



# SPARK-ICS

## Exploration de l'application à la bioinformatique de l'architecture « Big Data » Apache Spark

7 septembre 2018

Coordinateurs: jerome.gouzy@inra.fr & ludovic.legrand@inra.fr (LIPM/BBRIC)

Nous contacter si vous souhaitez: i) plus de détails sur un ou plusieurs points du document ii) de l'aide pour organiser une (in)formation iii) organiser un transfert de connaissance iv) utiliser tout ou partie de ce document v) si vous souhaitez contribuer au groupe de travail.

Rédacteurs: jerome.gouzy@inra.fr & ludovic.legrand@inra.fr (LIPM/BBRIC) axel.verdier@inra.fr (LIPM/SUNRISE)

Relecteurs: Erika Sallet, Corinne Rancurel, Céline Noirot, Ludovic Cottret, Sébastien Carrère, Nathalie Vialaneix, Nicolas Lapalu, Alexandre Dehne-Garcia, Bernhard Gschloessl

Contributeurs: Axel Verdier, LIPM/SUNRISE; Ludovic Legrand LIPM/BBRIC; Xavier Garnier IRISA/IFB; Erika Sallet LIPM/BBRIC; Alexandre Dehne-Garcia CBGP/BBRIC; Nicolas Lapalu BIOGER/BBRIC; Martial Briand IRHS/BBRIC; Franck Dorkeld CBGP/BBRIC; Bernhard Gschloessl CBGP/BBRIC; Corinne Rancurel ISA/BBRIC; Martine Da Rocha ISA/BBRIC; Sébastien Carrère LIPM/BBRIC; Ludovic Cottret LIPM/BBRIC; Céline Noirot MIAT/Bios4Biol; Olivier Filangi IGEPP/BBRIC; Nathalie Vialaneix MIAT; Jérôme Gouzy LIPM/BBRIC.

# Contenu

# I. Objectifs

Les objectifs initiaux étaient de mettre en place une architecture basée sur « Apache Spark », très utilisée dans l'univers du « Big Data » industriel. D'une part pour tester sa pertinence dans le domaine de la bioinformatique (et plus particulièrement l'analyse haut débit de données de séquence) et d'autre part pour acquérir et partager des compétences techniques dans les technologies du « Big Data ». Nous avions ainsi prévu d'évaluer la technologie pour répondre à des problématiques d'assignation taxonomique (utilisée dans plusieurs omiques) et l'annotation de génomes qui atteignaient déjà les limites des solutions courantes sur des gros jeux de données.

Prévu pour 6 mois, le projet exploratoire aura duré 2 ans et aura nécessité un investissement en matériel et personnel bien supérieur au prévisionnel. Néanmoins cette extension aura permis de voir apparaître des logiciels développés pour Spark par des équipes bioinformatiques reconnues (Broad Institute/CeBiTec). Cela nous a permis d'enrichir notre analyse en testant leurs logiciels dans le domaine de l'assignation taxonomique mais aussi pour la détection de variants qui est également un problème compliqué sur des gros génomes avec des dizaines de Tb de données de séquences (ex: tournesol). De plus, alors que dans le projet initial nous n'avions (volontairement) prévu de traiter seulement les aspects technologiques et calculatoires du « Big Data », l'extension de la phase exploratoire nous a permis de commencer à aborder les aspects méthodologiques du « Machine Learning » et de commencer à évaluer les solutions intégrées dans Spark dans nos propres développements.

Le dernier objectif était de partager les connaissances acquises au sein des différentes communautés de l'INRA. C'est l'objet de ce présent rapport. Même si nous n'avons pas fini et que nous apprenons encore tous les jours, nous pensons, qu'après 2 ans, il est temps de formaliser et partager notre analyse de la technologie Spark.

# II. Introduction à l'architecture et au fonctionnement d'Apache Spark

Cette section d'introduction à Spark a pour objectif de donner une vision globale du fonctionnement de l'architecture et cela passe par des analogies et des raccourcis. Les sections ci-après préciseront les différents points.

Comme indiqué sur le site https://spark.apache.org/, Apache Spark™ est un « unified analytics engine for large-scale data processing ». L'environnement intègre dans un même écosystème un ensemble de ressources et d'outils informatiques pour exécuter de façon efficace des calculs sur des données. Comme Hadoop qui a popularisé le MapReduce, les données sont distribuées au plus près des ressources de calculs afin qu'un traitement puisse être effectué dans une enveloppe de mémoire vive limitée (quelques Gb, sachant que plusieurs exécuteurs pourront fonctionner en parallèle). Mais contrairement à Hadoop, Spark peut enchaîner les traitements en mémoire [1] sans passer par une sérialisation sur disque ce qui est beaucoup plus rapide et potentiellement très adapté à la bioinformatique qui utilise de nombreux pipelines enchaînant les étapes de filtrages et d'analyses.

Un environnement Spark pour du « Big Data » intègre:

- une distribution intégrant de nombreux outils pour mettre à disposition et s'assurer de la disponibilité des ressources de calcul et de stockage physiques.
- des logiciels pour ordonnancer les calculs sur les ressources disponibles: l'équivalent de SLURM, SGE, etc.
- l'optimisation de la distribution des calculs sur les ressources de calcul demandées. Ainsi, Spark joue un rôle **d'ordonnanceur** (DAGScheduler) pour les tâches en les distribuant sur les ressources à sa disposition comme le ferait SGE avec les « array jobs », **d'optimiseur** en réordonnant les tâches comme un moteur SQL et **de compilateur** en optimisant le code à la volée.
- des logiciels/API pour requêter les données : l'équivalent du SQL des bases de données relationnelles
- des APIs/outils pour analyser les données : l'équivalent de nombreux packages R sur le Machine Learning par exemple
- la possibilité d'exécuter des programmes non Spark sur les ressources de calcul (pipes)

On dispose donc d'un environnement intégrant tous les outils pour développer des applications distribuées. Cela permet de ne se préoccuper qu'un minimum de la distribution des calculs et du reformatage des données pour passer d'un logiciel d'analyse à un autre. En outre Spark peut être instancié sur un ordinateur, un cluster ou dans un cloud, ce qui rend les applications Spark potentiellement hautement « scalable ».

# II. Architecture matérielle et logicielle mise en place

Par défaut les outils « Big Data » se reposent sur HDFS (Hadoop-FS) qui est un système de fichiers distribué résilient et performant. La mise en place d'un cluster Hadoop et de son écosystème applicatif nécessite de nombreuses briques logicielles. L'installation des briques une à une est complexe et nécessite une bonne connaissance des nombreux outils et des compatibilités entre versions.

Heureusement des distributions permettent d'installer simplement un cluster. Il existe 3 distributions majeures : Cloudera (le leader du marché), Hortonworks et MapR. Il existe d'autres distributions proposées par IBM ou Oracle mais elles ont pour base une des distributions majeures.

## II.1. Distribution MapR

**MapR** est une distribution qui facilite le déploiement d'architectures « Big Data » type Hadoop et de l'écosystème associé comme Spark. Il y a une version payante orientée haute disponibilité mais la version « community » est largement suffisante pour notre petit cluster d'évaluation.

**MapR** est la distribution la moins connue mais elle apporte des innovations très intéressantes notamment au niveau du système de fichiers. HDFS et MapR-FS ont en commun de découper les données en blocs et de distribuer ces blocs sur les disques des nœuds du cluster. Chaque bloc est tripliqué, ce qui diminue d'autant la capacité utile du système de stockage mais permet d'avoir un système performant et résilient. MapR-FS utilise des conteneurs de 32Go pour stocker les blocs et ce sont les conteneurs qui sont distribués et répliqués et non les blocs directement. De plus, si MapR-FS est un système de fichiers distribué propriétaire, certaines de ses fonctionnalités complémentent certains défauts de HDFS. Parmi ces fonctionnalités on retrouve:

- La compatibilité avec les systèmes de fichiers POSIX, y compris au travers de NFS
- La compression native des données
- Une distribution des métadonnées des fichiers sur l'ensemble des nœuds (vs un ou deux nœuds dédiés pour HDFS)

Les deux premiers points ont été déterminants dans notre choix d'utiliser MapR pour notre cas d'utilisation. La compression native des données (peu de systèmes de fichiers proposent cette fonctionnalité) permet de compenser partiellement la réplication x3 et la compatibilité POSIX permet de transférer les fichiers sur HDFS avec les commandes standards Unix de manipulation de fichiers. Il est à noter qu'un système de fichiers compatible POSIX est bien adapté à un système orienté calcul, dans lequel le système de fichiers est un espace temporaire. Par contre, pour un projet orienté Data Lake où le système de fichiers devient le point central pour la conservation des données, le fait de pouvoir modifier des données via POSIX deviendrait un point négatif.

Il doit être gardé à l'esprit que les conclusions/informations ci-après sont liées aux versions testées et que cela peut changer pour les versions futures de MapR.

## II.2. Installation et mise à jour

L'installation du cluster se fait en 2 étapes. La première étape consiste à installer l'ensemble des serveurs avec une distribution linux, si possible supportée par MapR, sans faire de RAID sur les disques destinés à MapR-FS. Dans notre cas chaque serveur possède 6 disques de 1To avec une répartition simple : 1 disque pour le système Linux/Debian et MapR et 5 disques pour MapR-FS.

La première installation sur 4 serveurs avec MapR 5.1 (été 2016) a été compliquée à cause d'un changement de politique de sécurité au niveau des dépôts *apt*. Il était impossible d'installer les paquets de MapR par *apt*, tout a du être fait manuellement via *dpkg*. Lors de la seconde installation sur 8 serveurs (été 2017), nous avons installé une Debian Stretch qui n'est pas officiellement supportée par MapR-FS mais qui reste assez proche de Ubuntu qui est supportée. Au prix de petits changements pour faire passer une Debian pour une Ubuntu (il suffit de remplacer les informations Debian par des informations Ubuntu dans /etc/lsb-release[i]), la configuration du cluster sous MapR 5.2 s'est déroulé sans problème.

La seconde étape consiste à télécharger et exécuter un script d'installation sur un des nœuds du cluster, ce nœud devient (par défaut) un nœud d'administration et ne fera pas de calcul. A partir de ce moment-là, on a accès à une interface web qui nous permet d'enregistrer les serveurs du cluster, de lister les ressources (CPU, RAM et disques) et d'allouer les disques au système de fichiers.

La distribution intègre de nombreuses briques mais celles que nous avons installées en mode « Custom » sont :

- **MapR-FS** (compatible HDFS) qui rend accessible l'espace de stockage clustérisé à travers NFS. Il organise les fichiers en conteneurs de 32Go.
- Apache **Drill** qui permet de requêter en **SQL** des bases de données NoSQL et des fichiers sur un système de fichiers comme **MapR-FS**.
- Apache **Hive** est un entrepôt de données qui se base sur HDFS pour le stockage des données et un metastore pour stocker le schéma des données. Il joue un rôle important dans Hadoop pour le stockage des schémas, c'est un composant indispensable.
- **MapR-DB** est une base de données NoSQL multi-modèles propre à MapR.
- le scheduleur **YARN** qui va permettre de gérer les ressources et la charge du cluster spark. C'est l'équivalent de SGE pour un cluster de calcul standard. Tout comme SGE il y a un daemon sur les nœuds « maîtres » gérant le cluster et un daemon tournant sur tous les nœuds du cluster.
- l'environnement de calcul distribué **Spark**

Il y a aussi des briques pour garantir la cohérence du cluster et fournir les services :

- Apache **Zookeeper**, c'est un service haute disponibilité installé (et actif) sur 3 nœuds. Il gère les configurations et l'état des services du cluster.
- MapR **CLDB** est un service propre à MapR-FS, il est en mode « master/slave ». Il gère la localisation et l'état des conteneurs ainsi que les quotas et règles de MapR-FS.
- MapR **NFS** est le service permettant aux clients de monter en NFS le système de fichiers (non réplicable avec la version « community »)

L'installation est simple, l'outil graphique s'occupe de tout, une fois les serveurs et les disques définis. L'installateur va assigner des rôles aux différents serveurs et installer les composants nécessaires. Il est possible de relancer le processus en cas d'erreur (après inspection des logs et correction du problème bien sûr).

Si l'installation est simple, le découpage automatique des rôles avec 8 serveurs ne nous convient pas totalement car il met de côté 3 serveurs qui servent à assurer la cohérence du cluster et donc ne font pas de calculs (mais participe à MapR-FS). Nous n'avions pas rencontré ce cas lors de la première installation avec 4 serveurs. A l'heure actuelle nous n'avons pas changé la configuration mais il est prévu d'externaliser la partie « administration » du cluster sur des machines virtuelles (VM) pour récupérer les 3 serveurs physiques dans le pool de calculs.

A ce jour, nous avons fait deux installations, la première avec 4 serveurs et la seconde avec les 8 serveurs. Nous n'avons pas fait de mise à jour d'un cluster avec une montée de version. C'est prévu pour 2018 car une nouvelle version de la distribution MapR avec une version de Spark, intégrant une nouvelle version de la librairie de "Machine Learning" va être mise à disposition d'ici la fin de l'année. A cette occasion nous pourrons évaluer la montée de version et les possibilités d'ajout de nouveaux nœuds avec 3 VMs pour prendre en charge les services d'administration du cluster.

Pour utiliser le cluster Spark, nous avons mis en place une VM qui sert de frontal pour soumettre les applications Spark. Pour installer ce serveur, il faut ajouter les dépôts de MapR et installer les paquets suivants :

- **mapr-client**, permet d'interagir avec le cluster
- **mapr-loopbacknfs**, permet d'installer le client NFS pour MapR-FS
- **mapr-spark**, permet d'avoir les outils Spark comme spark-shell

Pour mapr-client, un script de configuration permet de spécifier les serveurs et les services du cluster.

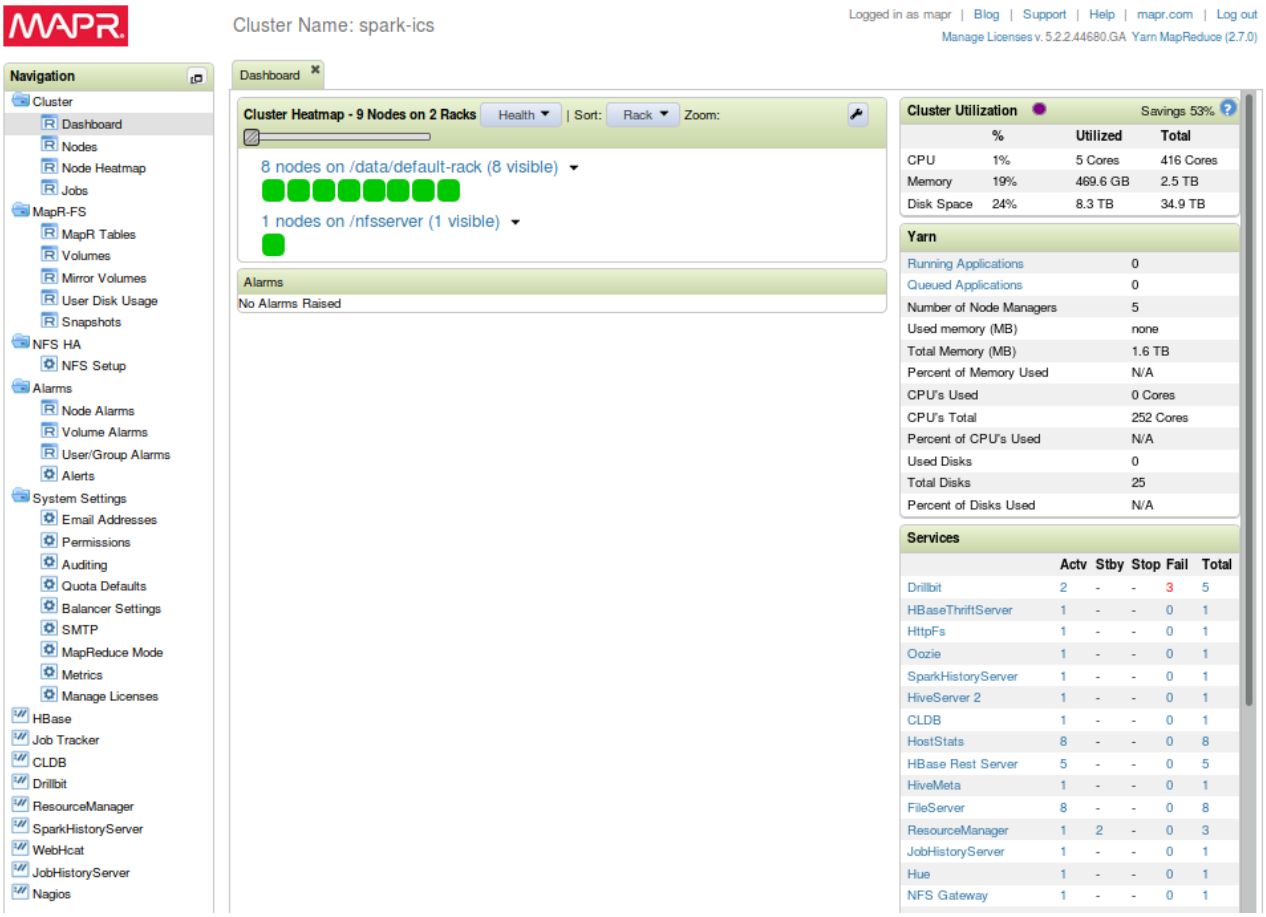
La configuration de YARN et Spark est moins « user friendly ». Il faut copier des fichiers de configuration et des librairies depuis le cluster. Pour Spark, il est préférable de modifier le fichier de configuration spark-defaults.conf pour forcer l'utilisation de YARN et éviter de lancer du Spark sur le frontal. L'installation du frontal est moins détaillée dans la documentation de MapR et a été plus fastidieuse que la partie cluster.

## II.3. Administration système et monitoring

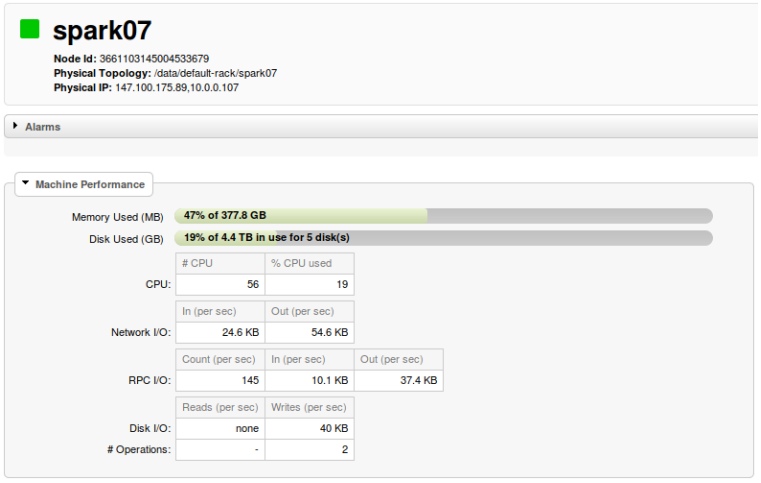
### II.3.1. Administration système

MapR fournit un « dashboard » web permettant de surveiller l'état du cluster et de redémarrer des services. Via cette interface, il n'est pas possible de modifier le rôle d'un serveur, de modifier les services installés ni de modifier la configuration des différentes briques logiciel. Au final cette interface est assez limitée en dehors de la visualisation de l'état global du cluster et nécessite le passage en ligne de commande pour des opérations de configurations.

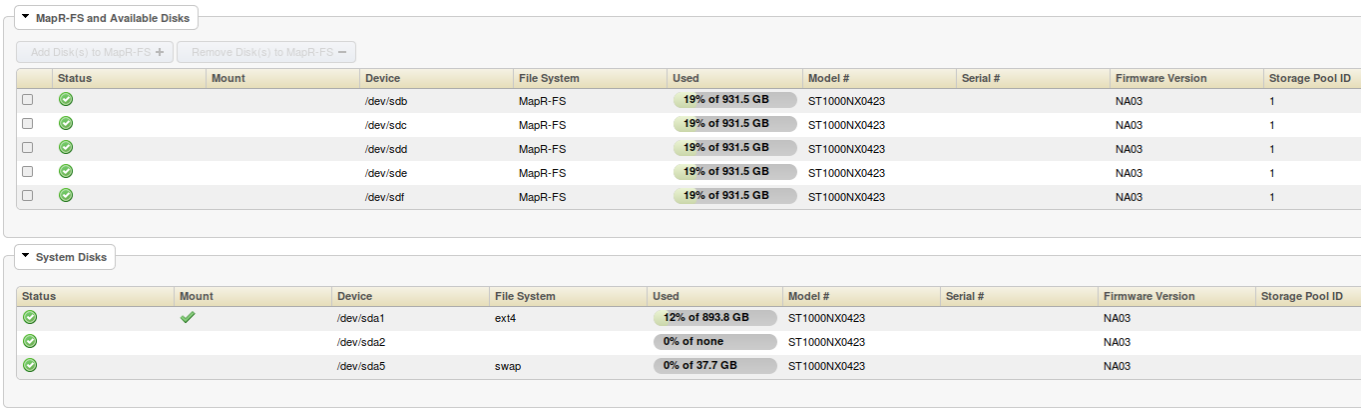
Néanmoins les figures ci-dessous illustrent différentes vues proposées par le « dashboard ».



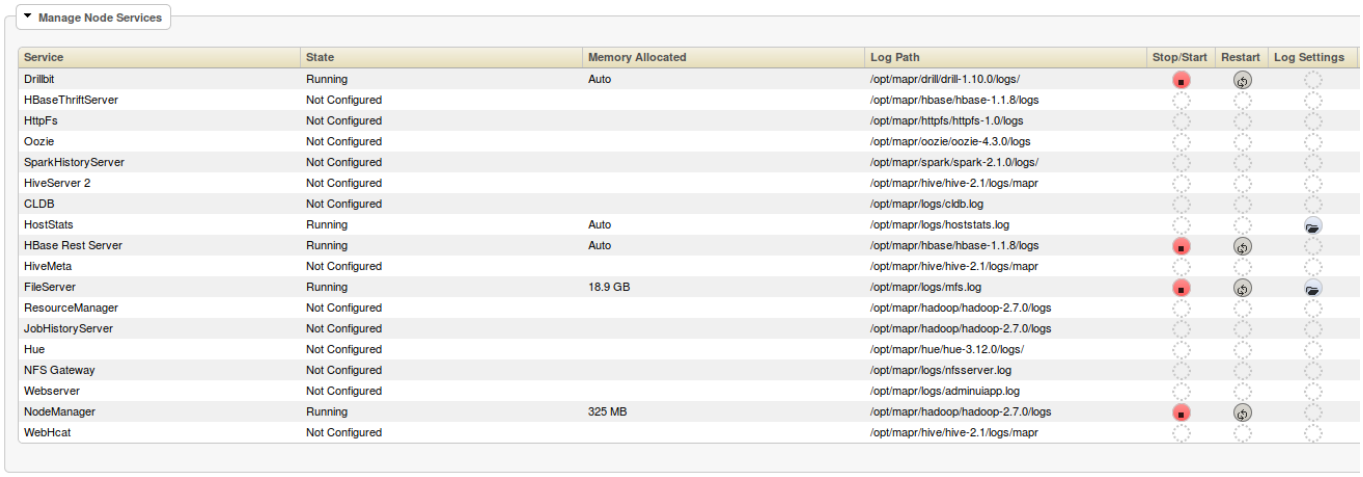

*Dashboard MapR.*




*Indicateurs généralistes pour un nœud.*




*Indicateurs du stockage pour un nœud.*




*Etat des services du nœud.*

MapR fournit aussi un outil en ligne de commande *maprcli* qui permet de voir de manière plus poussée l'état des services sur les nœuds et de faire quelques actions mais sans pouvoir modifier les rôles. Cet outil a l'avantage de pouvoir être encapsulé dans des scripts pour faciliter la création de comptes mais ne semble pas permettre une administration complète du cluster. Nous n'avons pas encore testé de manière poussée la partie administration mais c'est un point négatif des versions actuelles de MapR de ne pas proposer quelque chose de plus complet. A contrario, la distribution Cloudera semble offrir un outil beaucoup plus avancé d'administration d'un cluster sans devoir modifier en ligne de commande les fichiers de configuration.

Nous avons sécurisé l'accès au frontal mais à partir d'une connexion sur le frontal on peut voir la totalité des données et des jobs du cluster. Nous ne pouvons donc pas nous prononcer sur la facilité de la mise en place d'une véritable politique de sécurité.

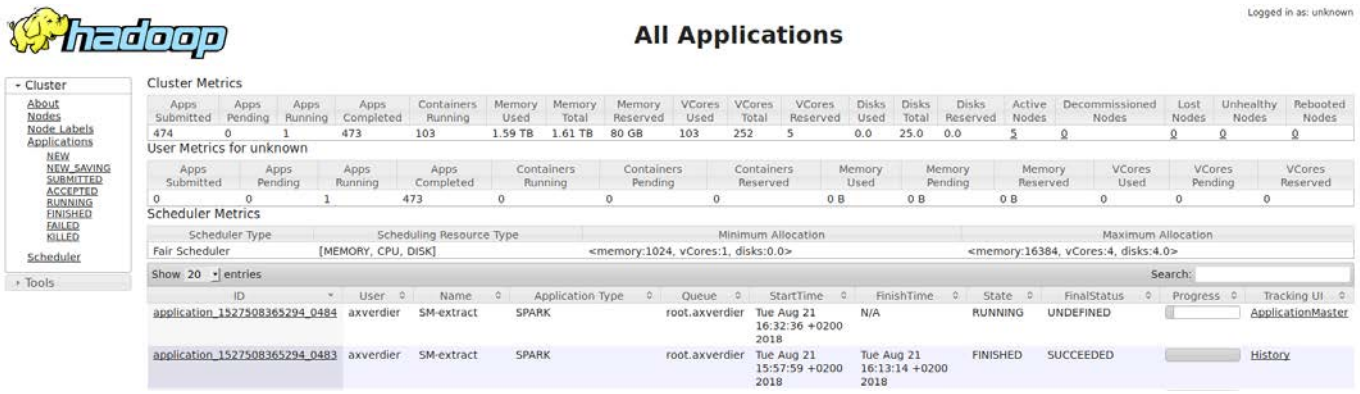
### II.3.2. Monitoring du cluster

Pour la partie monitoring, MapR fournit plusieurs outils :

- le « dashboard » qui permet d'avoir une vue en temps réel sur quelques indicateurs génériques concernant les nœuds et de positionner des alarmes par email.
- un monitoring plus poussé est disponible avec comme base de données OpenTSDB et Grafana pour la visualisation. Il faut installer et configurer le module lors de l'installation.
- pour les logs il est possible aussi d'installer des modules pour faire une indexation dans Elasticsearch et visualiser les informations dans Kibana

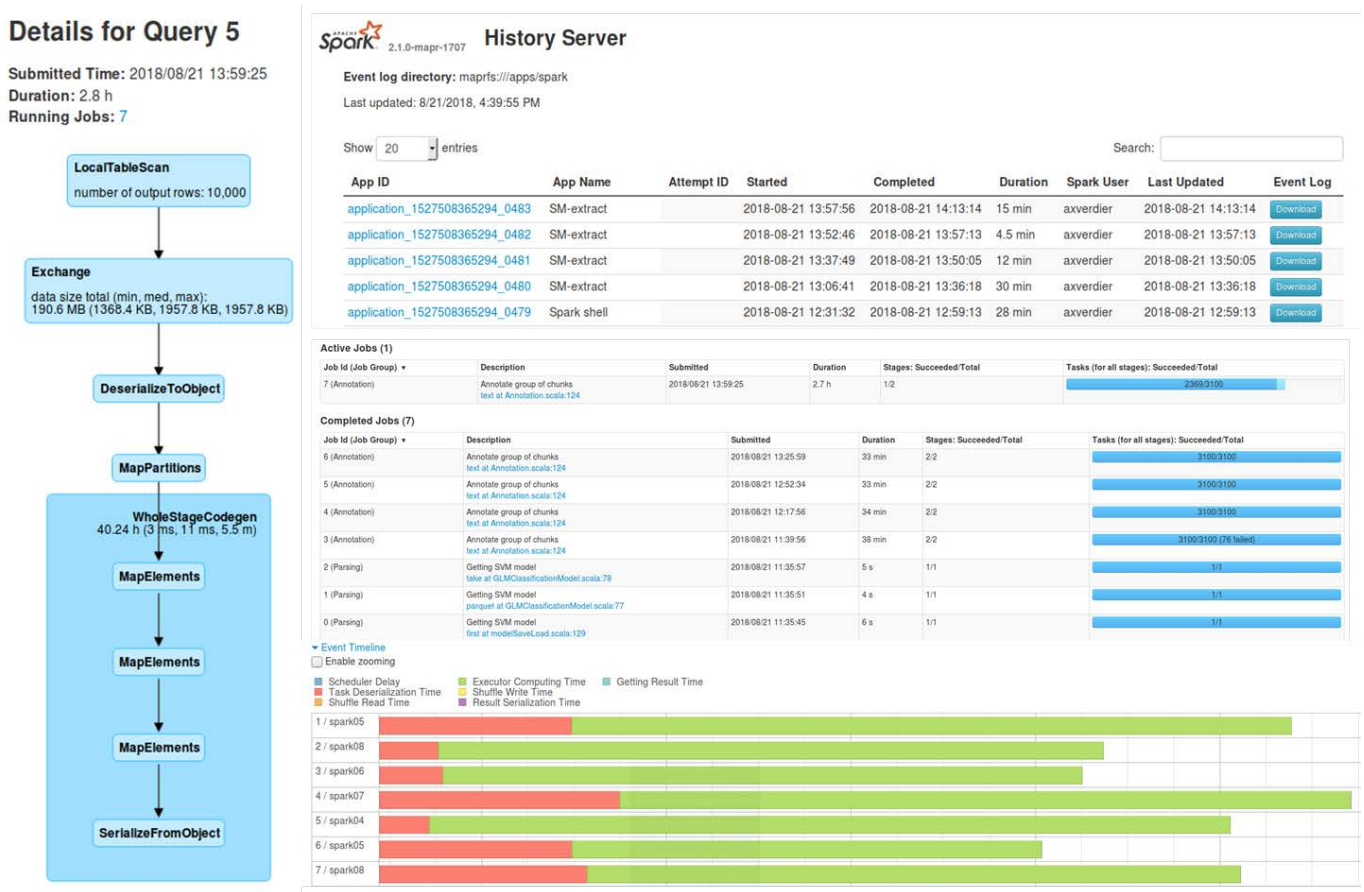
Les outils proposés étant très proches de ceux utilisés pour les serveurs du LIPM, nous avons préféré utiliser nos outils pour monitorer le cluster avec une sonde de collecte Telegraf, une base de données InfluxDB et Grafana pour visualiser les métriques. Nous n'avons donc pas testé la partie monitoring propre à MapR pour le cluster.

Spark ne fait pas partie du cœur de la distribution MapR, c'est un module externe appartenant à l'écosystème « Big Data ». Par défaut, il n'y a pas de monitoring particulier fourni par MapR pour les applications Spark, on ne voit que les informations issues du Ressource Manager (YARN) via une application web dédiée.




*Interface web du Resource Manager (YARN).*

Quand on installe le module Spark, il y a un *History Server* propre à Spark installé sur les nœuds d'administration pour suivre les jobs Spark. Cette application web permet à la fois de suivre les jobs Spark mais aussi d'avoir un « profiling » complet de l'application. C'est une interface très importante lors du développement des applications car elle permet d'avoir accès au découpage de l'application en *jobs*, *stages* et *tasks,* au DAG (Directed Acyclic Graph) généré pour chaque *stage* de l'application, aux indicateurs de performances et aux optimisations effectuées par Spark (quand on utilise les structures de données *DataFrame).*




*Interface web de l'History Server (Spark). A gauche le DAG d'un job. En haut un historique des applications exécutées, au milieu le découpage en jobs de l'application et en bas l'utilisation des nœuds.*

Finalement, il y a beaucoup d'interfaces et d'outils pour surveiller les différents services et les jobs du cluster mais ils ne sont pas actuellement unifiés au sein d'une interface regroupant l'ensemble des informations.

## II.4. Configuration matérielle du cluster et du frontal

Le cluster actuel est constitué de 2 Dell C6320 intégrant 4 serveurs physiques chacun et d'un serveur frontal qui est une VM basée sur un hyperviseur Xen (serveur bbricxen). Tous les serveurs physiques ont un lien 1Gb pour l'accès au réseau du data center (internet et autre) et un lien 10Gb sur un switch du LIPM.

Les serveurs du premier C6320 achetés dans le cadre du projet DTN/BBRIC possèdent les caractéristiques suivantes :

- 2 sockets Intel Xeon E5-2650 de 12 cœurs à 2.20GHz soit 48 cœurs avec l'hyperthreading
- 256Go de RAM (5,3Go par cœur/HT)
- 6 disques 1To NLSAS 7.2krpm

Les serveurs du second C6320 achetés dans le cadre du PIA SUNRISE possèdent les caractéristiques suivantes :

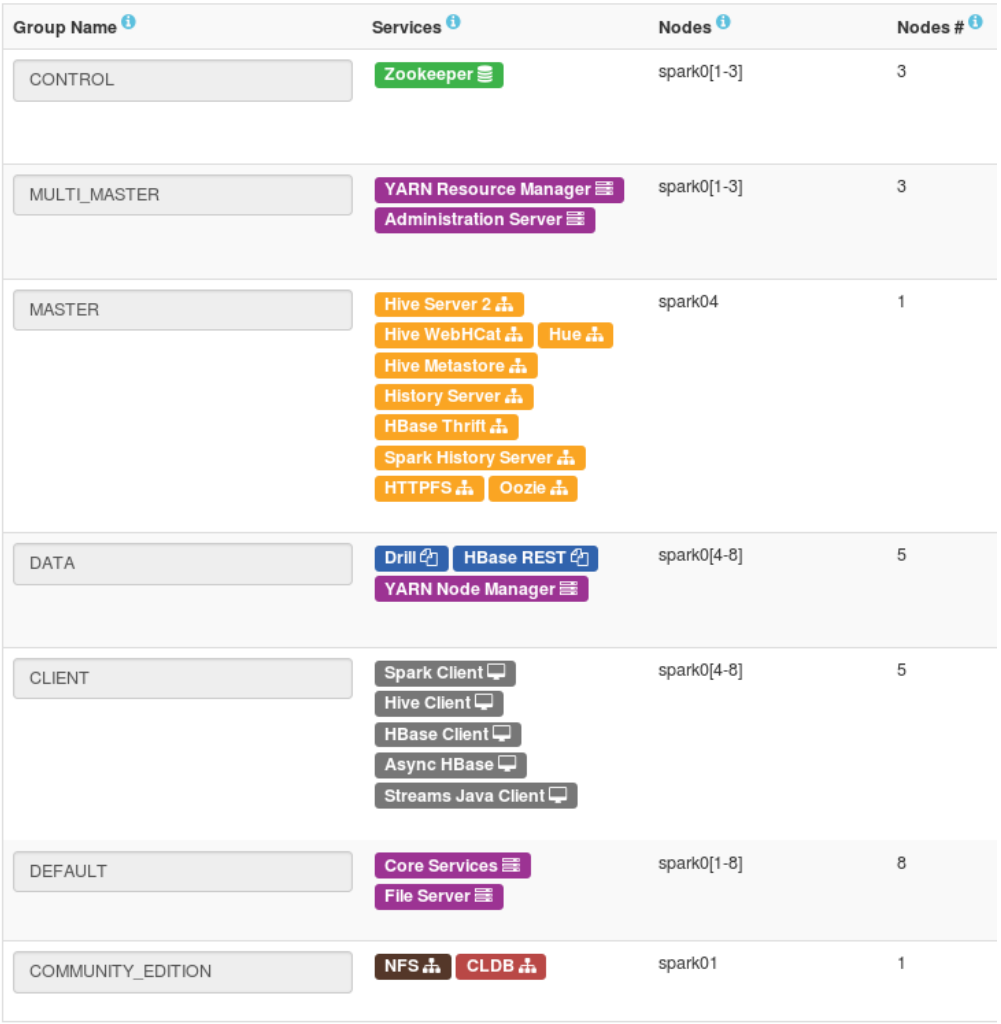
- 2 sockets Intel Xeon E5-2680 de 14 cœurs à 2.40Ghz soit 56 cœurs avec l'hyperthreading
- 384Go de RAM (6,8Go par cœur/HT)
- 6 disques de 1To NLSAS 7.2krpm

L'utilisation des Dell C6320 nous permet d'avoir une densité importante (2*4 serveurs dans 4U pour 3kW) mais limite le stockage. Le premier point de fragilité de la configuration est le disque système qui n'est pas redondé, en cas de crash le nœud est perdu et il faudra le réinstaller. Ce point n'est pas critique au niveau opérationnel car le cluster est capable d'encaisser la perte de plusieurs nœuds grâce à la redondance des données et des services. Le second point est la présence de 5 disques pour MapR-FS pour environ 50 cœurs soit 1:10 alors que le ratio recommandé est entre 1:2 et 1:4. Actuellement nous n'avons pas vu d'impact sur les accès disques donc nous sommes plutôt satisfaits de ce ratio pour nos applications. Nous n'avons pas évalué intensivement les outils de base de données disponibles dans le cluster (Drill et Mapr-DB) qui pourraient eux montrer les limites d'un tel ratio disque/CPU.

Enfin le serveur frontal Spark possède 4 vCPUs, 48Go de RAM et il est relié par un lien 1Gb au cluster. Il ne fait pas partie du cluster mais est capable d'y accéder pour permettre la soumission des jobs via YARN.

L'infrastructure actuelle du cluster répond bien à nos besoins actuels (et à notre budget!). Nous n'avons pas constaté de point limitant fort au niveau matériel à l'exception du disque système fortement sollicité lors de certaines étapes de consolidation/répartition des structures de données (*shuffle*). La seule alternative possible pour optimiser serait de mettre un SSD comme disque système mais il faut un disque autour de 1To pour absorber les données écrites lors de ces étapes ce qui aurait un coût non négligeable.

Nous allons tenter prochainement d'optimiser deux points au niveau de la configuration. Le premier concerne l'externalisation des services d'administration sur des VMs et le second est le réseau car notre configuration Spark dispose de 2 réseaux (1Gb et 10Gb) mais semble utiliser préférentiellement le réseau 1Gb.



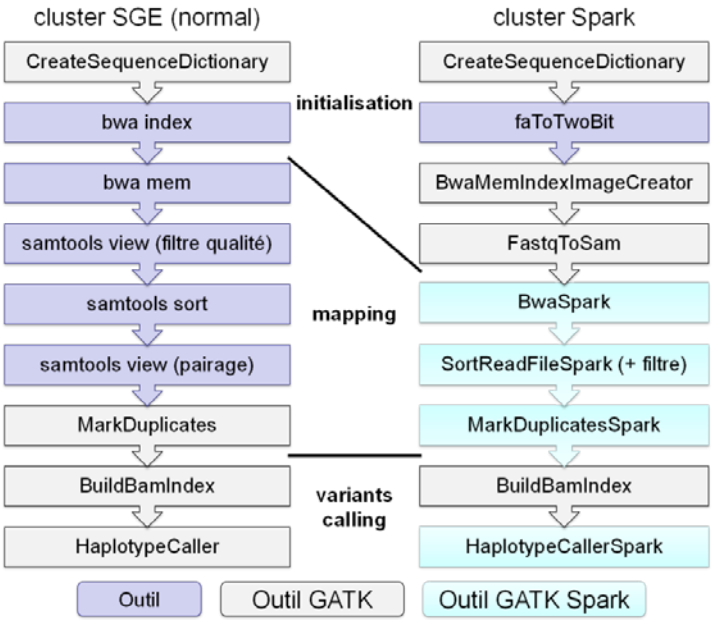

*Répartition des services sur les nœuds.*

# III. Evaluation de logiciels bioinformatiques Spark

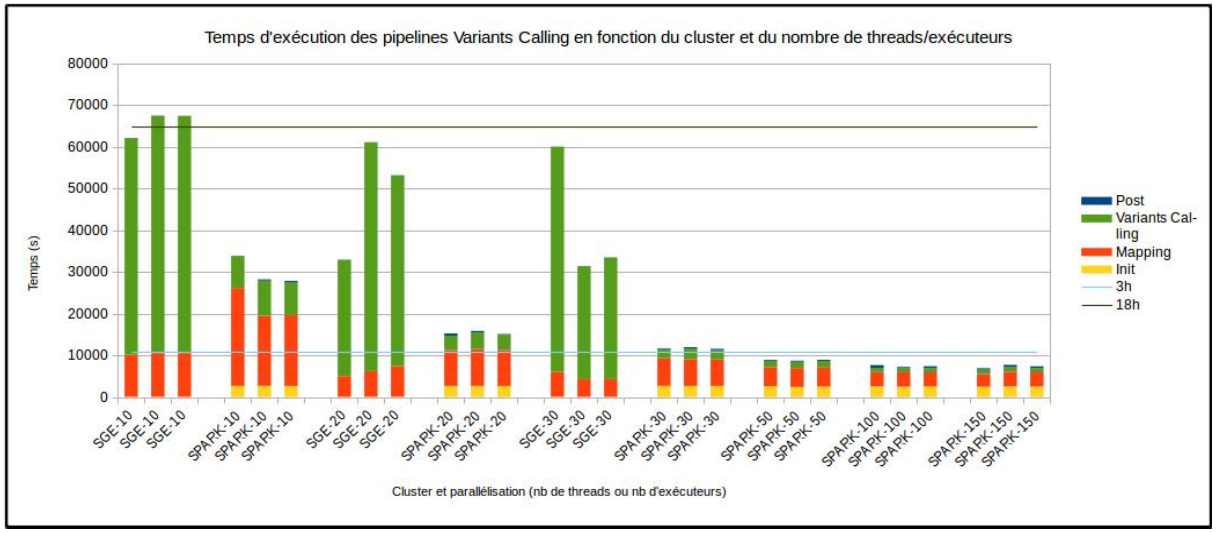
## III.1. GATK: détection et l'analyse du polymorphisme.

GATK, développé par le Broad Institute, est sûrement l'outil le plus connu/utilisé en bioinformatique qui ait été porté réellement sur Spark. Ainsi la version 4 diffusée en janvier 2018 implémente le choix fort d'utiliser l'architecture Spark pour accéder aux ressources distribuées de calcul. Néanmoins, cette version intègre de nombreux outils en version bêta. A ce jour (30 août 2018) elle n'est donc pas recommandée pour la production.

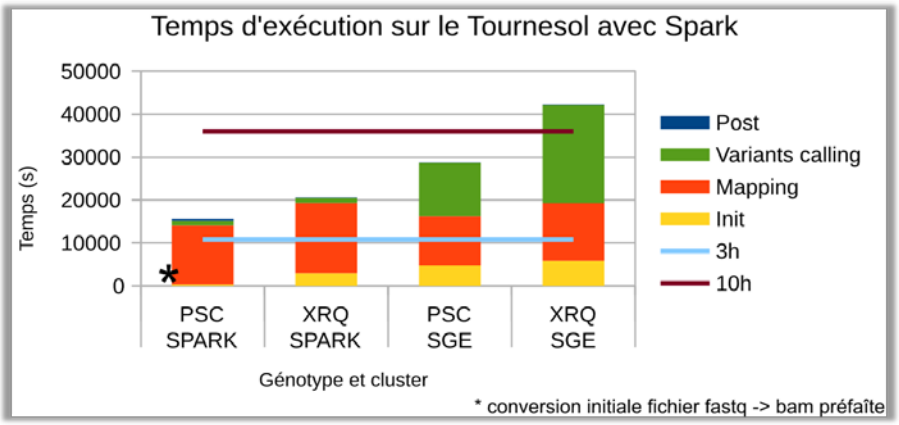
Pour comparer les performances (axel.verdier@inra.fr), les deux pipelines ont tout d'abord été implémentés sur le même cluster physique en mode natif. C'est-à-dire que la version précédente a été implémenté en mode SGE et la dernière version en mode Spark. Les deux pipelines sont décrits ci-dessous.




Pour une première expérience, nous avons utilisé deux jeux de données: un petit génome d'oomycètes (80Mb) sur lequel ont été mappées les 100x de données paired end illumina ayant servi à l'assemblage (le génome étant très homozygote on s'attend à très peu de polymorphisme). Afin d'évaluer d'éventuels effets de bord, les analyses ont été exécutées trois fois. Les temps de calcul sont reportés ci-dessous.



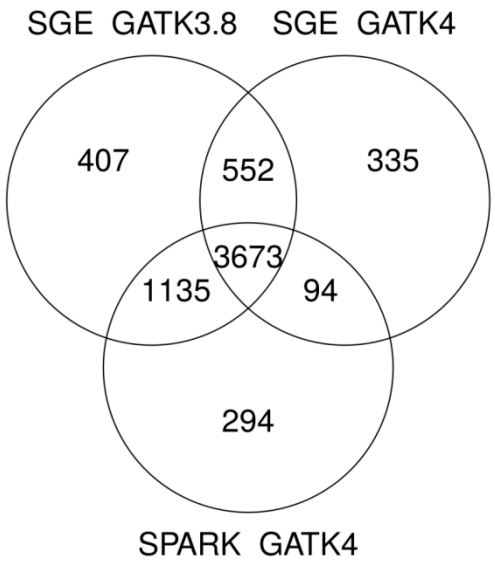

Pour changer d'échelle, la deuxième expérience a été faite sur le génome du tournesol (3Gb soit ~30 fois plus gros) sur lequel nous avons mappé les données du même génotype que celui ayant servi à assembler la référence (XRQ) et un deuxième génotype (PSC). Dans les deux cas nous avons mappé ~15x de données (fastq pairés 2*100nt), sans répétition.




La version SGE est moins performante, particulièrement la partie HaplotypeCaller. Nous avons fait de nombreux tests pour essayer de l'optimiser en mode SGE (paramétrage garbage collector, tests multi-threading) mais clairement le logiciel HaplotypeCaller non Spark sous-utilise les ressources de calcul. La réécriture et le changement de l'algorithme lors de la migration sous Spark ont nettement amélioré ses performances.

Durant le passage à l'échelle du tournesol nous avons dû travailler sur le partitionnement des données afin de ne pas saturer la mémoire des exécuteurs. Pour le tournesol, la comparaison SGE vs Spark arrive aux mêmes conclusions favorables à Spark pour les temps de calcul.

En termes qualitatifs, la comparaison des résultats obtenus sur l'oomycètes *Plasmopara halstedii* (positions homozygotes/alternatifs uniquement) avec les différentes versions semble montrer que la version 4 en mode Spark semble plus proche de la version précédente que la version 4 non Spark (1135 en commun vs 552) tout en produisant moins de résultats spécifiques (294 vs 335). Cette analyse préliminaire nous incite donc à creuser plus encore l'utilisation de la version Spark.




Notre analyse peut être complétée par le manuscrit bioxriv Heldenbrand, J. R. *et al.* qui présente une autre évaluation des versions de GATK.

Au final, certaines phases du protocole Spark doivent être améliorées. C'est le cas de la phase de conversion des fichiers Fastq en fichiers BAM triés par nom de lectures qui est très lente et qui pourrait être facilement recodée pour générer directement les fichiers triés sans multiplier les conversions/tris. Néanmoins, l'utilisation de Spark pour faire l'analyse du polymorphisme nous semble tout à fait possible pour de très gros jeux de données.

## III.2. Sparkhit (métagenomique, polymorphisme, etc.)

Sparkhit est un logiciel développé par le laboratoire CeBiTec en Java [3], il propose plusieurs outils pour la métagénomique et le variant calling ainsi que quelques outils de statistiques.

Sparkhit propose 2 outils pour le mapping de lectures NGS :

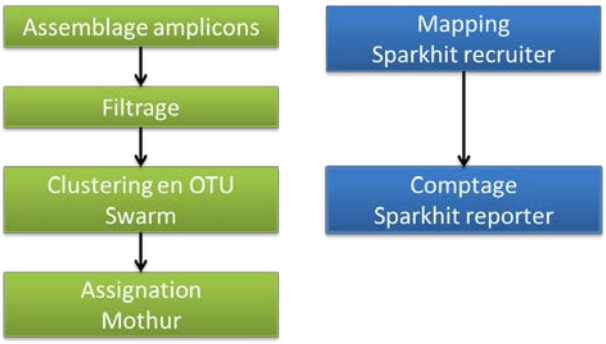
- **mapper**, acceptant peu de mismatchs
- **recruiter**, acceptant plus de mismatchs et donc plus adapté à la métagénomique

Les 2 outils fonctionnent de la même manière avec un index de kmer (taille 11 par défaut) et des algorithmes différents permettent de filtrer les hits. Ces outils sont simples d'utilisation et supportent en entrée les formats habituels de la bioinformatique à savoir Fasta et Fastq. Il y a quelques points problématiques :

- le format de sortie des 2 outils de mapping est un format type TSV spécifique à l'outil et non un format SAM/BAM.
- il n'utilise pas l'information de distance des paired-end car les lectures sont traitées indépendamment.

Pour l'évaluation de l'outil en métagénomique nous avons utilisé un jeu de données bactérien déjà analysé précédemment avec des outils classiques en métagénomique mais la comparaison entre les 2 méthodes est restée assez succincte car il est difficile de faire une comparaison exhaustive sans faire la totalité de l'analyse.

Le jeu de données est composé d'environ 30M de lectures paired-end MiSeq soit environ 2x15Go au format Fastq compressé. Dans le cas de Sparkhit nous n'avons utilisé que la read1 de la paire. La banque de référence *gyrB* comporte 30525 séquences soit 75 Mbases. La figure suivante décrit les deux protocoles d'analyses.




Au niveau des métriques globales du pipeline Swarm/Mothur nous avons: 18.955.647 amplicons valides, 12.106.766 amplicons assignés dans un OTU *unclassified* au niveau du genre et 6.813.538 amplicons assignés dans un OTU avec au moins 1000 représentants au niveau genre.

Pour le pipeline Sparkhit nous avons: 16.062.187 amplicons ayant un match avec au moins 80% d'identité sur la banque *gyrB,* 2.240.862 amplicons ayant une assignation *unclassified* au niveau du genre et 13.775.869 amplicons assignés à un genre ayant au moins 1000 représentants

Ci-dessous, le tableau de comparaison des abondances des 10 genres les plus abondants montre des résultats cohérents pour les effectifs les plus importants.

| Genre | % Swarm/Mothur | % Sparkhit | Position Sparkhit |
|---|---|---|---|
| *Sphingomonas* | 49,6% | 30,2% | 1 |
| *Pseudomonas* | 5,6% | 5,1% | 4 |
| *Variovorax* | 5% | 6,3% | 2 |
| *Acidovorax* | 3,7% | 4,6% | 5 |
| *Brevundimonas* | 3,3% | 5,8% | 3 |
| *Methylibium* | 2,7% | 3% | 9 |
| *Methylobacterium* | 2,4% | 1,9% | 12 |

| | | | |
|---|---|---|---|
| *Janthinobacterium* | 2,4% | 2,6% | 10 |
| *Burkholderia* | 2,1% | 1,2% | 19 |
| *Hymenobacter* | 2% | 0,7% | 24 |

Il est à noter que Sparkhit est particulièrement efficace pour faire de l'assignation taxonomique à partir d'une banque de séquences de référence comme il en existe pour les ARN 16S/18S ou *gyrB*. Néanmoins, l'exhaustivité de la banque est un facteur important car tout ce qui ne matche pas avec les séquences de la banque n'est pas pris en compte, contrairement aux pipelines classiques où les OTUs ne sont pas forcément assignés mais sont conservés et peuvent jouer un rôle important.

En conclusion, seulement avec les abondances il reste difficile de se faire une idée précise de la qualité des résultats de Sparkhit par rapport au pipeline Swarm/Mothur. En revanche la facilité d'utilisation de Sparkhit et ses performances permettent de traiter rapidement des jeux de données, sans considération de taille. Dans notre cas les 15Go de données ont été traités en 1h15 avec 125 cœurs.

### III.3. Autres

Le document bioxriv (Guo *et al.*, [4]) propose une version récente des logiciels disponibles sous Spark. Nous n'avons donc pas inclus notre revue (ni toutes les références des logiciels) dans ce document.

A l'automne 2017, nous avons essayé de tester les tout premiers logiciels bioinformatiques développés pour Spark : PASTASpark, SparkBWA et HAlign2, mais cela n'a pas été possible de les faire fonctionner correctement à cause de problèmes de code (ArrayOutOfBound, chemins hardcodés et code obscur difficile à corriger dans HAlign2), d'une complexité de l'installation et des dépendances (PASTASpark) et d'une configuration de Spark différente notamment sur les chemins (SparkBWA et PASTASpark).

## IV. Développement en Scala pour Spark

### IV.1. Environnement, tutoriels, outils et librairies

En janvier 2017, afin de faciliter la configuration de l'environnement pour ceux qui voulaient commencer à développer en Scala, un Docker (intégrant Eclipse/Scala) ainsi qu'un tutoriel pour soumettre des jobs sur le cluster ont été mis à disposition des membres de la liste (ludovic.legrand@inra.fr, https://lipm-git.toulouse.inra.fr/spark-ics/, [6]). Plus tard nous avons évalué et adopté l'IDE Intellij qui grâce à de nombreux plugins est très efficace pour développer en Scala/Spark (à l'exception des objets non-serializables au milieu d'un job qu'il ne détecte pas et qui engendrent un plantage lors de la génération du DAG d'exécution).

A l'automne 2017, deux QuickStarts ont été formalisés et présentés (axel.verdier@inra.fr) : la mise en œuvre de l'outil SBT qui est recommandé pour la compilation des projets Scala et la librairie ScalaTest pour les tests fonctionnels. Toujours dans l'idée de faciliter la prise en main du langage Scala, nous avons d'une part développé en Scala une petite librairie pour la gestion des paramètres, la gestion des flux (compressés) et des formats usuels en bioinformatique que nous utilisons depuis des années en Perl au LIPM (lipmutils ➔ scala-utils). Cet outil a été présenté lors d'une visioconférence (axel.verdier@inra.fr) ainsi qu'un tutoriel basé sur JNI pour faire du binding entre C/C++ et Scala (erika.sallet@inra.fr).

En 2018, un ensemble de méthodes permettant de calculer la grande majorité des indicateurs de classification binaires utilisés en Machine Learning (https://en.wikipedia.org/wiki/Confusion_matrix) a été intégré à la librairie scala-utils (axel.verdier@inra.fr). Une bibliothèque *plotter* permettant de visualiser des graphiques basiques (plots, heatmaps, bars, …) a également été développée ; utilisée conjointement avec scala-utils, elle permet de visualiser facilement les courbes statistiques de bases pour le Machine Learning (ROC, PR, TP/TN en fonction du score).

L'ensemble de ces outils et documentations est très utile pour se mettre à la programmation Scala sans avoir l'impression/la nécessité de repartir à zéro en termes d'outils/d'APIs. Le compte rendu de la formation Scala reporté en annexe fournit des sources d'information plus complètes sur le langage et les points importants à maîtriser car si les outils/APIs sont nécessaires, ils ne sont pas suffisants pour bien maîtriser un langage.

Les trois principaux verrous que nous avons pu identifier pour exploiter au mieux l'architecture dans nos codes sont les suivants :

- il est critique de bien maîtriser l'utilisation quasi exclusive de données non mutables (et leur structuration en DataFrame/DataSet) car c'est la connaissance du fait que la donnée ne sera pas modifiée qui rend possible le parallélisme automatique (Scala/Spark) ainsi que l'optimisation des tâches par le DAGScheduler de Spark (qui va réécrire/optimiser le code si l'état de la structure de données est figé).
- il faut éviter (autant que possible bien sûr) de faire des analyses qui concernent l'ensemble des données dans un état intermédiaire du processus d'analyse (exemple: compter combien il reste d'éléments après un filtre avant de faire autre chose). En effet chaque étape qui nécessite une consolidation des données va non seulement déclencher un échange de données entre les nœuds (et/ou le frontal) mais aussi en amont bloquer le processus d'optimisation (DAGScheduler) qui permet l'enchaînement en mémoire des traitements sur les partitions de données.
- il faut s'assurer que les calculs puissent s'effectuer dans l'enveloppe mémoire (de quelques Gb) allouée aux exécuteurs et cela demande des ajustements au niveau du code lorsque la quantité de données est importante. De façon classique on s'autorise à avoir des processus qui consomment beaucoup de mémoire et beaucoup de threads mais pour bien exploiter un cluster Spark l'objectif est de lancer de très nombreux calculs sur de multiples partitions afin de pouvoir enchaîner les différentes phases du traitement dans la mémoire de l'exécuteur en minimisant les échanges. Dans l'idéal que nous avons visé dans nos développements, un exécuteur exécute un thread et la totalité des nœuds de calcul sont actifs (cela permet aussi d'optimiser les accès disques aux données distribuées sur le cluster HDFS). Pour cela, l'enveloppe mémoire de chaque exécuteur doit être inférieure à la mémoire totale du nœud de calcul divisé par le nombre de threads/cores du nœud. Le programmeur doit garder à l'esprit cet objectif et doit rendre possible le partitionnement des calculs. Il est également possible d'allouer plusieurs cores par exécuteur mais on ne l'a pas évalué car cela complique énormément le debugging (les exécuteurs partageant la JVM du conteneur).

Les développements et documents évoqués dans ce document sont disponibles sur le site https://lipm-gitlab.toulouse.inra.fr/spark-ics (les personnes avec un email INRA peuvent créer un compte, en cas de problème, contacter ludovic.legrand@inra.fr et jerome.gouzy@inra.fr).

## IV.2. Réingénierie du logiciel d'annotation SpliceMachine en Scala/Spark (SVM)

Le logiciel EuGene, développé à l'INRA de Toulouse (http://eugene.toulouse.inra.fr/) pour prédire les modèles de gènes de génomes eucaryotes, nécessite l'intégration de résultats issus de nombreux outils et en particulier ceux d'un prédicteur de sites d'épissage (dinucléotides GT (donneurs) et AG (accepteurs) pour les sites canoniques).

Ce prédicteur doit scorer l'ensemble des sites donneurs et accepteurs potentiels du génome en fonction de leurs potentiels (sur la base du contexte génomique proche, typiquement 70nt en amont et en aval). Pendant de nombreuses années le logiciel SpliceMachine exploitant les SVM a été utilisé comme prédicteur de sites d'épissage (Degroeve *et al.* [5]). Malgré ses bonnes performances, il rendait l'utilisation du pipeline EuGene très compliquée pour plusieurs raisons: i) le logiciel est sous licence ii) il a des dépendances à SVM$^{light}$ iii) il n'est plus maintenu depuis longtemps et on devait utiliser de vielles versions des dépendances iv) l'entrainement des modèles était long (plusieurs semaines) et fastidieux même après avoir essayé de l'automatiser.

Ces différents problèmes ont conduit l'équipe EuGene à abandonner SpliceMachine pour le remplacer par des modèles WAM (Weight Array Model) plus faciles à entraîner mais moins performants. Le déficit en performance sur la prédiction *ab initio* des sites d'épissage est alors compensé par d'autres sources d'informations plus fiables (alignement des données RNAseq).

Les SVM sont intégrés à l'API de Machine Learning de Spark (spark.mllib [RDD] et spark.ml [DataFrame]) et nous avons décidé de nous familiariser avec le développement d'application Scala/Spark en redéveloppant entièrement le logiciel SpliceMachine.

Les performances sur des génomes de plantes de différentes tailles sont présentées ci-après mais on peut résumer les enseignements du projet en quelques points :

- à partir du moment où l'on a construit une matrice de variables (« features ») sous la forme d'un DataFrame, la partie entraînement et évaluation du modèle prend quelques dizaines de lignes de code Scala (grâce à spark.mllib et spark.ml).
- il n'y a plus de dépendances à un logiciel externe SVM$^{light}$
- contrairement au SpliceMachine originel qui passait par des étapes de sélection de variables (lourdes en calculs) nous entrainons les modèles avec la totalité des 22608 features (contexte génomique de 70nt en amont et en aval du site).
- désormais, l'annotation des sites d'un nouveau génome est faite à partir de l'exécution d'un seul programme qui intègre l'entrainement des modèles, leur évaluation et leur utilisation pour annoter tous les sites d'un génome.
- les entrées du programme se composent de seulement deux fichiers: un génome et le résultat du mapping d'un assemblage de transcriptome (qui doit être calculé pour une autre étape de l'annotation avec EuGene) et en sortie on dispose de deux fichiers GFF3 avec les scores de tous les accepteurs et donneurs.
- si le cluster Spark de 252 cœurs est dédié, on peut annoter les sites d'un génome de 100Mb en quelques dizaines de minutes, un génome de 500Mb en 3h et un génome de 3Gb en une douzaine d'heures.
- les performances intrinsèques de la prédiction *ab initio* des sites d'épissage sont bien meilleures que ce que l'on avait avec l'ancien SpliceMachine ou avec les WAM multi-espèces. Nous n'avons pas les valeurs des aires sous la courbe des anciennes analyses mais on peut constater que le point de la courbe Spécificité vs Sensibilité où les valeurs de sensibilité et la spécificité sont équivalentes est désormais autour de 0.95 alors qu'il était à 0.80 pour l'ancien SpliceMachine (uniquement pour les meilleurs modèles) et autour de 0.65 pour les WAM multi espèces plantes. L'architecture de calcul intégrée et optimisée que l'on utilise ici de façon très brutale (beaucoup plus de données pour entrainer et beaucoup plus de features simultanément) permet bien au SVM de mieux caractériser les sites en un temps très raisonnable.
- La totalité du code spécifique (des librairies génériques ont été externalisées) représente à ce jour 1800 lignes de code Scala.
- Le développement et les tests ont duré moins de 6 mois (en équivalent temps plein d'un développeur jeune mais néanmoins expérimenté axel.verdier@inra.fr). Cela semble très raisonnable sachant que c'était un premier projet sur une nouvelle technologie.
- Il reste du travail pour intégrer les résultats de SpliceMachineSpark au pipeline d'annotation EuGene : au niveau de l'intégration des résultats SpliceMachineSpark pour optimiser la remise à l'échelle des scores mais aussi au niveau du pipeline (qui contrôle l'exécution des analyses) pour mettre en place un environnement dans lequel on pourra basculer du cluster SGE/Slurm à un cluster Spark en fonction des analyses. Ces travaux seront traités dans le cadre du projet EuGene.

Le tableau ci-dessous résume les processus de génération des modèles et d'annotation des sites d'épissages de trois espèces de plantes avec une taille de génome variable.

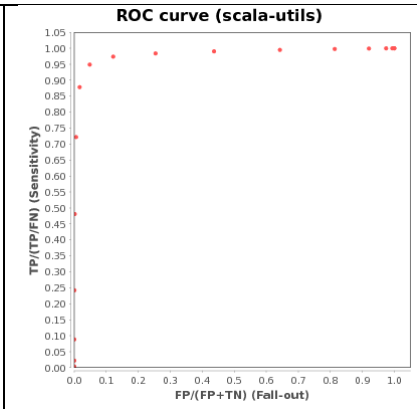
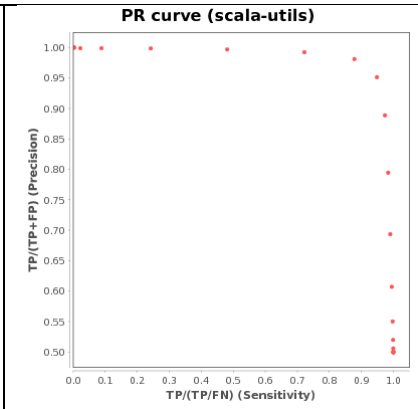
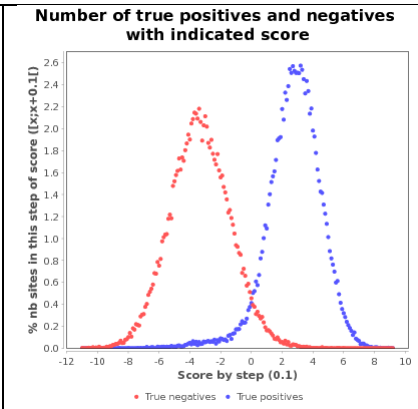
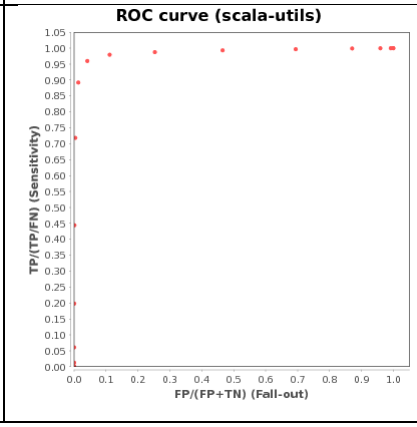
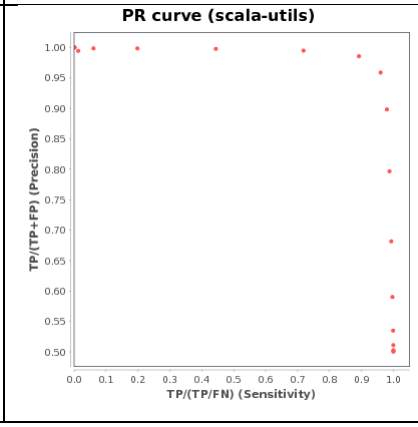
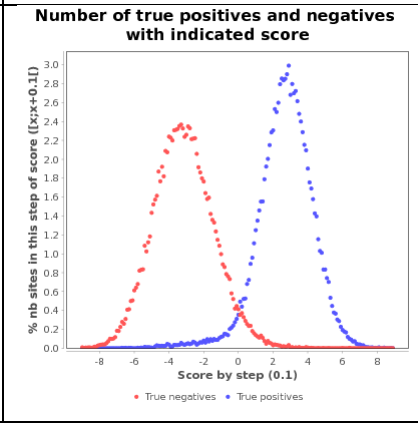
| | Construction et évaluation des SVM | | | | | Annotation (scoring) des sites du génome | | | Durée totale (elapsed) |
|---|---|---|---|---|---|---|---|---|---|
| Espèce | Sites<br>ACC: accepteurs<br>DON: donneurs | Nombre d'exemples positifs disponibles | Nombre d'exemples négatifs disponibles | Temps d'extraction (s) | Temps de construction & d'évaluation du modèle (s) | Nombre de sites à scorer | Temps (s) | Taille fichier (MB) | |
| *Arabidopsis thaliana* (At) (125Mb) | ACC | 76 175 | 76 102 | 358 | 447 | 14 209 102 | 1 098 | 1 036 | 3 349s |
| | DON | 75 521 | 75 305 | | 442 | 12 495 795 | 1 004 | 874 | 0,93h |
| *Rosa chinensis (Rose)* (520Mb) | ACC | 108 944 | 109 470 | 2 458(*) | 584 | 62 904 582 | 3 798 | 4 999 | 10 680s |
| | DON | 105 898 | 105 968 | | 558 | 53 212 472 | 3 282 | 4 070 | 2,97h |
| *Helianthus annuus (Tournesol)* (3Gb) | ACC | 105 929 | 105 377 | 992 | 563 | 319 335 873 | 18 316 | 25 401 | 39 339s |
| | DON | 104 330 | 104 165 | | 609 | 337 727 714 | 18 859 | 25 855 | 10,9h |

(*) L'extraction inclut la suppression de la redondance et dépend donc de la redondance dans les données transcriptomiques qui ont été mappées (pour la rose nous avons utilisé plusieurs assemblages, et un seul pour At et le tournesol).

Nous avons vérifié que le ratio de 1:1 pour le nombre d'exemples positifs et d'exemples négatifs était le plus pertinent pour entrainer les SVM. 60% des exemples positifs et négatifs sont utilisés pour entraîner et 40% pour tester les modèles. L'annotation des sites sur les trois génomes a été effectuée avec 200 exécuteurs et 5Gb max de mémoire par exécuteur. La mémoire maximale utilisée sur le frontal a été de 21G car certaines étapes sont exécutées sur le frontal (préparation des DataFrames, consolidation des résultats, etc.). Des optimisations postérieures à ces analyses ont permis de limiter significativement l'utilisation mémoire sur le frontal mais il semble nécessaire d'avoir une bonne configuration en termes de ressources de calcul et de mémoire sur la machine qui soumet les jobs.

Les indicateurs caractérisant la performance des modèles sont présentés dans les tableaux ci-dessous : les aires sous les courbes ROC et PR pour les trois espèces, ainsi que les courbes ROC, PR et la courbe TP/TN (True positives/True Negatives) vs Score seulement pour le tournesol. Les sorties présentées ci-dessous (et d'autres) nécessaires à l'évaluation du modèle sont générées par défaut par le programme et formatées dans un document HTML.

| Espèces | Sites | AreaUnderROC | AreaUnderPR |
|---|---|---|---|
| *Arabidopsis thaliana* | ACC | 0.987 | 0.988 |
| | DON | 0.989 | 0.989 |
| *Rose* | ACC | 0.951 | 0.963 |
| | DON | 0.950 | 0.963 |
| *Helianthus annuus* | ACC | 0.984 | 0.986 |
| | DON | 0.988 | 0.989 |

| | | | |
|---|---|---|---|
| ACC |  |  |  |
| DON |  |  |  |

*Indicateurs pour les des modèles d'Helianthus annuus (Tournesol, 3Gb)*

S'il est connu depuis longtemps que les SVM sont bien adaptés à la caractérisation des sites d'épissage, les performances obtenues grâce à la possibilité de construire des modèles via Spark sans se préoccuper de la quantité de données et de features rendent encore plus pertinent ce type de caractérisation.

Le programme est actuellement en cours d'extension pour la prédiction des sites de début et de fin de traduction pour lesquels les jeux d'entraînements seront moins importants.

## IV.3. Autres projets de développement Scala/Spark en cours

Le premier projet vise à redévelopper sous Spark le pipeline développé au LIPM pour la prédiction des précurseurs de miRNA sur la base de données de séquençage sRNA-seq (erika.sallet@inra.fr). Utilisé depuis 10 ans dans plusieurs publications ce pipeline combine des étapes de mapping, de calcul de structures secondaires (très couteux en calcul) et de nombreux filtres. Il correspond à plusieurs scripts Perl avec de nombreuses dépendances à des programmes compilés. Le portage sous Spark semble particulièrement adapté car le mapping est exact (sans mismatch donc revient à de la recherche de mots) et chaque locus peut être analysé de façon indépendante avec une empreinte mémoire limitée. Nous avons validé le binding Scala avec du code C (via JNI) afin de pouvoir exploiter les librairies standards du domaine pour le calcul des structures secondaires mais nous n'avons pas encore eu le temps d'aller plus loin. Ce projet sera poursuivi dès que possible car il est désormais indispensable de faire l'annotation des précurseurs de miRNA lorsque l'on annote un génome et ce besoin n'est pas couvert par le pipeline EuGene.

Le deuxième projet vise à développer un programme automatisant complètement la détection de contigs contaminants dans des assemblages de génomes ou de transcriptomes (jerome.gouzy@inra.fr). Ce problème de contamination est très fréquent lorsque l'on travaille sur des organismes pathogènes ou symbiotiques. On cherche à utiliser les méthodes de Machine Learning disponibles dans la librairie spark.ml (ici les Random Forests) pour caractériser les séquences de l'organisme cible et celles du contaminant sur la base de features intrinsèques aux séquences (ex: GC%), spécifiques à l'expérimentation (ex: profondeur de séquençage différentes) ou externes (ex: similarités avec des protéines d'origine connue). Cette partie semble donner de bons résultats sur des échantillons fortement contaminés car c'est un cas où il est relativement aisé de définir automatiquement des critères (sur un nombre significatif de séquences) pour catégoriser les séquences en "cible" ou "contaminant".

Mais nous n'avons pas encore eu le temps d'évaluer et de mettre au point sur des échantillons faiblement ou pas du tout contaminés (il faudra utiliser autre chose que les Random Forest). De plus, nous n'avons pas eu le temps d'intégrer tous les calculs lourds pour la définition des features sous Spark. Nous envisageons de le faire soit via l'utilisation de l'API C de minimap2 soit via l'exécution de commandes pipes qui sont difficilement contrôlable par l'exécuteur Spark (en cas d'échec du à un dépassement mémoire le job est relancé indéfiniment) et qui ne sont pas recommandées sous Spark. Néanmoins c'est une solution à creuser car elle limiterait les coûts de développement.

# V. Actions d'acquisition et de diffusion des connaissances

Une liste de diffusion a été créée en janvier 2016 spark-ics@listes.inra.fr (81 messages) puis migrée sur Renater en décembre 2017 spark-ics@groupes.renater.fr (33 messages). 27 inscrits au 30 août 2018.

En avril 2016 nicolas.lapalu@inra.fr a diffusé le compte rendu de la journée « Spark du réseau LoOPS » organisée par le LAL d'Orsay qui nous a permis d'avoir un premier retour d'expérience sur Spark d'autres communautés de recherche.

Un premier « spike » technologique de 2 jours a été réalisé en juin 2016 au CBGP (organisé par alexandre.dehne-garcia@inra.fr) afin d'évaluer l'installation de Spark et Hadoop avant la réception du matériel « spark-ics ». Le deuxième « spike » de 1 jour en décembre 2016, en marge d'une formation BBRIC, pour apprendre à écrire des programmes simples en Scala sur des problèmes basiques de bioinformatique (lecture Fasta, conversion format)

Un document technique sur l'installation de Spark sur notre cluster (5pg) a été diffusé au sein de la liste de diffusion en septembre 2016.

La première réunion a eu lieu le 21 mars 2016 (en satellite des journées bioinformatique INRA) pour discuter essentiellement administration système. Les autres réunions se sont déroulées par visioconférence. Ainsi, entre octobre 2017 et avril 2018 ont été organisées 7 visioconférences. Sur la base de présentations d'une quinzaine de pages, l'objectif était de formaliser et présenter, l'utilisation d'une technologie ou d'un outil liés à Spark, une API développée et mise à disposition, ou des résultats de benchmarks. Les comptes rendus des formations Scala/Spark organisées par le LAL ont également été présentés. Ces deux documents rapportés en annexe sont des points d'entrée intéressants aussi bien pour commencer en programmation fonctionnelle Scala que pour la compréhension de Spark.

Le projet spark-ics s'est entrelacé avec deux programmes « investissement d'avenir ». D'une part le projet SUNRISE (génomique/génétique du tournesol, via l'équipe informatique du LIPM) et avec l'IFB via Christophe Caron (par ailleurs « garant » spark-ics pour la DTN) qui a demandé l'accès au cluster spark-ics pour explorer la technologie dans le cadre d'un de ses projets dans le cadre de l'IFB. A partir d'octobre 2017, le groupe a ainsi bénéficié des connaissances acquises sur le développement Scala sur Spark par deux ingénieurs en CDD (SUNRISE 24 mois, en cours et IFB 8 mois). Le projet a également pu bénéficier du soutien financier de SUNRISE pour doubler la taille du cluster, nécessaire aussi bien pour les tests de passage à l'échelle que pour travailler sur des jeux de données de plusieurs Tb que l'on doit analyser pour de gros génomes (ex tournesol)

En mars et avril 2018, nathalie.vialaneix@inra.fr nous a fait une introduction (2 * 1h) aux méthodes du Machine Learning (classification, Random Forests, SVM, réseaux de neurones) que l'on a pu exploiter dans la foulée. En effet, s'il est très facile de construire des modèles Random Forest ou SVM dans Scala/Spark, il est très important de bien comprendre les aspects les plus importants des méthodes si l'on veut qu'ils soient performants (en particulier leurs hyper-paramètres, les tailles des jeux d'entrainements et la balance entre nombres de vrais positifs et vrais négatifs dans les jeux d'entrainements).

Communication externe

- La partie administration système a été présentée en juin 2016 au réseau d'informaticiens toulousains CAPITOUL (ludovic.legrand@inra.fr).
- En avril 2018, la partie architecture de spark-ics a été présentée à Paris lors d'une réunion Ingenum/CTIG puis aux informaticiens de l'unité GenPhyse (génétique animale) à Toulouse (ludovic.legrand@inra.fr).
- Le poster spark-ics ci-dessous a été présenté en juillet à JOBIM 2018 pour résumer les résultats du groupe de travail.

# Spark-ICS

## Exploration de l'application de l'architecture Apache Spark à la bioinformatique

Axel Verdier, Ludovic Legrand, Xavier Garnier, Erika Sallet, Alexandre Dehne-Garcia, Nicolas Lapalu, Martial Briand, Franck Dorkeld, Bernhard Gschloessl, Corinne Rancurel, Martine Da Rocha, Sébastien Carrère, Céline Noirot, Olivier Filangi, Jérôme Gouzy

Le projet SPARK-ICS vise à évaluer une solution technologique du « Big Data » pour résoudre des classes de problèmes bioinformatiques qui atteignent la limite des architectures actuelles.

La quantité et la diversité des données ne font que croître grâce à l'amélioration des débits et la baisse des coûts induisant des problèmes de passage à l'échelle et augmentant la complexité des analyses. En outre, une meilleure compréhension des jeux de données complexes nécessite de plus en plus l'utilisation du Machine Learning pour capter l'information la plus pertinente.

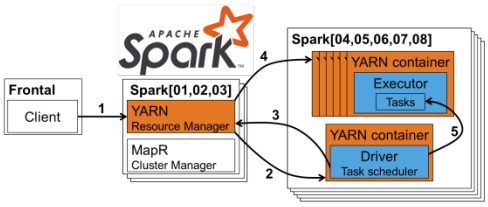
L'architecture Apache Spark intègre une grande variété d'outils et de méthodes du Machine Learning dans un environnement de calcul pouvant être instancié sur un poste de travail, un cluster de calcul ou dans le cloud. Depuis deux ans notre groupe de travail a évalué cette architecture pour répondre à des questions de bioinformatique.

## Objectifs

- Evaluer Spark dans le cadre de la bioinformatique
  - Evaluer des outils existants
  - Développer de nouveaux outils
- Accélérer et améliorer l'analyse des données
- Acquérir des compétences en « Big Data » avec Spark
  - Développement en Scala
  - Mise en place d'un cluster
  - Machine Learning

## Fonctionnement de Spark

1. Soumission d'un job par le client avec X exécuteurs. YARN valide ou met en attente le job en fonction des ressources disponibles
2. Instanciation d'un conteneur et d'un driver Spark
3. Construction d'un graphe de tâches et demande de ressources à YARN
4. YARN instancie des conteneurs et le driver démarre les exécuteurs dans les conteneurs
5. Le driver répartit les tâches sur les exécuteurs et gère les erreurs

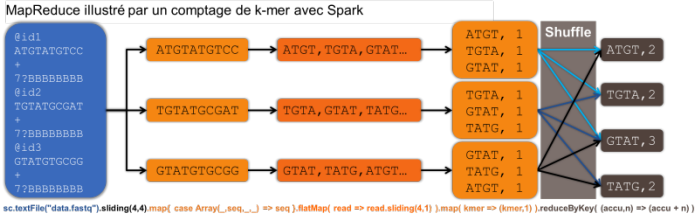



## Cluster Spark

- Cluster de 8 nœuds physiques
  - 252 cœurs disponibles
  - 1,4To de RAM utilisables
  - 34To de disque brut
- 4 nœuds avec 386Go de RAM et CPU à 2,4GHz
- 4 nœuds avec 256Go de RAM et CPU à 2,2GHz
- Un serveur frontal pour soumettre les jobs

*Note: le même cluster est aussi accessible via le scheduler SGE*

MapReduce illustré par un comptage de k-mer avec Spark

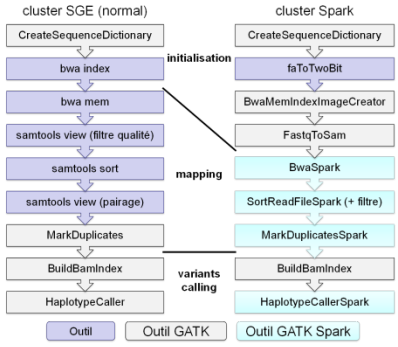



```
sc.textFile("data.fastq").sliding(4,4).map( case Array(_,seq,_,_) => seq ).flatMap( read => read.sliding(4,1) ).map( kmer => (kmer,1) ).reduceByKey( (accu,n) => (accu + n) )
```

## MapReduce[1] avec Spark

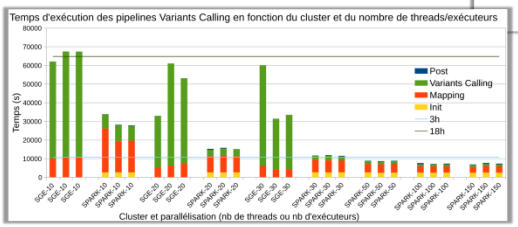
- Jusqu'à **100x plus performant que MapReduce avec Hadoop**
  - Utilisation de la **RAM** au lieu des disques entre les tâches
  - Réorganisation et optimisation des tâches avant l'exécution
  - Une API plus riche et plus généraliste
- Distribution et reprise sur erreur gérées par Spark
- Le **Shuffle** qui implique du trafic réseau et de l'écriture sur disque, reste une étape critique pour les performances

## Benchmarking

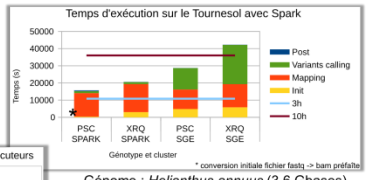



### Pipeline GATK4[2]

- Outil pour la détection de variants (v4.0.1.2)
- Utilisation des outils **HaplotypeCaller** et **HaplotypeCallerSpark**
- Les versions non-Spark et Spark ont un algorithme différent
- Benchmark SGE et Spark sur les mêmes serveurs




Génome : *Plasmopara halstedii* (75 Mbases), fastq.gz : 2x12Go, 3 répétitions



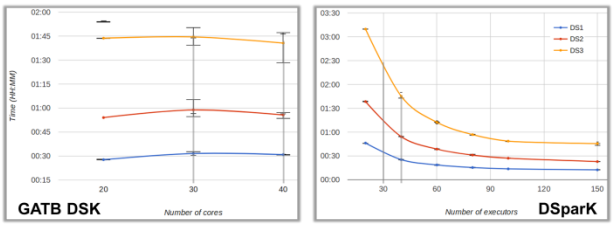

Génome : *Helianthus annuus* (3,6 Gbases)
2 génotypes : PSC et XRQ
fastq.gz : 2x12Go, sans répétition

- La version non-Spark est peu parallélisée
- La version Spark est encore en bêta
- **Gain de temps** conséquent avec la version Spark

## Réingénierie en Scala/Spark d'outils bioinformatiques

### DSparK

- Porter un outil C++ pour le comptage de k-mer
  - DSK fait partie de **GATB-Core**[3]
- Jeux de données (format fasta)
  - DS1 : 800 millions de reads
  - DS2 : 1,6 milliards de reads
  - DS3 : 3,2 milliards de reads




- Spark est moins performant que le C++ à ressources équivalentes
- Spark permet un **meilleur passage à l'échelle**
- Développement en Spark plus rapide

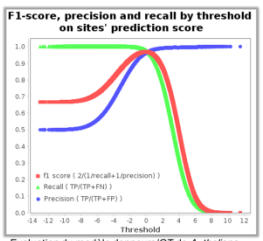
### SpliceMachine[4]

- **Objectif** : Détection des sites d'épissage par utilisation de Linear Support Vector Machine
- Développement de l'outil SpliceMachine sous Spark

- **Entraînement en quelques heures** contre plusieurs jours pour l'ancienne version
- Spark permet une **implémentation simple** du **Machine Learning**, quelques dizaines de lignes Scala pour entrainer le modèle

### Entraînement du modèle

- Extraction des features dans une fenêtre de +/- 70nt autour des sites donneurs (GT) ou accepteurs (AG)
- **22608 features** en 3 classes :
  - 11472 **Positional Information** : composition en 1 à 3-mers aux positions
  - 10752 **Compositional Information** : présence/absence des 4 à 6-mers
  - 384 **Coding Potential** : Présence des codons pour chacun des 3 cadres de lecture.
- Construction d'un jeu de 87454 sites positifs et 87454 sites négatifs à partir de données de transcrits *A. thaliana*




Evaluation du modèle donneurs/GT de *A. thaliana*
Area Under ROC = 0,993; Area Under PR = 0,991

[1] Zaharia, M. et al., 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI. https://cs.stanford.edu/~matei/papers/2012/nsdi_spark.pdf
[2] McKenna, A. et al., 2010. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. GENOME RESEARCH 20:1297-303
[3] Genome Analysis Toolbox with de-Bruijn graph (GATB). https://gatb.inria.fr/
[4] Degroeve, S. et al., 2005. SpliceMachine: Predicting splice sites from high-dimensional local context representations. Bioinformatics, 21(8), 1332–1338.

# VI. Bilan

Par rapport aux objectifs initiaux, nous n'avons pas encore pu aller jusqu'à l'utilisation de pipelines basés sur Spark en production. D'une part, parce que nous n'avons pas fini les développements mais aussi car nous testons sur de gros jeux de données pour trouver les limites. D'autre part, vu la dimension réduite de notre cluster il s'avère difficile de partager la même architecture Spark pour du développement et pour de la production.

Conformément aux attentes en 2015, nous avons pu vérifier que Spark offre bien la possibilité de développer des applications où la distribution des calculs est optimisée par une architecture très répandue dans l'industrie du « Big Data » (si l'on développe en suivant certaines règles comme explicité précédemment). Nous avons commencé à le vérifier en testant des outils développés pour Spark par d'autres équipes de bioinformatiques (Sparkhit pour la métagénomique et GATK pour l'analyse de variants). Puis en développant nos propres outils pour l'annotation des génomes où l'utilisation des API déjà implémentées sur Spark permettent à tout moment du processus d'analyse, et en quelques lignes, d'appliquer des méthodes de classification ou d'apprendre, tester et utiliser des modèles. Ces caractéristiques font de Spark une solution qui pourrait faciliter le développement mais aussi faciliter la maintenance sur le long terme de protocoles d'analyses. Si on peut évidemment implémenter les mêmes protocoles en dehors de Spark (c'est ce que l'on fait depuis 20 ans), c'est au prix de nombreuses dépendances logicielles difficilement maintenables et d'adaptations récurrentes pour accéder aux ressources de calculs (dont l'utilisation en bioinformatique est rarement optimale via SGE, SLURM, etc).

Concernant la diffusion des connaissances acquises auprès des communautés informatiques et bioinformatiques de l'INRA, les actions de communication passées (et futures) ainsi que le présent rapport technique correspondent au livrable.

Ci-après, quelques éléments d'analyse au-delà des livrables annoncés lors du dépôt du projet exploratoire.

Même si nous n'avons pas poussé très loin la gestion automatique des priorités entre les deux, nous avons pu valider qu'un même cluster physique de configuration classique pouvait être exploité simultanément pour SGE et pour Spark, annulant ainsi le risque sur l'investissement matériel.

Dans le projet, nous avons choisi Scala comme langage de développement car c'est le langage natif de Spark et qu'il est compatible Java. C'est un langage fonctionnel qui nécessite un apprentissage qui prend du temps pour ceux qui n'ont pas l'habitude. Ce coût d'apprentissage semble pertinent car changer de paradigme pour concevoir ses structures de données permet vraiment de faciliter la distribution (automatique) des calculs sur Spark. Des APIs pour les langages très utilisés en bioinformatique comme Python et R sont disponibles et celle pour Python semble même très utilisée dans la communauté Spark et à jour. Si nous ne les avons pas testées pendant cette phase du projet elles pourraient faciliter l'entrée dans l'écosystème Spark à de nombreux bioinformaticiens. Mais pour concevoir des applications vraiment optimisées il nous semble important de mentionner qu'il faudra prendre le temps de bien comprendre le fonctionnement de Spark.

Les actions d'animation ont intéressé une douzaine de personnes pour les réunions techniques et plus encore pour les deux sessions plus méthodologiques de formation au « Machine Learning » (par nathalie.vialaneix@inra.fr). Cette activité de veille et de partage de connaissances technologiques et méthodologiques aura donc été bénéfique pour l'amélioration de la culture informatique. La plupart des réunions ont donné lieu à des comptes rendus et au partage des documents sur la liste de diffusion.

Concernant les activités liées à l'évaluation et au développement, la quasi-totalité des résultats a été produite par l'équipe bioinformatique du LIPM. Ce n'est pas très étonnant car, au-delà du LIPM, les personnes qui en 2015 s'étaient montrées intéressées par le projet l'étaient plus par curiosité que par volonté de s'investir massivement dans une nouvelle technologie. Nous avons pu vérifier que le coût en temps d'acquisition des compétences est loin d'être négligeable (et comme l'architecture est relativement jeune, les API critiques comme spark.ml évoluent encore) et qu'il s'avère très difficile d'arbitrer positivement l'acquisition de nouvelles compétences sur un projet exploratoire au détriment des projets urgents des communautés servies. A posteriori, il semble évident que sans l'apport de personnels en CDD dédiés au développement Scala/Spark (Axel Verdier aura joué un rôle moteur très important) nous n'aurions pas pu faire une évaluation correcte et réaliste sur tous les aspects de la technologie.

# VII. Perspectives

On évaluera l'instanciation et l'utilisation d'un cluster Spark dans le cloud Amazon (calcul & stockage) afin de valider la portabilité et évaluer le passage à l'échelle (scalabilité) des applications Spark que nous développons sur les trois types de ressources de calcul: poste de travail, cluster Spark, cloud. Disposer d'un même code exécutable sur tous les types d'environnement de calculs sera un atout pour augmenter la durée de vie des applications d'analyses (il ne reste plus qu'à trouver les financements pour tester, estimation à ~8k€)

Nous allons continuer à développer et à consolider nos compétences en développement Scala pour exploiter au mieux les architectures Spark. Une formation en développement Scala sera organisée fin 2018/début 2019 par l'ISA à Sophia Antipolis (contact: corinne.rancurel@inra.fr) avec un programme et une durée très similaire a celui de la formation organisée par le LAL à Orsay en janvier 2018 qui a vraiment permis de consolider les compétences de 4 membres du groupe de travail.

Dès que possible nous évaluerons l'API Python ainsi que GraphX pour la manipulation de graphes.

Nous allons également tester ADAM (Big Data Genomics : http://bdgenomics.org/), une bibliothèque d'analyses génomiques pour Spark.

Dans l'année qui vient, nous comptons finaliser les applications bioinformatiques Scala/Spark en cours de développement pour l'analyse de génomes (prédiction de sites d'épissages, détection de contaminants, annotation des précurseurs de miRNA).

Nous évaluerons l'intégration Spark/TensorFlow afin d'évaluer l'utilisation des méthodes de Deep Learning, utilisées actuellement pour l'analyse d'images, pour résoudre des problèmes d'annotation des génomes.

# VIII. Calendrier effectif du projet

## VIII.1. Principaux évènements

- Juillet 2015 – Réponse à l'appel à projet de la DTN INRA
- Mai 2016 - Arrivé du financement DTN INRA au LIPM
- Juin 2016 - « Install party » au CBGP
- Aout 2016 - Installation du cluster de 4 serveurs et ouverture aux participants du projet
- Septembre 2017 - Réinstallation complète et ajout de 4 serveurs
- Octobre 2017 - Recrutement au LIPM d'un CDD sur le PIA Sunrise (Axel Verdier)
- Janvier 2018 - Ecole programmation fonctionnelle Scala et Spark au LAL à Orsay (2x3j) avec rédaction et présentation des comptes rendus
- Mars/avril 2018: 2 sessions de formation sur les méthodes de Classification/Machine Learning (Arbres de décisions, Random Forest, SVM) par Nathalie Vialanex.
- Juillet 2018 – Poster « Groupe de travail » à JOBIM 2018 « Spark-ICS: Exploration de l'application de l'architecture Apache Spark à la bioinformatique »
- Août 2018 – Rédaction du document de synthèse du projet exploratoire spark-ics

## VIII.2. Principales phases du projet

| | |
|---|---|
| mai 2016 – août 2016 | Achat matériel et administration système du cluster spark (4 nœuds, 192 cœurs) |
| sept 2016 – juin 2017 | « Standby » (focus sur projets génomes importants) |
| juil 2017 – sept 2017 | Extension matérielle et réinstallation complète du cluster (➔ 8 nœuds, 416 cœurs) |
| oct 2017 – nov 2017 | Mise en place d'outils et partage de connaissances pour le développement logiciel en Scala sur Spark (SBT, scala-utils, binding Scala/C, etc) |

| | |
|---|---|
| nov 2017 - avril 2018 | Evaluation du logiciel GATK 4 pour la détection de polymorphismes. Benchmarking version GATK/Spark et GATK/SGE. |
| avril 2018 – août 2018 | Initiation de l'implémentation d'un logiciel de détection de contaminants dans les assemblages (Scala/Spark/Random Forest)<br>Evaluation du logiciel Sparkhit en métagénomique<br>Réimplémentation sous Spark du logiciel d'annotation SpliceMachine (SVM) |

# IX. Financements

Les évaluations et résultats présentés dans ce document ont été produits à partir de ressources financées par différentes entités. Les principaux postes de dépenses sur la période (printemps 2016-août 2018) sont reportés ci-dessous. Il faut garder à l'esprit que si l'analyse *a posteriori* des coûts « du projet » est intéressante, elle reste imparfaite pour déterminer le coût du projet car: i) les coûts permanents sont basés sur des estimations ii) certains postes marginaux ont été occultés (ex: implication permanent < 1 mois, VM frontal, etc.) iii) l'architecture informatique n'a pas été dédiée au projet spark-ics pendant toute la période (les serveurs de calculs ont été aussi utilisés en mode classique SGE et le cluster Spark a été mis à disposition d'un projet de l'IFB entre octobre 2017 et mai2018) iv) les ressources (matérielles/documentaires/logicielles) continueront à être utilisées par et pour d'autres projets (ex: PIA SUNRISE)

| | | |
|---|---|---|
| **Matériel** | | |
| 2016: 1ère tranche<br>DELL C6320, 2U, 1400W, 4 nœuds (24 cœurs/48 HT 2,2Ghz/256Gb/6Tb) | INRA DTN<br>BBRIC (INRA SPE) | 21k€<br>4k€ |
| 2017: 2e tranche<br>DELL C6320, 2U, 1600W, 4 nœuds (28 cœurs/56HT 2,4Ghz/384Gb/6Tb) | LIPM (PIA SUNRISE) | 25,5k€ |
| **Hébergement Data Center INRA Toulouse** | | |
| C6320-2016-1400W juillet 2016- aout 2018 | LIPM | 2,8k€ |
| C6320-2017-1600W aout 2017-aout 2018 | LIPM | 1,6k€ |
| **Personnels** | | |
| IE CDD octobre 2017-août 2018 (AV)<br>(*) | LIPM (PIA SUNRISE) | 27k€ |
| Permanents (2016-août 2018)<br>5 mois IE (4 LL, 1 ES), 3 mois IR (JG) | INRA (BBRIC) | ~32k€<br>(**) |
| **Formations** | | |
| Pas de frais pédagogiques pour la formation Spark/Scala organisée en 2018 par le LAL a Orsay (2*3j) mais des missions Toulouse-Paris (LL, AV) et Rennes-Paris (OF) | LIPM, IGEPP | ~3,5k€ |
| **Total** | | ~117,4k€ |

(*) L'architecture a été mise à disposition de l'IFB et l'ingénieur associé aux réunions discussions du groupe mais ses travaux ont été gouvernés dans le cadre de l'IFB et donc ses résultats informatiques seront reportés à l'IFB par les coordinateurs des travaux IFB (IRISA/IGEPP). (**) Chargés/approximés mais sans environnement projet car déjà compté en grande partie dans les autres postes.

# X. Matrice des contributions

… aux travaux et actions reportés dans ce document pendant la période: janvier 2016 à août 2018

| | Labo<br>CATI | Adminis-tration Système | Dévelop-pement d'API et d'outils | Présenta-tions<br>(intra & externe) | Participa-tion aux réunions/visio | Tests formalisés d'outils bioinfo | Rédaction de comptes rendus et partage de documents |
|---|---|---|---|---|---|---|---|
| Axel Verdier | LIPM | | *** | *** | *** | *** | *** |
| Ludovic Legrand | LIPM<br>BBRIC | *** | * | *** | *** | ** | ** |
| Xavier Garnier | IRISA<br>IFB | | Travaux IFB | * | *** | Travaux IFB | * |
| Erika Sallet | LIPM<br>BBRIC | | * | * | ** | | * |
| Alexandre Dehne-Garcia | CBGP<br>BBRIC | * | | | ** | | |
| Nicolas Lapalu | BIOGER<br>BBRIC | | | | ** | | * |
| Martial Briand | IRHS<br>BBRIC | | | | * | | |
| Franck Dorkeld | CBGP<br>BBRIC | | | | ** | | |
| Bernhard Gschloessl | CBGP<br>BBRIC | | | | ** | | |
| Corinne Rancurel | ISA<br>BBRIC | | | | ** | | |
| Martine Da Rocha | ISA<br>BBRIC | | | | ** | | |
| Sébastien Carrère | LIPM<br>BBRIC | | | | ** | | |
| Ludovic Cottret | LIPM<br>BBRIC | | | | * | | |
| Céline Noirot | MIAT<br>Bios4Biol | | | | ** | | |
| Olivier Filangi | IGEPP<br>BBRIC | | | | *** | | * |
| Nathalie Vialaneix | MIAT | | | *** | * | | |
| Jérôme Gouzy | LIPM<br>BBRIC | * | * | * | *** | * | ** |

## XI. Bibliographie et URLs

La revue récente Guo *et al.* propose le catalogue des logiciels portés sur Spark, nous n'avons donc pas reporté notre propre inventaire. Si vous devez lire deux papiers en priorité ce sont coté informatique le papier fondateur sur des RDD (Zaharia, M. *et al.*) et pour la bioinformatique la publication sur Sparkhit (Huang *et al*.).


1. Zaharia, M. *et al.* Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

2. Heldenbrand, J. R. *et al.* Performance benchmarking of GATK3.8 and GATK4. *bioRxiv* (2018).

3. Huang, L., Krüger, J. & Sczyrba, A. Analyzing large scale genomic data on the cloud with Sparkhit. *Bioinformatics* (2017). doi:10.1093/bioinformatics/btx808

4. Guo, R., Zhao, Y., Zou, Q., Fang, X. & Peng, S. Bioinformatics applications on Apache Spark. *Gigascience* (2018). doi:10.1093/gigascience/giy098

5. Degroeve, S., Saeys, Y., De Baets, B., Rouzé, P. & Van de Peer, Y. SpliceMachine: Predicting splice sites from high-dimensional local context representations. *Bioinformatics* **21,** 1332–1338 (2005).

6. https://lipm-gitlab.toulouse.inra.fr/spark-ics (les personnes avec un email INRA peuvent créer un compte, en cas de problème, contacter ludovic.legrand@inra.fr et jerome.gouzy@inra.fr)

7. http://www.nathalievialaneix.eu/seminars2018.html (A short introduction to statistical learning)


# XII. Annexes

# XII.1 Compte rendu de la formation Spark

janvier 2018

Axel Verdier[1], Ludovic Legrand[1], Olivier Filangi[2] et Xavier Garnier[3]

[1]UMR INRA Laboratoire des Interactions Plantes-Microorganismes
[2]UMR INRA Institut de Génétique, Environnement et Protection des Plantes
[3]UMR Institut de Recherche en Informatique et Système Aléatoire

Ce compte rendu synthétise les 3 jours de la formation « Programmation Fonctionnelle et Spark » organisée par le Laboratoire de l'Accélérateur Linéaire à Orsay en janvier 2018. Cette formation Spark fait suite à la formation Scala de la même école.

L'objectif de ce document est de fournir les informations clés pour l'utilisation de Spark. Une première partie présente le fonctionnement interne de Spark. Une seconde partie explique les bonnes pratiques et les outils utiles. La dernière partie présente des spécificités de Spark.

# Table des matières

# Spark

## Écosystème

### Apache Umbrella

Le projet Apache Umbrella est constitué des sous-projets hive, hdfs, hbase, yarn et hadoop mapreduce.

### Hadoop

#### *HDFS*

Hadoop utilise un système de fichiers HDFS pour le stockage Il s'agit d'un système de fichiers distribué. C'est la pierre angulaire du Big Data avec MapReduce. Le concept est d'opérer un calcul au plus proche des données.

Un block HDFS est une *partition*. Les partitions sont répliquées. Un block HDFS a une taille de 128 MB (ext4 a des blocks de 4KiB soit 0.004MB). Cela permet d'accéder plus rapidement aux données de gros fichiers, au détriment de la place prise par un petit fichier. Cela **favorise l'utilisation de quelques gros fichiers** à la place d'une multitude de petits fichiers.

HDFS n'est pas POSIX.

HDFS est constitué de 2 types de nœuds :

- le `namenode` fait le registre, c'est le notaire du cluster. Il sait où sont les blocks.
- le `datanode`, il stocke les données

#### *Hive*

Hive est un système de requête SQL sur les données dans HDFS qui est intégré dans Hadoop. Il permet de retranscrire les requêtes SQL en MapReduce et jobs Spark.

#### *PIG*

Pig est une plateforme de haut niveau pour la création de programme MapReduce utilisé avec Hadoop. Le langage de cette plateforme, appelé le Pig Latin3, s'abstrait du langage de programmation Java MapReduce et se place à un niveau d'abstraction supérieur, similaire à celle de SQL pour systèmes SGBDR. Pig Latin peut être étendu en utilisant des UDF (User Defined Functions) que l'utilisateur peut écrire en Java, en Python, en JavaScript, en Ruby ou en Groovy4 et ensuite être utilisé directement au sein du langage.

C/P de Wikipédia

#### *MapReduce*

Hadoop est basé sur le principe de programmation MapReduce. Il s'agit de travailler sur une large quantité de données de la manière suivante:

1. Les données sont découpées en blocs sur le système de fichiers HDFS et sont répartis sur les datanodes (« invisible »pour l'utilisateur)
2. Une opération est effectuée sur chaque bloc de données par les datanodes => **map**
3. Réorganisation des données dans les blocs pour permettre le reduce => **shuffle**
4. Fusion des données réorganisées pour données un résultat => **reduce**

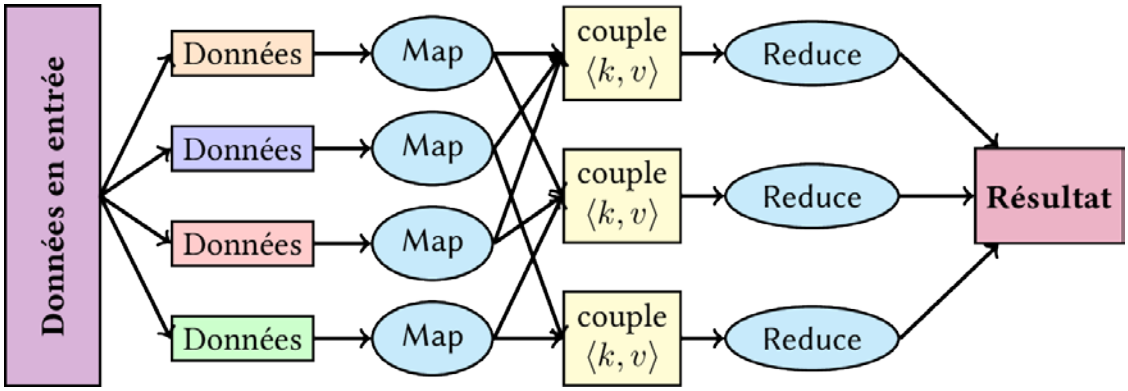



*Figure 1 : Schéma du principe de MapReduce*

**Il faut garder à l'esprit que le transfert de données entre les nœuds impacte fortement les performances.**

### *HBase*

HBase est une base de données NoSQL qui repose sur  HDFS. Son modèle d'organisation des données est orientée colonnes.

### MapR-FS

MapR-FS est un système de fichier prévu pour manipuler du Big Data en gardant l'API de HDFS et respectant le POSIX ce qui permet de faire un montage NFS. Il est un des rares systèmes de fichiers à proposer la compression des données.

Ainsi, c'est un système de fichiers compatible avec les outils utilisant HDFS ou POSIX.

### Spark

Spark est un framework de calcul : il n'est pas un *resource manager* ou un *filesystem* mais dépend fortement d'eux. Il propose cependant 2 modes de "gestion légère" des ressources pour une mise en place simple d'un cluster : *local* et *standalone* (Cf. partie Resource manager).

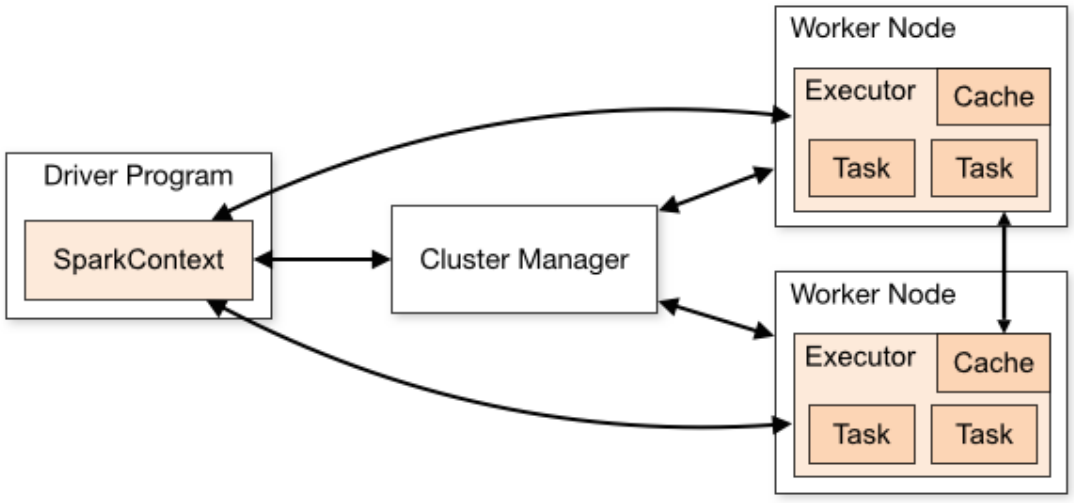



*Figure 2 : Manager de ressources dans Spark*

Spark se positionne sur la partie MapReduce mais travaille en RAM contrairement a Hadoop
Spark est constitué de 1 driver et de plusieurs exécuteurs.

Spark propose un shell interactif: `spark-shell`.
Spark est capable de faire de la reprise sur erreur car le driver connait tout ce qui se passe sur les workers.

#### ElasticSearch

ElasticSearch est un gestionnaire de données orienté documents. Il propose une implémentation native pour Spark et HDFS.

### Resource manager

Il y a 4 façons de gérer les ressources pour un cluster Spark:

#### local

Exploite tous les cœurs disponibles sur la machine. Il n'y a pas d'exécuteur, c'est le driver qui fait tout.

#### standalone

C'est le *resource manager* built-in de Spark. Les ressources sont statiques pour tous les serveurs, utilisable si on a une infrastructure dédiée. En gros 1 nœud == 1 JVM avec tous les cœurs.
Les ressources pour 1 task ne sont pas garanties vu que pour 1 JVM il y a X tasks.

#### Yarn

C'est le plus utilisé car il vient par défaut avec les distributions Spark/Hadoop.
Il part du principe que 1 task se lance dans 1 container avec 1 cœur et X Go de mémoire.

Avantages :
- bonne isolation entre les tasks
- chaque task a les mêmes ressources garanties

Inconvénients :
- overhead RAM pour la JVM (entre 400 et 800Mb)
- overhead temps de démarrage des JVM

On peut aussi faire comme pour le mode standalone, c'est à dire plusieurs tasks dans une JVM si on donne plusieurs cœurs, en modifiant certains paramètres.

Quelques particularités :
- *dynamic provisioning* : si besoin Yarn peut dépasser les ressources demandées dans la limite d'un max et redonne les ressources quand ce n'est pas utilisé
- bonne intégration pour la sécurité avec Kerberos

#### Mesos

Proche de Yarn, mais permet de cloisonner les ressources d'un pool de serveurs entre plusieurs services comme Spark, base de données, serveur web, ...

#### Kubernetes

Kubernetes est un projet open-sources de distribution de ressources par conteneurs comme Docker.
Son implémentation est actuellement expérimentale.

## Lexique & Définitions

### Driver

Le driver est le nœud qui orchestre les opérations, il ne manipule pas de données sauf pour certains actions spécifiques qui collectent tout sur le driver (exemple: `collect()`). Le driver peut être sur la machine locale (mode client) ce qui est obligatoirement le cas pour le mode interactif comme **spark-shell** ou sur un des nœuds du cluster (mode cluster).

### Worker

C'est la machine sur laquelle le container travaille. Le container est constitué d'un exécuteur et d'un cache (Cf. Figure 2).

### Partition

C'est la plus petite unité d'une donnée, elle est indivisible. Toutes les données sont découpées en partition d'au plus 128MB ou 256MB (MapR). Par défaut, Spark fait toujours 200 partitions. Il est possible de limiter la taille maximale d'une partition avec le paramètre `spark.files.maxPartitionBytes` (ce qui augmentera le nombre de partitions si nécessaire).

Réduire la taille des partitions (et donc augmenter leur nombre) permet de réduire la mémoire utilisée par les exécuteurs.

Spark ne gardera en mémoire que les partitions complètes, il n'accepte pas des partitions ne pouvant être stockées que partiellement.

### RDD (Resilient Distributed Dataset)

Les *RDDs* contiennent les données. Ce sont les structures sur lesquelles sont exécutées des suites d'instructions : les opérations. Ils sont immuables : à chaque transformation, un nouveau *RDD* est généré.

Les *RDDs* sont donc des collections distribuées sur un cluster. Les *RDDs* peuvent être stockés en RAM ou sur le disque. Si une panne machine intervient, Spark peut reconstruire un *RDD* à partir de son parent.

Un *RDD* est composé d'une ou plusieurs partitions et la parallélisation se fait sur celles-ci. Pour le cache, le mécanisme se fait aussi par partition. Si un *RDD* ne peut mettre que 2 partitions sur 5 complètements en cache (la 3ème serait tronquée), il ne mettra que les 2 : il ne fait pas de cache partiel, c'est du tout ou rien.

Bien qu'ils soient toujours à la base de la gestion des données en interne de Spark, on ne travaille plus directement avec les *RDDs* mais plutôt avec les *DataFrames*.

Il est recommandé de suivre les points suivants :

- une partition doit tenir en RAM, c'est tout ou rien, mais on peut repartitionner
- mieux vaut réaliser une multitude de petites tâches avec des petites partitions pour refaire facilement un traitement qui a planté (genre il y a 2-3min)
- pas de référence à un *RDD* dans un lambda : une lambda est une opération sur un *RDD* et un *RDD* est une suite d'opération, ça peut créer une boucle. Plus généralement, on évite de mettre des données conséquentes dans une lambda car cela oblige à sérialiser une grande partie des données.
- tout doit être sérialisable en java => transparence référentielle obligatoire
- mieux vaut effectuer un map compliqué que plein de map
- les *RDDs* ne font aucunes optimisations (mieux vaut utiliser les *DataFrames*).

Il y a plusieurs types de *RDD* (GeoRDD, CassandraRDD, EsSPark (ElasticSearch)).

Un *RDD* peut être créé à partir :

- d'une collection scala : `val rdd = sc.parallelize( List(1, 2, 3) )`
- d'un fichier distribué : `val rdd = sc.textFile( "path/to/file.txt" )`

### DataFrame

Les *DataFrames* sont une surcouche aux *RDDs* et sont faiblement typées. Elles apportent une simplification d'utilisation mais aussi de l'optimisation. En effet, Spark est capable à partir des *DataFrames* de générer un graphe d'exécution, un *DAG* (Directed Acyclic Graph), et de l'optimiser (Cf. partie ultérieure Compilateur).

Les *DataFrames* ont plusieurs avantages :

- données structurées
- inférer le schéma depuis les données (non recommandé en production)
- spécifier le schéma
- optimisation des exécutions (comme le moteur SQL)
- réorganisation des clauses `where`
- nettoyage des colonnes non utilisées
- meilleurs performances que les *RDD*
- performances uniforme quelques soit le langage (Java, Scala, Python ou R)
- nombreuses sources supportées : parquet, avro, posgresql, mysql, json, csv, HDFS, S3, hive, hbase, elasticsearch, cassandra ...

Il est possible d'utiliser au choix soit l'API des *DataFrames* soit le Spark SQL qui ressemble beaucoup au SQL. Les 2 APIs permettent de faire exactement la même chose. Néanmoins certaines opérations sont plus simples à écrire avec l'API des *DataFrames* et d'autres avec Spark SQL.

Les D*ataFrames* peuvent être générées à partir d'un *RDD* ou d'un fichier distribué.
Les *DataFrames* de Spark sont inspirées de celles de panda de python.
En plus d'avoir de meilleures performances que les *RDDs*, elles ne dépendent pas du langage (python, scala, R).

#### *Données structurées en entrée*

Les *DataFrames* acceptent des sources de données suivant un schéma (comme un format de fichier). Si lors de la récupération des données, on ne définit pas un schéma de la source, alors un sample (10% des données) est utilisé pour produire le schéma.

```
spark.read
  .option("header", "true")
  .option("delimiter", "\t")
  .option("inferSchema", "true")
  .csv(csvFile)
  .printSchema()
```

Cependant il est recommandé de définir le schéma à la main :

```
val csvSchema = StructType(
  List(
    StructField("timestamp", StringType, false),
    StructField("site", StringType, false),
    StructField("requests", IntegerType, false)
  )
)

spark.read
  .option("header", "true")
  .option("delimiter", "\t")
  .schema(csvSchema)
  .csv(csvFile)
  .printSchema()
```

Les formats *parquet* et *avro* encodent le schéma dans le fichier, il n'y a donc pas d'inférence.

### *DataSet*

Alors qu'un *DataFrame* est faiblement typé, un *DataSet* est fortement typé.
On peut dire :

- *DataSet* = *DataFrame* + case class
- *DataFrame* == *DataSet*[Raw].

### *Méthodes*

#### *select()*

Équivalent au *select* de SQL

```
df.select($"firstName", $"age").show(5)

+---------+---+
|firstName|age|
+---------+---+
|Erin     | 42|
|Claire   | 23|
|Norman   | 81|
|Miguel   | 64|
| Rosalita| 14|
+---------+---+

df.select($"firstName", $"age" > 49).show(5)
```

#### *filter()*

Filtre les lignes en fonction d'un prédicat.

```
df.filter($"age" > 49).select($"firstName", $"age").show(5)

+---------+---+
|firstName|age|
+---------+---+
|Norman   | 81|
|Miguel   | 64|
+---------+---+
```

#### *groupBy()*

Groupe par valeur d'une colonne.

```
df.groupBy("age").count().show()

sqlContext.sql("SELECT age, count(age) FROM names "GROUP BY age")

+---+-----+
|age|count|
+---+-----+
| 39|    1|
| 42|    2|
| 64|    1|
+---+-----+
```

#### *Autres*

- `orderBy()` : tri des lignes
- `as()` / `alias()` : renommer une colonne
- `join()` : jointure entre N *DataFrames*
- User Defined Function (UDF) : facile à déclarer, faire et utiliser mais limité a un map
- User Defined Agregation Function (UDAF) : plus complexe a faire car il y a une phase d'agrégation ce qui n'est pas le plus évident à faire en distribué
- `printSchema()` : imprime le schéma de la *DataFrame* sur le *stdout*
- `show(n)` : permet de voir les 20 (n pas défaut) première lignes des données sous forme d'une table comme sous SQL
- `explain(true)` : imprime sur le *stdout* l'optimisation de la requête
- `partitions` : info sur les partitions
- `limit(x)` : limite le nombre de ligne
- `distinct()` : valeurs uniques

### Executor

C'est le processus qui effectue les opérations à l'intérieur d'un worker.

Chaque exécuteur a sa propre JVM. Un exécuteur peut avoir plusieurs cœurs selon le *resource manager*.

### Job

C'est le travail pour la génération d'un *RDD* à partir de 1 ou plusieurs *RDDs*. C'est l'unité la plus haute, correspondant à toutes les opérations entre une entrée et une action.

Une application est constituée de plusieurs *jobs* qui ont plusieurs *stages* effectuant plusieurs *tasks* (applications / *jobs* / *stages* / *tasks*).

### Stage

Suite d'opérations d'un job sans interactions externe à l'exécuteur sur une suite de *RDDs* (cette suite peut n'en contenir qu'1). Un stage est un ensemble de tasks entre deux *shuffles*, toutes les transformations dans un *stage* sont effectuées sur le même *executor* , dans le même container et en mémoire.

### Task

Suite d'opérations pour un seul *RDD* dans un *stage*. Chaque *task* est une transformation.

### Opérations

Dans Spark, il y a 2 types d'opérations : les transformations et les actions.

Afin d'éviter de répéter certains blocs d'opérations lorsqu'ils sont requis par plusieurs opérations, il est possible de les mettre en cache à leur première exécution avec `.cache()`.

#### *Transformations*

Les transformations ne s'exécutent pas à leur déclaration, elles sont lazys (rien n'est effectué tant qu'une action n'est pas demandée). Toute transformation retourne un *RDD* modifié à partir d'un autre *RDD*.

List non-exhaustive :

- `map()` : applique une fonction sur l'ensemble des éléments d'une collection
- `flatMap()` : idem que map() mais aplatie la collection (voir Scala)
- `distinct()` : comme SQL retourne les valeurs uniques
- `filter()` : ne réduit pas les partitions mais peut amener un déséquilibre de quantité de données entre les partitions, voire amener certaines à d'être vides.
- `groupByKey()` : Non recommandé, fait un groupeBy sur les exécuteurs puis échange la totalité des données pour les réorganisé par clé (induit un shuffle) et fait le reduce. Si on a une clé qui regroupe trop de données ça explose la RAM. A utiliser que si l'on connaît ses données.
  `RDD( (A,1), (A,2) ).groupByKey() => ShuffledRDD( (A, CompactBuffer(1,2) )`
- `coalesce()` : réduit les partitions à un certain nombre *n* sans faire de *shuffle*.
- `mapPartitions()` : permet d'accéder à l'itérateur de chaque partition et donc d'enchaîner les calculs sur les éléments d'une même partition tout en évitant une répétition des pré- et post-étapes (par exemple, la (dé)connexion. Cela évite d'ouvrir une connexion par élément mais par partition. Utile pour la partie output sur une DB ou autre.
- `reduceByKey()` : Optimisation de groupeByKey + reduce. Il fait groupBy puis un reduce partiel sur l'exécuteur puis refait un reduce sur les données partiellement reduce. Le shuffle se fait sur des données plus petites grâce au reduce partiel.
- `repartition()` : c'est le coalesce extrême => force la recréation des partitionnements sur une clé et un nombre de partition. Réalise un *shuffle*.
- `mapPartitionsWithIndex()` : comme mapPartitions mais fournit un index en plus. Cela permet de savoir où l'on en ait dans les logs.
- `sortByKey()`
- `partitionBy()`
- `sample()` : échantillonnage d'un *RDD* par exemple pour faire des test sur des vrais données mais plus petite
- `join()` : fusionne deux *RDD*, comme pour SQL il y a plusieurs méthodes de jointure.

```
val rdd1 = sc.parallelize(List("lion", "tiger", "tiger", "peacock", "horse"))
val rdd2 = sc.parallelize(List("lion", "tiger", "parrot", "horse"))
rdd1 .distinct().collect() => Array(peacock, lion, tiger, horse
rdd1.union(rdd2).collect() => Array(lion, tiger, peacock, horse, lion, tiger, parrot,
horse)
rdd1.intersection(rdd2).collect => Array( lion, tiger, horse
```

- `pipe()` : à éviter.

Tout appel de transformation doit contenir une lambda sérializable. C'est-à-dire qu'il ne faut pas générer de nouvelles instances (new), avoir une référence à un *RDD* ou créer/fermer une connexion à une BDD.

### *Actions*

Les actions s'exécutent directement et exécutent les transformations nécessaires. À leur appel, les actions vont exécuter le *DAG*, c'est-à-dire les opérations indiquées par ce dernier. Une action créé et retourne un *RDD* à partir d' 1 ou plusieurs données.

List non-exhaustive :

- `reduce()`
- `collect() : rapatrier toutes les données sur le driver. Dangereux pour la RAM du driver. A éviter quand il y a beaucoup de données.`
- `saveAsTextFile()`
- `count()`
- `saveAsSequenceFile()`
- `first()`
- `saveAsObjectFile()`
- `take()`
- `foreach()`
- `takeOrdered()`

Les actions **saveAs*** et **foreach** se font sur les workers donc par partitions. Par exemple `saveAsTextFile()` génère autant de fichiers que de partitions.

### Shuffle

Il s'agit d'une étape intermédiaire de calcul nécessitant des transferts entre les exécuteurs. C'est une opération qui peut être coûteuse lorsque beaucoup de données transitent entre les exécuteurs. Il y a systématiquement un cache fait avant un *shuffle*. Afin de permettre un accès aux données, chaque exécuteur écrit ses données sur le disque système et ouvre un service pour permettre aux autres exécuteurs de requêter les données.

Les shuffles sont déclenchés par certaines opérations. Cela a un impact fortement négatif sur les performances, il faut donc éviter un maximum d'utiliser ces opérations :

- `repartition()`
- `coalesce()`
- `...ByKey()` à l'exception de countingByKey
- `cogroup()`
- `join()`

À savoir que `reduceByKey` effectue un `groupBy + reduce` (comme attendu) mais optimisé. De même, `sortByKey` effectue un `groupBy + sort` (comme attendu) mais optimisé. **Il est fortement déconseillé d'utiliser** `groupBy`**.**

Comme `filter` ne réduit pas les partitions, il ne fait pas de shuffle.

exchange => PRE-SHUFFLE + POST-SHUFFLE

`spark.conf.set("sparql.sql.shuffle.partitions",8)` ==> permet de changer le nombre de partition par défaut de spark (200) lors d un shuffle à un nombre plus spécifique au dataset.

### Broadcast

Le *broadcast* consiste à fournir une/des valeur(s) (val, object, etc...) a tout les exécuteurs en mode read-only. Cela est pratique pour avoir des séquences de référence accessibles sur tous les nœuds par exemple.

Les *broadcasts* ne sont pas envoyés sur le même nœud plusieurs fois. Après avoir créé un broadcast d'un élément, ce dernier **ne doit pas être modifié**.

Méthodes :

- `value` : accesseur de la valeur.
- `id` : retourne l'id unique du broadcast.
- `destroy()` : détruit toutes les données et métadonnées liées à ce broadcast.

```
import org.apache.spark.broadcast.Broadcast

val bcAnswer: Broadcast[Int] = sc.broadcast( 42 )
val i: Int = bcAnswer.value // Access


case class Ref ( val id: String, val seq: String )
val bcRefDNA: Broadcast[Ref] = sc.broadCast( Ref("myseq", "atgtgccgta" )
val refid: String = bcRefDNA.id // Access
val refseq: String = bcRefDNA.seq // Access
```

## Performances : fonctionnement et astuces

### Compilateur

Spark travaille en générant un *DAG* (Directed Acyclic Graph) à partir des *DataFrames* et des opérations à effectuer. Il optimise ensuite celui-ci, notamment en réorganisant les opérations, puis exécute les blocs en remontant le graphe pour trouver les dépendances.

Spark recompile le code, en suivant le *DAG*, en java et utilise la mémoire "unsafe" : il n'y a donc pas de GC (Garbage Collector). Cette recompilation et l'utilisation de la mémoire "unsafe" sont invisibles pour l'utilisateur bien qu'il soit possible d'obtenir les ordres d'exécutions dans la web-UI de monitoring.

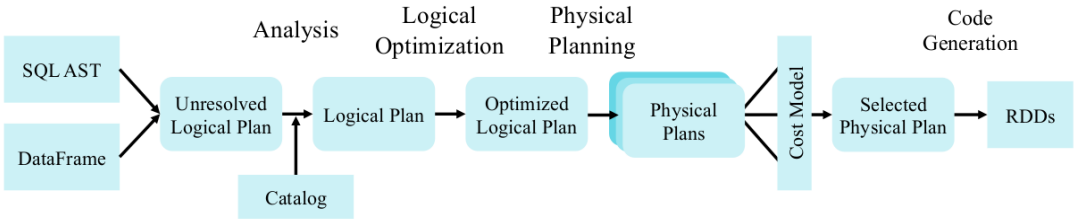



*Figure 3 : Workflow d'optimisation de code par Spark*

### Cache

Dans Spark si l'on fait N actions sur la même suite de transformation, il refera l'ensemble des opérations pour chaque action.
Afin d'éviter de recalculer 2 fois un même bloc, il est possible de mettre le résultat de celui-ci en cache.
On pose un cache sur un *RDD* pour pouvoir faire des opérations sans refaire tout le DAG précédent, c'est un cache de type LRU (Last Recently Used)

Il y a plusieurs stratégies de cache:

- MEMORY_ONLY: seulement de la RAM et pas sérialisé. Par défaut. (++)
- MEM_SER_ONLY : seulement RAM mais sérialisé. Cela a un coût CPU mais fait de la compression. (+)
- MEM_AND_DISK : RAM et disque, non sérialisé. (-)
- M_A_D_SER : RAM et disque, sérialisé. (--)
- DISK_ONLY : utilisé pour éviter de solliciter le réseau sur du S3 (cout) => on stocke en local. Bien pour garder un dataset couteux à récupérer. (---)

Pour savoir quelles partitions en cache existent, il faut aller dans l'onglet "storage" de la Spark UI ( port 4040 ).

Il est nécessaire de matérialiser un cache avec une action. Par exemple `data_rdd.cache.count`.
Attention: Le cache sur disque se fait sur le système de fichiers de la machine et pas sur HDFS.
Lors d'un **shuffle,** Spark fait un cache automatique sur disque afin de gérer les problèmes de perte de nœuds et pour réorganiser les données.

## Scheduling

Le job de Spark est de faire du scheduling et de gérer la reprise sur erreur.
Workflow:

1. *RDD* + transformations + action
2. création d'un DAG à partir des *RDDs*
3. découpage en stages des tasks
4. exécution des tasks et retry si erreur

## Interface web 4040

Une interface web est disponible sur un nœud ou le master via le port 4040 et 8088 (Hadoop). Par exemple : `spark01:4040` et `spark01:8088`.
Les informations des vieux jobs sont accessibles par l'*history server* au http://spark04:18080/?showIncomplete=true.
L'interface 4040 dispose d'une multitude d'informations :

Attention: le nœud hébergeant les services peut changer

### Jobs

Cet onglet donne les jobs en cours et l'historique.
L'***Event Timeline*** permet d'observer la création et destruction des executors. Cela permet notamment de savoir lesquels utilisent trop de RAM.
Attention, un job Spark peut stagner : les exécuteurs sont tués car ils consomment trop de mémoire et de nouveaux sont créés en boucle. Pour corriger cela il est possible d'augmenter la mémoire max des executors jusqu'à une certaine limite.
Si cela est toujours insuffisant, il peut être nécessaire de réduire la taille des partitions comme vu dans la partie Partition.

### Stages

Cet onglet donne tout les stages d'un job avec des statistiques pour toutes les partitions et agrégés.
**locality level** :

- PROCESS LOCAL (JVM)
- NODE LOCAL (nœud)

- RACK LOCAL (rack)
- ANY (reste du monde)

L'***Event Timeline*** montre le temps passé pour le stage avec le type de travail :
- `Executor Computing time` (vert) doit être très majoritaire > 95%, c'est le temps effectif de calcul
- `Scheduling delay` => si trop haut c'est que les jobs sont trop courts et le scheduling n'est pas efficace, partition trop petite
- `Deserialization Time` => doit reste bas sauf si beaucoup de données

### Storage

Montre l'état du cache dont ce qui est en cache. Permet également de savoir de quel type de cache il s'agit/
Storage Level:
- `memory deserialized` => MEMORY_ONLY
- `memory serialized` => MEM_SER_ONLY
- `disk serialize` => M_A_D_SER
- `disk deserialized` => MEM_AND_DISK
- `disk only` => DISK_ONLY

## Streaming

Il est possible de faire du streaming avec Spark. C'est-à-dire avoir une application qui tourne en boucle et lui fournir les données au fur-et-à-mesure.
Plusieurs modes de récupération des donnes sont possibles, par exemple le mode receiver et le mode DirectStreaming.
Le mode receiver n'est pas recommandé car il bloque un ou des exécuteurs à cette tâche.
Pour faire du DirectStreaming, il faut utiliser **Kafka** qui va absorber les données dans des partitions et attribuer un exécuteur à une partition pour le traitement de celles-ci. Kafka micro-batch toutes les N unités de temps les DirectStreams puis génère des *RDD*/*DataFrames*.

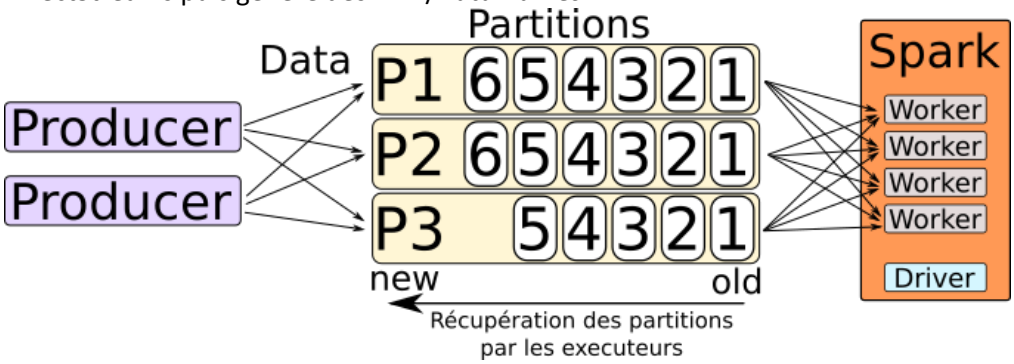



*Figure 4 : Fonctionnement de Kafka. Répartition des données sur les partitions Kafka (P1, P2, P3), transformation des données en partitions Spark puis récupération de celles-ci par Spark.*

Kafka fait du commit log distribué. C'est le point central de temporisation des données. Kafka ajoute les données entrantes comme blocs de partitions. Il fait du « append-only » mais ce n'est pas une queue.

## Machine Learning

Il existe 2 bibliothèques spark pour le machine learning : `spark.ml` utilisant les *DataFrames* et `spark.mllib` utilisant les *RDDs*.
Il est recommandé d'utiliser `spark.ml` pour cela, plus intuitive.

Il y a 2 types de processus :
- `supervised` : avec corpus annoté pour un X on sait => Y
- `unsupervised` : X+ fonction norme => classification : quel point correspond à quelle classe où X est une feature et Y un Label ( le résultat, ce qu'on cherche ).

## Format

### Avro

Format binaire orienté ligne avec un schéma évolutif, on peut modifier le schéma dans une certaines limites
Principalement utilisé quand on doit parcourir toutes les données et pour les applications temps réel.

### Parquet

Format binaire orienté colonne et avec un schéma évolutif (like avro)
Principalement utilisé quand on travail sur une partie des colonnes.

# XII.2. Compte rendu de la formation Scala
## janvier 2018

Ludovic Legrand[1], Axel Verdier[1], Olivier Filangi[2] et Xavier Garnier[3]

[1]UMR INRA Laboratoire des Interactions Plantes-Microorganismes
[2]UMR INRA Institut de Génétique, Environnement et Protection des Plantes
[3]UMR Institut de Recherche en Informatique et Système Aléatoire

Ce compte rendu synthétise les 3 jours de formation Scala suivie lors de l'école « Programmation Fonctionnelle et Spark » organisée par le Laboratoire de l'Accélérateur Linéaire à Orsay en janvier 2018.

L'objectif de ce document est de donner des clés pour débuter la programmation fonctionnelle avec Scala. Une première partie présente le langage et ses similitudes avec Java pour la programmation orientée objet. La seconde partie introduit les concepts de la programmation fonctionnelle et les grandes lignes de l'API Scala d'un point de vue fonctionnel.

# Table des matières

# Scala

## Généralités

Utilisé par plusieurs grands noms de l'industrie informatique : Twitter, Netflix, La poste, AXA

### Écosystème

- **SBT** : build et résolution des dépendances
- application web, API REST**: Play! framework**
- gestion de la concurrence : message passing : **Akka**
- calcul distribué : **Spark**

La syntaxe est plus légère en Scala que Java.

### Définition

Scala est un langage de programmation multi paradigme (c'est un langage objet, mais aussi fonctionnel). Il est compilé en Bytecode pour tourner sur la JVM (Permet d'utiliser toutes les librairies Java existantes). Le typage est statique.

A ce jour Scala ne garanti pas la compatibilité entres les versions (2.10, 2.11 et 2.12). La version actuelle est la 2.12.4.

Dans un langage fonctionnel, les variables sont immuables, et les fonctions sont traitées comme des valeurs.

C'est aussi un langage objet, avec le système de classe, méthodes, héritage *etc*.

### le REPL

Le REPL (Read-Eval-Print Loop) est la console scala. Peut-être lancé par la commande `scala` ou par SBT avec `sbt console`.

Ammonite est un REPL pour scala plus complet, il apporte une coloration syntaxique, l'édition des lignes déjà tapées, l'import de bibliothèque *etc*.

### Scala

- **???** : permet de ne pas implémenter la méthode tout de suite mais de pouvoir compiler quand même par contre au runtime une exception **Not Implemented** sera levé si la fonction est appelée
- **==** : comparaison des objets au niveau des champs et pas au niveau référence comme pour Java
- **val** : déclaration d'une variable immuable (finale en Java)
- **var** : déclaration d'une variable mutable
- **def** : déclaration d'une fonction
- **Boolean**: ne peut pas être *Null*, il peut prendre que *true* ou *false* comme état.

### *Lazy*

On peut rendre une variable **lazy** avec le même mot clé. L'évaluation de la variable est faite quand elle est demandée et le résultat est stocké dans la variable.

```scala
import scala.util.Random

class RandomPoint {
    val x = { println(s"creating x"); Random.nextInt}
    val y = { println(s"creating y"); Random.nextInt}
}

val point = new RandomPoint()
creating x
creating y
// initialisation des 2 variables a l'instantiation

class RandomPoint {
    val x = { println(s"creating x"); Random.nextInt}
    lazy val y = { println(s"creating y"); Random.nextInt}
}

val point = new RandomPoint()
creating x  // y n'est pas évalué pour le moment

println(s"Location is ${point.x}, ${point.y}")
creating y // évaluation de y au 1er appel
Location is -597462083, -1297147190

println(s"Location is ${point.x}, ${point.y}")
Location is -597462083, -1297147190  // en cache il ne bouge plus
```

## Le langage POO

### Classe

Les champs de la classe sont déclarés directement dans la définition de la classe s'il y a **val** ou **var**.

```scala
class Car(val name: String, val seat: Int = 4) {
    // ...
}
val car = new Car("test")
car.seat // retourne 4

val car = new Car(seat = 5, name = "test2")
car.seat // retourne 5
```

Ici **name** et **seat** sont les champs de la classe **Car**.

Exemple avec des paramètres simples :

```scala
class Car(name: String, seat: Int) {
    val carName = name
}
val car = new Car("K2000", 1)

car.carName // retourne K2000
car.seat    // error: value seat is not a member of Car
```

Dans l'exemple, **name** et **seat** ne sont pas des champs de la classe **Car**. Ces paramètres peuvent initialiser des champs comme **carName** mais une fois l'instance crée, ils n'existent plus.

Quelques remarques :

- on peut mettre une valeur par défaut comme pour **seat** dans le 1er exemple
- on peut nommer les champs à la construction pour ne pas suivre l'ordre

### *Constructeur*

Par défaut il y a un constructeur utilisant les arguments de la classe mais il est possible de définir un constructeur spécifique de cette manière.

```
class Car(val name: String, val seat: Int) {
    def this(name: String) = {
        this(name, 1)
    }
}
val carFull = new Car("2CV", 4)
val car = new Car("4CV")
car.seat // retourne 1
```

### *Case class*

Une **case class** est une classe avec des méthodes et un **companion object** déjà disponibles:

- **hashcode** : méthode retournant une clé unique pour les collections comme Map et Set
- **equals** : méthode permettant de comparer 2 classes au niveau des champs
- **toString** : méthode permettant l'affichage sous forme de String des champs
- **copy** : méthode pour faire une copie de l'instance (changement des valeurs possible)
- **apply** : méthode permettant d'instancier la classe sans **new**
- **unapply** : méthode permettant de récupérer les champs dans une collection d'Option

Par contre tous les champs sont **immuables** et **publics** par défaut. Il faut préciser la portée si on veut autre chose que des champs publics.

Raisons de ne pas utiliser les **case class** :

- champs mutables
- héritage (pas recommandé)

Utilisation recommandée :

- Stockage de données (exemple Car)

```
case class Car(val name: String, val seat: Int)
val car1 = Car("2CV", 4)
val car2 = Car("2CV", 4)
car1 == car2 // true
```

### *Classe abstraite*

Comme pour Java, on utilise le mot clé **abstract**. Les classes abstraites peuvent définir des méthodes abstraites mais avec un typage explicite.

Les classes qui héritent d'une classe abstraite devront implémenter les méthodes abstraites.

### Objet

C'est l'équivalent d'un singleton de Java, il existe qu'en un seul exemplaire et toutes ses méthodes sont statiques. Il n'existe pas de mot clé **static** en Scala, il faut utiliser les **Object**. Il a les mêmes propriétés qu'une classe, il peut hériter d'une classe et des traits.

### *Companion object*

Un **Object** avec le même nom et dans le même fichier qu'une classe est appelé **companion object**. On peut implémenter les méthodes suivantes :

- **apply**: permet d'instancier la classe sans le mot **new**
- **unapply**: permet d'extraire les champs de l'instance dans une collection d'**Option**

```
class Person( val name: String, val surname: String ) {}

object Person {
    def apply( name: String, surname: String ): Person = new Person( name, surname )
    def unapply( p: Person ): Option[(String, String)] = {
        if( p.name.nonEmpty && p.surname.nonEmpty ) Some((p.name, p.surname))
        else None
    }
    val jedi: Person = Person( "Luc", "Skywalker" )
    jedi.name
    jedi.surname
    Person.unapply(jedi)
}
```

### *Application / main*

Une façon de déclarer un **main** est de d'implémenter un **object** avec le trait **App** :

```
object Exemple extends App {
    ???
}
```

Il est aussi possible de simplement déclarer une fonction `main( args: Array[String]): Unit` dans un **Object** :

```
object Exemple {
    def main (args: Array[String]): Unit = {
        ???
    }
}
```

Si il y a plusieurs **main** dans une application, il faudra préciser lequel utiliser.

### Scope

Par défaut tout est public et tout est immuable.

Si on veut modifier la portée d'une fonction ou valeur il faut utiliser les mots clés suivant :

- **private[this]**: restreint l'accès à l'instance uniquement (peu utilisé)
- **private**: restreint l'accès aux instances de la même classe
- **private[myPackageName]**: restreint l'accès aux instances des classes du package et au dessous
- **protected**: restreint l'accès aux instances des classes qui héritent de la classe

```
class Scope(val publicImmutable:String, private val privateScopeImmutable: String, priv
ate[this] val privateInstanceImmutable: String)
```

### Héritage

C'est très similaire à Java, une classe peut hériter que d'une seule classe.

Le mot clé est **extends** et il faut assigner ou transmettre les arguments pour la classe mère. Dans l'exemple, **VeryImportantPerson** transmet les arguments à **Person**

```
class Person(val name:String, private[this] var gender:String, private val birthYear:
Int) {
  def isOlder(anotherPerson:Person) = birthYear > anotherPerson.birthYear
  protected def getGender = gender
  def getBirthYear = birthYear
}

class VeryImportantPerson(override val name:String, private[this] var gender:Strin
g, private val birthYear:Int) extends Person(name, gender, birthYear) {
  override def isOlder(anotherPerson:Person) = false
  override def getGender = super.getGender
}
//une instance de VeryImportantPerson donne accès a name, isOlder et getGender

class UnknownPerson(gender:String, birthYear:Int) extends Person("John Doe", gender,
birthYear)
//une instance de UnknownPerson donne accès a name et isOlder comme Person
```

Il est obligatoire d'utiliser **override** devant une méthode surchargée par la classe fille, en Java c'est optionnel.

Pour utiliser explicitement une méthode de la classe mère il faut utiliser le mot clé **super** comme pour **getGender** dans **VeryImportantPerson**. La méthode **getBirthYear** est disponible dans la classe **VeryImportantPerson** et utilisera la méthode de la classe mère par défaut.

L'héritage se place en premier donc s'il y a un héritage et N **trait** il faudra mettre la classe à hériter après le **extends** et les **trait** avec **with**.

### Trait

Ça ressemble aux interfaces de Java, une classe peut implémenter plusieurs **trait** contrairement à l'héritage.

On utilise **extends** pour le premier **trait** comme pour l'héritage et/ou **with** pour les **trait** suivant. L'ordre est important, si 2 traits définissent une même chose (par exemple une fonction), le dernier **trait** implémenté est celui qui sera effectif.

Dans les **trait** on peut mettre des fonctions ou des signatures de fonctions

```
trait CanFly
trait Feather
class Animal(name:String)
// héritage
class Lion(name:String, sound:String) extends Animal(name)
// héritage et 2 traits
class Bird(name:String, sound:String) extends Animal(name) with CanFly with Feather
```

On peut limiter la portée des **trait** au fichier avec le mot clé **sealed**, seule les classes dans le même fichier pourront utiliser le trait mais il sera quand même visible de l'extérieur. Exemple si on fait un traitement avec un **pattern matching**, on veut être sûr qu'on connaît tous les objets à traiter.

On peut forcer un **trait** à être dépendant d'une classe.

```
trait CanFly {
    self:Vehicule =>
// uniquement les classes héritant de Véhicule peuvent utiliser CanFly

    def takeoff : println("ca décolle")     // méthode spécifique au trait
    override def fix: Unit : self.fix
}
```

On a la possibilité d'instancier une classe avec un trait à l'instanciation. Cela permet pour du code test de surcharger des comportements comme l'accès à une base de données tests.

```
new Voiture("dolorean") with CanFly
```

### Generic

Fonctionne comme en Java avec **[]** au lieu de **<>** comme symbole.

```
// signature de List ressemble à ca: class List[A]
// je déclare une List de type Int
val listInt: List[Int] = List[Int](1,2,3)
// ou de string
val listString: List[String] = List[String]("1","2","4")
```

## Programmation fonctionnelle

Le programme est une suite de fonctions imbriquées. Les variables sont immuables, une fois définies, elles ne changent pas de valeurs. Les fonctions peuvent-être passées en paramètres d'autres fonctions.

Pas forcement la meilleur façon d'exploiter les ressources d'une machine ou d'un CPU mais par contre dans un environnement distribué ça simplifie énormément le développement car il y a pas ou très peu d'effet de bord.

### Effet de bord

Une fonction qui impact sur autre chose que ce qui est dans sa signature n'est pas pure. Son comportement n'est pas sûr car sa signature n'est pas suffisant pour savoir ce qu'elle va faire réellement.

Le problème c'est que le monde entier est un effet de bord, une connexion à une base de données peut ne pas fonctionner, l'écriture dans un fichier peut être impossible ... C'est pour ça qu'il y a les **monades** pour gérer les effets de bord.

### Transparence référentielle

Une fonction transparente renvoie toujours le même résultat pour une même entrée. Une fonction renvoyant un nombre random n'est pas transparente.

L'intérêt c'est de permettre de faire des optimisations avec des caches (**mémoïsation**) car on sait que f(3)=5 quelques soit le contexte ou l'état de l'application.

### Immuabilité

C'est par défaut dans le langage Scala, il permet mais n'encourage et ne facilite pas l'utilisation des variables mutables.

### Fonction d'ordre supérieur et lambda

En Scala une fonction est une valeur comme une autre donc on peut passer en argument une ou des fonctions à une autre fonction qui peut retourner une fonction aussi. Cela permet de faire des compositions de fonctions élémentaires pour construire des fonctions plus complexes.

Une fonction d'ordre supérieur doit au moins avoir une des propriétés suivantes:

- prendre en argument une fonction
- retourner une fonction comme résultat

```
def addOne(x:Int) = x + 1
def compositionFun(f: (Int) => Int): Int = {    //f: (Int) => Int : une fonction f avec u
n argument Int retournant un Int
    return f(0) + f(1)
}
compsitionFun(addOne)
```

Les **lambdas** sont un cas particulier, se sont des fonctions anonymes.

```
val addOne2 = { x:Int => x+1 } // fonction anonyme assignée à addOne2

compositionFun(x => x + 1)
f(0), x=0 et résultat: 0+1 = 1
f(1), x=1 et résultat: 1+1 = 2
```

Attention à la notation:

- *compositionFun(_ + 1)* : _ représente le premier paramètre puis le paramètre n+1...
- *compositionFun(x => x + 1)* : permet de faire ce qu'on veut et est plus facile à lire

Exemples:

- *compositionFun(_ + 1) == compositionFun(x => x + 1)*
- *compositionFun(_ + _) != compositionFun(x => x + x)*
- *compositionFun(_ + _) == compositionFun((x,y) => x + y)*

### Boucle / Récursivité

Faire une boucle oblige à avoir un compteur qui s'incrémente ou un itérateur qu'on parcourt. L'incrément ne respecte pas l'immutabilité car il faut déclarer un variable qui va changer au cours du temps et l'itérateur a un effet de bord car on modifie l'état de l'itérateur à l'extérieur de la fonction. Il faut utiliser la récursivité et la dernière opération doit être la fonction de récursion seule pour optimiser au mieux et éviter des **stackoverflow**.

La récursivité implique de garder l'état en paramètre de la fonction et de faire le changement d'état dans l'appel de la fonction de récursion.

```
// Finale, tailed-end
def add( x: Int, y: Int ): Int = {
    if( y > 0 ) { add( x+1, y-1) }
    else { x } //return x
}
add(4, 100006) // Return 100010

// Non-finale
def badAdd( x: Int, y: Int ): Int = {
    if( y > 0 ) { badAdd( x, y-1)+1 }
    else { x }
}
badAdd(4, 100006) // Problem !
```

On peut utiliser **@tailrec** qui est une annotation pour dire qu'on veut et attend une fonction récursive terminale. Si elle ne l'est pas on aura une erreur de compilation

### Curryfication

C'est la décomposition d'une fonction à N arguments en une fonction avec N fonctions de 1 argument. Permet de faire des fonctions partielles.

```
def sum(x:Int, y:Int, z:Int) = x+y+z
def curryfiedSum(x:Int)(y:Int)(z:Int) = x+y+z

def partial1 = curryfiedSum(10)_
//uniquement le 1er argument (x) a 10 les autres sont pas encore renseigne

def partial1bis = curryfiedSum(10,_,_)
// idem mais permet de remplir y au lieu de x si on veut
```

L'objectif est de réutiliser des fonctions génériques en fonctions partielles en fixant un des termes.

```
def curryfiedSum(x:Int)(y:Int)(z:Int) = x+y+z
val curryfiedAdd = curryfiedSum(0)_
// transformation de la fonction sum en add en fixant le premier argument
```

Transformer une fonction en fonction *curried* rapidement:

```
val curriedSum = (sum _).curried
curriedSum: Int => (Int => (Int => Int)) = <function1>
```

### Fermeture

C'est une fonction qui utilise une variable défini en dehors de son bloc

## API Scala

### Pattern matching

C'est l'équivalent d'un switch sous stéroïde, utilisable sur la plupart des types de Scala et on peut y mettre des conditions. L'ordre des cases a un sens car il s'exécute dans l'ordre.

```
//cas classique comme un switch
val i = 1
i match {
    case 0 => println("0")
    case 1 => println("1")
    case _ => println("default")
}

//cas un poil plus avancé
val a = i match {
    case x if x%2 == 0 => true
    case _ => false
}

def a(i:Int) = i match {
    case x if x%2 == 0 => true
    case _ => false
}

sealed trait Vehicle
case class Car(seat:Int) extends Vehicle
case class Bicycle() extends Vehicle
def s(vehicle:Vehicle) = vehicule match {
    case Bicycle() => ???
    case Car(1) => ???
    case Car(4) => ???
}
```

### Collection

#### *Set*

Collection où les éléments sont uniques.

#### *Map*

Collection type clé-valeur.

#### *Seq*

Ce sont des listes.

#### *IndexeSeq*

- accès rapide à chaque élément
- ex: **Array**, **Range**, **Vector**

Attention à **Array**, c'est le type de Java qui est utilisé et non une conversion vers un type Scala immuable. Par conséquence les **Array** sont mutable.

```
val array = Array(1,2,3)
array(0) retourne 1
array(0) = 2
array(0) retourne 2
```

Le **val** protège la réassignation de **array** par une autre **Array** mais par contre la collection est mutable donc on peut changer les valeurs à l'intérieur.

Si l'on fait la même chose avec une **List** ou une autre collection de base dans Scala on a une erreur car on ne peut pas modifier une collection par défaut. Il faut utiliser des collections particulières qui ont la propriété d'être mutable.

### *LinearSeq*

- accès et insertion rapide du 1er élément
- **List**, **Stream**

***Nil*** représente la liste vide, et ***::*** permet de concaténer des éléments à une liste.

```
val list1 = 1::2::3::Nil
val list2 = List(1,2,3)
```

### *Tuple*

Les collections demandent d'avoir une homogénéité dans les valeurs stockées. Si on mélange les types et laisse Scala inférer le type d'une collection, il va chercher à créer une collection avec le type dénominateur commun qui peut aller jusqu'a **Any**.

Les tuples peuvent être hétérogènes mais l'ordre importe et est limité à 22 éléments (34 éléments en 2.12)

```
val tuple = (1, "hello", true)
tuple _1 // accès au premier élément du tuple
```

### *Range*

Génère une collection entre 2 valeurs avec un **step** qui par défaut vaut 1.

```
val range = 1 to 100
val range = 1 to 100 by 2

val range = 1 until 100 by 2
range.start  // 1
range.end    // 98
range.step   // 2
```

### *Opération*

- `Seq ++ Seq` : concaténation de 2 collections
- `take(n)` : prendre les n premiers éléments d'une séquence
- liste vide: `val emptyList = Nil`
- liste initialisée: `val list = List(1,2,3)`

### *Filter*

Avec un prédicat (opération booléenne) on va garder que les éléments valides d'une collection pour retourner une nouvelle collection. La nouvelle collection aura au pire autant d'éléments.

```
val list = List(1,2,3,4,5)
// list: List[Int] = List(1, 2, 3, 4, 5)
val list2 = list.filter(x => x%2 == 0)
// list2: List[Int] = List(2, 4)
```

### *Map*

Applique une fonction à tous les éléments d'une collection et retourne une nouvelle collection. La nouvelle collection aura le même nombre d'éléments.

```
List(1,2,3).map(x => List(x,x+3,x+6))
//List[List[Int]] = List(List(1, 4, 7), List(2, 5, 8), List(3, 6, 9))
```

### *FlatMap*

Applique une fonction à tous les éléments d'une collection et retourne une nouvelle collection « aplatie ».

```
List(1,2,3).flatMap(x => List(x,x+3,x+6))
//List[Int] = List(1, 4, 7, 2, 5, 8, 3, 6, 9)
```

### *Reduce*

Réduit une collection en appliquant une opération sur 2 valeurs pour en produire une.

```
List(2,3,4,5).reduce((x,y) => x*y)   // x == accumulateur, y == elem courant
/*
2 3 4  5
  6 4  5
    24 5
       120
*/
```

C'est une opération terminale car elle ne retourne pas une collection.

### *Fold**

Proche de **reduce** mais permet d'avoir un accumulateur initial autre que le 1er élément. Si la collection est vide, elle retourne la valeur de l'accumulateur donc 5 ici.

```
List(2,3,4,5).foldLeft(5)((x,y) => x*y)
/*
5, 2  3  4   5
   10 3  4   5
      30 4   5
         120 5
             600
*/
```

**foldRight** fait la même chose mais en partant de la fin.

Il est préférable d'utiliser **foldLeft** si l'opération est commutative car plus efficace et récursive terminale

C'est une opération terminale car elle ne retourne pas une collection.

### *Zip*

Cette méthode permet d'associer 2 collections dans une nouvelle collection de tuple2.

### *zipWithIndex*

C'est un zip sur une collection et un range de sa taille. Cette méthode génère une collection de tuple (valeur, index)

### *TakeWhile*

Retourne les éléments d'une collection tant qu'ils répondent au prédicat.

### *Span*

À partir d'une collection et d'un prédicat, retourne 2 collections. La première avec les éléments respectant le prédicat et la seconde pour le reste.

## For comprehension

Sucre syntaxique (facilité) pour:

- rendre plus lisible les imbrications de map, flatMap, filter ...
- itérer sur plusieurs collections

On peut y parcourir N collections. Le **type retourné** est celui de la **première déclaration**.

```
for {
    x <- 1 to 3   //collection 3
    y <- 4 to 6   //collection 2
    z <- 7 to 9   //collection 1, les collections peuvent être externe
    if y % 2 == 0 //filtre sur collection2, remplaçable par une fonction si on veut
} yield x * y * z //opération
/*
Vector(28, 32, 36, 42, 48, 54, 56, 64, 72, 84, 96, 108, 84, 96, 108, 126, 144, 162)
1*4*7
1*4*8
1*4*9
1*6*7
...
*/
```

Équivalent à cette version moins lisible :

```
(1 to 3).flatMap(x => (4 to 6).filter(y => y % 2 == 0).flatMap(z => (7 to 9).map(a => x*
z*a) ) )
//Vector(28, 32, 36, 42, 48, 54, 56, 64, 72, 84, 96, 108, 84, 96, 108, 126, 144, 162)
```

## Monades

Permet d'encapsuler le monde extérieur pour obtenir la transparence référentielle.

- **functor** implémente map
- **monade** implémente flatMap

Les monades retournent toujours une valeur parmi 2 possibles:

- List[A] = List[A] ou Nil

- Option[X] = Some(value:X) ou None
- Try = Success ou Failure
- Either[A,B] = Left(A) ou Right(B)
- Future = Success ou Failure

### *Gestion erreur et null*

Idéalement on évite de lever des erreurs pour ne pas casser la transparence référentielle. Une exception n'est pas indiquée dans la signature de la méthode/fonction et elle va se propager ou pas (présence d'un **try** ou pas) donc ce n'est pas idéal.

L'autre problème c'est l'impossibilité de chaîner les traitements de manière sûre. Si une exception est levée la chaîne est cassée.

Pour gérer les cas d'erreurs, on peut utiliser **Try**, **Either** et **Option**.

Tous les types suivants sont des monades donc ils ont les méthodes pour faire des traitements et les chaîner (**map**/**flatMap**)

### *Option*

**Option** est un moyen de dire qu'il y aura un résultat ou pas mais ce n'est pas grave. Au lieu de retourner un **Null** qui provoquera une **Exception**, on utilise une **Option**.

On récupère le résultat avec **getOrElse**, ce qui permet de retourner une valeur par défaut si l'option est vide (**None**).

Il ne faut pas utiliser **get** car s'il y a **None**, cela génère une erreur.

```
// on sait pas qu'il y a pas de résultat c'est tout
def divide(x:Double, y:Double): Option[Double] = if (y == 0) None else Some(x/y)

// a l'utilisateur de gérer l'absence de résultat
divide(2.0,0.0).getOrElse("division par 0")


divide(2.0,0.0).get
// java.util.NoSuchElementException: None.get
```

### *Either*

Similaire à **Option** il permet de retourner un message d'erreur au lieu de **None**. **Either** est composé de **Left** pour les erreurs et **Right** pour la valeur. C'est une convention il n'y a pas d'obligation à respecter ce sens.

```scala
// on est informé du problème
def divide(x:Double, y:Double): Either[String,Double] = if (y == 0) Left("Can't divide by 0") else Right(x/y)

// monade oblige on peut chaîner des traitement sans tenir compte des cas en erreur
def add(eX:Either[String, Double], eY:Either[String, Double]) = eX.right.flatMap { x => eY.right.map { y => x + y } }

// idem en for comprehension
def add(eX:Either[String, Double], eY:Either[String, Double]) = for {
    x <- eX.right
    y <- eY.right
} yield x+y
```

### *Try*

A la mode de Java, on peut récupérer une Exception et la traiter.

```scala
val result = Try { /* code susceptible de générer une erreur */ }
result match {
    case Success(value) => value
    case Failure(e:NullPointerException) => ???
    case Failure(_) => ???
}
```

### Implicit

Les valeurs, fonctions et classes peuvent être déclarées avec le mot clé **implicit**.

C'est un moyen pour Scala de faire de l'injection de dépendances.

L'ordre de résolution des implicites est à prendre en compte mais si 2 implicites ont la même signature le compilateur génère une erreur car il y a ambiguïté

Ordre de résolution:

1. scope actuelle
2. import explicite (import package.myClass)
3. import generic (import package._)
4. companion object

Les implicites sont une des raisons de la lenteur de compilation du langage. A chaque implicite il doit rechercher la définition qui va bien.

C'est un outil puissant mais à utiliser avec parcimonie car ça rend plus difficile la lecture du code et le debugging.

### *Implicit sur les valeurs*

Sur des valeurs ça peut être intéressant pour assigner une valeur dans toutes les fonctions qui déclare l'implicite. Ca évite de devoir passer systématiquement des paramètres.

Exemple d'utilisation:

- paramètres de connexion d'une base de données
- contexte d'exécution pour le multithreading

```
import scala.concurrent._
import ExecutionContext.Implicits.global

implicit val ec: ExecutionContext = scala.concurrent.ExecutionContext.Implicits.gl
obal

case class Employee(id: Int, name: String)
case class EmployeeWithRole(id: Int, name: String, role: String)

def getEmployee(id: Int)(implicit e: ExecutionContext): Future[Employee] = ???
def getRole(employee :Employee)(implicit e: ExecutionContext): Future[Role] = ???

val bigEmployee: Future[EmployeeWithRole] =
  getEmployee(100).flatMap { e =>
    getRole(e).map { r =>
      EmployeeWithRole(e.id, e.name,r)
    }
  }
```

### *Implicit sur les fonctions*

Permet de faire des conversions automatiques, d'ailleurs dans Scala c'est très utilisé pour çà. Toutes les conversions entre les types Java et Scala ou entre les types Scala se font à l'aide de fonctions implicites.

### *Implicit sur les classes*

Les implicites sur les classes permettent d'ajouter des méthodes à des classes déjà existantes dans des librairies externes. On peut ajouter mais pas **override** des fonctions déjà existantes.

C'est très utilisé par le langage Scala pour étendre des types java, une grande partie des méthodes sont issues des implicites.

Il y a 2 façons de faire des implicites:

- utilisé le mot clé **implicit** et à ce moment il y aura création d'un nouvel objet pour faire ce qu'il y a faire puis il est mis à la poubelle. Ça peut poser des problèmes de performances si on a des implicites dans une boucle par exemple
- utilisé le mot clé **implicit** et faire hériter de la classe **AnyVal**. Çà implique d'avoir que des paramètres immuables mais c'est beaucoup plus efficace car les fonctions seront rendues **inline** où c'est possible au moment de la compilation.

```
import java.time.LocalDate

case class Person(name: String, age: Int, birthDate: LocalDate)
val ryan = Person("ryan",21,LocalDate.parse("1997-01-01"))

println(ryan.isAdult)
// error: isAdult is not a member of Person

// ajout d'une méthode par implicit
implicit class PimpPerson[T <: Person](val p: T) extends AnyVal {
    def isAdult(): Boolean = p.age > 18
}
println(ryan.isAdult)
// true
```

### Concurrence

Il n'y a pas dans Scala d'avancées particulièrement innovantes pour la concurrence. Il part du principe que la limitation (voir l'absence) d'effets de bord facilite l'utilisation de la concurrence.

#### *Par*

Pour paralléliser un traitement sur une collection, on utilise la fonction **par**.

```
(0 to 100).par.map(x => x*2)
```

http://docs.scala-lang.org/overviews/parallel-collections/overview.html

Autant c'est sans risque (mais pas forcément efficace) sur beaucoup de méthodes (map, flatMap...) autant il faut faire attention avec les méthodes du type **reduce** et **fold*** car si la fonction d'agrégation n'est pas commutative (même résultat quelque soit le sens de calcul) on aura des résultats imprévisibles.

#### *Thread*

Les threads s'exécutent dans un contexte d'exécution qu'il faut déclarer soit en implicite soit en explicite.

```
implicit val executionContext = ExecutionContext.global  //mode implicite
import ExecutionContext.Implicites.global                 //mode explicite
```

Par défaut il y a un contexte global qui utilise tous les cœurs d'une machine sans limitation mais on peut fixer son propre contexte d'exécution avec :

```
implicit val executionContext = ExecutionContext.fromExecutor(new ThreadPoolExecutor(//
conf))
```

Les différents types d'exécuteurs:

- **FixedThreadPool** : un nombre fixe de thread utilisés, les taches en plus sont dans une queue. Idéal pour les traitements lourds
- **CachedThreadPool** : génère autant de threads que de tâches et peut réutiliser les threads. A utiliser pour des taches courtes (bdd).

- **ForkJoinPool** : mode par défaut de la JVM.

### *Future*

Pour jouer avec les threads en mode asynchrone il faut utiliser **Future** qui est une sorte de promesse. Le traitement va être exécuté un jour et il te retournera un résultat ou une erreur mais potentiellement pas tout de suite. **Future** est une *monade* aussi donc on peut le chaîner ou le mettre dans un **for-comprehension** sans problème.

```
import scala.concurrent._
//déclaration explicite d'un contexte d'exécution par défaut
import ExecutionContext.Implicits.global
import scala.util.{ Success , Failure }

val future:Future [ Int ] = Future { Thread.sleep(10000) ; 1}

future onComplete {                                                    //callback
    case Success(value) => println(value)                             // qu'est ce que je
fais si c'est ok
    case Failure(throwable) => println( throwable.getMessage() )    // qu'est ce que je
fais si problème
}

// comportement avec une chaîne
val future1 = Future { Thread.sleep(10000) ; 1}
val future2 = Future { Thread.sleep(10000) ; " Finished " }
val future3 = future1.flatMap( x => future2.map( y => println((x,y) ) ))`
```

Quand on chaine des **Future**, les actions se réalisent en parallèle mais on garde l'ordre de sortie.

On peut aussi vouloir synchroniser les threads et pour ça on utilise la fonction **Await**

```
import ExecutionContext.Implicits.global
import scala.concurrent._
import scala.concurrent.duration._

val start = System.currentTimeMillis()

val future1 = Future { Thread.sleep(20000) ; 1}
val future2 = Future { Thread.sleep(10000) ; " Finished " }
val future3 = future1.flatMap( x => future2.map( y => println((x,y) ) ))
// on attend tous les futures avant de continuer le programme
// Duration.Inf veut dire on attend indéfiniment
Await.result(future3, Duration.Inf)

val end = System.currentTimeMillis()
println("Le programme a mis " + (end-start)/1000 + " secondes")
```

### *Akka, modèle acteur*

C'est une librairie externe qui utilise des messages pour faire communiquer des acteurs en eux. Un acteur est une unité de calcul avec une file qui reçoit des traitements à faire.

Les messages peuvent être **fire&forget** ou avec **accusé réception**. Les acteurs se contactent en utilisant leur nom et pas leur type

Les acteurs appartiennent à un ActorSystem

### Test

- **ScalaTest** : rien de nouveau, on utilise déjà.
- **ScalaCheck** : permet de faire des tests génériques avec des jeux de données importants et randomisés (*forAll*)

## Références

### web

- https://www.scala-exercise.org : exercice sur scala et quelques librairies
- https://www.scala-lang.org/ : API du langage Scala

### Livres

- Programming in Scala (Martin Odersky, Lex Spoon, Bill Verners)
- Scala in Depth (Joshua Suereth)
- Functionnal Programming in Scala (Paul Chiusano, Runar Bjarnason)

### Outils

- http://ammonite.io/ : REPL ammonit
- http://scala-ide.org/ : IDE Eclipse pour Scala
- https://www.jetbrains.com/idea/ : IDE IntelliJ IDEA community

[i] MapR ne gère pas debian mais la version LTS de ubuntu, pour se faire passer pour la bonne distribution il faut éditer /etc/lsb-release

```
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=14.04
DISTRIB_CODENAME=trusty
DISTRIB_DESCRIPTION="Ubuntu 14.04.4 LTS"
```