Le génie pour l'industrie

Rapport de PFE

Présenté à l'École Technique Supérieure

Modernisation d'ADVANCE avec ADAM et Spark

Simon Grondin

GROS15059104

Département de Génie Logiciel

Professeur Alain April

Montréal, le 1 mai 2016

Automne 2015 - Hiver 2016

Table des matières

Modernisation d'ADVANCE avec ADAM et Spark

Terminologie employée

CRCHUM: Centre de Recherche de l'hôpital CHUM (Centre Hospitalier Universitaire de Montréal). Le CRCHUM recevra et utilisera les livrables de ce projet.

Dataset: Un ensemble de données formant un tout cohérent. Le CRCHUM utilise plusieurs datasets différents, tels qu'ADVANCE, MONICA et autres.

ADVANCE: Le nom d'un des datasets utilisés par le CRCHUM. Il s'agit de données sur des patients, les tests qu'ils ont suivis, ainsi que les résultats du génotypage de ces patients.

Postgresql: Un système de base de données relationnel. C'est un logiciel libre. Les données du CRCHUM sont actuellement chargées dans Postgresql.

ADAM: Une hiérarchie de types représentant les entités en lien avec les activités de recherche en génétique.

Spark: Un logiciel libre facilitant la coordination et l'exécution de tâches génériques sur des systèmes distribués de grande taille.

Parquet: Un format de fichier orienté colonnes, basé sur des types hiérarchiques. Ce format convient bien à Spark et ADAM.

SQL: Langage déclaratif basé sur la logique relationnelle. Utilisé par Postgresql ainsi que Spark via Spark SQL.

Python: Langage de programmation simple, utilisé majoritairement dans un style impératif et synchrone. Ce langage est connu et utilisé par les bioinformaticiens du CRCHUM.

Scala: Langage de programmation multi-paradigme, sur lequel Spark et un bon nombre de technologies dites "Big Data" sont basées.

Aperçu

Problèmes

Le CRCHUM a un besoin urgent de moderniser son infrastructure logicielle.

- Cohérence: À l'heure actuelle, le système existant est excessivement complexe dû à des années de croissance incontrôlée. Des tables sont importées de bases de données complètement disjointes sans contrôle de cohérence. Les identifiants uniques de certaines tables ne correspondent à rien, certaines colonnes de référence ne mènent nulle part, etc.

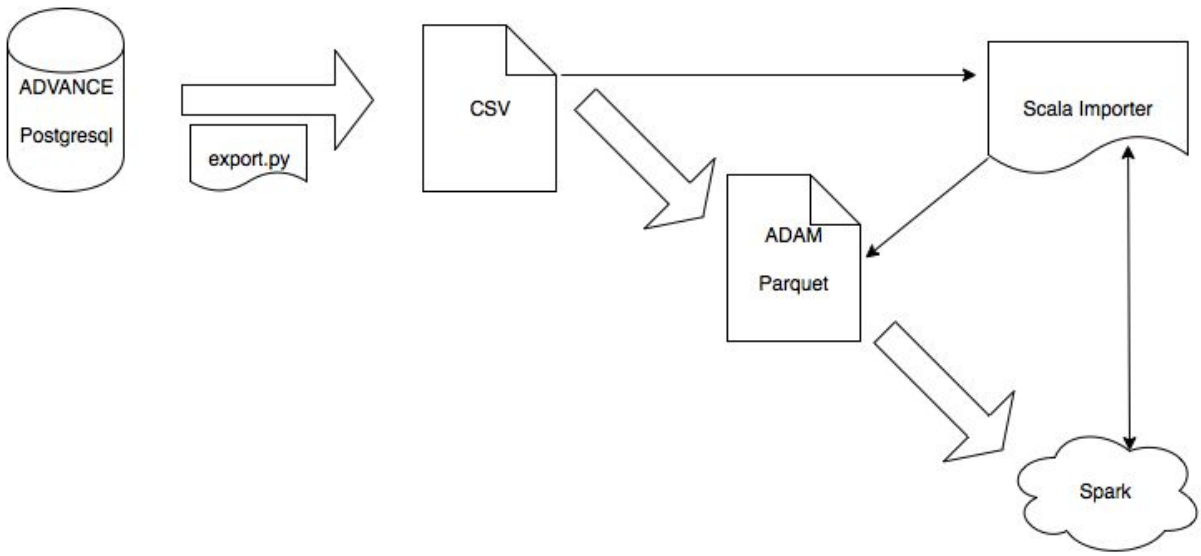
- Performance: L'architecture actuelle peine à suffire à la taille des données, ce qui limite le potentiel d'effectuer des analyses et des études sur des datasets plus grands.

- Traçabilité: Il n'y a aucune traçabilité ou historique. Il est impossible de recréer et de valider les résultats d'une étude, car le dataset sur lequel une étude est basée est en constante évolution.

Objectif

Exporter la base de données ADVANCE, présentement dans le format utilisé par Postgresql, vers un état intermédiaire basé sur des fichiers CSV. Ensuite, réécrire ces fichiers intermédiaires dans le format Parquet pour pouvoir les lire et les interroger avec Spark SQL.

Ces étapes sont représentées par le diagramme simplifié suivant:



Architecture

Il y a ainsi 2 modules dans ce système. Celui d'exportation, écrit en Python, et celui d'importation, écrit en Scala. Je référerai à ces modules par les noms anglais "Exporter" et "Importer".

L'idée de séparer le processus en 2 étapes vient de David Lauzon. En utilisant un format intermédiaire simple, l'Importer devient générique et peut ainsi accepter des données provenant de n'importe quel dataset. Le CRCHUM planifie justement de créer une version modifiée de l'Exporter pour chacun des datasets qu'ils désirent regrouper dans le système unifié final fonctionnant sur Spark.

Résolution

Les 3 problèmes majeurs suivants seront résolus à la suite de ce projet.

- Cohérence: L'Exporter corrige le plus de problèmes de cohérence que possible et s'assure que les données exportées sont correctes.

- Performance: Spark règle ce problème de manière transparente. En étant conçu dès le départ pour régler des problèmes de grande taille sur des systèmes distribués, Spark permet de gérer des centaines ou même des milliers de "noeuds" (nodes) d'exécution, qui travaillent chacun sur une fraction du problème. Spark assemble ensuite une version finale des résultats. S'attaquer à un plus gros problème ne demande souvent que de connecter plus de serveurs au système.

- Traçabilité: Les données chargées dans Spark sont immuables. L'unité de base dans Spark est le RDD (Resilient Distributed Dataset). Ceux-ci ne peuvent être changés sans créer un nouveau RDD avec les changements appliqués. Un nouveau RDD ne signifie pas automatiquement une copie des données. Une requête particulière sur un RDD particulier est garantie de toujours retourner les mêmes résultats. En gardant un historique des requêtes SQL utilisées dans une étude et sur quel RDD ces requêtes ont été effectuées, on peut ainsi reproduire les résultats exacts de cette étude.

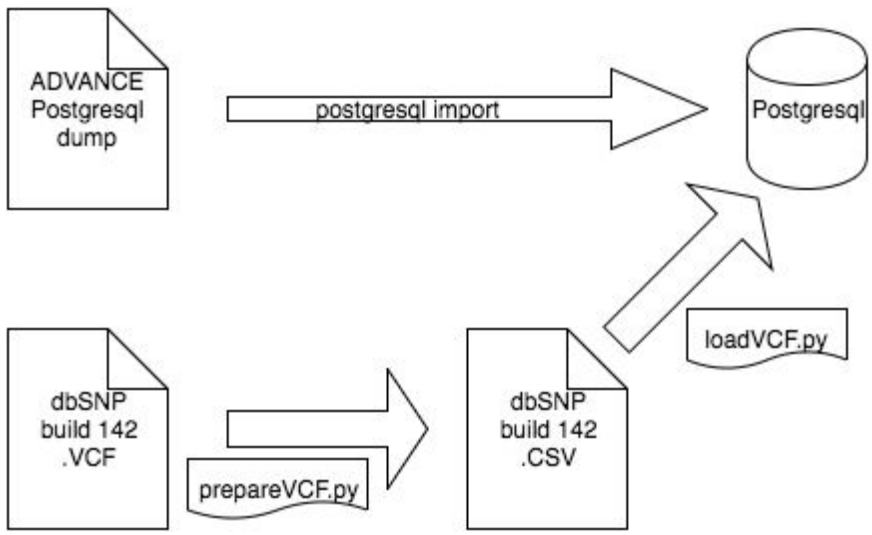
Implémentation de l'Exporter

Choix techniques

L'Exporter est entièrement écrit en Python. Il consiste de 4 fichiers: export.py, utils.py, prepareVCF.py et loadVCF.py. La raison pour le choix de ce langage provient du fait que les bioinformaticiens du CRCHUM pourraient avoir besoin d'y apporter des modifications et que Python est bien connu et facile à apprendre. Le style du code est très rudimentaire. Python est de plus un des langages ayant une librairie mature pour Spark. Il est beaucoup plus simple que Scala et il semble que les futurs développements de ce projet seront majoritairement écrits en Python.

Le fichier requirements.txt (nom standard) peut être utilisé pour installer automatiquement la bonne version de la librairie psycopg2, utilisée pour communiquer avec Postgresql.

Chargement initial des données dans Postgresql



Une fois les données d'AVANCE chargées dans Postgresql, il faut aussi aller télécharger le dataset dbSNP (Single Nucleotide Polymorphism Database), version 142, disponible sur le site du NCBI (National Center for Biotechnology Information)

Le script prepareVCF.py crée un fichier CSV (5.6Gb) à partir du fichier VCF (26Gb) de dbSNP. Le script loadVCF.py peut ensuite charger ce CSV avec un import direct dans Postgresql, sans passer par la couche SQL. Il crée ensuite un index unique sur la colonne rs.

L'approche suggérée par David Lauzon était plutôt de convertir le VCF en un fichier Parquet et de charger les données de dbSNP dans Spark durant la phase d'importation. Le problème avec cette approche est qu'elle requiert une requête à Spark pour chaque variant du génotype de chaque patient, ce qui représenterait 10 milliards d'accès aléatoires et

séquentiels en lecture sur le dataset dbSNP. Spark n'est pas du tout optimisé pour cette tâche.

En ayant dbSNP dans Postgresql, il devient possible de faire une jointure entre les annotations de variants et dbSNP. Avec l'index mentionné précédemment, c'est même rapide et efficace.

prepareVCF.py

Arguments:

- fichier dbsnp.vcf

Le fichier de sortie est le nom du fichier d'entrée suivi de ".csv".

Ce script lit le VCF une ligne à la fois en ignorant les lignes commençant par le symbole #. Toutes les autres lignes sont converties vers un format de style CSV, sauf que le séparateur est le signe de dollar ($) au lieu de la virgule (,). La raison est que cela évite d'avoir à gérer des cas spéciaux pour les champs contenant des virgules.

Les lignes converties sont ajoutées à un "tampon". Lorsqu'il atteint 1000 éléments, son contenu est transformé en chaîne de caractères et écrit à la fin du fichier de sortie. Ceci apporte un gain de performance important en évitant d'aller faire un appel système pour chaque ligne. Il y a environ 150 millions de lignes dans dbSNP. Ce nombre varie par version.

loadVCF.py

Arguments:

- fichier dbsnp.vcf.csv provenant de prepareVCF.py
- hôte postgresql
- nom de la base de donnée postgresql
- nom d'utilisateur pour postgresql
- mot de passe pour postgresql

```
CREATE TABLE vcf
(
  id integer NOT NULL,
  chromosome text NOT NULL,
  "position" integer NOT NULL,
  rs character varying(32) NOT NULL,
  ref text NOT NULL,
  alt text NOT NULL,
  quality text,
  filter text,
  info text,
  CONSTRAINT vcf_pk PRIMARY KEY (id)
)
```

Les données sont chargées avec la command `COPY` et le délimiteur `$`. Finalement un index unique est créé pour permettre une jointure efficace.

```
CREATE UNIQUE INDEX vcf_index_rs
ON vcf
USING btree
(rs COLLATE pg_catalog."default")
```

export.py

Arguments:

- Dossier où écrire les fichiers intermédiaires
- hôte postgresql
- nom de la base de donnée postgresql
- nom d'utilisateur pour postgresql
- mot de passe pour postgresql

Structure du dossier après complétion, exemple avec 2 patients:

```
output/
├── 1
│   ├── Individuals_1_Phenotypes.tsv
│   ├── Individuals_1.tsv
│   ├── Individuals_1_v1_Genotypes.tsv
│   ├── Individuals_1_v2_Genotypes.tsv
│   └── Individuals_1_Visits.tsv
├── 2
│   ├── Individuals_2_Phenotypes.tsv
│   ├── Individuals_2.tsv
│   ├── Individuals_2_v1_Genotypes.tsv
│   ├── Individuals_2_v2_Genotypes.tsv
│   └── Individuals_2_Visits.tsv
└── Batches.tsv
```

Le nom de chaque sous-dossier correspond à l'identifiant unique du patient dans ADVANCE.

Il y a 5 types de fichiers émis, en ordre de création : Individuals, Visits, Phenotypes, Genotypes, Batches.

Individuals

1 requête SQL.
Tables ADVANCE utilisées: Person, Batches
Type de jointure: LEFT JOIN

Cette requête lit toutes les informations utiles pour tous les patients d'un coup et crée leur sous-dossier avant d'aller créer leur fichier Individual. Le type de jointure utilisé permet d'aller chercher l'entièreté des patients, même ceux n'étant pas associés à une batch (environ 1500 sur 5000). Une batch est une étude sur plusieurs patients.

Une particularité de ce fichier est que le nombre de colonnes dans le CSV est variable. La relation Patient - Batch étant de type Un à Plusieurs, chaque fichier Individual aura un nombre de colonnes équivalent à 11 plus le plus grand nombre de batch par patient détecté dans le dataset. Pour ADVANCE, ce nombre est 3, donc chaque fichier individu contient 3 colonnes dynamiques: `batch0`, `batch1`, `batch2`.

Les fonctions `ARRAY_TO_STRING` et `ARRAY_AGG` lorsque combinées permettent de créer une aggrégation qui concatène des chaînes de caractères. En d'autres mots, c'est un `SUM()` pour les types `text` et `varchar`. Cette technique est utilisée pour regrouper les batches par patient.

Visits

1 requête SQL.
Table ADVANCE utilisée: Visit
Aucune jointure

Toutes les visites de tous les patients sont lues d'un coup. En les triant par numéro de patient et en comparant le numéro de patient de la ligne en cours avec celui de la ligne précédente, on peut écrire chaque fichier Visits (un par patient) en une seule passe et changer à un autre patient seulement lorsque le numéro change.

Phenotypes

1 requête SQL.
Tables ADVANCE utilisées: Phenotype, Measure
Type de jointure: LEFT JOIN

Ce fichier fonctionne exactement de la même façon que le précédent, sauf que les données proviennent de 2 tables au lieu d'une. Pas tous les phenotypes ne sont associés à une mesure, mais lorsqu'ils le sont, c'est du Un à Un, ce qui explique le type de jointure. Il y a aussi une ligne apparemment corrompue qui doit être exclue dans la clause WHERE.

Genotypes

2 requêtes SQL.
Tables ADVANCE pour la première: Snp_genotype, Person
Type de jointure: INNER JOIN
Tables ADVANCE pour la deuxième: Snp_annotation, Vcf (chargée par loadVCF.py)
Type de jointure: LEFT JOIN

C'est ici que plus de 99.9% du temps d'exécution se passe. Chaque patient est associé à (présentement) 2 versions de génotype. La première requête extrait les données du patient et ses données de génotypage. Ces données sont représentées par une longue chaîne de 1863892 (près de 2 millions) caractères de long. Il y a une telle chaîne par patient par version, donc plus de 10 000 au total. Chaque position dans la chaîne est liée à une annotation, contenant des informations sur ce variant.

Par exemple, l'annotation 5 contient des informations qui sont valides pour tous les variants à l'index 5 de toute chaîne. On peut donc réutiliser les résultats de la deuxième requête pour les 10 000 chaînes. Cette optimisation à elle seule fait en sorte que le script export.py prend un peu moins d'une journée à s'exécuter au lieu d'environ 36 jours. Cette technique n'aurait pas été possible sans avoir dbSNP dans Postgresql avec ADVANCE.

Tout comme pour Visits et Phenotypes, c'est avec une clause ORDER BY que ce gain de performance est possible. Les données de génotypage sont retournées triées par
`person_id ASC, genotype_version ASC`
et les annotations par
`annotation_version ASC, genotype_index ASC`

Supposons que chaque genotype ne contienne que 3 variants (au lieu de 2 millions), que nous avons 2 patients (au lieu de 5000) et que tous les patients ont 2 versions de leur genotype (ce qui exact).

Résultats de la première requête:

ID	version	genotype
1	1	21x
1	2	01x
2	1	242
2	2	142

Résultats de la deuxième requête:

index	chromosome	position	etc	etc
0	7	31183071		
1	12	189807684		
2	X	216957281		

Ce qui est écrit dans le fichier Genotype_1_v1:

value	chromosome	position	etc	etc
2	7	31183071		
1	12	189807684		
x	X	216957281		

Ce qui est écrit dans le fichier Genotype_1_v2:

value	chromosome	position	etc	etc
0	7	31183071		
1	12	189807684		
x	X	216957281		

Le curseur de la deuxième requête est "rembobiné" au début après avoir consommé chaque chaîne `genotype`.

Et ainsi de suite pour les fichiers Genotype_2_v1 et Genotype_2_v2.

Batches

2 requêtes SQL.
Table ADVANCE pour la première: Batches
Aucune jointure
Table ADVANCE pour la deuxième: Samples
Aucune jointure

Il n’y a qu’un seul fichier Batches. Il contient les infomations à propos des batches dont les patients peuvent faire partie.

utils.py

Ce fichier n’est pas exécuté directement. Il contient des utilitaires et des fonctions utilisées par export.py. En autre, c’est là que se trouve toute la logique de nettoyage des données ADVANCE avant l’insertion dans un CSV, ainsi que la logique pour générer des fichiers CSV automatiquement à partir de curseurs SQL. Une connaissance de la syntaxe de “tranchage de listes” (list slicing) en Python est requise.

Implémentation de l'Importer

Choix techniques

L'Importer est écrit en Scala pour plusieurs raisons. La plus importante est que c'est le langage derrière Spark, ADAM et une bonne partie de l'écosystème "Big Data". ADAM est modélisé dans le format AVPR (Avro Protocol) et des classes Java sont automatiquement générées à partir de ce modèle. Scala étant un langage fonctionnant sur la JVM, il est ensuite possible de créer des objets de ces classes Java. Adam-IBS, un projet d'Ivan Kizema, alumni d'Alain April, comprend un module "Persistence", capable d'écrire des fichiers Parquet à partir de classes ADAM. Ce module a été réutilisé et adapté pour l'Importer.

Structure du projet

SBT (build.sbt)

L'Importer est construit sur SBT, le Scala Build Tool. SBT offre un format de définition des dépendances beaucoup plus attirant que Maven. SBT télécharge automatiquement toutes les dépendances et même la version exacte de Scala requise par le projet. Démarrer l'Importer est aussi simple que `run <arguments>` dans SBT.

Ivan Kizema a réussi un tour de force en trouvant une combinaison (assez complexe) de versions des librairies étant compatibles entre elles. Un grand désavantage des langages JVM est que chaque librairie ne peut être incluse qu'une seule fois dans un projet et donc il est parfois impossible de trouver une combinaison sans conflits de versions. De plus, l'insistance de David Lauzon pour explorer Adam-IBS à la recherche de composantes réutilisables a aussi grandement contribué à ce projet.

L'Importer est composé de 4 fichiers: Importer.scala, DirectoryHandler.scala, IndividualRecord.scala et ParquetWriter.scala.

Importer.scala

Ce fichier contient le "main", donc le point d'entrée.
Arguments:

- Version du génotypage à utiliser (exemple: v2)
- Nom des données de références (exemple: NCI142)
- Nombre de threads

Ces arguments sont passés au programme par SBT.

L'Importer commence tout d'abord par charger le fichier Batches.tsv. Chaque batch lue est indexée dans une HashMap qui sera référencée plus tard. La liste des sous-dossiers est ensuite transformée en une liste parallèle supportée par le nombre de threads spécifié en argument. Les structures de données parallèles sont une fonctionnalité intéressante de

Scala. Cette liste parallèle appelle le DirectoryHandler sur le dossier et passe le résultat au ParquetWriter. Le programme se termine lorsque tous les dossiers ont été consommées.

DirectoryHandler.scala

C'est dans ce fichier que les objets ADAM sont créés et sérialisés. L'appel au DirectoryHandler prend un nom de sous-dossier en argument. Ce sous-dossier est lu et chaque fichier est ensuite passé dans une expression régulière avec extraction pour aller chercher le type de fichier (Genotype, Phenotype, etc). Le `pattern matching` de Scala rend cette tâche simple et le résultat élégant.

De nombreux objets ADAM sont donc créés, mais puisque ni l'ordonnancement des fichiers lus et ni la présence de tous les fichiers nécessaires ne sont garantis, ces objets ADAM sont placés dans une structure temporaire de type IndividualRecord, décrite au chapitre suivant. Une fois tous les fichiers consommés, la méthode isReady de cette structure est appelée. Si tous les éléments requis sont présents, l'objet ADAM Individual avec toutes ses composantes est assemblé et immédiatement sérialisé en un Buffer qui sera consommé par le ParquetWriter.

Une note importante sur le `parsing`. ADAM est fortement typé: certains champs sont des Long, d'autres des chaînes de caractères, etc. Les champs dans un CSV ne sont que des listes de chaînes de caractères, il faut donc faire une conversion pour les types Date, Long et Boolean. Cette conversion peut échouer et lancer une exception lorsque le champ est vide dans le CSV ou lorsque la valeur ne peut être convertie correctement. Cette situation est commune et pour éviter des problèmes, les conversions sont faites à l'intérieur d'une lambda `ignore` créée pour ce projet, qui comme le nom l'indique, ignore les erreurs de conversion et passe au champ suivant.

IndividualRecord.scala

Classe très simple pour contenir les objets ADAM intermédiaires créés par le DirectoryHandler et les assembler une fois que toutes les données nécessaires sont présentes.

ParquetWriter.scala

Ce fichier a été récupéré du projet Adam-IBS. Chaque instance de type ParquetWriter est associée à un fichier Parquet sur le disque. Un Buffer est passé en argument et immédiatement écrit dans un fichier utilisable par Spark.

Autres aspects techniques

Taille des données

On a ici affaire à environ 50Gb de données brutes (ADVANCE + dbSNP) et dans le futur le CRCHUM prévoit avoir à traiter des dizaines ou même des centaines de fois plus de données. La performance du code n'est pas critique, mais à cette échelle même des détails d'implémentation mineurs ont un impact.

Il faut aussi penser en terme d'itérateurs au lieu de listes ou de tableaux. Les lignes doivent être lues une à la fois pour éviter de faire planter l'application avec une erreur OOM (Out Of Memory).

Le format du résultat final est affecté par cette contrainte. Au lieu d'un seul "mégafichier" contenant tous les individus, l'Importer crée donc un fichier Parquet par individu. Le format Parquet est orienté colonnes, c'est donc impossible d'écrire un "mégafichier" sans avoir tous les individus en mémoire. Néanmoins, une fois que tous les individus ont été exportés sur le disque dans le format Parquet, il est possible d'utiliser la commande `coalesce` de Spark pour les joindre en un seul fichier.

Fragilité

ADVANCE contient énormément de champs NULL et peu de `constraints` SQL. C'est ensuite exporté vers le format CSV, où tout est une chaîne de caractères. Finalement, c'est importé dans le format ADAM, qui a zéro validations ou contraintes et presque tous les champs peuvent être NULL. C'est extrêmement facile d'introduire des fautes de frappe ou d'oublier des lignes en choisissant le mauvais type de jointure. Aucune erreur n'est détectée automatiquement, il y a donc de bonnes chances que certains problèmes se soient faufilés malgré toute l'attention portée aux détails.

Parallélisme

L'Exporter est limité par la vitesse du disque plus que la lenteur de Python, il n'y a pas de gain réalisable en parallélisant le code. L'Importer doit lire 45Mb et écrire 16Mb par Individu, donc on est limité encore une fois par la vitesse du disque, mais puisqu'instancier et sérialiser des structures ADAM est assez lourd sur le CPU, passer un nombre de threads égal à 2 à l'Importer peut doubler le taux d'Individus par minute (à environ 20 par minute avec un SSD), car un thread peut utiliser le disque de manière synchrone pendant que l'autre maximise le CPU.

Futurs développements

Il est à noter que le projet actuel est toujours en flux. Les bioinformaticiens ont demandé que quelques changements soient apportés au format CSV intermédiaire et aux données ADAM. L'Importer actuel requiert une version non-officielle du format ADAM, qui se trouve dans un `repository` du groupe GELOG sur Github. Les changements demandés par le CHUM font en sorte qu'il faudra encore une fois créer une nouvelle classe ADAM. Ces modifications au standard ADAM sont loin d'être certaines d'être un jour approuvées par l'AMPLab de Berkeley.

Ce sont là des défis auquel Mohamad Awada, le successeur pour ce projet, devra répondre.