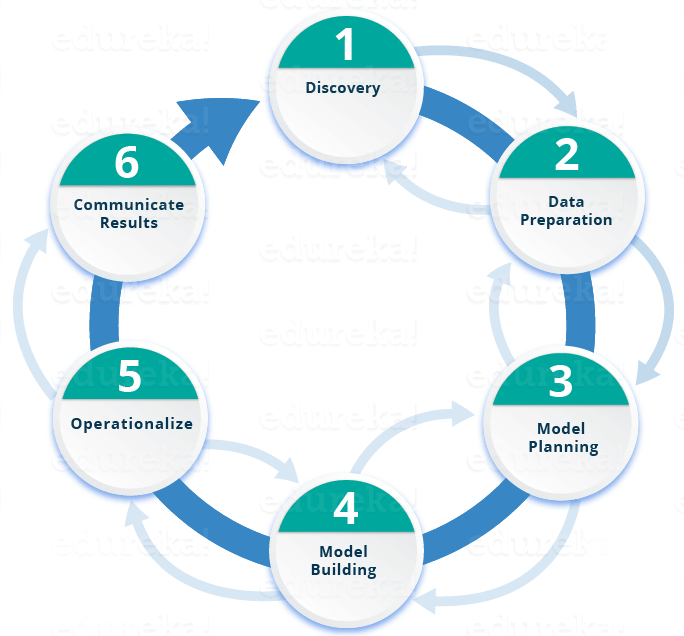
# Infrastructure pour le traitement des données

Après avoir expliqué en détails l’architecture qui nous a permis d’extraire et de stocker les 1.4 To de données, préalablement désencapsuler de leur format pcap d’origine, nous sommes maintenant en capacité d’étudier plus en détails les données fournies. Pour cela, nous allons appliquer la méthodologie utilisée pour la plupart des projets de data science (voir figure 1). Dans un premier temps, nous l’appliquerons dans le cadre de données « batch », puis nous montrerons comment nous avons répercuter cette analyse pour faire de même dans le cadre de données réceptionnées et analysées en temps réel *(simulé ici)*.

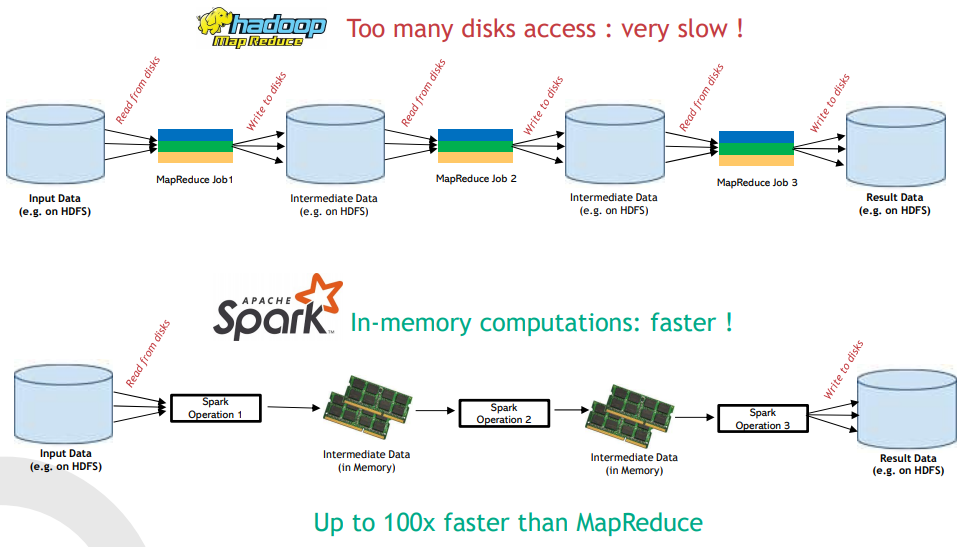
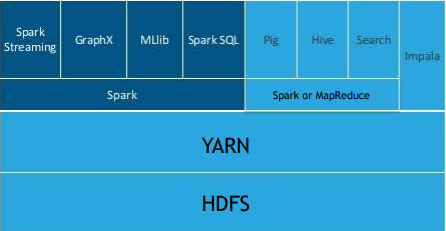
*Note au lecteur : L’ensemble du code python (ainsi que la configuration des différents nœuds Spark) qui nous a permis de mener à bien cette étude est disponible en fin de rapport.*

## Implémentation sur Spark et configuration des différents nœuds

### Justification du choix de Spark (pour l’analyse batch et temp-réel)

Les données fournies par Activus étant volumineuses, variables et variées, il est paru rapidement pertinent de distribuer la charge du travail des données (nettoyage, requêtes, entraînement des modèles et prédiction à venir) à venir sur plusieurs nœuds.

Hadoop Map Reduce et Spark étant les deux principales technologies utilisées (toutes deux étudiées cette année) pour le traitement des données massives découpées et stockées en HDFS, nous nous sommes finalement orientés vers Spark pour sa rapidité (stockage en mémoire cache des données contrairement à Hadoop Map Reduce dont la multiplication des appels de lecture et d’écriture sur disque auraient pu se faire sentir lors du clean et de l’apprentissage) – 100 fois plus rapide que Map Reduce, sa simplification dans le requêtage, son interactivité et sa flexibilité (RDD, Spark DataFrame et Spark SQL, Spark DataFrame API) et l’apprentissage mis-en-place directement sur les nœuds. Enfin, l’utilisation de Spark était, de toute façon, obligatoire pour notre analyse de données en temps-réel.



### Configuration des nœuds Spark

Partie Kader. *Besoin d’un schéma avec le driver program, le cluster manager et les 7 workers. Et une photo des workers en activité (peut-être au moment de la pred)*

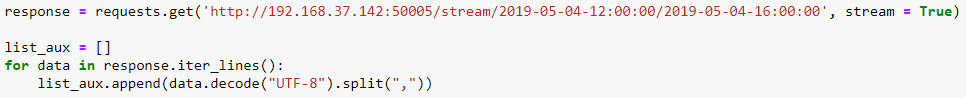
Notre cluster Spark contient un nœud master (driver node) et 7 nœuds travailleurs qui communique via un cluster manager. Le driver est responsable du déroulement du programme Spark : il crée le contexte et envoie des tâches aux nœuds travailleurs pour qu’ils les exécutent. Les nœuds travailleurs font tourner le résultat et se chargent d’envoyer la réponse au nœud master.

# Analyse des données batch

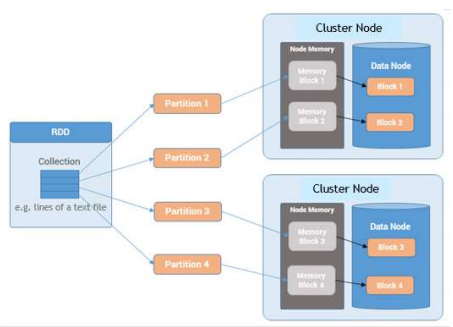
## Obtention des données grâce au 1er micro-service

## 

Grâce au 1er micro-service, détaillé dans la partie précédente, il est possible d’obtenir directement, toutes les trames radars qui ont été reçus par les radars sur une période de temps déterminée.



## Stockage de nos données en format RDD : utilisation de la librairie PySpark



Une fois les données entièrement reçues, nous les avons parallélisé sur Spark grâce au format Spark RDD pour garantir une scalabilité accrue dans le cas d’une requête sur une année complète par exemple. Un RDD correspond à une collection de données distribuées et stockées en mémoire sur les différents nœuds du cluster précédemment présenté. Ainsi, pour toutes les requêtes que nous avons effectués, même si nous ne manipulions localement qu’un objet, il était en fait distribué sur l’intégralité du cluster.

## Découverte et Nettoyage des données

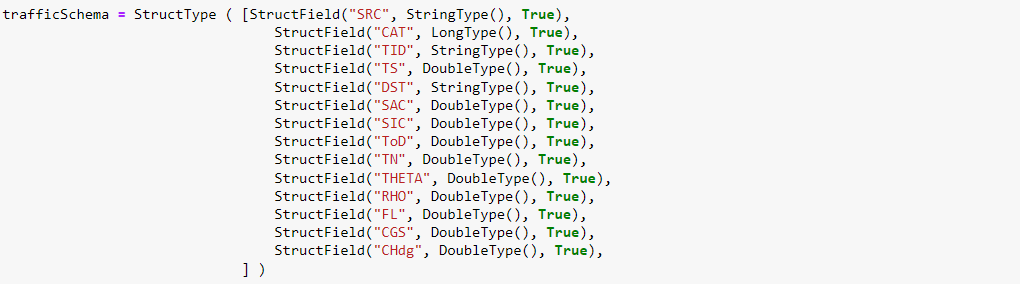
Dans la première partie de ce rapport, nous avions défini les champs présents dans les trames et ceux que nous souhaitions conservés pour analyse (resp. SRC\_MAC\_ADDRESS, CAT, TID, TS, DST, SAC, SIC, ToD, TN, THETA, RHO, FL, CGS, CHDG).

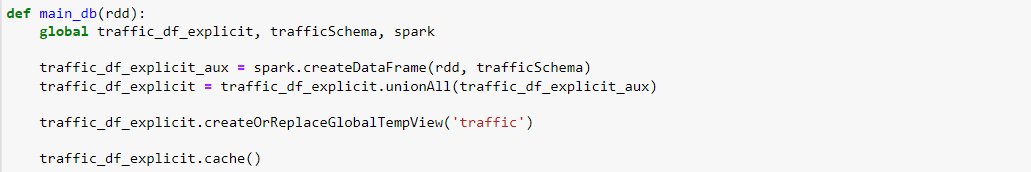
Avant analyse (requêtage, apprentissage, prédiction) et visualisation grâce à Spark SQL, nous avons souhaité apporter quelques modifications aux données reçues, notamment convertir au format numérique les variables quantitatives et remplacer toutes les valeurs manquantes par le même champ NULL. Tout cela s’est fait grâce à une opération de « transformation » par mapping et filtering du RDD existant et l’utilisation de lambda fonctions.

En outre, le fonctionnement de Spark nous a permis de réaliser de multiple transformations (dont le résultat n’était pas calculé immédiatement) et dont une action qui, elle, nous donnait un résultat immédiatement. Aucun traitement n’est effectué tant qu’aucune action est appelée ce qui minimise le temps de traitement et le trafic réseau et donc optimise le processus en réduisant la latence.

Nous avons également profité de la possibilité que nous offre Spark de garder en mémoire le RDD à analyser avant prochaine utilisation.

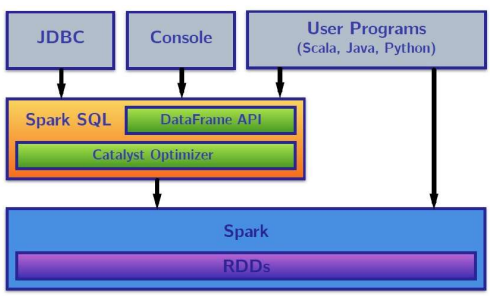
## Utilisation de Spark SQL et création d’une vue temporaire pour le requêtage

Nos données sont maintenant prêtes à être visualisé. Néanmoins, si nous devions requêter nos données depuis les RDD, un problème se poserait. Plus les requêtes deviendraient complexes, plus le risque d’augmentation de data shuffling (trafic réseau entre les nœuds travailleurs et le driver node) entre les nœuds s’accentuerait. Nous devons minimiser au maximum le data shuffling, même avec des requêtes extrêmement complexes. Une solution nous est offerte avec Spark SQL en passant d’un RDD à une dataframe pyspark.sql et une vue SQL temporaire (accompagné de son schéma) de cette dataframe.

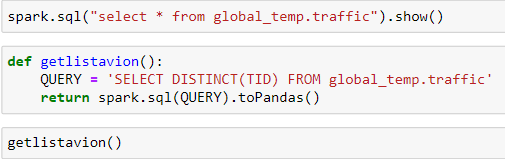


Ainsi, Spark SQL nous évite de nous poser des questions quant à l’ordre dans lequel nous devons utiliser les fonctions de mapping et de reducing (reduceByKey, groupByKey) puisqu’il optimise le code automatiquement en fonction de la requête SQL : il minimise tout seul le data shuffling.

C’est un véritable gain de temps pour l’utilisateur qui n’a juste qu’à écrire une requête SQL qui elle est traduite en RDD et réalise les opérations voulues sur toutes les partitions.



*NB : A ce stade, aucune action n’a encore été faite, seulement des transformations. Le traitement n’a donc pas encore lieu et n’aura lieu que lorsqu’une action sera effectuée (show() pour une dataframe, take() ou collect() pour un RDD).*

* Spark SQL est similaire à Hive
* Les exemples, ici, sont effectués sur un micro-sous-ensemble de la base des pcap. Cela fonctionnerait à l’identique pour les 1.4 Tera de la base tout entière.

## Visualisation des données

## Prédiction