Université de Bordeaux

Mémoire du Projet de Programmation

VisualMapReduce

Auteurs :
Kinda Al Chahid
Marc-Alexandre Espiaut
Imen Hentati
Oualid Yjjou

Client :
Alexandre Perrot
Chargé de TD :
Maria Predari

2 avril 2017

Table des matières

Introduction

Avec l'augmentation [1] du volume de données à traiter, le BigData (Données Massives) est une problématique qui s'est progressivement introduite en entreprise.[JF14] MapReduce est un algorithme qui tente d'apporter une réponse satisfaisante au traitement d'un grand volume de donnée.

MapReduce exploite le concept de `"Diviser-Pour-Régner"` dont le fondement est de diviser un large problème en petits sous-problèmes qui sont indépendants les uns des autres dans le but de les traiter en parallèle[LD10]. C'est donc un pradigme de programmation servant à traiter et à générer de grands jeux de données avec un algorithme distribué et parallelisé.

L'exécution d'un programme MapReduce se fait sur un cluster [2] contenant généralement un grand nombre de machines, ce qui rend le débogage compliqué. En effet, si une des machines ne parvient pas à exécuter le programme correctement, le retour n'est pas fait à l'utilisateur et l'identification de l'origine du problème est difficile. Le but de VisualMapReduce est de simuler le fonctionnement d'un programme MapReduce sur un cluster de machines homogènes (c'est-à-dire qu'elles possèdent toutes les mêmes caractéristiques matérielles) afin de pouvoir le tester avant son application dans le monde réel pour éviter ce genre de problèmes.

MapReduce [NT16] décrit deux tâches principales : la tâche `map()` qui converti un jeu de donnée A en un jeu de donnée B, où les éléments individuels sont découpés en tuples (paire clé/valeur), et la tâche reduce() prends en entrée le résultat produit par map() pour combiner les tuples de données produites en un plus petit jeu de tuples. Comme le nom MapReduce l'indique, la tâche `reduce()` s'effectue toujours après la tâche `map()`.

Plan du mémoire

plan a mettre sous forme de paragraphe (sans section)

1. 2.5 trillions d'octets de données par jour.
2. Un *cluster* est une grappe de machines sur un réseau.

Chapitre 1

Présentation du projet

1.1 Hadoop et MapReduce

Hadoop[PG13] est une implémentation open source de MapReduce en Java distribuée par Apache. Les deux grandes parties de Hadoop sont :

— Traitement des données : le framework MapReduce.

— Stockage des données : Hadoop Distributed File System.

Le principe est de distribuer les données (avec HDFS [1]) et de faire des traitements sur ces données là où elles sont stockées (grâce à MapReduce) afin de paralléliser des opérations.

Hadoop exécute une tâche de en commençant par diviser les données en entrée en blocs de données de même taille. Ensuite, chaque bloc est planifié pour être exécuté.

1.2 Contexte du projet

Dans le cadre de notre projet de programmation, l'idée est de retranscrire la distribution des données en appliquant l'algorithme de mapReduce sans utiliser un framework [2] ou un système de gestion de fichiers.

Avec cela, le but est d'arriver à apporter une visualisation de la simulation sur un cluster de machines.

A noter : Dans le projet, il n'y a pas de cluster concret derrière. Notre travail est simplement de simuler le comportement de mapReduce sur des machines fictives qui n'existent pas sur un serveur ou autre support matériel.

La représentation graphique doit comprendre les phases de map et de reduce, toutes les deux fournies comme fonctions par l'utilisateur de l'application. Il faut pouvoir visualiser les connections qui existent entre map et reduce sur les différentes machines.

L'objectif du projet est de déceler des bugs [3] dans la simulation. Ces bugs peuvent correspondre à une mauvaise répartition des données dans l'ensemble du cluster qui sont invisibles pour l'utilisateur. En effet, le retour du process de mapReduce est un ensemble de paires clé-valeur, il n'est pas possible de connaître quel processus fait quelle tâche ou quel processus ne fait rien. Grâce à VisualMapReduce, l'utilisateur peut voir visuellement s'il a donné par exemple un nombre de machines/coeurs important par rapport à ses données traitées et vis-versa.

1. Hadoop Distributed File System.
2. dont le plus connu est Hadoop
3. en Français : bogue

Cette application, destinée à l'enseignement, permettra à des étudiants en Master 2 Informatique de simuler un cluster et comprendre le fonctionnement de MapReduce.

1.3 Description détaillée du process de mapReduce

1.3.1 Notions importantes

Avant de lister les différentes phases de mapReduce, il y a certaines notions importantes à savoir. Nous citons :

— **Slot** : Le cluster d'éxécution comprends `m` machines avec `c` coeurs, ce qui représente **m*c** slots d'éxécution. Un slot peut accueillir un mapper ou un reducer. Le slot est libéré lorsque l'exécution du mapper ou reducer se termine.
— **Job** : Un job est un ensemble de map et reduce. Il dispose en entrée d'un jeu de données.

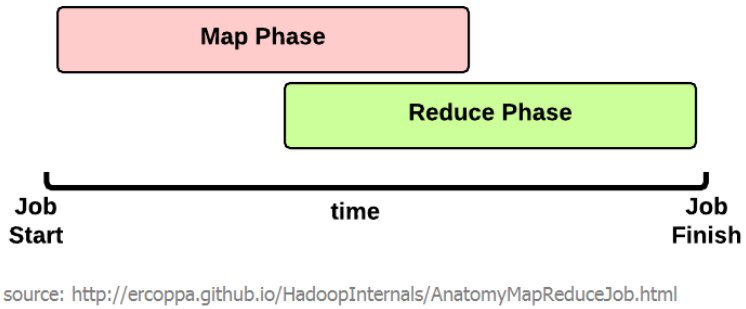


FIGURE 1.1 – Job dans mapReduce

— **Task** : On trouve des mapTasks et des reduceTasks. une phase de map comprend plusieurs mapTasks et pareil pour une phase de reduce.
— **Mapper/Reducer** : Ce sont les définitions des fonctions map et reduce. Un mapper est appliqué sur chaque slot. On a donc autant de mappers que de slots du cluster. [4]
Le mapper est appliqué sur chaque entrée du résultat de la fonction de découpage "split()" qui sera expliqué plus tard dans le document. Un reducer est appliqué sur toutes les valeurs associées aux mêmes clés générées.

Dans mapReduce, l'utilisateur définit un mapper et un reducer avec les signatures suivantes[LD10] :

$$map : (k_1, v_1) \rightarrow [(k_2, v_2)]$$
$$reduce : (k_2, [v_2]) \rightarrow [(k_3, v_3)]$$

1.3.2 Phases dans mapReduce

L'algorithme MapReduce se décompose en plusieurs phases : **Split, Map, Combine, Partition, Shuffle** et **Reduce**.

— **Split** : C'est une fonction qui récupère l'ensemble des données d'entrées et retourne des objets structurés qui serviront comme entrée de map. Dans la figure 1.2, l'étape de split répartie un fichier textuel selon son nombre de lignes.

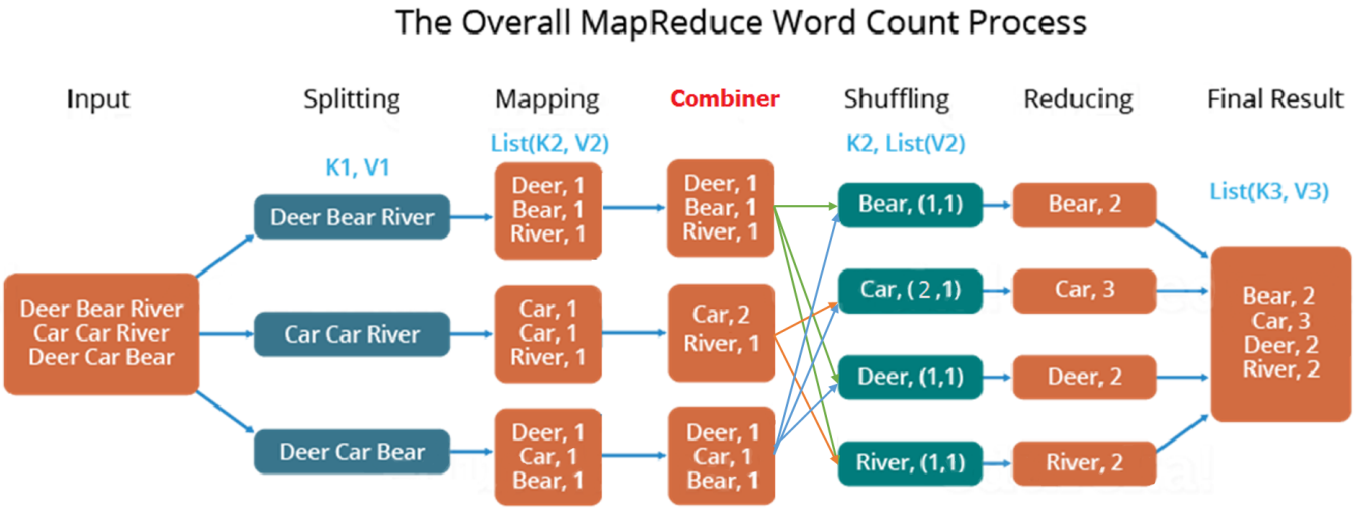
4. Il faut distinguer entre Mapper et MapTask. Un mapper peut accueillir plusieurs mapTasks. Une mapTask est une tache individuelle. Pour simplifier : un mapper = map+map+...+map

— **Map** : Prend en entrée une ou plusieurs données. Les résultats sont stockés sous forme de paires <key, value> (<clé, valeur>) dites intermédiaires.
— **Combine** : Optionnel. Il possède les mêmes propriétés d'entrée/sortie que map. Son but est de faciliter la tâche de reduce en efectuant un "merge" partiel[DG04] sur les résultats du map.
Dans la figure 1.2, la phase de combine n'existait pas dans le schéma, mais dans le but de montrer son utilité, nous avons modifié la figure et rajouté le combiner à l'exemple.
— **Partition** : contrôle le partitionnement des clés des résultats de map. Par défaut, le partitioner utilise une fonction de hashage.Le nombre total de partitions est le même que le nombre des tâches de reduce. Le partitionneur intervient entre les phases de map et shuffle. Pour l'exemple de Wordcount de la figure 1.2, il agirait pour faire le partitionnement des sorties de `map` sur les 4 reducers qui existent.
— **Shuffle** : C'est la phase de transfert des données des mappers aux reducers. Plus clairement, elle fournie les entrées de reduce.
Le shuffle est de la forme :

$$shuffle : [(k_2, v_2)] \rightarrow (k_2, [v_2])$$

— **Reduce** : C'est la phase de calcul du process. Elle accepte les clés intermédiaires et un ensemble de valeurs d'une même clé. Les valeurs sont alors traitées selon la fonction `reduce` fournie.

Pour mieux comprendre l'exécution de mapReduce, nous proposons l'exemple de WordCount qui compte le nombre d'occurences des mots dans un texte.



Source : https://www.edureka.co/blog/mapreduce-tutorial/

FIGURE 1.2 – Exemple MapReduce : WordCount

Chapitre 2

Analyse des besoins

2.1 Besoins fonctionnels

2.1.1 Besoins essentiels

2.1.1.1 Importation

- **Importer un fichier de données (sous format csv)**

L'utilisateur choisit parmi sa propre bibliothèque de fichiers un fichier sous format `csv` qui contient des lignes de données sur lesquelles il veut appliquer son code mapReduce. L'extension doit être en .csv mais en revanche nous ne limitons pas la taille du fichier (surtout qu'on est dans un contexte de Big Data). Plus de détails sur le fichier est retrouvé dans le chapitre 4 Implémentation.

- **Importer un fichier contenant les fonctions de mapReduce**

Donner la main à l'utilisateur pour charger son propre code mapReduce avec les 3 fonctions à implémenter "map", "reduce" et "getPartition". Ce sont ces fonctions qui serviront pour le traitement des données en `csv` qu'il a fournit. Le fichier doit donc être sous format `js` mais par contre il n'est pas nécessaire d'en fournir un.

2.1.1.2 Configuration

- **Configurer le cluster de map**

Offrir la possibilité à l'utilisateur de fournir le nombre de machines ainsi que le nombre de coeurs par machine. Par contre, il est limité par un maximum de 20 machines à 24 coeurs chacune pour des raisons expliquées plus tard dans ce document.

- **Configurer le nombre de Reducer du programme**

Il est possible de préciser que le code mapReduce sera traité avec un nombre de tâches de reduce (Reduce Task) bien défini. En effet, la simulation peut varier si on a une seule tâche de reduce, dans ce cas toutes les données seront transférées vers un seul `Slot` et le temps de traitement sera plus long. Par contre, utiliser plusieurs reducers permet une bonne parallélisation selon les cas. Pour ce nombre, il est obligatoire de ne pas dépasser le nombre de slots existants dans la phase de map.

2.1.1.3 Modification

- **Modifier les fonctions de mapReduce**

C'est un besoin étroitement lié à "Importer un fichier contenant les fonctions de mapReduce". Quand l'utilisateur joint son fichier .js, celui-ci est directement chargé dans une zone de texte totalement modifiable. Ainsi, il peut voir son propre code contenu dans le fichier mais aussi effectuer directement des modifications dessus sans avoir à l'ouvrir ailleurs dans un éditeur de texte. C'est une fonctionnalité très utile pour pouvoir tester son code sur plusieurs cas et voir à chaque fois la simulation obtenue.

2.1.1.4 Visualisation

- **Visualiser la simulation de l'exécution de mapReduce**

C'est le service principal offert par notre application. Suivant les entrées de l'utilisateur (données csv, code mapReduce en js et configuration du cluster), un graphique est généré en dessous de la zone de code. Il est affiché alors le process de mapReduce selon ses spécifications.

- **Consulter les résultats sur console**

En plus du graphique, l'utilisateur peut notamment voir les résultats des variables de sorties des différentes phases. Ce besoin est intéressant quand il veut consulter les résultats des phases de partitioner et de shuffle qui ne sont pas inclus dans la simulation graphique pour ne pas allourdir la lisibilité de ce dernier.

- **Consulter les données exécutées sur un slot**

Une fois la simulation affichée et le graphique généré, on obtient l'ensemble des machines avec leurs différents coeurs ainsi que les liaisons qui montrent la distribution des données. Pour consulter les données exécutées sur un slot en particulier, l'utilisateur clique sur ce slot. Une zone à droite de la page est alors remplie avec les différentes lignes exécutées sur ce map ou bien reduce (un slot peut contenir soit un map soit un reduce).

2.1.1.5 Exportation

- **Exporter le résultat de la simulation**

L'utilisateur peut récupérer les données de sortie du process de mapReduce sous forme d'un fichier de données `csv`. Ce fichier contient toutes les lignes de résultats ayant chacune la structure suivante : "clé ; valeur".

2.1.2 Besoin optionnel

- **Exporter les fonctions générées en langage Java**

Ceci permet d'avoir le code Java du process MapReduce équivalent à celui en Javascript utilisé pour traiter les données. Ce besoin est facultatif en vue du manque de temps que nous puissions rencontrer.

2.1.3 Priorité des besoins fonctionnels

En plus du découpage besoins essentiels/besoins optionnels, nous pouvons indiquer la priorité des besoins. (Voir tableau ci-dessous)

Besoins	Priorité attribuée
Importer un fichier des fonctions	Priorité élevée
Configurer les fonctions	Priorité élevée
Visualiser la simulation	Priorité élevée
Importer un fichier de données	Priorité élevée
Configurer le cluster de map	Priorité moyenne
Consolter données d'un slot	Priorité moyenne
Consulter résultats sur console	Priorité faible
Configurer le cluster de reduce	Priorité faible
Exporter le résultat de la simulation	Priorité faible
Exporter les fonctions en java	Priorité faible

2.2 Besoins non fonctionnels

Lisibilité du résultat
L'utilisateur doit pouvoir lire sans difficulté le résultat du *MapReduce*. Il faut pouvoir zoomer et naviguer dans le graphique réalisé par *FATuM* pour comprendre les liens entre le *mapper* et le *reducer*. L'utilisateur doit pouvoir cliquer sur les slot pour obtenir une liste des données contenues, et ainsi comprendre leurs répartitions.

Pour des raisons de visibilité à l'écran et pour simplifier la lecture du graphique, c'est la distribution des données entre map et reduce (avec pour chacun ses entrées et ses sorties) qui est générée. La distribution suit les configurations apportées.

Affichage des données dans la console
Le résultat des fonctions `map()` et `reduce()` doit être affiché dans la console Javascript pour permettre à l'utlisateur d'obtenir le résultat sans avoir à passer par l'interface graphique générée par *FATuM*.

2.3 Prototypes d'interface

Nous montrons à travers des illustrations suivantes les prototypes d'interface[1] de notre application telle que nous l'avons imaginé au début du projet. Cette étape a été importante durant la réalisation du projet parce qu'elle a permis d'avoir une idée globale sur les différents composants graphiques (boutons, zones de texte,etc...) ainsi que les interactions possibles entre l'utilisateur et le programme.

1. ou WireFrames en anglais.

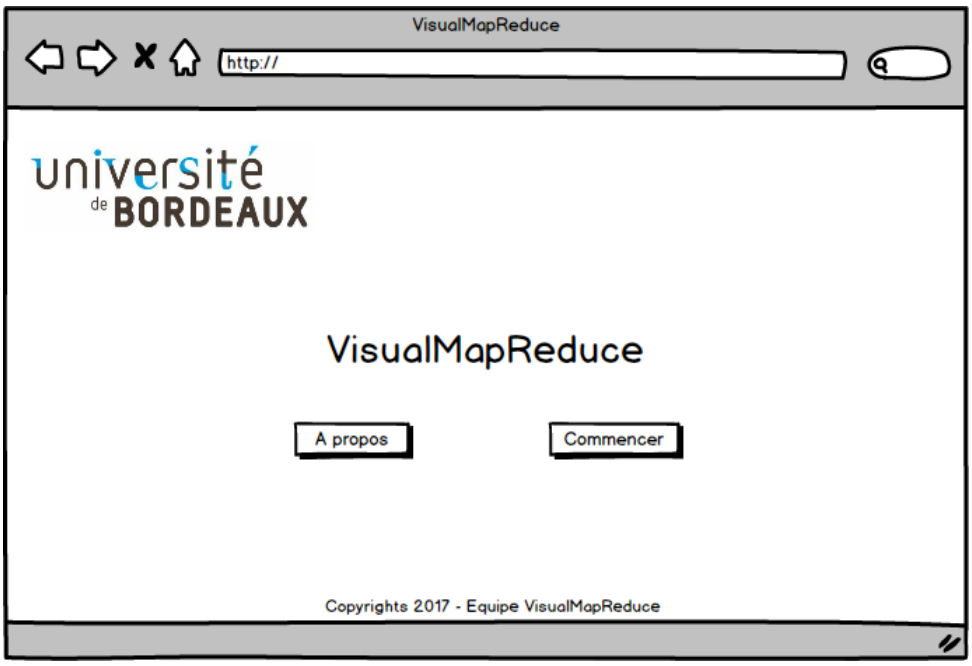


FIGURE 2.1 – Page d'accueil

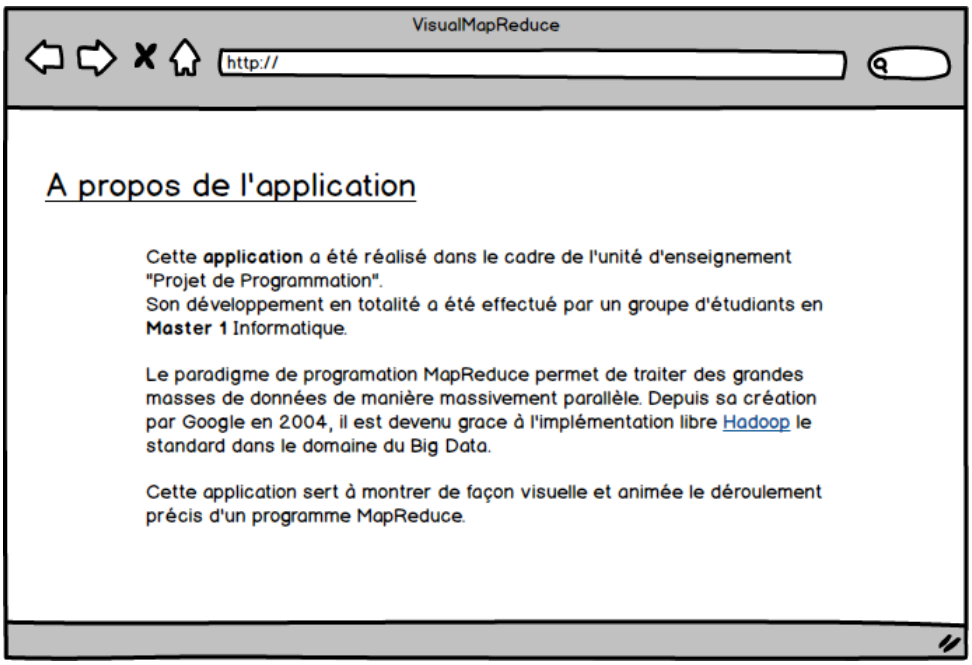


FIGURE 2.2 – A propos

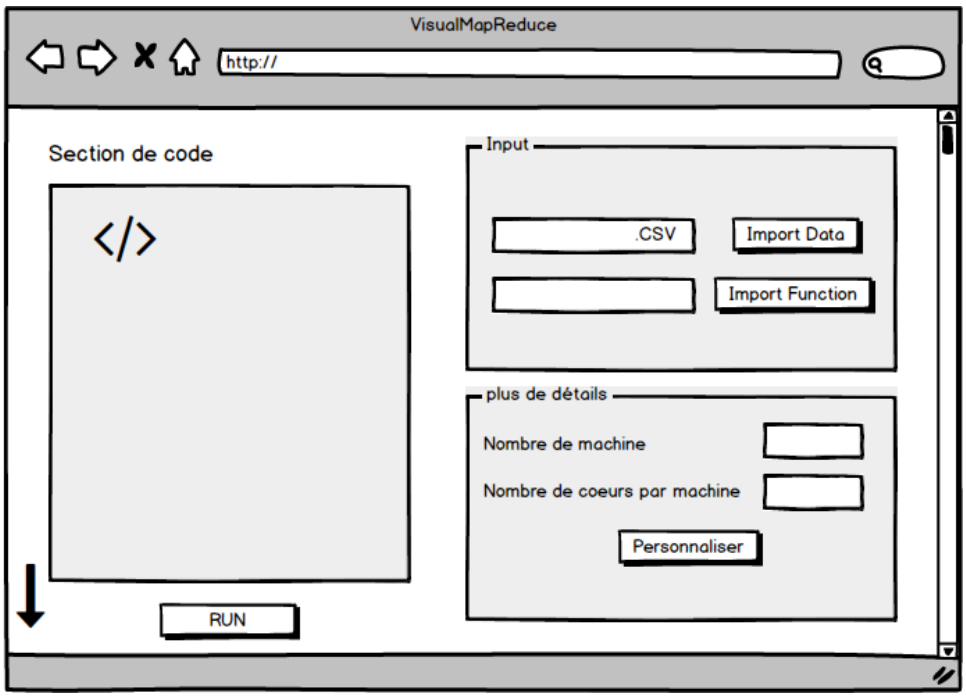


FIGURE 2.3 – Page d'interprétation

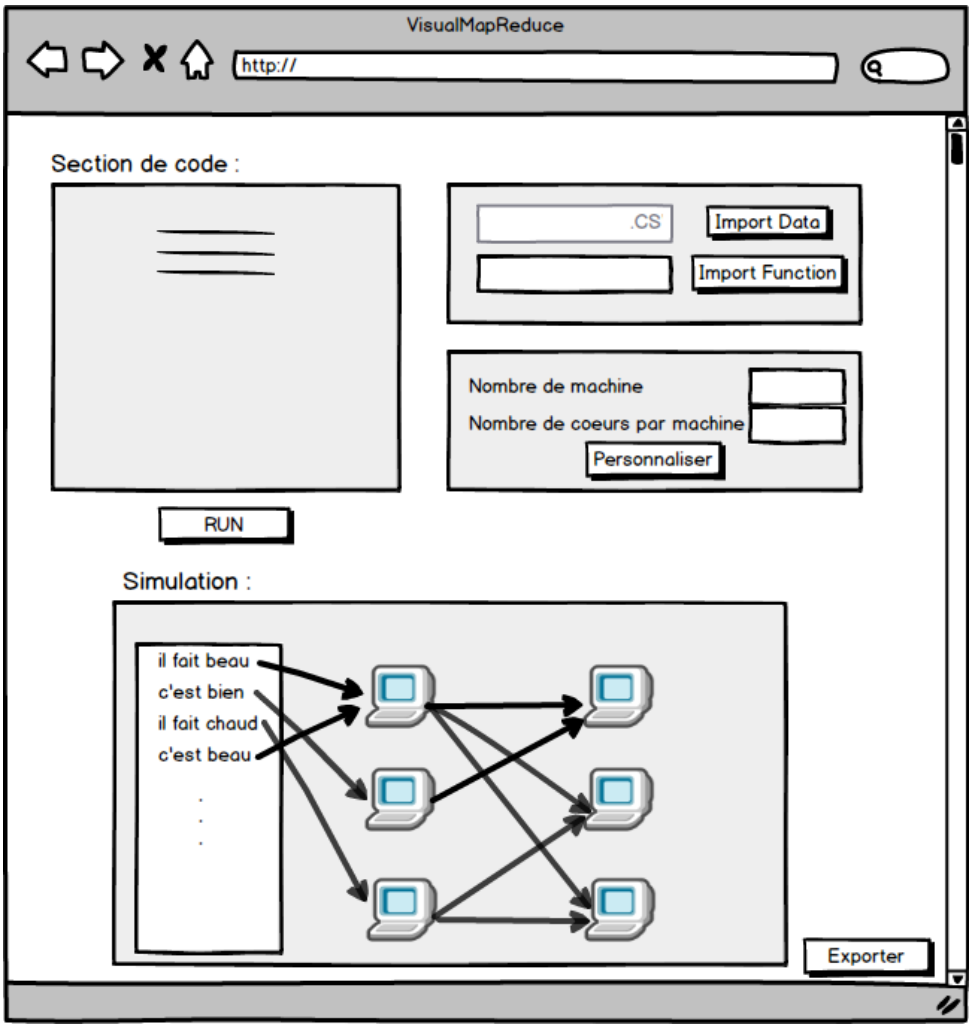


FIGURE 2.4 – Affichage du résultat

L'application étant monopage, le résultat de la simulation est directement affichée sur la même page où l'utilisateur remplit les données. Le graphique apparaît en dessous de la première section.

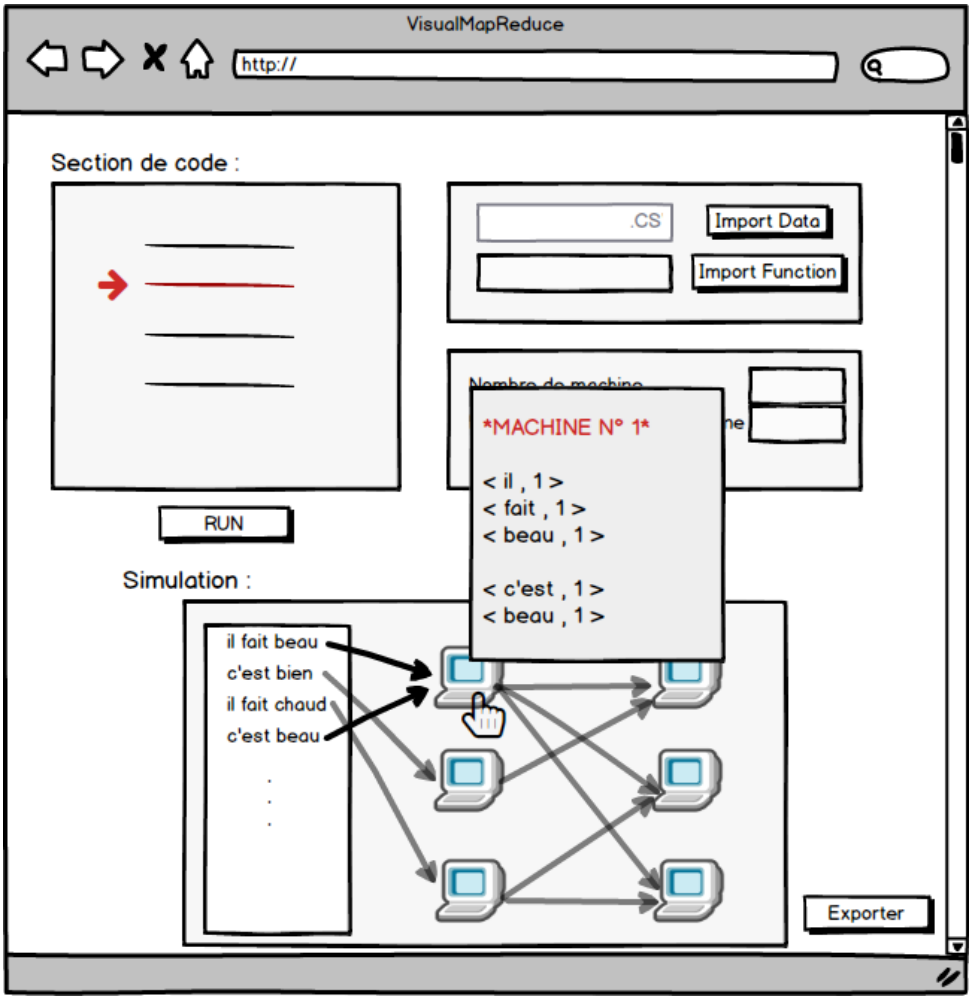


FIGURE 2.5 – Détail d'une machine

Comme illustré dans la figure ci-dessus, l'utilisateur a la possibilité de consulter les détails d'exécution qui concernent une machine particulière. Lorsque l'utilisateur appuie sur l'icône de l'une des machines, l'ensemble des propriétés qui concernent celle-ci s'affichent à l'écran. Au début du projet, ce besoin n'était pas très clair et nous avons cru comprendre qu'il peut aussi voir quelle partie de la section de code est exécutée sur cette machine. Or, nous nous sommes rendu compte plus tard que ce qu'il peut consulter c'est les données traitées sur chaque slot exécutant une des phases map ou reduce.

Chapitre 3

Architecture

3.1 Présentation de l'architecture du projet

3.1.1 Diagramme de flux des données

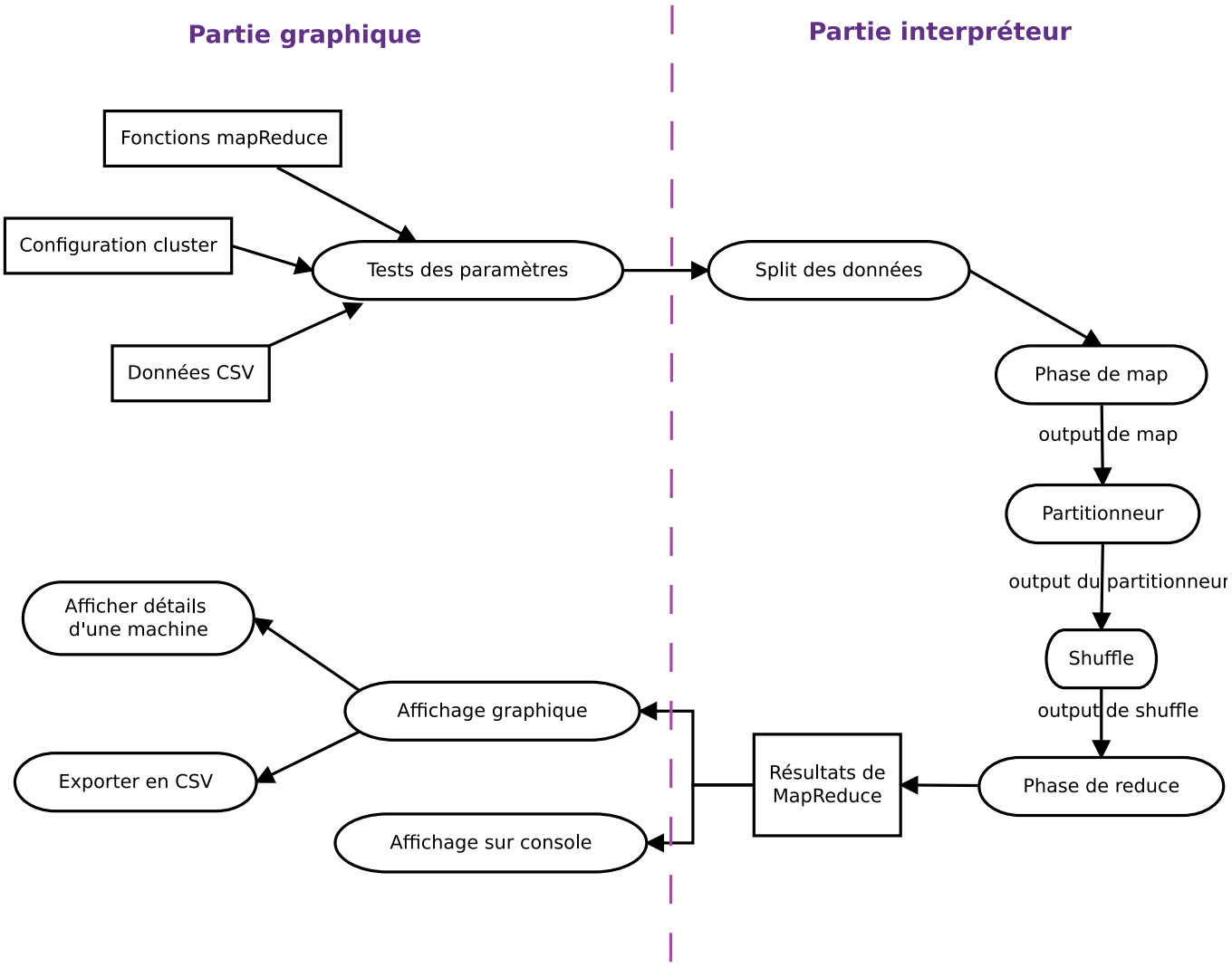


FIGURE 3.1 – Diagramme de flux des données

Ce diagramme de flux des données résume le passage des données en entrée dans une "boite noire" pour ensuite récupérer le résultat final en sortie. Lorsque l'application est lancée, les données sont traitées dans la partie graphique (zone de gauche) puis sont envoyées à la partie interpréteur (zone de droite).

3.1.2 Architecture client-lourd | 2-tiers

Notre projet consiste en un site web qui ne possède pas de base de données. Le serveur ne sert qu'a accéder à son URL [1] c'est-à-dire qu'il n'est utilisé que pour héberger le contenu du site. Le traitement dans sa totalité se fait du coté du navigateur directement et aucune donnée n'est envoyée/conservée ou reçue d'un serveur. Ce type d'application correspond donc à l'architecture **Client-lourd**.

D'autre part, on remarque que ce projet peut être divisé en deux parties indépendantes, l'une qui va traiter le process de mapReduce, et l'autre qui s'occupe du coté visuel/graphique. On parle ici d'une séparation entre le `traitement` et `l'affichage graphique`.
Nous avons donc eu l'idée de suivre l'architecture **2-couches** pour représenter cette séparation.

Ainsi, notre projet reflette une architecture **client-lourd** | **2-tiers** combinées pour améliorer l'organisation du travail.

3.1.3 Diagrammes de séquence

3.1.3.1 Chargement de la page

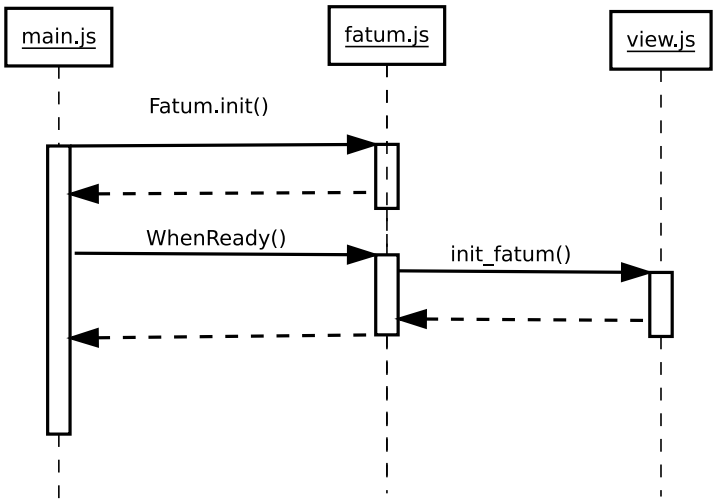


FIGURE 3.2 – Chargement de la page

Quand l'URL est saisi, la page se met à charger sans aucune intervention de l'utilisateur. La figure 3.2 représente les différents appels de fonctions lors du chargement de la page. En effet, le chargement se fait par l'initialisation de FATuM appelé dans le fichier `view.js` qui appelle la bibliothèque `fatum.js`.

1. Uniform Resource Locator

3.1.3.2 Lancement de la simulation

Lorsque le bouton `Run` est cliqué, une série d'actions se produit. D'abord, le cluster est affiché (sans les connections). Ensuite, le traitement de mapReduce est effectué. Enfin, les connections entre les slots sont affichées.
Ces actions sont représentées par trois parties dans les figures 3.3, 3.4 et 3.5.

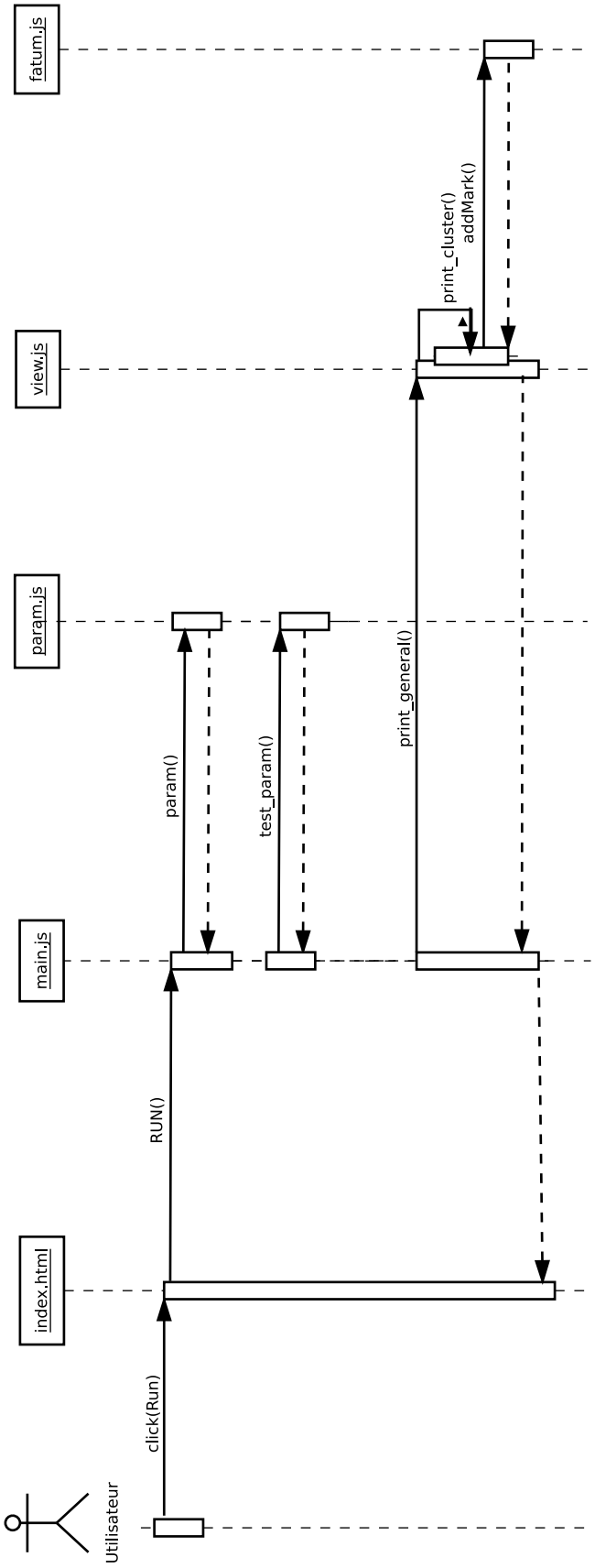


FIGURE 3.3 – Lancement du bouton RUN

Lors de l'affichage du cluster, les données fournies par l'utilisateur doivent respecter certaines conditions imposées par la fonction `Test_param` (par exemple entrer un nombre supérieur à 1 pour le nombre de PC). Une fois les conditions d'utilisation passées, on appelle la bibliothèque

FATuM pour afficher les différents slots du cluster de Map et de Reduce ainsi que les numéros des PC (affichage sans les connections entre les slots).

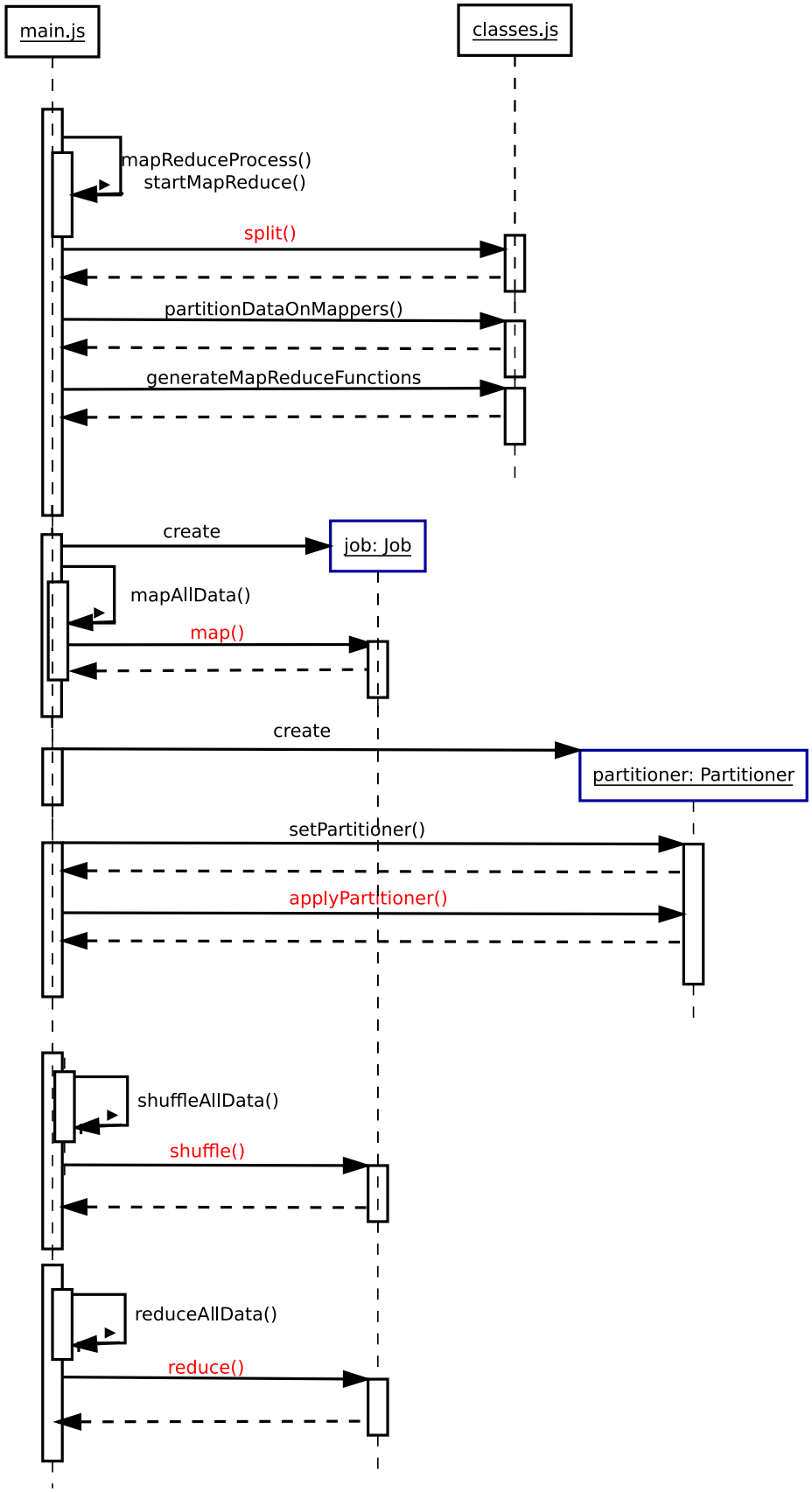


FIGURE 3.4 – Process de MapReduce

Une fois les éléments du cluster de la simulation affichés, le traitement des données s'effectue. Après chaque traitement, on effectue un lien entre chaque slot du cluster pour indiquer le transfert des données au sein de ce dernier représenté par des flèches provenant de FATuM.

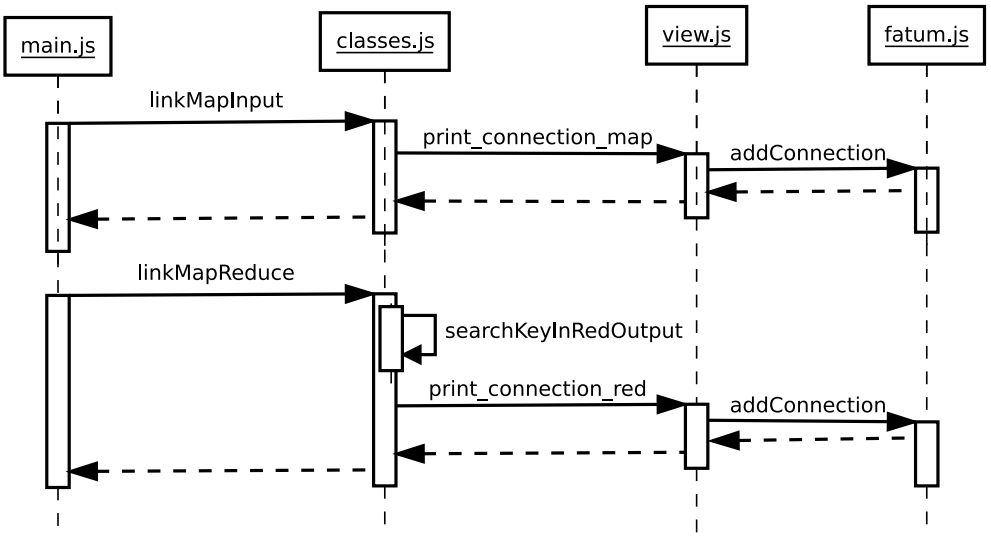


FIGURE 3.5 – Affichage des connections

3.1.3.3 Récupérer les données traitées

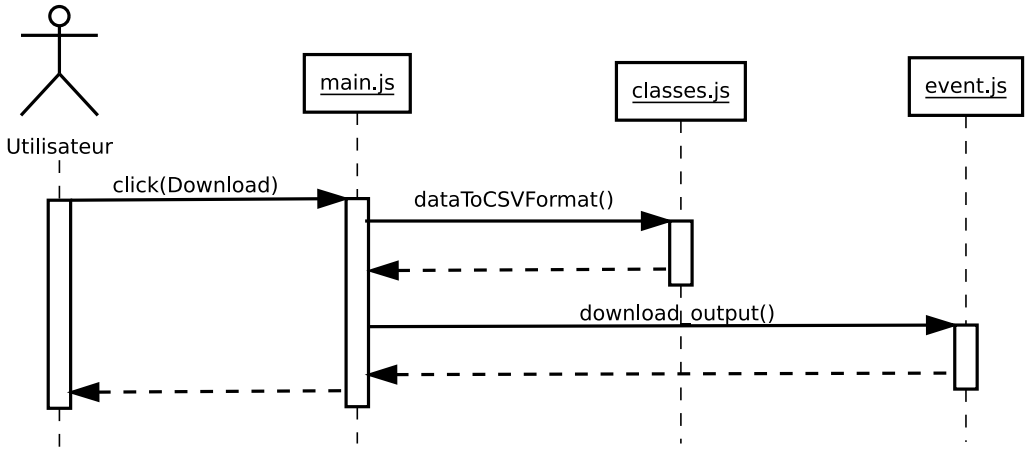


FIGURE 3.6 – Lancement du bouton Download

Une fois la simulation et le traitement des données effectué. L'utilisateur peut récupérer ces données transformer par ses fonctions mapReduce en les téléchargeant dans un format CSV.

Chapitre 4

Implémentation

4.1 Partie graphique

4.1.1 Coté interface utilisateur

4.1.1.1 MaterializeCSS

Pour l'interface utilisateur, nous avons dû utiliser la bibliothèque graphique `MaterializeCSS` qui fournit un dossier compressé comprenant tous les fichiers de base d'un site web. Ceci permet d'avoir une architecture de fichiers déjà existante et fournir les fichiers CSS nécessaires pour avoir un visuel optimisé et un design attirant.

Ainsi, il suffit d'appeler un élément graphique de HTML, de lui donner la classe correspondante implémentée dans MaterializeCSS et le visuel est prêt.

4.1.1.2 Structure du site

Le site est en `monopage`, c'est à dire qu'il n'y a pas de présence de plusieurs fichiers HTML à charger ou d'un serveur avec une base de données pour les différents élements.

Pour passer d'un contenu à un autre, nous utilisons l'élément HTML de "navigation". Ainsi, cela donne l'effet de trois pages : "Acceuil", "Commencer" et "A Propos" alors qu'il ne s'agit que d'un seul fichier HTML.

Dans la partie "Commencer", nous pouvons séparer cette partie en trois sous-catégories. La première permet à l'utilisateur de saisir ces paramètres (les fichiers de données, ses fonctions map/reduce et les paramètres du cluster). La seconde permet soit d'écrire les fonctions map/reduce directement sur le site, ou de visualiser le code fournis pour le corriger directement. Enfin la dernière section concerne la simulation avec FATuM pour le cluster et une colonne à droite de la page pour l'affichage des données contenues dans un slot.

4.1.1.3 Les Loaders

FIGURE 4.1 – Loader

Le temps de chargement du site au démarrage ainsi que lors du lancement de la simulation sont long. Cette lenteur peut être interpréter comme un mauvais chargement des données par l'utilisateur. C'est pourquoi notre objectif était de rajouter un "loader" (voir Figure 4.1) au démarrage de la page pour signifier le chargement de FATuM et un autre pour le temps de calcul de la simulation. Le loader du démarrage fonctionne, malheureusement, celui de la simulation n'a pas pu être implémenter. En effet, le site se bloque le temps de calcul ce qui empêche l'utilisation du loader.

4.1.1.4 Les paramètres

Les `paramètres` données par l'utilisateur doivent remplir certaines conditions.

— Le cluster du Map doit au grand maximum contenir entre 1 et 20 pc et entre 1 et 24 coeur .
Cette limitation est visuelle car au delà, la lisibilité n'est plus assuré. Il s'agit là, d'un choix suite aux conseils du client.
— Le cluster de Reduce doit être comprise entre 1 et le nombre de slots total du cluster de Map.
— Le fichier js des fonctions map/reduce n'étant pas nécessaire pour l'utilisation car nous avons fournit un exemple de code dans "section code" qui est utilisé pour la suite de l'application.
— Le fichier csv doit être impérativement fournit et contenir des données. Si l'utilisateur rafraîchit la page, cette variable disparaît, il faut donc la charger de nouveau.

4.1.2 Simulation graphique(Fatum)

Comme demandé par le client, nous avons utilisé la bibliothèque graphique FATuM développée au LABRI. Cette bibliothèque permet d'afficher la simulation du cluster (voir Figure 4.2) avec différents composants graphiques et ne peut être utiliser pour l'interface utilisateur contrairement à MaterializeCSS.

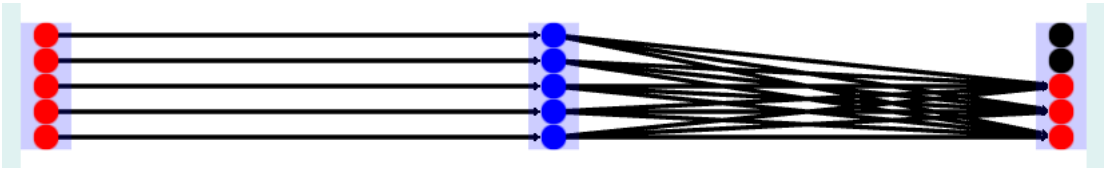


FIGURE 4.2 – FATuM - Simulation

Nous utilisons plusieurs composants de FATuM :

— Les Marks
— Les Connections
— Le zoom
— Une partie de la gestion d'un "clic souris"

FIGURE 4.3 – FATuM : Marks

Un `Mark` (comme dans la figure ci-dessus) est un élément sous forme de cercle qui représente un slot du cluster. Ils sont séparés entre eux lorsque la limite de slots par PC est atteinte. Ainsi, chaque PC sont séparés graphiquement. Ces éléments graphiques sont dépendants des données fournies par l'utilisateur.

Les `connections` sont les flèches qui vont d'un Mark vers un autre. Ils représentent le transfert de données entre les slots. On trouve des connections entre l'input de map et map ainsi que des connections entre map et reduce.

Le `zoom` permet (si l'on pose le pointeur de la souris dans la zone de simulation gérée par FATuM) la gestion de la molette de la souris. Ainsi, en cas d'un gros cluster, l'ensemble reste lisible grâce à ce zoom.

Enfin, la gestion du `"clic souris"`. Lors d'un clic dans la zone de simulation FATuM, les coordonnées récupérées sont celles de la fenêtre. Elles n'ont donc rien à voir avec celles de FATuM. La fonction `windowToModel` nous a permis de transformer les coordonnées du clic en coordonnées compréhensibles par FATuM pour pouvoir exécuter le traitement suivant la zone de clic.

4.1.2.1 Précision sur la fonction "search_mark"

```
function search_mark(x, id) {
    var header_data, tmp_header;
    //type of the mark
    switch (x) {
        case indice_fatum_1:
            tmp_header = "Map Input--";
            break;

        case indice_fatum_2:
            tmp_header = "Map Output--";
            break;

        case indice_fatum_3:
            tmp_header = "Reduce Output--";
            break;
        default:
            return false;
    }
    var min = nb_slot + gap;
    var max = min + nb_slot;
    //researh the true id without the gap
    for (var i = 0; i < nb_pc; i++) {
        if (0 <= id && id < nb_slot) {
            header_data = "Slot " + id + ": --"; //exple: Slot 1: --Map Task
                --
            print_data(x, id, header_data + tmp_header);
            break;
        } else
        if (min <= id) {
            if (id < max) {
                id = id - (gap * (i + 1));
                header_data = "Slot " + id + ": --";
                print_data(x, id, header_data + tmp_header);
                break;
            }
```

```
            }
            min = max + gap;
            max = min + nb_slot;
        }
    }
```

Cette fonction est particulière nécessite des précisions.

En effet, pour différencier les blocs de pc, nous avons décider de mettre un décalage (variable gap) entre ces blocs. Ce décalage créer des erreurs d'id de mark par la suite. En effet, lorsque l'on clique sur un slot, sont id correspond à sa position dans l'axe y. Mais a cause de ce décalage, l'espacement était également considérer comme un bloc et les id était décaler.

D'autre part, cette fonction permet également de détecter quelle type de donnée on souhaite afficher. Pour cela, on utilise l'axe des x pour se repérer.

4.1.2.2 La sortie console

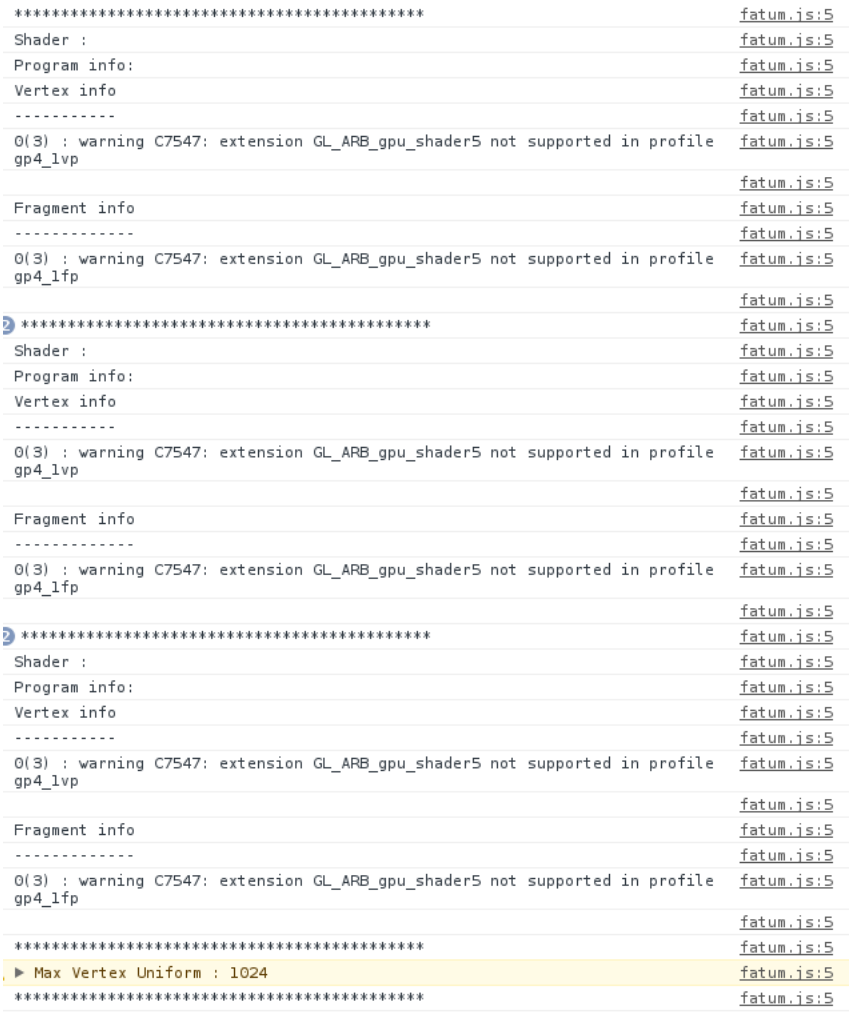


FIGURE 4.4 – Sortie console au démarrage

Lors du démarrage de l'application, l'initialisation de FATuM se fait, il est donc normal de voir en console (voir Figure 4.4), des informations concernant la bibliothèque.

4.1.3 Difficulté lié à l'affichage

Nos difficultés se sont principalement portées du fait que nous étions des débutants en web. Ainsi lorsqu'il a fallu utiliser la bibliothèque FATuM, les documentations n'étaient pas évidentes à comprendre au début. Surtout du fait que nous avions des exemples d'implémentation de FATuM dans une ancienne version de la documentation qui nous était fort utile pour comprendre les fonctions mais qui n'existaient plus pour certaines dans la nouvelle implémentation de FATuM.

Dans la nouvelle version de la documentation, il n'y a que les énoncés des fonctions, et c'est avec le temps que nous avons pris l'habitude de voir les exemples d'implémentation de FATuM dans l'ancienne documentation puis nous référer à la nouvelle documentation pour voir ce qui a changé.

4.1.4 Amélioration possible

Plusieur éléments peuvent être rajouté à la partie graphique pour améliorer l'utilisation de l'utilisateur et que nous n'avons pas pu implémenter

— le scrolling horizontal des données
— un loader pour la simulation
— Une optimisation visuelle du cluster

Concernant la partie visualisation, nous avons pensé que, plutôt que de rendre le cluster linéaire, c'est à dire : entrée Map puis sortie Map puis sortie Reduce chacun en ligne, le rendre cyclique.

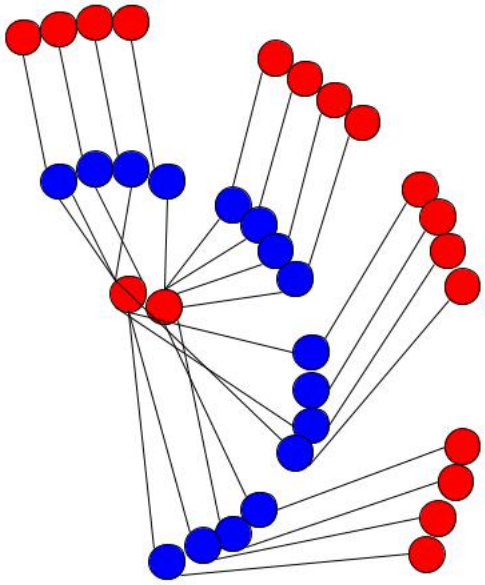


FIGURE 4.5 – Exemple d'affichage

Ainsi, un très gros cluster serait plus ludique et plus compréhensible mais pourrait créer deux problèmes : il n'y a plus la linéarité du map reduce comme indiqué dans la littérature et les connections de fatum sont moins lisibles car elles vont traversé les marks intérieur.

Enfin, il y existe un conflit avec webGL. Nous ignorons à quoi ce bug est dû mais, il arrive que le navigateur ait des soucis avec cette bibliothèque. Ce problème peut être lié au cache. Mais nous n'avons pas de solution à proposer

4.2 Partie interpréteur

4.2.1 Les classes

En Javascript, l'implémentation des classes peut se faire de plusieurs manières. Il existe la façon avec le mot clé '`class`'. Cette manière, bien que plus claire et plus facile à comprendre, n'est pas supportée par tous les navigateurs. Et comme la portabilité est l'un de nos besoins de qualité, nous avons préféré la manière "classique" de créer des classes en Javascript.

La déclaration est par contre un peu différente des autres langages Orientés Objet comme Java ou C++. Nous donnons l'exemple suivant de l'implémentation de la classe `Job` :

```
function Job(map, reduce) {
    this.map = map;
    this.reduce = reduce;
}

Job.prototype.applyMap = function(data) {
   ... //Appel a this.map
}

Job.prototype.applyReduce = function(data) {
    ... //Appel a this.reduce
}
```

4.2.2 Retranscription du process mapReduce

Pour les besoins de notre projet, nous n'avons pas basé notre implémentation du code sur les classes mais plutot sur les fonctions. Pour cela, nous avons centré le code de l'interpréteur dans un seul fichier .js qui contient, en plus des classes du projet, les fonctions qui interprétent le code mapReduce pour fournir les sorties (outputs).

4.2.2.1 Fonction split

4.2.2.2 Fonction eval

4.2.2.3 Fonction partitionDataOnMappers

4.2.3 Les expressions régulières

L'un des problèmes qui peuvent s'imposer est que le retour à la ligne n'est pas le même sous les différents systèmes d'exploitation (Linux, Windows et macOS). En effet, sous Windows le retour à la ligne est décelé avec "`\n`" alors que sous MacOS c'est fait avec "". Nous avons donc utilisé une expression régulière dans la fonction `split()` pour séparer les lignes du fichier `CSV` quelque soit le système d'exploitation sous lequel il a été écrit.

Dans la même fonction split, on utilise une autre expression régulière

4.2.4 Difficultés rencontrées

- Apprentissage du process de mapReduce (a pris du temps etc)
- Retranscrire la distribution normalement faite par Hadoop.

- Récupérer les fonctions en JS écrites par l'utilisateur et les appliquer sur les données.

4.2.5 Limitations

- Pas de free pour les tasks terminées.
- Malheureusement, pour des contraintes de temps, nous n'avons pas réussi à implémenter la phase de combiner ni l'exportation du code mapReduce en Java.
- Gestion d'erreurs :

Amélioration possible : rajouter un analyseur syntaxique pour le code javascript entré et faire un retour à l'utilisateur sur les erreurs potentiellement commises sur son code.

4.3 Limitations

4.4 Indentation du code

Chapitre 5

Fonctionnement et Tests

5.1 Tests de validation

Les tests de validation permettent de tester les besoins énumérés par le client. Nous pouvons vérifier à l'aide d'un outils qui s'intitule "**Selenium**" que les besoins seront respectés. C'est à dire , en effectuant des actions sur les sites nous obtenons bien ce qui est demandé de la part du client ,nous pouvons en énuméré quelque-uns :

- le test `"d'upload de fichier"` qui permet de charger en mémoire les fichiers à analyser.
- le test `"d'éxecution du projet"` qui permet aprés avoir renseigné tous les champs tels que le : le nombre de PCs , le nombre de cœurs et le code dans la partie section code.
- le test `"affichage des données"` qui permet d'avoir les informations sur un nœud du coté map et reduce.

Nous donnons plus tard dans ce même chapitre plus de précision sur Selenium.

Afin de garantir le bon déroulement du développement du programme et d'identifier rapidement les régressions, nous avons conçu plusieurs tests de validation au début du projet. En voici une liste exhaustive.

Entrée	Attendu	Erreurs éventuelles	Importance
Code MapReduce	Syntaxe correcte. Respecte le format imposé.	Erreurs de syntaxe	Critique. Empêche le programme de se lancer.
Données	Fichier .csv	Fichier manquant. Données aléatoires.	Critique. Empêche le programme de se lancer.
Paramètres	Nombre supérieur à zéro.	Nombre inférieur à zéro. N'est pas un nombre. Manquant.	Critique. Empêche le programme de se lancer.

- **Code MapReduce** (*optionnel*) : Code en JavaScript qui comprend les fonctions du Mappeur/Reduceur. Erreurs de syntaxes dûes à l'utilisateur dans son code. Nous ne savons au moment de l'écriture comment prévenir ce problème.
- **Données** : Jeux de données sous format `.csv`.
- **Paramètres** : Nombre de machines et nombres de cœurs par machine.

Selenium IDE

C'est un module de firefox qui permet de tester l'interface du projet en réalisant un ensemble de scénarios. Nous citons comme par exemple le clic sur un bouton ou le remplissage d'une zone de texte. Ce module nous permet d'automatiser ces scénarios à l'aide d'enregistrement sous forme d'une vidéo.

Test du résultat de l'interpréteur

Afin de vérifier l'adéquation du produit aux exigences fonctionnelles du client, on teste que le produit réalisé est le bon d'une manière textuelle explicative. Dans ce qui suit le test de la fonction de visualisation de la simulation.

Désignation	Condition requise	Démarche à suivre	Résultat attendu
Visualiser la simulation.	Les données initialement sous format CSV sont transformées et affichées dans une section à gauche. Chaque ligne est traitée par une machine. La simulation est affichée et est valide.	Sélectionner une machine du cluster à visualiser en cliquant sur son icône correspondante.	Les données traitées sur la machine sélectionnée apparaissent à l'écran.

5.2 Tests unitaires

Le bon fonctionnement de l'ensemble des fonctions est vérifié par une liste de tests unitaires dont le principe est de vérifier la sortie d'une fonction après un appel préalablement conditionné.

Les tests sont réalisés sans framework et son appellés par un fichier `.html` séparé.

5.3 SonarQube

Dans le but d'améliorer au maximum notre code source, nous avons cherché à utiliser un outil pour indiquer les éventuelles erreurs plus ou moins graves dans le code et ainsi connaître les failles. C'est pourquoi, pour rendre des sources de code propres, nous avons demandé conseil auprès d'étudiants de Master 2 qui nous ont indiqué l'utilisation de SonarQube.

SonarQube est un logiciel qui permet d'utiliser des métriques qui sont des algorithmes qui permettent par exemple de repérer les bugs ainsi que de voir la complexité des fonctions voire même des fichiers. Il regorge de métriques qui sont assez complexes à comprendre dans son ensemble. Nous avons pu constater qu'en l'utilisant dans notre projet nous avions quelques bugs majeurs et d'autres mineurs. Ces bugs étaient assez simple à résoudre dans l'ensemble.

En testant une première fois notre code avec le logiciel, nous obtenant les résultats affichés dans la figure suivante.

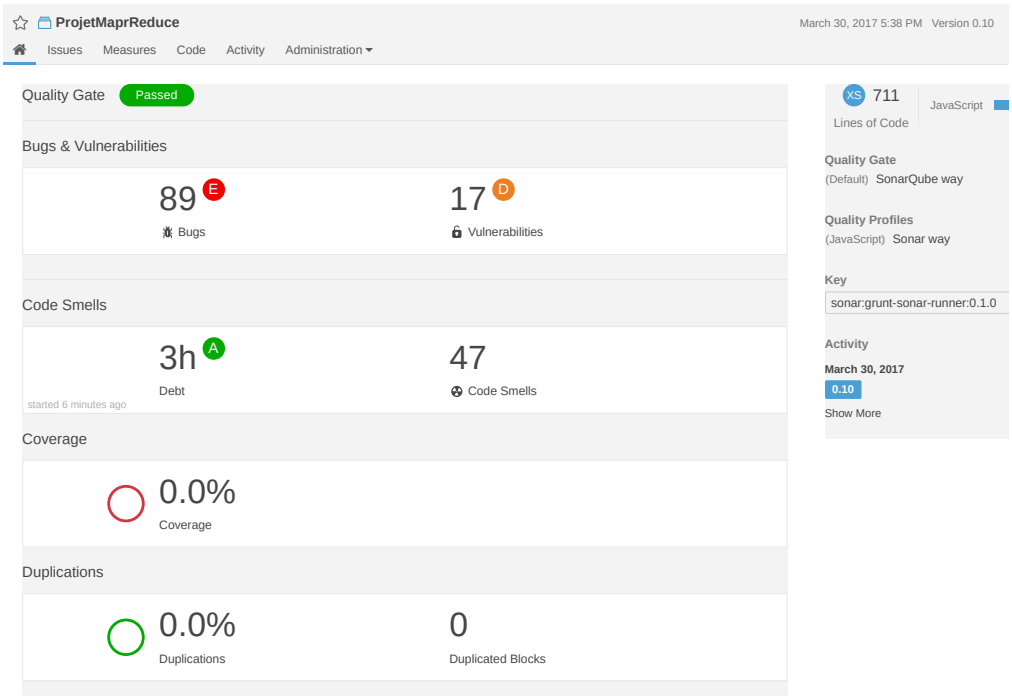


FIGURE 5.1 – Résultats de SonarQube

Nous remarquons que 83 bugs est un nombre assez importants et cela peut être amélioré. Aussi, il y a 17 vulnérabilités dûes principalement aux **console.log** rencontrés dans les fonctions de tests unitaires et dans le résultat du mapReduce process. Nous laissons quand même ces console.log, même si déconseillé dans les projets, pour les besoins du client et des tests unitaires. Les failles de sécurité viennent aussi de l'utilisation de la fonction *eval()* dont nous avons donné l'intérêt dans le chapitre d'Implémentation. Nous laissons quand même cette fonction parce que, dans le contexte de notre projet, il n'est pas très grave de l'utiliser surtout que le site est hébergé sur les serveurs du CREMI.
D'autre part, les résultats sont satisfaisants vis-à-vis de la duplication de code (un taux nul) et des heures de dettes techniques [1].

Après avoir revu le code en vue de correction de bugs, les nouveaux résultats obtenus par SonarQube sont ceux de la figure ci-dessous.

1. Les dettes étant le temps qu'une entreprise aurait mis pour déboguer le code.

Chapitre 6

Résultats obtenus par l'application

6.1 Interface graphique

Vous trouverez ci-joint l'affichage de l'application web avec ces différentes sous parties : "Acceuil" (Figure 6.1), "Commencer"(Figure 6.2 et 6.3), "A Propos" (Figure 6.4).

6.1.1 Page d'accueil

FIGURE 6.1 – Page d'acceuil

6.1.2 Page de simulation

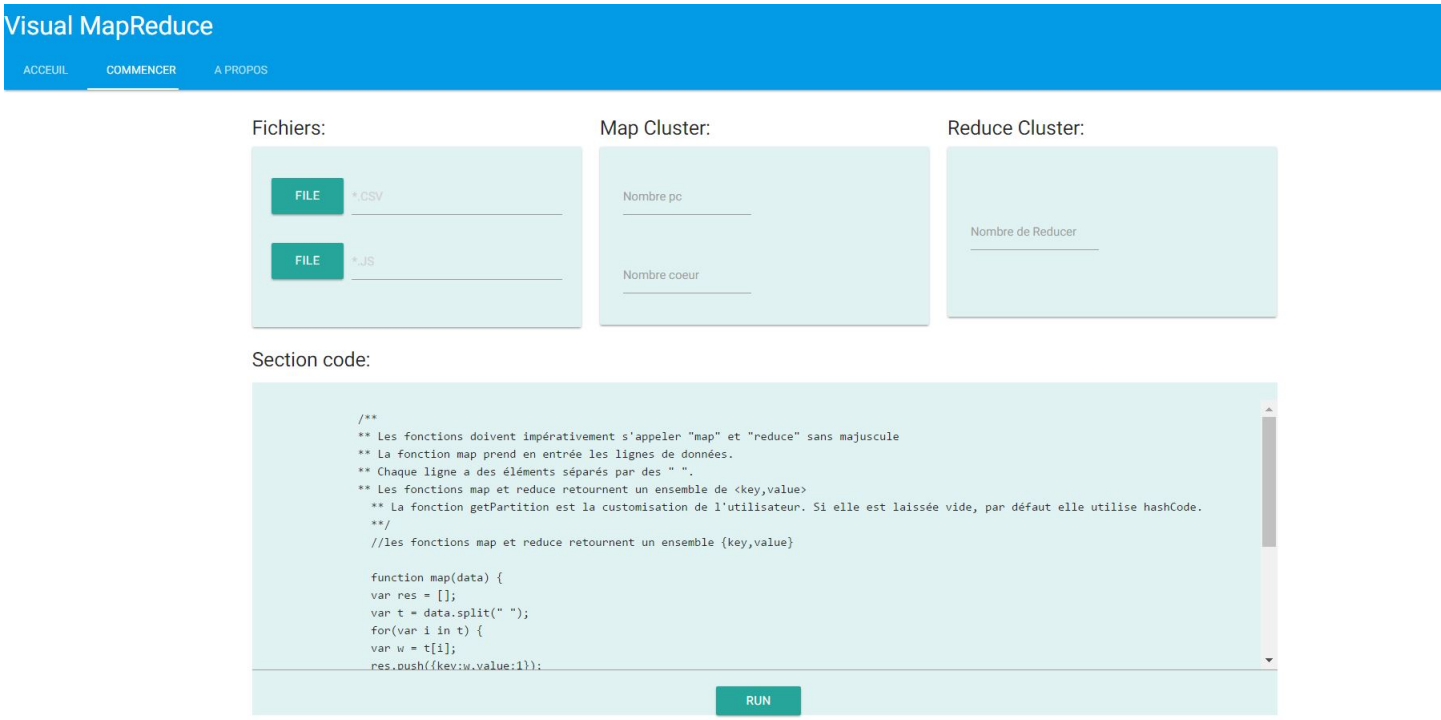


FIGURE 6.2 – Simulation - Paramètres

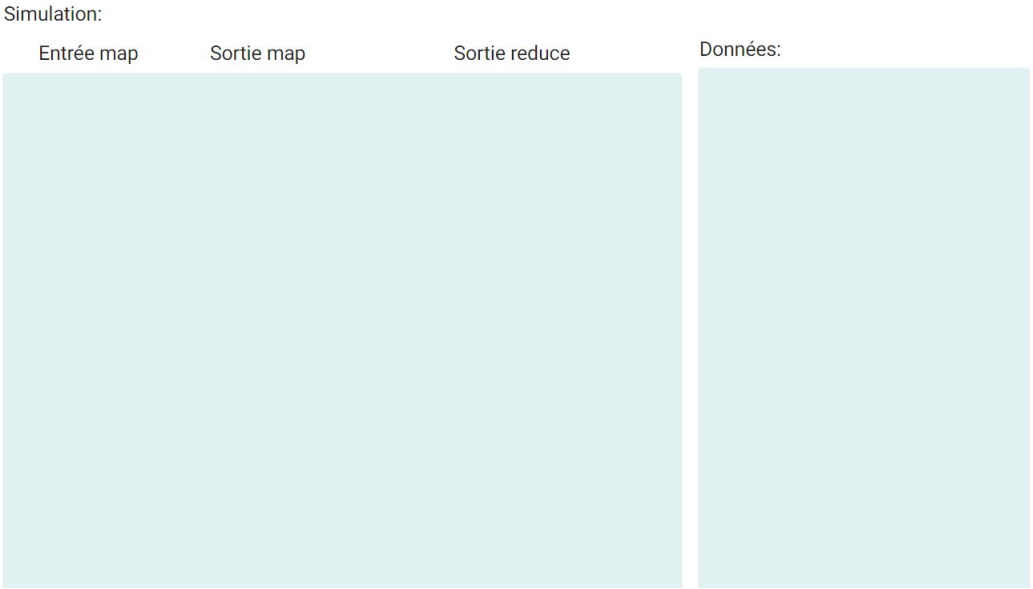


FIGURE 6.3 – Simulation - Cluster et données

6.1.3 Page "A propos"

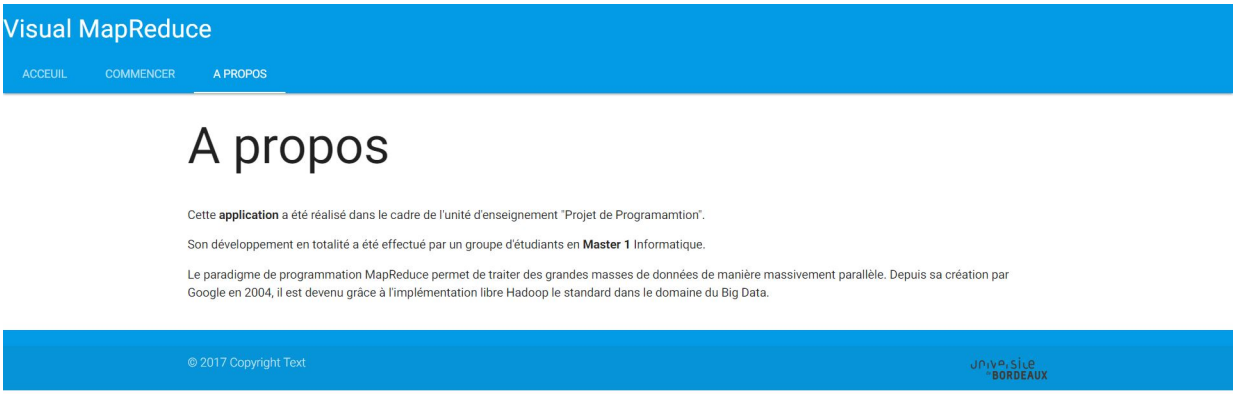


FIGURE 6.4 – Page "A propos"

6.2 Résultats de la simulation

La simulation (Figure 6.5) est permise grâce à FATuM, ce bloc html est donc uniquement affiché par cette bibliothèque.

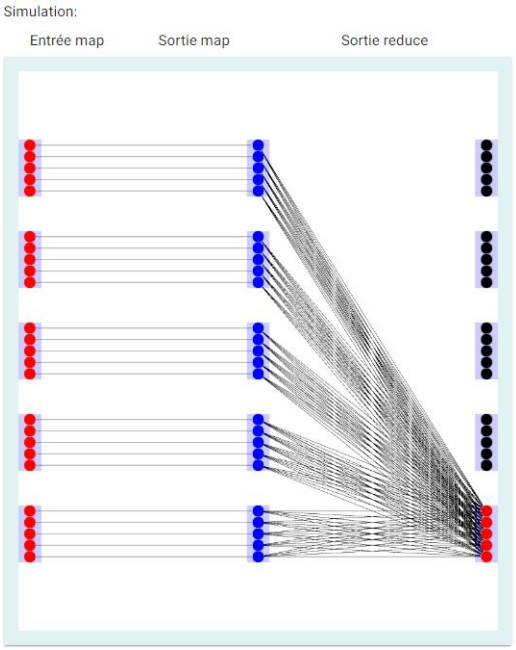


FIGURE 6.5 – FATuM

6.3 Résultats sur console

Enfin, à la demande du client, nous avons mis les résultats des différentes étapes en sortie de console (Figure 6.3).

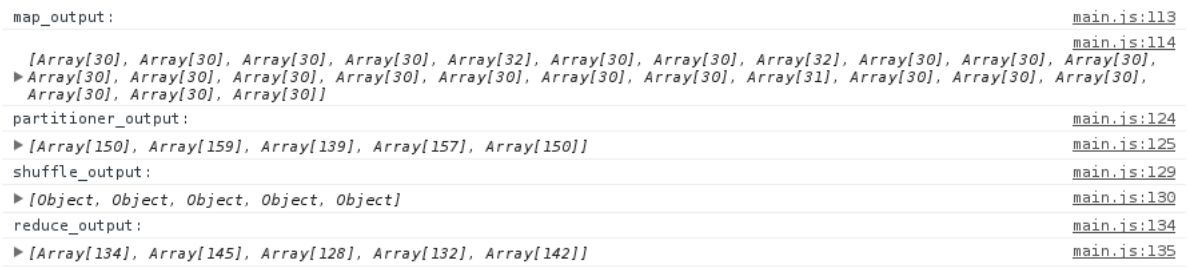


FIGURE 6.6 – Sortie Console

Chapitre 7

Gestion du projet

Ce projet de programmation est basé sur plusieurs contraintes telles que le temps et les conditions imposées par le client. Il est donc nécessaire de bien définir les besoins et de correctement se répartir la charge de travail.

Pour cela, nous nous sommes imposés des délais pour chaque tâche et utiliser des techniques pour la répartition du travail.

7.1 Répartition des tâches

Pour une optimisation du travail durant tout le projet, nous avons utilisé l'outil en ligne Trello. Chaque membre du groupe a sa liste avec les tâches à effectuer, les idées pour l'amélioration du projet ou les retours des réunions avec la chargé de TD ou le client. Dans la figure ci-dessous on donne un aperçu de l'outil Trello.

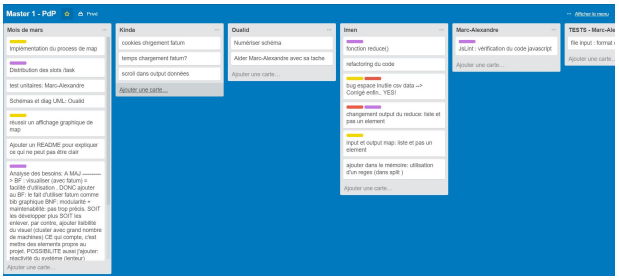


FIGURE 7.1 – Trello - Utilisation

Lorsqu'une tâche est réalisée, elle est archivée. C'est pourquoi sur la figure du dessus, on ne voit que les tâches ou idées de la fin du PdP.

Nous avons également établi un code couleur selon l'importance des tâches comme suit :

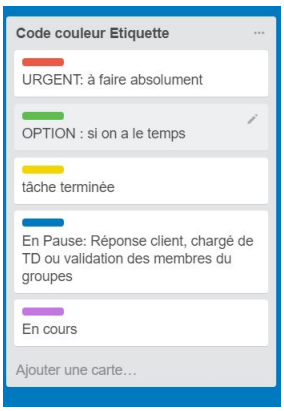


FIGURE 7.2 – Trello - Code couleur

Ainsi chacun peut connaître l'avancement des autres, et connaître leurs occupations actuelles pour éviter que deux personnes ne travaillent sur la même tâches.

7.2 Répartition des fonctions

Pour éviter des conflits permanents avec le dépôt, nous avons décidé de séparer au maximum les fichiers concernant la partie graphique et la partie interpréteur. Ainsi depuis le début du projet, chaque partie est indépendante (dans un répertoire à part).

Lorsque le projet a commencé à arriver à son terme, nous avons fusionné les deux parties. Pour permettre cette fusion, certaines fonctions dans le main sont communes aux deux parties.
Ainsi, dans le répertoire js/ nous avons deux dossiers issus de cette séparation du début : interpreteur/ et view/. La liaison entre la partie graphique et la partie interpréteur se voit dans le fichier "`main.js`" et dans le fichier "`event.js`".
Cette répartition est aussi visible dans la section qui suit où on définit l'arborescence.

Arborescence du dépôt du projet

Nous avons pris soin de bien organiser le dépôt du projet. En effet, il est structuré comme indiqué dans la figure ci-dessus le la manière suivante : deux répertoires principaux pour le contenu du projet (code source) et pour le mémoire (ainsi que le cahier des besoins). Par défaut dans Savane[1], un répertoire nommé "website" est créé automatiquement et ne peut donc pas être supprimé. C'est un répertoire non utilisé pour le projet.

1. Serveur du CREMI

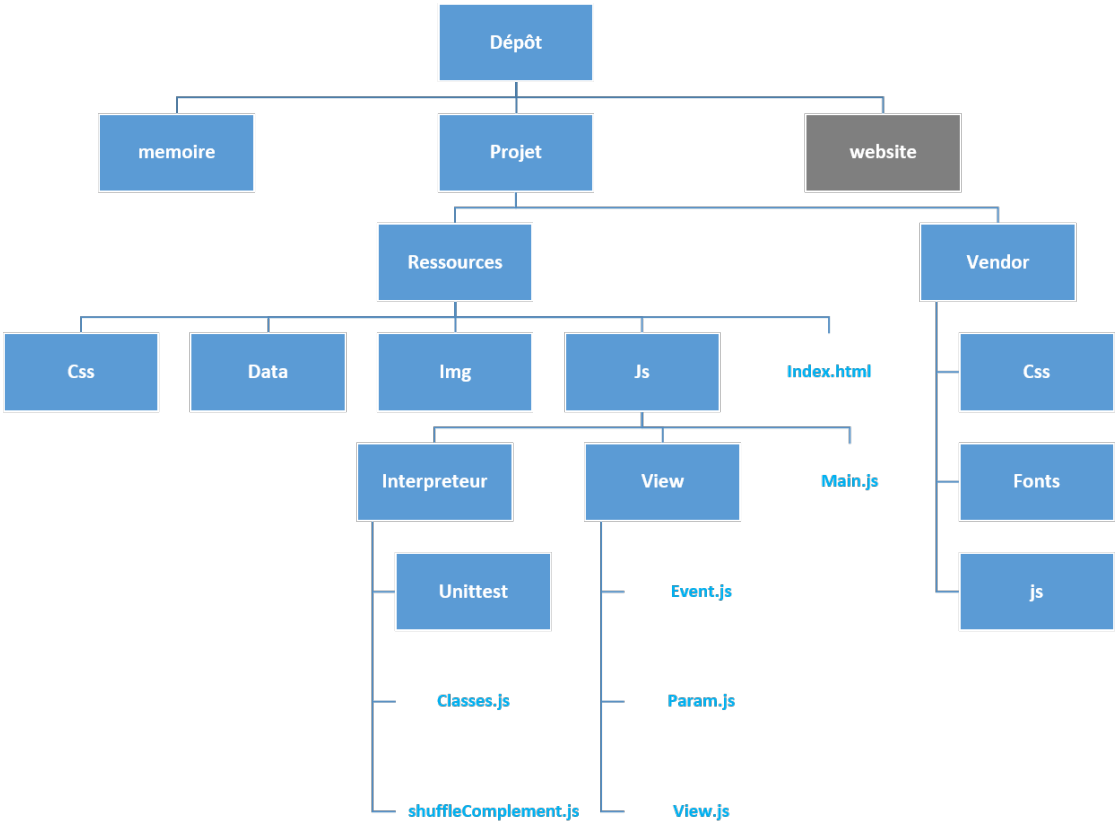


FIGURE 7.3 – Arborescence du dépôt

7.3 Gestion du temps

La gestion du temps a été la plus critique. Nous nous sommes basés sur les rendez-vous avec le chargé de TD pour cette gestion. Ainsi, chaque semaine, chaque membre du groupe avait une tâche à réaliser avec comme délai imparti la veille du TD. Nous avions donc des contraintes de temps plus serrées pour mieux gérer notre temps.

Conclusion

Avant toutes choses, le projet *VisualMapReduce* a été l'occasion pour nous de se familiariser avec le concept de *MapReduce*, enseigné en Master 2.

Le projet *VisualMapReduce*, de part le faible nombre de contraintes imposées par le client, a été l'occasion de découvrir le langage Javascript, ainsi que l'environnement de développement associé, notamment l'éventail des outils fournis nativement par les navigateurs web Firefox et Chrome.

La réalisation d'un projet à 4 personnes, divisible en deux parties distinctes (rendu visuel avec la librairie FATuM, interpréteur du code MapReduce) nous a permis de nous confronter à la réalité de la gestion de projet. La répartition des tâches (architecture, programmation, tests unitaires, documentation, rédaction du mémoire) et le temps dédié à chacunes d'entre elles.

Bibliographie

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce : Simplified data processing on large clusters. https://static.googleusercontent.com/media/research.google.com/fr//archive/mapreduce-osdi04.pdf, 2004. [18 Janvier 2017].

[JF14] Laurent Jolia-Ferrier. *Big Data. Concepts et mise en œuvre Hadoop.* Éditions ENI, feb 2014. [ISBN : 978-2-7460-8688-3].

[LD10] Jimmy Lin and Chris Dyer. Data-intensive text processing with mapreduce. https://lintool.github.io/MapReduceAlgorithms/MapReduce-book-final.pdf, 2010. [31 mars 2017].

[NT16] Raphaël Fournier S'niehotta et Philippe Rigaux Nicolas Travers. Mapreduce, premiers pas. http://b3d.bdpedia.fr/mapreduce.html, 2016. [4 Janvier 2017].

[PG13] Srinath Perera and Thilina Gunarathne. Hadoop mapreduce cookbook. http://barbie.uta.edu/~jli/Resources/MapReduce&Hadoop/Hadoop%20MapReduce%20Cookbook.pdf, 2013. [31 mars 2017].