end_position
from marqueurs
where chromosome=4
and position > 168052827;
```

Query Favorites ▾ Query History ▾

nom	end_position
rs2309396	168053128

En représentation graphique, on aurait donc le résultat suivant. Cependant, les 2 points du rsid ne sont pas distincts et apparaissent comme un seul point.

1	4	27 955	début chromosome
2	4	168052827	début rs2558128
3	4	168053128	fin rs2558128
4	4	190 939 665	fin chromosome



Application Django backend:

- Développement d'une preuve de concept d'une application python avec django qui utilise le framework ADAM pour faire des requêtes au cluster apache Spark.
- Développement d'un package python qui interface avec le framework adam, mais en python.
- Lecture et recherche dans les scripts de migration de données vers le format adam.

http://127.0.0.1:8000/kmers/

(TTAG,4) (CCTA,6) (ACCC,4) (GTTG,8) (CTTG,4) (TTTC,24) (TGTA,11) (CTGC,9) (CGTT,2) (ACCA,4) (ATGA,3) (TCTG,1) (TCAC,9) (ACGG,2) (CAAA,10) (AGAT,3) (CACA,5) (GAGT,4) (TGGA,8) (AAAA,18) (GAAC,2) (ATAT,8) (CTGG,7) (ATGT,8) (GATC,2) (AAGA,9) (CGGT,1) (GACG,1) (GCCC,8) (GAAG,2) (GGGA,4) (TAGT,3) (CCCG,1) (TGTG,4) (CCAC,5) (TTTG,11) (CACG,2) (AACG,3) (CCTT,5) (TCCC,11) (AGAA,4) (ATTG,5) (TTAT,14) (AGCT,9) (TAAA,13) (GGAG,4) (GTCT,4) (CGAT,2) (GTTA,2) (TCCT,9) (GTAC,2) (TTTT,26) (CGGG,1) (TCGT,1) (GGTG,7) (CCAT,7) (GCTA,4) (CGCT,1) (AAAC,11) (TCTC,11) (GTAA,8) (CCCC,3) (ATTC,6) (GAAA,6) (TTGA,8) (ATAC,2) (ATCT,6) (ATGC,7) (CACC,8) (GGGG,3) (GGGT,4) (CCAA,13) (AACC,3) (TAAG,5) (ACGC,1) (AGGA,5) (ACAA,7) (CCTG,10) (GAGG,6) (ATTA,6) (GGTT,5) (GTGG,5) (GTTC,2) (ACTT,5) (AGAG,5) (CGTG,1) (GATG,4) (CAAC,4) (ATAG,2) (GCTT,3) (TGGG,5) (TTGT,8) (AGTA,7) (AAAT,9) (CTTA,3) (TGCC,14) (AGAC,3) (CGCG,1) (CAAT,6) (CGCC,3) (TGCT,5) (TTTA,20) (TCCA,6) (CTCT,6) (GTGT,4) (CTAC,4) (GGAA,6) (TGGT,3) (CAGC,9) (GACA,2) (CATC,5) (CATG,9) (CCGG,1) (GACT,6) (CTAG,4) (TGCA,7) (GGAC,3) (TTGC,4) (CAGA,2) (CCAG,9) (GTGC,6) (ACTC,3) (GCAT,5) (GCAG,6) (ACGT,2) (GGCC,8) (TACA,4) (AGTC,5) (CATT,4) (AGGG,7) (TAGA,2) (CCCA,16) (ACAC,2) (TACT,5) (AGTG,10) (GATA,3) (ATCC,4) (AGGC,7) (GGGC,4) (GGTC,4) (GTCC,4) (GCGC,1) (GTTT,10) (GCAC,6) (CTGA,4) (ATAA,10) (CTTT,25) (GGCT,6) (TCAA,9) (GAGC,5) (GGCA,6) (CCTC,11) (ATTT,18) (GGAT,2) (AGCA,7) (CAAG,12) (AGCC,5) (ATCA,5) (AAGG,2) (GCTG,8) (ACAG,9) (ACTG,9) (TCCG,2) (AATG,6) (CAGG,11) (GAGA,2) (AAGC,6) (CCGC,3) (TAGG,8) (TCGC,1) (GAAT,5) (CGCA,1) (CTAT,5) (TCTA,2) (TTCA,6) (AGTT,4) (CTCG,3) (TTAA,11) (AATT,12) (TGGC,7) (CTTC,4) (AACA,10) (CCCT,6) (CCGA,1) (TAAT,11) (TTAC,2) (CTAA,6) (TTCC,9) (AATA,7) (TGAT,6) (GCCT,13) (AAAG,10) (TCGG,1) (TATA,9) (GCAA,3) (TAGC,1) (CTCC,11) (AAGT,11) (TATC,4) (GTAT,5) (TGAG,9) (CTGT,8) (ACTA,6) (TGAA,5) (ACAT,3) (ACCT,3) (TTCT,21) (GTGA,7) (GATT,4) (TGAC,2) (TCAT,6) (TACC,2) (TCAG,6) (TGTT,12) (CGGC,2) (GCCA,8) (CTCA,14) (GCTC,7) (GTCA,4) (AATC,4) (TATG,4) (TGTC,3) (TCTT,22) (CACT,6) (ACCG,2) (TAAC,7) (ATGG,4) (AGGT,8) (CAGT,8) (CATA,3) (TTGG,7) (ACGA,1) (TATT,15) (GTAG,4) (AACT,7)

Difficultés rencontrées :

- Manque de temps dû à la semaine d'examens
- Si un rsId est unique et représente toujours les mêmes chromosomes/positions, pourquoi ceux-ci doivent-ils être paramétrables dans le téléversement?

Éléments bloquants :

- Concept de représentation graphique à valider.
- Les modifications faites au front-end dans les fichiers React.js ne sont pas pris en compte.
- Les tâches gulp ne semblent pas aider…
- Accès au script de conversion de données

Ordre du jour SCRUM :

- Tour de table sur les tâches accomplies dans la semaine précédente
- Tâches qui seront accomplies dans la prochaine semaine
- Éléments bloquants

Ce qui s'est dit pendant la rencontre :

- Béatriz va fournir une spécification technique pour un modèle de données plus simple
- Alain va envoyer des gabarits de rapport de proposition pour nous aider

Plan d'action de la semaine:

- Écriture d'un rapport de proposition
- Exploration de la spécification qui sera fournie par Béatriz
- Solidifier la structure du package python-adam et ajouter des fonctionnalités

23 février 2017

Ce qui a été fait:

- Cibler et régler le problème avec React.js (6 heures)
- Ajout d'une page au niveau du front-end pour commencer le nouveau AreaSelection. (2 heures)
- Ajout d'une librairie pour transférer un fichier sur la nouvelle page. (2 heures)
- Familiarisation et expérimentation avec les scripts créés par Simon Grondin (2 heures)
- Création d'une nouvelle méthode dans ADAM (2 heures)
- Rapport d'étape (4 heures)

Difficultés rencontrées :

- Incapacité à modifier les vues React

Éléments bloquants :

- Aucun pour le moment

Ordre du jour SCRUM :

- Tour de table sur les tâches accomplies dans la semaine précédente
- Tâches qui seront accomplies dans la prochaine semaine
- Éléments bloquants

Ce qui s'est dit pendant la rencontre :

- Discuter de ce qui a été fait dans le rapport d'étape, et ce qui reste à faire
    - Mettre les références et annexes
- Avancement du front-end
- Avancement du back-end

Plan d'action de la semaine:

- Max et Émile vont
- Raphaël et David vont :
    - Tenter de permettre d'uploader un fichier csv
    - Tenter de représenter un graphique quelconque avec Bokeh

2 mars 2017

Ce qui a été fait (heures travaillées : x heures total):

- Tentative de passer un fichier uploadé à une vue Django en passant par React et un POST asynchrone (5 heures)
- Représenter nos données dans un graphique Bokeh. Pour l'instant, une requête SQL sélectionne les positions min et max pour chaque chromosome et le représente dans un graphique (5 heures)
- Nous avons de nouveau cherché une manière de pouvoir accélérer le temps des requêtes. Nous avons finalement opté pour créer une copie de la table marqueurs, dans laquelle nous pourrons insérer des copies exactes des éléments que nous souhaitons utiliser dans notre interface. Ainsi, pour le moment, nous avons créé une requête SQL pour exporter les marqueurs correspondant aux positions minimum et maximum de chaque chromosome. On exporte ces données dans une nouvelle table afin d'accélérer le développement (2 heures)

```
INSERT INTO marqueurs_copy

SELECT ma.*

FROM marqueurs ma INNER JOIN
    (
        SELECT chromosome, MIN(position) MinPosition
        FROM marqueurs
        GROUP BY chromosome
    ) t ON ma.chromosome = t.chromosome AND ma.position = t.MinPosition;
```

- Suite des tests d'affichage sous Bokeh, nous pouvons maintenant afficher une forme de rectangle qui pourra être utilisée afin de représenter les chromosomes (2 heures)
- Migration des données actuelles vers le format ADAM (3 heures)
    - Les scripts utilisent une base de données PostgreSQL, nous devons donc migrer nos donnés vers une base de données PostgreSQL.
- Implémentation d'une nouvelle fonctionnalité dans ADAM afin de mieux comprendre le fonctionnement de la plateforme en prévision de l'implémentation du Area Selection. (3 heures)
    - Il sera possible d'implémenter la fonction d'Area Selection lorsque les donnés seront migrés.
- Finalisation du rapport d'étape (2 heures)

Difficultés rencontrées :

- Comprendre le flow entre React et Django pour envoyer des données dans une requête POST
- Comprendre le flow des informations entre la base de données et les graphiques Bokeh

- Les scripts fournis par Simon Grondin utilisent un des connecteurs PostgreSQL. Afin de pouvoir faire la migration, nous devons migrer les donnés actuels en format PostgreSQL.

Éléments bloquants :

- Incapable de faire un POST avec un fichier .csv

Ordre du jour SCRUM :

- Tour de table sur les tâches accomplies dans la semaine précédente
- Tâches qui seront accomplies dans la prochaine semaine
- Éléments bloquants

Ce qui s'est dit pendant la rencontre :

- Avancements
- Poursuivre le travail sur le front-end et le back-end.

Plan d'action de la semaine:

- Faire fonctionner le POST avec le .csv
- Poursuivre le graphique Bokeh (afficher les chromosomes, faire un match de rsId)

9 mars 2017

Ce qui a été fait:

- Upload d'un fichier csv. Analyse du fichier csv pour en extraire les entêtes. Passer ces entêtes à la vue afin de permettre à l'utilisateur de faire la correspondance entre les entêtes et les données nécessaires pour Area Selection. (3h30)
- Continuer le graphique bokeh : les chromosomes sont maintenant représentés sous forme de rectangles. (3h)
- Implémentation d'une requête allant sélectionner des rsid dans la table marqueurs, et affiche la correspondance sous forme d'un rectangle par-dessus les chromosomes (2h)
- Tentative d'utiliser/adapter les scripts de Simon Grondin afin de migrer la base de données actuelle. (2h)
- Documentation sur les différents mapping (1h)
- Création de nouveaux scripts pour migrer les données. (1h)
- Création d'une nouvelle solution pour le backend (3h)

Difficultés rencontrées :

- Nous trouvons Bokeh un peu difficile à manipuler
- Les scripts de Simon Grondin ne semblent pas migrer vers le format ADAM
- Difficile de comprendre le type des données actuelles ainsi que le mapping. (https://docs.google.com/spreadsheets/d/1uU1QM9q4uVrzSKtg7nelMKUik7RifDqrwAwR6mUTENo/edit#gid=0 et DBSNP)
- ADAM

Éléments bloquants :

- Fichier de mapping

Ordre du jour SCRUM :

- Tour de table sur les tâches accomplies dans la semaine précédente
- Tâches qui seront accomplies dans la prochaine semaine
- Éléments bloquants

Ce qui s'est dit pendant la rencontre :

- Backend:
    - Proposition d'utiliser Apache Spark pour Python
        - Proposition rejetée
    - L'équipe Backend devra migré les donnés vers le format VCF
        - Le format VCF peu être migré vers le format ADAM en utilisant la fonction vcf2adam d'ADAM

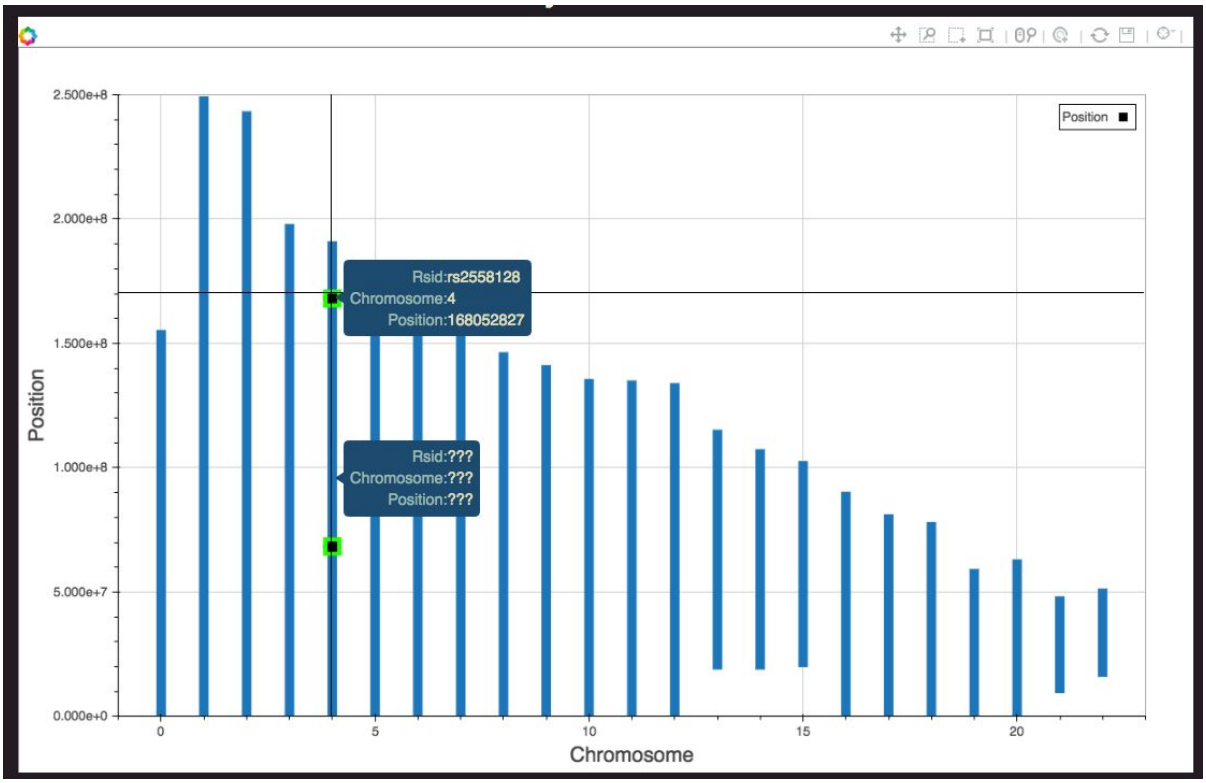
    - L’équipe Backend doit poser plus de questions afin de ne pas perdre l’objectif de vue

Plan d'action de la semaine:

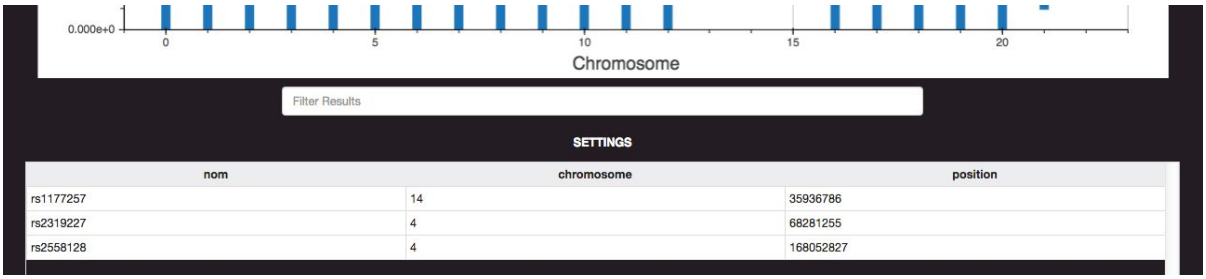
- Équipe Back-end: Mise en oeuvre d’une nouvelle solution afin de réduire le temps d’exécution des requêtes
    - Lecture sur Spark pour Python (25% fait)
    - Migration de la base de données en format .csv (100% fait)
    - Création de la nouvelle méthode en utilisant Apache Spark
    - Remplacer la méthode par la nouvelle utilisant Spark
    - Migration des données vers le format VCF
    - Conversion des données vers le format ADAM

16 mars 2017

Ce qui a été fait (heures travaillées : x heures totales):

- Lecture sur le format de donnée VCF et élaboration d'un plan d'attaque (2h)
- Installation de VCFtools afin de valider les données (1h)
- Test avec la méthode vcf2adam. (Avant d'utiliser la fonction vcf2adam nous devons nous assurer qu'elle fonctionne. Nous avons téléchargé des fichiers vcf de 1000genomes et avons fait plusieurs essais) (2h)
- Création d'un script pour migrer les donnés (3h)
- Migration de fichier VCF vers le format ADAM (3h)
- Migration des fichiers VCF vers le format ADAM (½ heures)
- Lecture des fichiers dans ADAM (½ heures)
- Ajout d'un HoverTool dans le graphique pour afficher les informations sur le Rsid pointé :
- 

- ...On peut voir les chromosomes ainsi que les Rsid sélectionnées. (1h)
- Affichage du tableau sous le graphique pour afficher les Rsids trouvés (1h)

- 

- Modification pour que le ToolTip ne soit affiché que sur le rsId (1h)
- Tentative de faire la requête SELECT qui fera un match avec les différents rsid, position et chromosome donnés dans le csv (2h)
- Permettre à l'utilisateur de faire la correspondance entre ses entêtes de son fichier csv et ceux utilisés par l'application. (2h)
- Passer la correspondance d'entêtes au contrôleur Django afin d'envoyer les informations sous format JSON à Area Selection pour générer le graphique. (2h)
- Création d'un objet JSON pour l'envoi à AreaSelection (1h)
- Tentative de rendre le graphique bokeh plus interactif : sélectionner des checkbox pour afficher/cacher des données (3h30)

Difficultés rencontrées :

- Les fichiers .vcf téléchargé étais espacé avec des espaces à la place de tabulation, cela a causé plusieurs problèmes puisque le message d'erreur affiché par ADAM ne laissait par voir cela.
- Le HoverTool apparait aussi pour le chromosome quand on pointe dessus. Il ne faudrait pas (corrigé).
- Faire la requête qui fait tous les matchs selon les valeurs fournies (liste de rsid+chromosome+position (résolu)

Éléments bloquants :

- Dans GenomeViewer StepII
    - À quels champs correspond Gênes dans la table marqueur?
    - À quel champ corresponds le link(dbSnp, genecard) dans la table marqueur?
    - Quel est le lien entre la table marqueurs et les Mutations dans la bd?
    - Quel est le lien entre la table marqueurs et les Phénotypes dans la bd?
- Incapable d'ajouter des checkboxes et de l'interactivité à Bokeh. De la manière qu'on l'utilise, je ne suis pas sûre que c'est possible.

Ordre du jour SCRUM :

- Avancements

Ce qui s’est dit pendant la rencontre :

- Discussion sur les problèmes avec Bokeh.
- Possibilités de modifier l’engin d’affichage des graphiques pour amCharts.

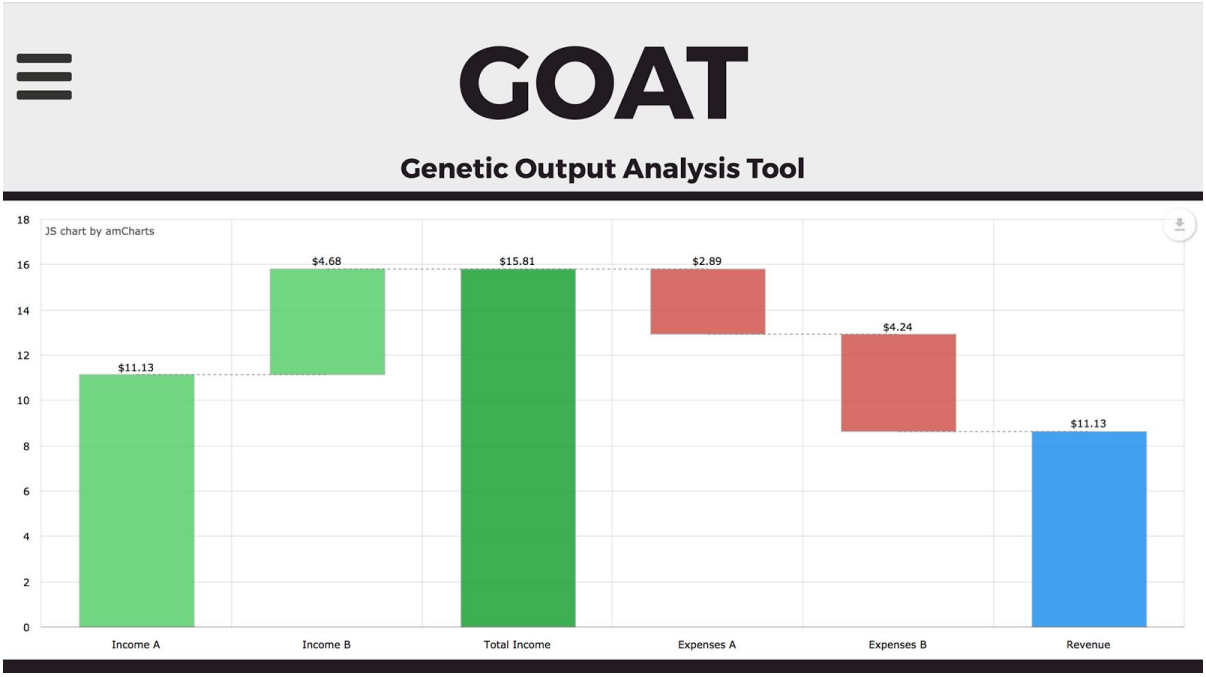
Plan d'action de la semaine:

- Intégrer amCharts pour la génération des graphiques.

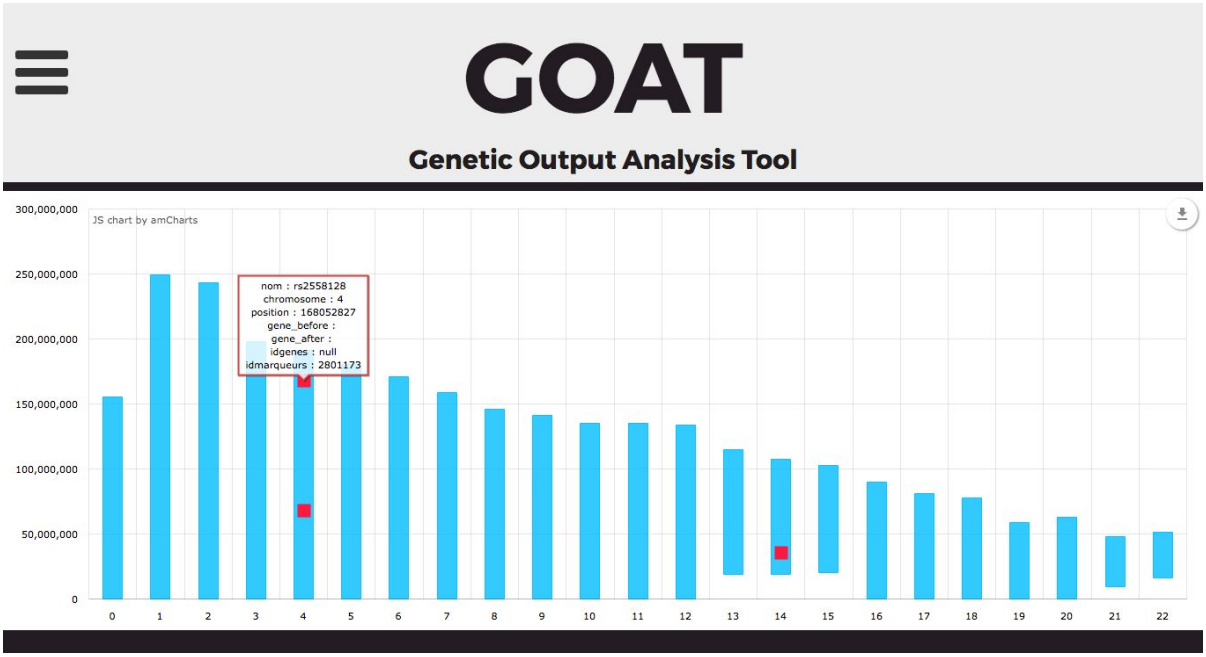
23 mars 2017

Ce qui a été fait (heures travaillées : x heures totales):

- Tentative d'installer Bokeh 0.12 (1h x 2)
- Installation de AmCharts dans le frontend et affichage d'un graphique de démonstration (3h30 x 2)



- Installation d'ADAM sur une instance EC2 (3h)
- Installation s3fs sur l'instance EC2 afin de relier le serveur S3 (2h)
- Téléchargement de HG19 et HG38 sur le serveur S3 (2h)
- Extraction du Chromose 22 (1h)
- Préparation pour la migration de HG19-CHR22 en format ADAM (1h) (6h de calcul sur le serveur) (Exemple: https://s3.amazonaws.com/goat-adam-genes/chr22.txt)
- Faire un affichage semblable à celui que nous avions sur Bokeh auparavant, mais avec AmCharts (5h30 x 2)
- Installation de Django et du nouveau backend sur l'instance EC2 (3h)

- 

Difficultés rencontrées :

- Nous n'avons pas réussi à faire fonctionner Bokeh 0.12 et avons préféré passer rapidement à AmCharts
- Difficulté dans l'installation de AmCharts à cause du Stack du projet. Nous avons eu du mal à mettre les "imports" aux bons endroits pour que ça fonctionne.
- Une fois avoir réussi a installé AmCharts, nous avons ajouté un graphique de démonstration. Cependant, l'affichage n'était pas fait correctement et le graphique grossissait infiniment. Nous avons fini par régler le problème.
- Ce que nous souhaitons faire avec AmCharts ne vient pas réellement "out-of-the-box", soit l'affichage de 2 types de données complètement différentes : des colonnes représentants des chromosomes et un nuage de point représentant des rsid.
- Encore une fois, c'est difficile de filtrer les données. La fonctionnalité native est de cacher/afficher des graphes entiers.
- Lenteur de la conversion des données VCF vers le format ADAM.

Éléments bloquants :

- Accès afin de créer une instance EMR (Courriel envoyé)

Ordre du jour SCRUM :

- Avancements

30 mars 2017

Ce qui a été fait (heures travaillées : x heures totales):

- Amélioration de la conversion de donnée (3h)
- Documentation sur les services Amazon et installation d'ADAM (2h)
- Mise en marche de Django sur l'instance EC2 avec ADAM (2h)
- Élaboration du rapport final (½ h)
- Création d'une nouvelle fonction ADAM afin de comparer le chromosome 22. (1h)
- Tentative de permettre de montrer/cacher des données dans le graphique selon certains attributs (4h)
- Tentative de faire une validation des rsid en validant la position et le chromosome (2h)
- Formatage et envoi des données du fichier csv téléversé pour l'affichage sur le graphique (3h)
- Affichage des données venant du fichier csv (1h)
- Amélioration de la fonctionnalité de téléversement du fichier csv (2h)

Difficultés rencontrées :

- Permettre de montrer/cacher des données selon des attributs (réponse de Amchart à essayer)

Éléments bloquants :

- Problème dans la conversion Json des résultats de plusieurs requêtes SQL Select (plusieurs Select sont nécessaires si on veut valider la position et chromosome)

Ordre du jour SCRUM :

- Ce qui reste à faire avant la fin du projet
- Rapport/présentation final

Ce qui s'est dit pendant la rencontre :

- Documenter où on est rendu et tout ce qu'on a fait

Plan d'action de la semaine:

- Travailler sur le rapport et la présentation final

6 avril 2017

Ce qui a été fait (heures travaillées : x heures total):

- Utilisations de Spark pour convertir les données (2h)
- Création d'une nouvelle architecture sur Amazon AWS (2h)
- Création de la méthode "matching" dans ADAM (4h)
- Documentation sur le travail accompli afin de permettre une transition simple et efficace (4h)
- Écriture du rapport (10h)
- Préparation pour la présentation orale (5h)